Appunti di Ingegneria del Software

Questa repository è relativa ai contenuti del corso di Ingegneria del Software dei prof. Carlo Bellettini e Mattia Monga, per l'anno accademico 2022/23.

Gli appunti sono presi in collaborazione secondo i principi dello sviluppo di software *open source*. Contribuire non è solo un modo per studiare, ma anche per utilizzare Git e alcuni concetti dell'Ingegneria del Software in un contesto reale.

I *maintainer* ufficiali sono stati:

- Marco Aceti;
- Daniele Ceribelli;
- Matteo Mangioni.

ma *chiunque* può contribuire e proporre *Merge Request*.

Nel file Autori sono presenti tutti i contributori per ogni lezione.

Se utilizzi questi appunti per studiare, non solo è *galante* contribuire ma è anche un modo per controllare e sistemare l'enormità di errori che prevediamo saranno presenti e per espandere o integrare nozioni e concetti.

I docenti del corso sono a conoscenza di questo progetto e sembrano apprezzarlo.

La repository del codice è pubblicata sui seguenti *remoti*:

- **GitLab Dipartimento di Informatica**;
- **GitHub di Marco Aceti**.

Autori

Maintainers

I maintainers hanno la responsabilità di definire la suddivisione degli argomenti, approvare e mergiare le Merge Request in `master` e assegnare il lavoro ai contributori.

Inoltre, possono avere ulteriori responsabilità specifiche.

- **Marco Aceti**: direzione del progetto, definizione workflow, supporto tecnico, pipeline CI/CD;
- **Matteo Mangioni**: stesura delle styleguides, responsabile dei contenuti;
- **Daniele Ceribelli**: capo revisori.

Argomenti

Si precisa che l'attività di **revisione** non è una mera rilettura, bensì di **refactoring completo del testo** privilegiando la forma, *senza modificare i contenuti*.

L'ordine nelle celle non è casuale.

#	Titolo	Integratori	Revisori
01	Introduzione	Daniele Ceribelli, Marco Aceti	Matteo Mangioni
02	Modelli di ciclo di vita del software	Daniele Ceribelli, Marco Aceti	Matteo Mangioni
03	eXtreme Programming	Daniele Ceribelli	Marco Aceti, Matteo Mangioni
04	Open source	Marco Aceti, Daniele Ceribelli	Matteo Mangioni
05	Software Configuration Management	Marco Aceti	Armani Islam
06	Git workflow	Andrea Cambiaghi	Marco Aceti
07	Progettazione	Daniele Ceribelli	Marco Aceti, Andrea Cambiaghi, Armani Islam
09	Patterns	Daniele Ceribelli, Matteo Mangioni	Matteo Mangioni
10	Mocking	Ilyass Ouardi	Francesco Protospataro, Marco Aceti
11	UML	Francesco Protospataro	Marco Aceti
12	Verifica e convalida	Matteo Yon, Marco Aceti	Marco Aceti, Matteo Mangioni
13	Testing e processi di review	Marco Aceti, Matteo Yon, Mattia Mendecino, Andrea Cambiaghi	Matteo Mangioni, Marco Aceti

#	Titolo	Integratori	Revisori
14	Reti di Petri	Daniele Ceribelli	Marco Aceti, Matteo Yon
15	Analisi di reti di Petri	Daniele Ceribelli	Marco Aceti
16	Reti di Petri temporizzate	Armani Islam, Matteo Mangioni	Matteo Mangioni

Come contribuire

Stack tecnologico

Tutti i contenuti sono scritti in Markdown e quindi convertiti in HTML automaticamente da mdbook. Per strutture complesse, è possibile embeddare dell'HTML (e del CSS) nel file Markdown.

CONSIGLIATO: installazione di tutte le dipendenze tramite Docker

Prima di iniziare, è necessario avere Docker installato. Quindi:

1. entra nel branch `master` e sincronizzalo con l'ultima versione remota:

   ```
   $ git switch master
   $ git pull
   ```

2. costruisci l'immagine Docker:

   ```
   $ docker build -t appunti_sweng .
   ```

3. crea un container ed eseguilo, ricordandoti di:

   - mappare le porte 3000/tcp sul tuo host;
   - mappare la cartella del progetto a `/usr/src/app` nel container;
   - mappare correttamente l'utente;

   In ambiente UNIX, ho creato uno script che permette di fare tutte le cose di cui sopra con un comando. Per eseguirlo fare:

```
$ ./docker-run.sh
```

Aprendo la pagina https://localhost:3000/ nel nostro browser potremo visualizzare un'anteprima della pagina HTML compilata, aggiornata ad ogni modifica del file Markdown originale. È estremamente consigliato arrivare a questo punto prima di continuare: non inviare patch prima di aver verificato che mdbook compili il file in una pagina sensata.

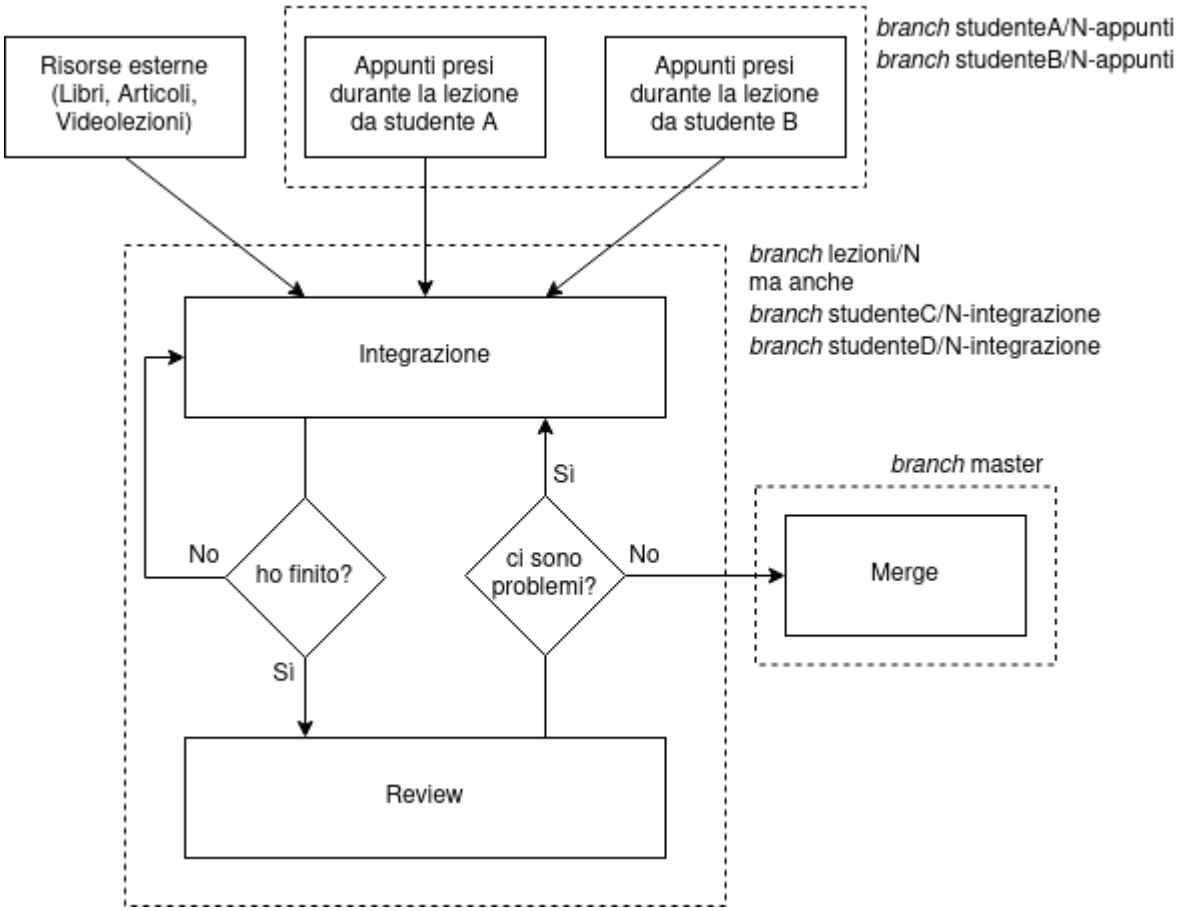
Oltre a mdbook, è naturalmente necessario avere Git installato sulla propria macchina. Come editor di testo, consigliamo VSCode (meglio ancora VSCodium) ma qualsiasi va bene.

Regole base di Git

- Utilizziamo GitLab e non GitHub perché abbiamo iniziato con GitLab e non abbiamo motivo per cambiare.
- Abilita l'autenticazione a due fattori.
- Consigliamo l'utilizzo dell'autenticazione SSH con GitLab.
- Imposta il tuo nome e cognome reale; per esempio: `git config --global user.name "Carlo Bellettini"`;
- Utilizza la tua email universitaria, se vuoi; per esempio: `git config --global user.email "carlo.bellettini@unimi.it"`. Ricordati di aggiungere l'email al tuo account GitLab.
- Consigliamo di impostare e attivare la firma dei commit tramite GPG.

The *"Silab Gang" Notes Engineering Development Process*

Per ogni **argomento** (spesso corrispondente a una *lezione*) N...



Ad ogni argomento corrisponde una directory sotto la directory di mdbook `src/`.

Organizzazione dei branch

La gestione dei branch è simile a GitFlow, ma non è uguale. Osserviamo le tipologie di branch:

- `master`: contiene l'ultima versione stabile di tutti gli appunti. Ad ogni commit viene azionata una GitLab CI che aggiorna la pagina su GitLab Pages. Solo i maintainer possono mergiare su questo branch.
- `lezioni/N` (ma forse avremmo dovuto chiamarli `argomenti/N`): contiene l'ultima versione *instabile* di uno specifico argomento. I contributori della lezione `N` pulleranno dal branch `lezioni/N` per sincronizzare i contributi degli altri nel proprio branch, e mergieranno (o chiederanno di mergiare) i propri contributi nel branch `lezioni/N`. Nel nome c'è un *leading zero*: `lezioni/04`, `lezioni/12`.
- **branch utente**: iniziano con il nome dell'utente (lower case e breve) e sono utilizzati come *"sandbox"* personale.

Tutti i branch devono essere creati partendo da `master`. È consentito e apprezzato il fast forward in caso di merge banali.

Essendo i branch `lezioni/N` condivisi, è **NECESSARIO** aggiornare il branch con il remoto facendo `git pull`. Prima di mergiare in `lezioni/N`, quindi:

1. entra nel branch `lezioni/N`: `git switch lezioni/N`;
2. scarica le ultime modifiche: `git pull`;
3. entra nel tuo branch: `git switch mio/branch`;
4. mergia il branch `lezioni/N`: `git merge lezioni/N`;
5. risolvi gli eventuali conflitti;
6. entra nel branch `lezioni/N`;
7. mergia il tuo branch: `git merge lezioni/N`;
8. carica le tue modifiche: `git push lezioni/N`.

> `lezioni/N` contiene sempre l'***ultima versione instabile*** e tutti i contributori la utilizzano come riferimento per quell'argomento. I contributi non mergiati in `lezioni/N` non saranno considerati da nessuno e sono quindi inutili. In ogni caso, non si committa mai direttamente a `lezioni/N` ma prima si passa sempre per un *branch utente*.

Esempio

Lo studente Carlo Bellettini prende appunti durante la (sua?) lezione 5 e crea un branch `carlo/05-appunti`. Anche lo studente Mattia Monga prende appunti e pubblica le modifiche su `mattia/05-appunti`. Carlo, da bravo contributore, si impegna a integrare gli appunti; crea il branch `carlo/05-integrazione` e mergia inanzitutto i suoi appunti (`carlo/05-appunti`) quindi quelli di Mattia (`mattia/05-appunti`).

Il secondo merge da parte di Carlo degli appunti di Mattia causerà sicuramente dei conflitti, che Carlo dovrà risolvere: non è codice, è testo, e due studenti prenderanno gli appunti in modo completamente diverso! Il concetto stesso di *integrazione* è proprio questo.

Una volta terminato il lavoro, Carlo mergierà il suo branch `carlo/05-integrazione` in `lezioni/05`, quindi aprirà una Merge Request da `lezioni/05` verso il branch `master`.

Inizia il processo di *review*: altri contributori (ovvero tutti a parte Carlo) controlleranno la correttezza e la **completezza** (!) degli appunti proposti. Se (ancora, per esempio) Marco trova dei problemi, può creare un proprio branch `marco/05-review` partendo dal branch `lezioni/05`, committare le proprie proposte e quindi rimegiarle in `lezioni/05`.

Infine, una volta che tutti i reviewer sono contenti, la Merge Request viene mergiata in master e gli appunti vengono aggiunti in GitLab Pages.

Issues e Merge Requests

Per coordinare il lavoro tra di noi, utilizziamo principalmente la funzione "Issue" di GitLab. Tutte le issues sono elencate qui.

C'è una issue per ogni argomento. Ogni issue...

- ha un titolo con il numero (corrispondente all'`N` nei nomi di branch) e al nome dell'argomento;
- ha una descrizione, contentente i riferimenti alle lezioni relative all'argomento (come la data) e altre note opportune (*"il prof. ha spiegato il pattern Observer in questa lezione"*, ...);
- ha un label per tracciare lo stato nel processo (Da Fare / In esecuzione / In attesa di review / Fatto);
- ha un epico per tracciare il progresso dei 4 macro argomenti del corso.
- ha un utente assegnato: solitamente è l'integratore principale della issue.

Nelle issue si può discutere e coordinare il lavoro, ma le review si fanno nelle merge request. Le osservazioni sul *processo* si fanno nell'issue, quelle sul *contenuto* nella merge request.

Le merge request sono collegate alla relativa issue semplicemente citandola. È possibile utilizzare la revisione GitLab per indicare i problemi: se trovi un problema sei invitato a risolverlo subito, per velocizzare il processo.

Convenzioni mdBook

Il Markdown scritto su mdBook è particolare e richiede l'utilizzo di alcune convenzioni, specialmente per lavorare insieme.

Nomi di file e intestazione

Tutti i *file* vanno creati in una sottocartella della cartella `src/`. Ogni file all'interno di `src/` DEVE avere come il seguente nome: `PROGRESSIVO_nome-argomento.md` dove PROGRESSIVO è il numero progressivo del file all'interno della cartella. I numeri progressivi iniziano da zero e hanno un *leading zero*.

Ogni file deve essere poi aggiunto in `SUMMARY.md`.

Diagrammi UML

(Quasi) tutti i diagrammi UML mostrati durante le lezioni dal prof. Bellettini sono generati utilizzando PlantUML, uno strumento open source che genera diagrammi in formato vettoriale partendo da del semplice testo. È quindi perfetto per il nostro caso d'uso (*pun intendend*).

La sintassi per generare un diagramma dal Markdown di mdBook è la seguente:

````
```plantuml
@startuml
Object <|-- ArrayList
Object : equals()
ArrayList : Object[] elementData
ArrayList : size()
@enduml
```
````

Informazioni complete sulla sintassi con esempi sono sul sito di PlantUML.

Oltre al plugin, per generare i diagrammi è necessario installare l'eseguibile `plantuml`. Nei sistemi UNIX-like:

1. in una cartella che vuoi (come nella HOME), scarica il file `.jar` con `$ wget https://github.com/plantuml/plantuml/releases/download/v1.2022.13/plantuml-1.2022.13.jar -O plantuml.jar`;
2. crea un file chiamato `/usr/bin/plantuml` avente come contenuto

```
#!/bin/bash
java -jar /path/to/plantuml.jar "$1" "$2"
```

3. rendi il file eseguibile: `$ sudo chmod +x /usr/bin/plantuml`.

Se possibile, **cerca sempre di utilizzare un diagramma UML al posto di uno screenshot**.

Convenzioni di stile e contenuto

Gli appunti devono essere chiari, concisi ma **completi**. L'obiettivo è creare la *bibbia* del corso: idealmente studiandola da zero si dovrebbe arrivare al 30L.

In tale prospettiva proponiamo una guida alle fasi di integrazione e di review che chiarifichi che cosa dev'essere presente negli appunti e lo stile di scrittura consigliato.

Naturalmente, queste indicazioni valgono per gli appunti proposti per il branch master: per gli appunti presi a lezione è assolutamente OK essere vaghi o brevi.

Guida all'integrazione

La fase di integrazione degli appunti dovrebbe servire per riunire gli appunti di tutti i partecipanti in un unico documento. Per agevolare la fase di review e riscrittura, tuttavia, questo non può limitarsi a un semplice *merge* dei rispettivi file: l'integratore ha il compito di fornire a colui che dovrà riscrivere gli appunti la miglior base possibile su cui lavorare.
Ecco dunque alcuni consigli utili in tal senso:

- Assicurarsi che **CI SIA TUTTO**: idealmente la fase di review dovrebbe solo fare "refactoring" degli appunti senza aggiungere nessun concetto, per cui è espressa responsabilità dell'integratore assicurarsi che il risultato finale sia assolutamente completo in quanto nessuno controllerà più i contenuti.

- **Una frase, una riga**: al termine di ciascuna frase (*ndr. una proposizione terminata da punto*) andare a capo in Markdown. Questo infatti non spezza il paragrafo, come si può vedere dalla preview, ma agevola moltissimo il versioning con Git in quanto ogni frase viene così trattata come una linea di codice indipendente dalle altre.

- Assicurarsi che le stesse cose non siano dette in più punti diversi e, nel caso, integrarle tra di loro;

- Tenere i propri appunti sottomano per accertarsi che ogni concetto citato a lezione sia riportato: è chiaro che all'esame viene chiesto *tutto*, compresi i riferimenti esterni, per cui occorre includere negli appunti ogni nozione rilevante;

- Organizzare gli argomenti in maniera logica, evitando salti logici in avanti e in indietro per agevolare il lavoro di review;

- Sfruttare le potenzialità di Markdown (*es. titoli di vario livello, tabelle, elenchi...*) e rispettarne per quanto possibile le convenzioni (*es. linea vuota dopo i titoli, nessuno spazio alla fine di una riga...*);

- Tenere **sempre** la preview di mdbook aperta per verificare che immagini e/o schemi vengano mostrati correttamente.

Guida alla review

Gli appunti definitivi dovrebbero costituire un discorso omogeneo e fluido, come fossero un piccolo libro di testo.
Per fare ciò, ecco alcune accortezze di stile e consigli utili durante la fase di review e riscrittura: si tratta solo di indicazioni (*"Just rules"* ^-^), per cui non sentitevi in dovere di seguirle alla lettera.

Contenuto

- Immaginare sempre di stare parlando con chi non sa nulla della materia: leggendo gli appunti dall'inizio alla fine si dovrebbe essere in grado di comprendere tutto. È quindi importante:
    - non citare concetti senza che siano stati già spiegati precedentemente: se invece sono già stati spiegati può essere utile richiamarli con una formula del tipo *"Come sappiamo..."* o *"Come abbiamo già visto..."* seguita da un breve accenno al concetto;
    - non dare per scontato nessuna conoscenza;
- Se qualcosa è preso pari pari dalle slide può essere un campanello d'allarme. Conviene dunque farsi le seguenti domande:
    - La frase si sposa bene con lo stile del discorso? Come potrei riscriverla in modo da rendere il fluire del discorso più omogeneo?
    - Il concetto espresso non è affrontato da nessun'altra parte? Se sì, tale ripetizione è davvero necessaria e funzionale?
- Mantenere convenzione "una frase, una riga" adottata nella fase di integrazione (*vd. sopra*): specialmente nella fase di review è importante che modificare una singola frase non comporti modificare interi paragrafi.
- Tenere i propri appunti sottomano per verificare ulteriormente che non manchi nulla: sebbene la fase di integrazione dovrebbe in teoria creare un documento completo di tutto, può capitare che qualcosa sia sfuggito.

Stile

- Adottare una **sintassi semplice**: gli appunti dovrebbero essere completi ma facili da seguire;
- Avere una qualche estensione di **controllo ortografico** attiva (*es. Code Spell per vscode*);
- Usare il più possibile l'**impersonale**: non *"possiamo fare X"* ma *"si può fare X"*;
- Sforzarsi di presentare gli argomenti nel modo più chiaro possibile, legandoli tra di loro in unico discorso logico. Per favorire questo approccio, ogni argomento dovrebbe essere affrontato nel modo seguente:
    1. **Presentare il problema**: es. *"Spesso capita di dover gestire X, Y e Z"*;

2. **Discutere e analizzare il problema**: es. *"Il problema ha queste queste e queste caratteristiche, che non possiamo risolvere con quanto visto finora"*;

3. **Proporre la/le soluzione/i al problema e discuterle**, confrontandole se più di una: es. *"In un primo momento si potrebbe pensare di risolvere così; tuttavia questo approccio ha questi difetti. Ecco allora che si è pensato di fare quest'altro"*;

4. **Concludere con un breve riassunto** su quanto visto, che servirà inoltre a introdurre il prossimo argomento: es. *"Abbiamo quindi visto come risolvere sta cosa; la soluzione pone però un nuovo problema..."*.

- Preferire i discorsi omogenei agli elenchi: usare gli elenchi **SOLO** quando necessari. Alcuni esempi dei pochi casi in cui un elenco è accettabile sono:

    - un **elenco puntato** quando si elencano più cose contrapposte tra di loro (*es. diversi approcci o soluzioni a un problema*)

    - un **elenco numerato** quando si specificano le varie fasi di un processo (*es. ciclo di vita del software*)

- Utilizzare la **separazione in paragrafi** in modo coscienzioso: all'interno di un paragrafo dovrebbe idealmente essere trattato *un unico concetto*. Diversi aspetti dello stesso concetto possono essere separati nello stesso paragrafo andando a capo (*con \ al termine della riga*), mentre quando si passa al concetto successivo è bene aprire un nuovo paragrafo.

- Utilizzare la **corretta punteggiatura**. Può essere utile in tal senso rileggere mentalmente gli appunti appena scritti per assicurarsi che il discorso fluisca in modo scorrevole, ricordando che:

    - la virgola (",") rappresenta una pausa brevissima utilizzata per riprendere fiato o per evidenziare tramite un inciso (*una frase compresa tra due virgole*) determinati concetti che espandono in modo significativo il discorso principale;

    - i due punti (":") rappresentano una pausa breve e sono usati per introdurre elenchi o proposizioni strettamente correlate con quella principale;

    - il punto e virgola (";") rappresentano una pausa media, e vanno utilizzati quando si vuole dare un legame debole alla proposizione con la precedente e al termine di ogni elemento di un elenco tranne l'ultimo (*dove invece si usa il punto*);

    - le parentesi ("(...)") vengono utilizzate per incapsulare proposizioni che espandono la frase principale in modo non significativo: idealmente esse potrebbero essere saltate nella lettura senza togliere nulla al discorso.

- Utilizzare il **grassetto** per evidenziare concetti chiave e il *corsivo* per sottolineare frasi importanti; all'interno delle parentesi si può inoltre utilizzare il corsivo per aumentare la rilevanza del contenuto.

- Usare le congiunzioni correttamente per legare le frasi tra di loro (*es. dunque, perché, perciò, allora, in quanto...*);

- **NON IGNORARE I COMMENTI**: si tratta di richieste di aiuto da parte di chi ha fatto l'integrazione, che chiede un consiglio su una determinata questione. È dunque importante che per il termine della review tale problema sia stato risolto: se vi trovate in difficoltà potete sempre chiedere sul gruppo!

Ingegneria, qualità e processi

In questa lezione verranno trattati i seguenti argomenti.

- **Informazioni logistiche**: orari e modalità d'esame.
- **Ingegneria del software**: di cosa si occupa la materia?
- **Qualità del software**: quali qualità *misurabili* ha un software?
- **Processo di sviluppo**: quali fasi e processi contraddistongono lo sviluppo di un software?

Informazioni logistiche

- Non ci sarà lo streaming però ci sono le videolezioni
- Teoria
  - Lun 14:30-17:00 Aula 403
  - Mer 14:30-17:00 Aula 403
- Laboratorio
  - Gio 13:30-17:30 due turni equivalenti
    - Turno A matricole pari
    - Turno B matricole dispari
  - Due persone per computer, a coppia
- Non c'è libro di testo, ma consigliati:
  - Software Engineering (Carlo Ghezzi, Dino Mandridi)
  - Design Patterns (Eric Freeman, Elisabeth Robison)
  - Handbook of Software and Systems Engineering (Albert Endres, Dieter Rombath)

Esami

- **Laboratorio**
  - prova pratica di laboratorio di 4 ore
  - OPPURE *per chi segue tutti i laboratori* ci saranno due laboratori valutati A COPPIE
- **Teoria**
  - prova orale
- Prima di fare l'orale bisogna fare il laboratorio
- La prova di laboratorio vale all'infinito

Ingegneria del software

Storia

Con la diffusione dei primi computer in ambito accademico, negli anni '50 e '60 si è subito colta la necessità di superare metodi di produzione "artigianale" del software: sebbene il cliente e il programmatore coincidessero e i programmi fossero prettamente matematici, si iniziavano già a vedere i primi problemi. Negli anni '70, si inizia dunque a pensare a dei metodi, dei processi e a degli strumenti che potessero migliorare e ***"assicurare"*** la **qualità del software**, sviluppando un approccio di tipo ingegneristico costituito da una serie di fasi.

Approccio ingegneristico

1. **Target**: ci si prefigge un obiettivo da raggiungere.
2. **Metric**: si definisce una metrica per misurare la qualità del software, ovvero quanto esso si avvicina al target prefissato.
3. **Method, Process, Tool**: si provano una serie di metodi, processi e strumenti per avvicinarsi all'obiettivo.
4. **Measurements**: si misura tramite la metrica stabilita se le strategie implementate sono state utili e quanto ci hanno avvicinato (o allontanato) all'obiettivo. A seconda dei risultati ottenuti vi sono due possibili strade:
   - risultati soddisfacenti (*aumento delle metrica*) - accettiamo come buoni metodi e processi utilizzati.
   - risultati insoddisfacenti (*diminuzione della metrica*)- ci sono dei peggioramenti o dei forti effetti collaterali, di conseguenza bisogna modificare il lavoro qualcosa: si possono cambiare target o metrica se ci si rende conto di non aver ben definito l'obiettivo, ma più comunemente bisogna rivedere i processi e metodi usati.

Ma **che cosa si intede per target**? Gli obiettivi da raggiungere possono essere di due tipi: la risoluzione dei **problemi nella progettazione del software** e l'assicurazione di una qualche **qualità che il software dovrà avere**. È dunque necessario chiedersi le seguenti domande:

- Quali problemi ci sono?
- Quali qualità deve avere il software?

Problemi principali

Vediamo allora a questo punto alcuni dei problemi che possono insorgere durante lo sviluppo di software, partendo dal presupposto che una delle più grandi fonti di problemi sono le **persone**. L'obiettivo della disciplina è infatti principalmente quello di risolvere i **problemi di comunicazione**, che possono essere:

- tra il **programmatore** e il **cliente**: sono esperti di domini diversi ed è difficile comprendersi;
- tra un **programmatore** e altri **programmatori**.

Un'altra fonte di problemi sono le **dimensioni** del software, che possono raggiungere valori molto elevati in termini di milioni di righe di codice e migliaia di *"anni uomo"* di lavoro. Lo sviluppo software non è più piccolo e domestico, e questo crea chiaramente problemi di manutenzione e gestione della codebase.

Il software è infine **facilmente malleabile**, ovvero modificabile nel tempo: il moltiplicarsi di versioni, evoluzioni e variazioni di target può creare non poche difficoltà.

Qualità

Per fare fronte ai problemi sopracitati si sviluppano allora una serie di processi per lo sviluppo software: essi non assicurano la bontà del programma finito, ma possono assicurare la presenza di *proprietà desiderabili* del prodotto, dette **qualità**. Le qualità del prodotto, che costituiscono a conti fatti un "valore per le persone", si dividono innanzitutto in due tipi:

- **qualità esterne**: qualità che vengono colte dal cliente;
- **qualità interne**: qualità che vengono esclusivamente colte dallo sviluppatore.

Le qualità interne influenzano molto le qualità esterne (per esempio se ho un codice ottimizzato ed efficiente, il mio software produrrà i risultati più velocemente). Prima di vedere quali siano le proprietà auspicabili in un software, però, facciamo un'importante distinzione a livello terminologico tra **requisiti e specifiche**:

- I **requisiti** sono quello che il cliente vuole che il software faccia. Spesso sono cambiati in corso d'opera oppure sono espressi in modo sbagliato, per cui è necessaria un'interazione continua.
- Le **specifiche** sono ciò che è stato formalizzato dal programmatore a partire dai requisiti: si tratta di una definizione più rigorosa di che cosa dovrà fare il software. Si noti però che se i requisiti sono stati espressi in modo non corretto anche le specifiche risulteranno inesatte (vd. G1).

Fatta questa doverosa distinzione, vediamo quali sono le qualità che un software dovrebbe idealmente possedere.

Qualità del software

Un software di qualità deve ***funzionare***, ***essere bello*** e ***"farmi diventare ricco"***.

Un software deve...	Qualità	Descrizione
Funzionare	**Correttezza**	Un software è corretto se soddisfa la specifica dei suoi *requisiti funzionali*. Si tratta di una proprietà *"matematica"* e relativamente dimostrabile in modo formale.
	Affidabilità	Un software è affidabile quando ci si può fidare del suo funzionamento, ovvero ci si può aspettare che faccia ciò che gli è stato chiesto. Se è molto difficile perseverare la correttezza, in quanto si tratta una proprietà assoluta, l'affidabilità è invece relativa: un software può essere affidabile (o *dependable*) nonostante contenga qualche errore.
	Robustezza	Un software è robusto se si comporta in modo accettabile anche in circostanze non previste nella specifica dei requisiti, senza generare effetti troppo negativi.
Essere bello	**Usabilità**	Un software è usabile (o *user-friendly*) se i suoi utenti lo ritengono facile da utilizzare. Si possono fare degli esperimenti (le grandi aziende lo fanno) per testare e quantificare l'usabilità del software ponendolo di fronte a dei soggetti umani (vd. NN23).
	Prestazioni	Ad ogni software è richiesto un certo livello di prestazioni. L'efficienza è una qualità interna e misura come il software utilizza le risorse del computer; la performance, d'altro canto, è invece una qualità esterna ed è basata sui

Un software deve...	Qualità	Descrizione
		requisiti dell'utente. Essa ha effetto sull'usabilità, e spesso viene considerata alla fine dello sviluppo software visto che vari avanzamenti tecnologici possono efficientare algoritmi e processi prima troppo costosi.
	Verificabilità	Un software è verificabile se le sue proprietà sono verificabili facilmente: è importante essere in grado di poter dimostrare la correttezza e la performance di un programma, e in tal senso la **leggibilità** del codice è fondamentale. La verifica può essere fatta con metodi formali o informali, come il testing. È considerata una qualità interna, ma alcune volte può diventare una qualità esterna: per esempio, in ambiti in cui la sicurezza è critica il cliente può chiedere la verificabilità di certe proprietà.
	Riusabilità	Le componenti del software che costruiamo dovrebbero essere il più riutilizzabili possibile così da risparmiare tempo in futuro: ciò può essere fatto non legandole troppo allo specifico contesto applicativo del software che stiamo sviluppando. Con la proprietà di riusabilità, utilizziamo un prodotto per costruire - anche con modifiche minori - un altro prodotto (vd. MI15).
Farmi diventare ricco	**Manutenibilità**	Per *manutenzione software* si intendono tutte le modifiche apportate al software dopo il rilascio iniziale. Questa proprietà può essere vista come due proprietà separate: • **Riparabilità**: un software è riparabile se i suoi difetti possono essere corretti con una quantità di lavoro ragionevole. • **Evolvibilità**: indica la capacità del software di poter evolvere aggiugendo funzionalità. È importante considerare questo aspetto fin dall'inizio: studi rilevano come l'evolvibilità decresce con il passare delle release (vd. L27-28).

Leggi rilevanti

Prima legge di R.Glass (G1).

La mancanza di requisiti è la prima causa del fallimento di un progetto.

Legge di Nielsen-Norman (NN23).

L'usabilità è misurabile.

Legge di McIlroy (MI15).

Riutilizzare il software permette di incrementare la produttività e la qualità.

Leggi di M. Lehman (L27-28).

Un sistema che viene utilizzato cambierà.

Un sistema che evolve incrementa la sua complessita a meno che non si lavori appositamente per ridurla.

Qualità del processo

Un progetto è di qualità se segue un buon processo.

Sappiamo che il prodotto è influenzato dal processo che viene utilizzato per svilupparlo, di conseguenza possiamo parlare anche di **qualità del processo**.

Anche un processo deve funzionare, essere essere bello e farmi diventare ricco, ma dobbiamo interpretare queste parole in maniera differente.

Quali caretteristiche ha un processo di qualità?

Un processo deve…	Qualità	Descrizione
Funzionare	**Robustezza**	Un processo deve poter resistere agli imprevisti, come la mancanza improvvisa di personale o il cambiamento delle specifiche. Esistono certificazioni (*CMM: Capability Maturity Model*) che valutano la robustezza di alcuni processi aziendali e che vengono per esempio considerate nei bandi pubblici.
Essere bello	**Produttività**	La produttività di un team è molto meno della somma della produttività individuale dei suoi componenti. È una metrica difficile da misurare: conteggi come il numero di linee codice scritte o la quantità di *tempo-uomo* richiesta per un lavoro si rivelano spesso un po' fallaci (per esempio, la gravidanza umana non è un'attività parallelizzabile, ma si potrebbe dire che servono 9 mesi-donna per creare un bambino).
Farmi diventare ricco	**Tempismo**	Un processo deve consegnare il prodotto nei tempi stabiliti, in modo da rispettare i tempi del mercato. È spesso conveniente la tecnica dello **sviluppo incrementale**, ovvero la consegna frequente di parti sempre più grandi del prodotto (es. compilatore ADA): essa permette infatti di conquistare il cliente ancor prima di avere il prodotto finito.

Il processo di produzione del software

Il processo che seguiamo per costruire, consegnare, installare ed evolvere il prodotto software, dall'idea fino alla consegna e al ritiro finale del sistema, è chiamato **processo di produzione software**.

Innanzitutto occorre riconoscere diverse problematiche.

- I **requisiti** imposti dal cliente possono cambiare spesso.
- Produrre software **non è *solo* scrivere codice** (alla Programmazione I).
- Bisogna risolvere i **problemi di comunicazione** tra tutte le diverse figure in gioco (tra sviluppatori, tra progettista e sviluppatori, ecc).
- Bisogna essere **rigorosi**, anche se può essere difficile. Ci sono lati positivi e negativi: la rigorisità può facilitare la comprensione di ciò che bisogna fare ma implica al contempo molta fatica extra, e viceversa.

Ipotesi di Bauer-Zemanek (BZh3): *Metodi formali riducono in maniera significativa gli errori di progettazione, oppure permettono di eliminarli e risolverli prima.*

Trovare gli errori prima della fase di sviluppo permette di facilitarne la risoluzione e di risparmiare sia tempo che soldi: tanto prima si individua un errore, tanto più facile sarà risolverlo.

- Ci sono **tanti aspetti** da considerare, che andranno affrontati uno alla volta. Per parlare di aspetti diversi ho bisogno di metodi comunicazione diversi, che interessano ruoli diversi in tempi diversi (*Aspect Oriented Programming*).

Tenendo a mente tutto queste problematiche è necessario decidere come organizzare l'attività di sviluppo software in modo da mitgarle. Per modellare un ciclo di vita del software, occorre dunque in primo luogo **identificare le varie attività necessarie** e quindi:

- deciderne le precedenze temporali;
- decidere chi le debba fare.

In particolare, ci si pone due domande:

- cosa devo fare adesso?
- fino a quando e come?

L'ingegneria del software prova a rispondere a queste domande per individuare quali siano le fasi necessarie per sviluppare un software e quale sia la loro migliore disposizione temporale. È dunque bene specificare da subito che lo sviluppo di un programma non è solo coding: tale presupposto genera conseguenze disastrose.

Inizialmente, infatti, nell'ambito dello sviluppo software è stato adottato il modello ***code-and-fix***, che consisteva nei seguenti passi:

1. scrivi il codice;
2. sistemalo per eliminare errori, migliorare funzionalità o aggiungere nuove funzionalità.

Ben presto però questo modello si è dimostrato pesantemente inefficace in gruppi di lavoro complessi, specialmente quando il cliente non era più lo sviluppatore stesso ma utenti con poca dimestichezza con i computer, generando codice estremamente poco leggibile e manutenibile.

Per organizzare meglio l'attività di sviluppo e non ricadere negli errori del passato gli ingegneri del software hanno dunque individuato diverse **fasi** del ciclo di vita di un software che, combinate, permettessero di produrre del software di qualità. Diamo dunque un'occhiata a quelle principali.

Le fasi del ciclo di vita del software

Studio di fattibilità

Lo studio di fattibilità è l'attività svolta prima che il processo di sviluppo inizi, per decidere se dovrebbe iniziare *in toto*. L'**obiettivo** è quello di produrre un **documento in linguaggio naturale** presentante diversi scenari di sviluppo con soluzioni alternative, con una discussione sui trade-off in termini di benefici e costi attesi.

Più specificatamente, il documento include:

- uno studio di diversi scenari di realizzazione, scegliendo:
  - le architetture e l'hardware necessario;
  - se sviluppare in proprio oppure subappaltare ad altri.
- stima dei costi, tempi di sviluppo, risorse necessarie e benfici delle varie soluzioni.

È spesso difficile fare un'analisi approfondita, a causa del poco tempo disponibile o di costi troppo elevati: spesso viene commissionata esternamente.

Analisi e specifica dei requisiti

L'analisi e specifica dei requisiti è l'attività più critica e fondamentale del processo di produzione del software. L'obiettivo è la stesura di un ***documento di specifica*** .

In questa fase i progettisti devono:

- comprendere il **dominio applicativo** del prodotto, dialogando con il cliente e la controparte tecnica;
- identificare gli **stakeholders**, ovvero tutte le figure interessate al progetto, e studiarne le richieste. Spesso non si tratta di figure omogenee (può essere il *top manager* fino al segretario) e le loro necessità sono molto diverse;
- capire quali sono le **funzionalità richieste**: la domanda più importante che deve porsi il programmatore è il *cosa* non il *come*; al cliente non devono infatti interessare gli aspetti tecnici e le scelte architetturali interne. Le **specifiche** vanno quindi viste dal punto di vista del cliente.
- stabilire un **dizionario comune** tra cliente e sviluppatore che può anche far parte della specifica per agevolare la comunicazione;
- definire **altre qualità** eventualmente richieste dal cliente: per esempio, *"la centrale non deve esplodere"* non è un dettaglio implementativo, ma un requisito. Queste ulteriori qualità, che non sempre sono solo esterne, si dicono **requisiti non funzionali**.

Lo scopo del *documento di specifica* è duplice: da una parte, deve essere analizzato e approvato da **tutti gli stakeholders** in modo da verificare il soddisfacimento delle aspettative del cliente, e dall'altra è usato dai programmatori per sviluppare una soluzione che le soddisfi, fungendo da punto di partenza per il design. È un documento contrattuale e deve essere scritto in modo formale per evitare contestazioni contrattuali e ambiguità.

Deve essere presente anche un **piano di test**, ovvero una collezione di collaudi che certificano la correttezza del lavoro: se questi test hanno esito positivo il lavoro viene pagato, altrimenti il progetto non viene accettato. A differenza dei lavori di altri tipi di ingegneria, per esempio l'ingegneria civile, dove il collaudo è diretto, nell'ingegneria del software è molto difficile collaudare tutti i casi e gli stati possibili.

Un altro output di questa fase può essere anche il **manuale utente**, ovvero la *"vista esterna"* (ciò che il cliente vuole vedere, evitando i dettagli implementativi) del sistema da progettare.

> **Legge di David**: Il valore dei modelli che rappresentano il software da diversi punti di vista dipendono dal punto di vista preso (assunto), ma non c'è nessuna vista che è la migliore per ogni scopo.

Progettazione (design)

Il *design* è l'attività attraverso la quale gli sviluppatori software strutturano l'applicazione a diversi livelli di dettaglio. Lo scopo di questa fase è quello di scrivere un **documento di specifica di progetto** contenente la descrizione dell'architettura software (i diversi linguaggi e viste).

Durante questa fase occorre quindi:

- scegliere un'**architettura software di riferimento**;
- **scomporre** in moduli o oggetti gli incarichi e i ruoli: si tratta del cosiddetto *object oriented design*, non necessariamente accompagnato da object oriented programming;
- **identificare i patterns**, ovvero problemi comuni a cui è già stata trovata una soluzione generale giudicata come *"bella"* dalla comunità degli sviluppatori (ne vedremo un paio più avanti nel corso). I pattern favoriscono alcune qualità, come il design.

Programmazione e test di unità

In questa fase le *"scatole nere"* - i moduli o oggetti definiti al punto precedente - vengono realizzate e per ognuna di esse vengono definiti dei **test unitari** che ne mostrano la correttezza. Vi è infatti spesso la brutta abitudine di non fare il testing durante lo sviluppo di

ciascun componente, ma solamente alla fine di tutto: questa usanza è molto pericolosa perché un problema scoperto solo alla fine è molto più oneroso da risolvere.

I singoli moduli vengono testati indipendentemente, anche se alcune funzioni da cui dipendono non sono ancora sono state implementate: per risolvere tale dipendenza si utilizzano allora moduli fittizzi (***stub***) che emulino le funzionalità di quelli mancanti. Altri moduli, detti ***driver***, forniscono invece una situazione su cui applicare il modulo che sto testando. Nei linguaggi più utilizzati esistono framework che facilitano le suddette operazioni al programmatore.

L'obiettivo di questa fase è avere un **insieme di moduli** separati **sviluppati indipendentemente** con un'interfaccia concordata e **singolarmente verificati**.

Integrazione e test di sistema

In questa fase i moduli singolarmente implementati e testati vengono **integrati** insieme a formare il software finito. In alcuni modelli di sviluppo (come nello sviluppo incrementale) questa fase viene accorpata alla precedente.

Nei test, i moduli *stub* e *driver* vengono sostituiti con i componenti reali formando un sistema sempre più grande fino ad ottenere il risultato richiesto. È poi fondamentale testare che l'intero programma funzioni una volta assemblato (non basta che le singole parti funzionino!): test di questo tipo vengono detti **test di integrazione**.

L'integrazione può essere adottata seguendo un approccio *top down* o *bottom up*. La fase finale è l'***alpha testing***, ovvero il testing del sistema in condizioni realistiche.

Consegna, installazione e manutenzione

Dopo aver completato lo sviluppo, il software deve essere **consegnato** ai clienti. Prima di consegnarlo a tutti, si seleziona un gruppo di utenti per raccogliere ulteriore feedback; questa fase è chiamata ***beta testing***.

L'**installazione** (deployment) definisce il *run-time* fisico dell'architettura del sistema. Per esempio, un servizio di rete potrebbe necessitare di apparecchiatura server da installare e particolari configurazioni.

Infine, la **manutenzione** può essere definita come l'insieme delle attività finalizzate a modificare il sistema dopo essere stato consegnato al cliente. La manutenzione può essere di tipo:

- **correttivo**: sistemare errori nel sistema;

- **adattivo**: adattare il software ai nuovi requisiti (vd. *evolvibilità*);
- **perfettivo**: migliorare certi aspetti interni al programma senza modificare gli aspetti esterni. Serve per migliorare la futura manutenzione riducendo il cosiddetto *debito tecnico*.

Come già detto, è necessario sviluppare avendo in mente la futura manutenzione di ciò che si sta scrivendo: infatti, il **costo** della manutenzione concorre al costo del software in una misura spesso superiore al 60%.

L'*output* di questa fase è un **prodotto migliore**.

Altre attività

Alle attività sopracitate se ne aggiungono altre:

- **Documentazione**: può essere vista come attività trasversale. Per esempio, un documento di specificazione contenente diagrammi UML e una descrizione narrativa che spiega le motivazione dietro certe decisioni può essere il risultato principale della fase di progettazione. È un'attività spesso da procastinare, perché le specifiche possono cambiare spesso. In alcuni gruppi esistono delle figure che si occupano di questa attività, anche se può essere pericoloso: non tutti possono capire ciò che un programmatore ha creato.
- **Verifica e controllo qualità** (Quality Assurance): nella maggior parte dei casi, la verifica è svolta attraverso review e ispezioni. L'obiettivo è anticipare il prima possibile la scoperta e la sistemazione degli errori in modo da evitare di consegnare sistemi difettosi. Andrebbe fatta costantemente e non tutta alla fine.
- **Gestione del processo**: gestione incentivi (premi di produzione), responsabilità, formazione del personale, perfezionamento del processo con l'esperienza guadagnata, eccetera.
- **Gestione delle configurazioni**: gestione delle relazioni inter-progettuali, ovvero delle risorse di sviluppo non appartenenti ad un singolo progetto. Un esempio potrebbe essere una libreria condivisa tra più progetti, i quali vorrebbero modificare la libreria stessa.

Tutte queste diverse attività saranno specificate successivamente entrando nel dettaglio.

Modelli di ciclo di vita del software

In questa lezione vedremo i principali modelli di ciclo di vita del software, ovvero famiglie di processi di sviluppo che si distinguono per il modo in cui le fasi di produzione viste nella scorsa lezione vengono organizzate in un processo unitario. Ognuno di tali modelli avrà i propri pro e i propri contro, ed è bene da subito capire che non esiste il modello giusto per ogni situazione.

- **Modelli sequenziali**

    - modello a cascata
    - modello a V
- **Modelli iterativi**
    - modello a cascata con singola retroazione
    - modello a fontana
- **Modelli incrementali**
    - modello prototipale
    - pinball life-cycle
    - metamodello a spirale
    - modelli trasformazionali
    - modello COTS
- **Metodologie Agile**
    - manifesto
    - lean
    - kanban
    - scrum
    - crystal
    - extreme programming

Modelli sequenziali

Il modo più semplice e immediato di organizzare le fasi del ciclo di vita di un software è sicuramente quello **sequenziale**: i vari passaggi vengono posti in un ordine prestabilito e vengono attraversati uno alla volta uno dopo l'altro. Da questa idea nascono i cosiddetti *modelli sequenziali*, di cui il più famoso è certamente il *modello a cascata*.

Modello a cascata

Caratteristiche e punti di forza


Modello a cascata

Nato negli anni '50 ma diventato famoso solo negli anni '70 grazie allo sviluppo di un grosso software per la difesa area chiamato SAGE (*Semi-Automated Ground Environment*), il modello a cascata organizza le fasi in una serie di step sequenziali: fatto uno si passa al successivo fino ad arrivare alla fine, come in una sorta di *catena di montaggio*. Viene infatti forzata una **progressione lineare** da una fase alla successiva; non è previsto in questo modello tornare indietro a uno step precedente.

Sebbene varino molto da processo a processo, la maggior parte dei processi che segue il modello a cascata include almeno le seguenti fasi organizzate in quest'ordine:

1. Requisiti
2. Progetto
3. Codifica
4. Testing
5. Prodotto

Ognuno di tali step produce un output, detto **semilavorato**, che è dato come input allo step successivo. In virtù dell'affidamento su tali semilavorati intermedi il modello a cascata si dice **document-based**: tra una fase e l'altra si crea infatti un documento che è il mezzo di trasmissione dell'informazione. Questo aspetto permette una **buona separazione dei compiti** tra i vari dipendenti che lavorano al progetto: ognuno è infatti specializzato in una singola fase e una volta prodotto il documento utile ad avviare la fase successiva il suo coinvolgimento nel progetto non è più necessario ed esso può essere assegnato ad altri lavori.

La linearità del modello rende inoltre possibile **pianificare i tempi** accuratamente e monitorare semplicemente lo stato di avanzamento in ogni fase: è infatti sufficiente stimare la durata di ogni fase per ottenere una stima del tempo di completamento dell'intero progetto. Si tratta però di una stima a senso unico: una volta finita una fase non è possibile ridurre il tempo speso, e in caso di inconvenienti l'unica opzione è cercare di assorbire il ritardo.

Criticità

Sebbene il modello a cascata abbia il grande pregio di aver posto l'attenzione sulla comunicazione tra gli elementi del progetto in un momento storico in cui il modello di sviluppo più diffuso era di tipo *code-and-fix*, esso soffre di numerose criticità.

In primo luogo il modello **non prevede una fase di manutenzione** del software prodotto: esso assume di non dover apportare modifiche al progetto dopo averlo consegnato, e impedisce dunque di *"tornare indietro"* in alcun modo. Ovviamente questa assunzione è un'illusione smentita nella quasi totalità nei casi: qualunque software è destinato ad evolvere, e più un software viene usato più cambia. Una volta finito lo sviluppo ciò che si può fare è rilasciare al più piccole patch, che tuttavia non fanno altro che disallineare la documentazione prodotta precedentemente con il software reale.

Il modello soffre inoltre di una generale **rigidità**, che mal si sposa con la flessibilità naturalmente richiesta dall'ambiente di sviluppo software. In particolare, l'impossibilità di tornare indietro implica un **congelamento dei sottoprodotti**: una volta prodotto un semilavorato esso è fisso e non può essere modificato; questo è particolarmente critico per le stime e specifiche fatte durante le prime fasi, che sono fisiologicamente le più incerte.

Infine, il modello a cascata adotta un approccio volto alla **monoliticità**: tutta la pianificazione è orientata ad un singolo rilascio, e l'eventuale manutenzione può essere fatta solo sul codice. Inutile dire che si tratta di una visione fallace, in quanto come già detto più volte il software è destinato ad essere modificato e ad evolvere.

Who's Afraid of The Big Bad Waterfall?

LIBRO: **The Leprechauns of Software Engineeering** di Laurent Bossavit.

In realtà, il modello a cascata non è mai stato veramente elogiato, ma è sempre stato utilizzato come paragone negativo per proporre altri modelli o variazioni. Nel corso del tempo la sua presentazione è stata erroneamente attribuita al paper *"Managing the development of large software systems: concepts and techniques"* di W.W. Royce, di cui veniva citata solo la prima figura: Royce stava a dire il vero presentando quel modello per descrivere la sua esperienza nello sviluppo software, per poi proporre altri concetti più moderni (come lo sviluppo incrementale) che non sono però mai stati colti dalla comunità scientifica.

Anche noi utilizziamo il modello a cascata solo come paragone negativo, e in generale nell'ambiente di sviluppo software esso non è più applicato alla lettera. Alcuni suoi aspetti si sono però mantenuti come linee guida generali (es. l'ordine delle fasi); è infatti bene chiarire subito che esistono due tipi di modelli:

- **prescrittivi**: forniscono delle indicazioni precise da seguire per svolgere un processo;
- **descrittivi**: colgono certi aspetti e caratteristiche di particolari processi esistenti, ma non obbligano a seguirli in modo rigoroso.

Tutti i modelli visti per ora ricadono perlopiù nell'ambito descrittivo, mentre i modelli AGILE che vedremo più avanti tendono ad essere più di tipo prescrittivo.

Riassunto pro e contro

Pro	Contro
• Document-based • Buona suddivisione dei compiti • Semplice pianificazione dei tempi	• Rigidità • Congelamento dei sottoprodotti • Monoliticità

Modello a V (denti di pesce cane)

Dal modello a cascata nascono poi numerose varianti che cercano di risolverne i vari problemi: tra queste spicca per rilevanza il **modello a V**, che introduce fondamentalmente una **più estesa fase di testing**.

Nonostante sia ancora un modello sequenziale come il modello a cascata, nel modello a V vengono infatti evidenziati nuovi legami tra le fasi di sviluppo, che corrispondono alle attività di **verifica** e **convalida**: alla fine di ogni fase si *verifica* che il semilavorato ottenuto rispetti la specifica espressa dalla fase precedente, e inoltre si richiede la *convalida* del fatto che esso sia in linea con i veri vincoli e necessità del cliente. Come si vede, questo modello pone l'accento sul rapporto con il cliente, che viene continuamente coinvolto con la richiesta di feedback su ciascun sottoprodotto generato.

Volendo formalizzare, le due nuove attività introdotte sono dunque:

- **verifica** (freccie grigie): controlla la correttezza rispetto alla descrizione formale delle specifiche;
- **validazione** (freccie bianche): controlla la compatabilità del sistema con le esigenze del cliente tramite feedback continuo.

Modelli iterativi

Osservando il modello a cascata e le sue varianti ci si è ben presto resi conto che la stringente sequenzialità delle fasi costituiva un grosso limite non conciliabile con la flessibilità richiesta dallo sviluppo software e con la naturale mutevolezza dei requisiti imposti dal cliente. Si inizia dunque a pensare di permettere agli sviluppatori di *ripetere* alcune fasi più di una volta, ciclando su di esse fino a ottenere un prodotto soddisfacente: nascono così i primi **modelli interativi**.

Modello a cascata con singola retroazione

Uno dei primi modelli iterativi è in realtà una variante del modello a cascata, in cui si permette di fare un'unico salto indietro; a parire da una fase si può cioè **ritornare alla fase precedente**: così, per esempio, si può *iterare* tra *Codifica* e *Testing* fino a consegnare il prodotto.

Anche in questo modello non si può però tornare indietro dalla consegna per eseguire attività di manutenzione; inoltre, l'introduzione di un'iterazione rende molto **più difficile pianificare** il lavoro e monitorarne l'avanzamento: si tratta questa di una caratteristica condivisa da molti modelli iterativi.

Modello a fontana


Modello a fontana

Nel 1993 nasce, in contrapposizione al modello a cascata, il cosiddetto **modello a fontana**, che amplia il concetto di iterazione permettendo in qualunque momento di **tornare alla fase iniziale**: se ci si accorge della presenza di errori si torna indietro all'inizio (*software pool*) e si ricontrollano tutte le fasi precedenti. Ovviamente questo non implica buttare tutto il lavoro già fatto, quanto piuttosto risolvere l'errore con un approccio che parta innanzitutto dalla modifica dei requisiti (se possibile), delle specifiche e solo dopo del codice, evitando di rattoppare solo quest'ultimo alla bell'e meglio come nel modello *code-and-fix*.

Il modello a fontana è inoltre il primo in cui sono previste delle azioni dopo la consegna; dopo l'ultima fase (*programma in uso*), infatti, si aprono ancora due strade: **manutenzione ed evoluzione**. La consegna del prodotto non è quindi più l'atto finale, ma solo un altro step del processo: ecco quindi che si aprono le porte ad una **visione incrementale** dello sviluppo software, che approfondiremo nel prossimo paragrafo.

Anche qui si perdono purtroppo le garanzie sui tempi di sviluppo: una volta ritornato all'inizio per sistemare un errore non è infatti affatto detto che riuscirò a ritornare alla fase da cui sono partito, ma potrei imbattermi in altri errori durante le fasi precedenti costringendomi a iterare su di esse più di una volta.

Modelli incrementali

Un modello incrementale è un particolare modello iterativo in cui nelle iterazioni è inclusa anche la consegna: questo permette di sviluppare il software a poco a poco, rilasciandone di volta in volta parti e componenti che costruiscano *incrementalmente* il programma finito.

Si noti la differenza tra incrementale e iterativo; si può parlare infatti di:

- **implementazione iterativa**: dopo aver raccolto le specifiche e aver progettato il sistema, *iterativamente* sviluppo i componenti, li integro nel prodotto finale, quindi consegno.
- **sviluppo incrementale**: l'iteratività interessa tutte le fasi, comprese quelle di specifiche e realizzazione.

Lo sviluppo incrementale riconosce la criticità della variabilità delle richieste e la integra nel processo. La manutenzione non è quindi più una particolarità ma è vista come normale e perfettamente integrata nel modello: in tal senso, la richiesta di una nuova feature o la correzione di un errore generano gli stessi step di sviluppo.

Modello prototipale

Un particolare modello incrementale è quello protitipale: in questo modello viene introdotto il concetto di **protitipi usa e getta** (*throw away*), interi programmi che vengono costruiti e poi vengono buttati via.

Lo scopo del prototipo **non è consegnare** un prodotto finito, ma **ricevere feedback** dal cliente per essere sicuri di aver compreso a pieno i suoi requisiti, oppure testare internamente un'idea o uno strumento. Per questo motivo tali prototipi vengono costruiti fregandosene di correttezza, pulizia del codice, leggibilità eccetera. I protitipi possono dunque essere:

- **pubblici**: per capire meglio i requisiti del cliente (vd. L3);
- **privati**: per esplorare nuovi strumenti, linguaggi, scelte per problemi difficili; inoltre, molto spesso una volta programmata una soluzione si capisce anche meglio il problema (*"do it twice"*).

La tentazione coi prototipi pubblici può essere quella di consegnarli come prodotto finito, ma c'è il **rischio** enorme di dover mantenere poi in futuro software non mantenibile, illeggibile e con altissimo debito tecnico.

Legge di Bohem (L3)

> La propotipizzazione riduce gli errori di analisi dei requisiti e di design, specialmente per le interfacce utente.

I problemi dei modelli incrementali

Come già detto nessun modello è perfetto, e anche i modelli incrementali soffrono di alcuni problemi.

Viene innanzitutto **complicato il lavoro di planning**: bisogna pianificare tutte le iterazioni e lo stato di avanzamento è meno visibile; inoltre, la ripetizione di alcune fasi richiede di avere sempre sul posto gli esperti in grado di eseguirle. Ad ogni iterazione, poi, dobbiamo rimettere

mano a ciò che è stato fatto, in un processo che potrebbe non convergere mai a una versione finale.

Ma cosa è un'iterazione, e quanto dura? Tagliare verticalmente sulle funzionalità non è infatti facile, soprattutto considerando che quando si consegna il prodotto esso dev'essere funzionante con tutti i layer necessari ed essere al contempo pensato per poter crescere con successivi attaccamenti. Ci sono dunque diversi rischi:

- voler aggiungere troppe funzionalità nella prima iterazione;
- overhead dovuto a troppe iterazioni;
- avere un eccessivo overlapping tra le iterazioni: non si ha tempo di recepire il feedback dell'utente (es. Microsoft Office 2020 e 2019 vengono sviluppati contemporaneamente).

Pinball Life-Cycle


Pinball Life-Cycle

Il *"modello meme"* del Pinball Life-Cycle, creato da Ambler come critica ai modelli incrementali, estremizza queste problematiche: l'ordine in cui faccio le attività è casuale, incoltrollabile. Qualunque passo è possibile dopo qualunque altro, e non si possono imporre vincoli temporal: il processo è **non misurabile**.

Si tratta ovviamente di una visione eccessivamente pessimistica, ma spesso nelle aziende non specializzate l'iter di sviluppo assomiglia effettivamente a questo.

Modelli trasformazionali


Modelli trasformazionali

Diametralmente opposti all'incubo del Pinball Life-Cycle troviamo i **modelli trasformazionali**: tali modelli pretendono infatti di controllare tutti i passi e i procedimenti in **modo formale**.

Partendo dai requisiti scritti in linguaggio informale, tali modelli procedono tramite una sequenza di **passi di trasformazione** dimostrabili tutti formalmente fino ad arrivare alla versione finale. Essi si basano infatti sull'idea che se le specifiche sono corrette e i passi di trasformazione sono dimostrati allora ottengo un programma corretto, ovvero di sicuro aderente alle specifiche di cui sopra. Inoltre, la presenza di una storia delle trasformazioni applicate permette un rudimentale versioning, con la possibilità di tornare indietro a uno stato precedente del progetto semplicemente annullando le ultime trasformazioni fatte.


Trasformazioni formali

Ad ogni passo si ottiene quindi un **protitipo** che differisce dal prodotto finale per efficienza e completezza, ma che è possibile trasformare in un altro più efficiente e corretto. Non si tratta tuttavia di un processo totalmente automatico, anzi: ad ogni passo di "ottimizzazione", ovvero applicazione di una trasformazione, è richiesto l'intervento di un decisore umano che scelga che cosa ottimizzare.

Viene quindi introdotto il concetto di **prova formale di correttezza** delle trasformazioni applicate; per via di questo approccio molto matematico questo tipo di modelli è nella realtà applicato quasi solo negli ambienti di ricerca e produzione hardware.

*Meta*modello a spirale


Modello a spirale

Introduciamo ora un metamodello, ovvero un modello che ci permette di rappresentare e discutere di altri modelli (una sorta di framework).

Nel metamodello a spirale l'attenzione è posto sui **rischi**, ovvero sulla possibilità che qualcosa vada male (decisamente probabile nell'ambiente di sviluppo software). Per questo motivo il modello è di tipo incrementale e pone l'accento sul fatto che non abbia senso fare lo studio di fattibilità una sola volta, ma ad ogni iterazione serva una decisione. Le fasi generali sono dunque:

- Determinazione di obiettivi, alternative e vincoli
- Valutazione alternative, identificazione rischi (decido se ha senso andare avanti)
- Sviluppo e verifica
- Pianificazione della prossima iterazione

Nella figura il raggio della spirale indica i **costi**, che ad ogni iterazione aumentano fisiologicamente.

Variante *"win-win"*

Esiste una variante al modello a spirale che fa notare come i rischi ad ogni fase non sono solo rischi tecnologici ma anche **contrattuali** con il cliente. Ad ogni iterazione bisogna dunque trovare con esso un punto di equilibrio *win-win* in entrambi le parti "vincono" (o hanno l'illusione di aver vinto), così da far convergere tutti su un obiettivo comune.

Modello COTS (Component Off The Shelf)


Modello COTS

Vediamo infine un modello che si concentra molto sulla **riusabilità**: si parte dalla disponibilità interna o sul mercato di moduli preesistenti sui quali basare il sistema, e che è dunque necessario solo integrare tra di loro.

Non si creda che si tratti di un approccio facile: questo modello di design necessita di far dialogare componenti che non necessariamente comunicano già nel modo voluto.

Si tratta tuttavia di un modello di sviluppo diverso perché richiede attività diverse. In particolare:

- *Analisi dei requisiti*
- ***Analisi dei componenti***: prima di progettare considero la disponibilità di componenti che implementano una parte o tutte le funzionalità richieste;
- ***Modifica dei requisiti***: stabilisco se il cliente è disposto ad accettare un cambiamento nei requisiti necessario per utilizzare un componente particolare;
- ***Progetto del sistema col riuso di componenti***: occorre progettare il sistema per far interagire componenti che non necessariamente sono stati originariamente progettati per interagire;
- *Sviluppo e integrazione*;
- *Verifica del sistema*.

Metodologie Agili

Finora i modelli visti erano di tipo prettamente descrittivo; vediamo ora dei modelli più prescrittivi, che dicano cioè che cosa fare effettivamente durante lo sviluppo.

Le metodologie agili *"nascono dal basso"*, ovvero solitamente da chi sviluppa, per colmare un disagio prevalente nell'usare i metodi tradizionali. Per tale motivo, di tali metodologie esiste un...

Manifesto

Nelle parole di Fowler e i suoi collaboratori, per migliorare il modo in cui sviluppiamo il software dobbiamo dare più importanza ad alcuni valori rispetto agli altri:

- Gli **individui** e la **collaborazione tra individui** è più importante di processi e strumenti.
- Il **software che funziona** è più importante della documentazione ben fatta.

- La **collaborazione con il cliente** è più importante del contratto.
- **Rispondere al cambiamento** è più importante che seguire un piano.

LIBRO: **Agile!** di Bertrand Meyer

Come si vede, si tratta di un drastico cambio di rotta rispetto allo sviluppo tradizionale, che si evolve anche in un business model diverso: piuttosto che farsi pagare a programma finito, adesso gli sviluppatori vogliono farsi pagare a tempo di sviluppo, dando però la garanzia al cliente di lavorare durante tale periodo esclusivamente per lui e al massimo delle proprie capacità. Al rapporto confluttuale con il cliente, in cui ciascuno cerca di fregare l'altro, si sostituisce dunque una collaborazione più estesa in cui, come vedremo, anche il cliente diventa parte del team di sviluppo.

Vediamo dunque adesso alcune delle più famose metodologie agili, mettendone in evidenza gli aspetti peculiari.

Lean Software

Nato dal progetto di *Lean Manufactioring* della Toyota, ha l'obiettivo di **ridurre gli sprechi**, ovvero quei prodotti e sottoprodotti che non vengono consegnati al cliente (es. testing, prototipi...) e dunque non generano valore: essi possono essere ignorati.

Un'altra idea interessante è quella di posticipare il più possibile le scelte vincolanti per aiutare a risparmiare risorse: più possibilità mi lascio aperte, più mi sarà facile adattarmi (a patto però che l'adattamento sia veloce).

Kanban

 Kanban

L'obiettivo è qui invece di **minimizzare il lavoro in corso** (work in progress), ovvero concentrarsi in ogni momento su una sola cosa in modo da evitare i continui *context switch* che costituiscono una perdita di tempo. Le attività possono per esempio essere organizzate in una tabella con 5 colonne:

- **backlog**: richieste dal cliente
- **da fare**: attività da fare in questa iterazione
- **in esecuzione**

- **in testing**
- **fatto**

La tabella dà a colpo d'occhio informazioni sullo stato del progetto per tutti. Ogni **card** (storia) è assegnata a uno sviluppatore (o coppia nel *pair programming*), in modo che nella colonna in esecuzione vi sia una sola card per sviluppatore (o coppia); qualora il lavoro di un altro blocchi il mio lavoro in qualche modo è poi mia responsabilità aiutarlo per rimuovere il blocco.

Scrum

L'obiettivo è **fissare i requisiti** durante le iterazioni (**brevi**, da 2 a 4 settimane), in modo da permettere agli sviluppatori di lavorare in pace senza doversi adattare continuamente a nuove richieste. Solo al termine di ogni iterazione, infatti, si permette al cliente di rimettere in discussione i requisiti.

Crystal

Sebbene non sia molto apprezzata o usata, questa tecnica introduce l'interessante concetto di **comunicazione osmotica**. Nel modello a cascata la comunicazione è fatta tramite documenti rigidi, ed è settorializzata; in Crystal la conoscenza viene condivisa nel team tramite *"osmosi"*, in modo che tutti sappiano un po' di tutto.

Questo rende il processo più robusto, perché l'assenza di una persona esperta in un campo non è più in grado di bloccare completamente i lavori. Il pair programming è in quest'ottica: tra i due componenti la conoscenza è condivisa; Crystal estende questo concetto all'intero team.

Si capisce però facilmente che questa tecnica funziona solo con team piccoli (max 8-10 persone), sebbene altre metodologie agili (*SAFE*) tentino di scalarla anche a team più massicci.

eXtreme Programming (XP)

Si tratta di una tecnica a cui dedicheremo una trattazione più approfondita nella prossima lezione. Per il momento accontentiamoci di enunciarne i due motti:

- **incrementa quindi semplifica**;
- **sviluppo guidato dal test** (*test-first*: prima testa poi sviluppa).

eXtreme Programming

In questa lezione verranno trattati i seguenti argomenti.

- **Test Driven Development**: *test-first + baby steps*
- **I fondamenti dell'XP**: variabili, principi, figure e responsabilità
- **Tecniche**: tutte e 13 le tecniche dell'XP
- **Relazione con il modello a cascata**
- **Documentazione**: carte CRC
- **Criticità**: quando non utilizzare XP, critiche di Meyer e discussione sui *mesi uomo*.

Test Driven Development

Il *test driven development* (TDD) è una **tecnica di progettazione** del software che mira a far emergere "dal basso" il design più semplice in grado di risolvere un dato problema. Non si tratta ne un'attività di verifica ne di scrittura del codice, quanto piuttosto un approccio alla scrittura di questi ultimi.

Il TDD si fonda su due concetti fondamentali, esplicitati nella seguente citazione:

> TDD = **test-first** + **baby steps**

Il significato di questa espressione è che per scrivere del codice che esalti la semplicità della soluzione è necessario **scrivere prima il test rispetto al codice** (*test-first*) e procedere a **piccoli passi** (*baby steps*), realizzando cioè piccole porzioni di codice, testandole e solo allora andando avanti. Questa tecnica mira infatti a stabilire un ciclo di *feedback istantaneo*: facendo piccoli passi e testando ogni volta ciò che si appena scritto è meno probabile buttare molto tempo su una soluzione che non funziona, e anche in caso di errore è più facile individuare cosa lo genera e come risolverlo.

Per applicare questo approccio *test-driven* allo sviluppo effettivo di software, il TDD ha sviluppato il seguente "mantra": **rosso**, **verde**, **refactoring**. Quando si scrive codice bisogna infatti seguire le seguenti tre fasi:

- Ogni volta che si deve aggiungere una feature **si scrive prima il test** che la provi; non essendo ancora stata sviluppata, tale test dovrà fallire (rosso).
- Si cerca poi di **soddisfare il test il più velocemente possibile**, facendolo diventare verde. Si ottiene così del codice corretto ma probabilmente molto brutto, quasi come fosse una bozza: tale codice serve però come feedback del fatto che l'algoritmo scelto funziona.

- Si compie infine un'azione di **refactoring** (*fattorizzazione*), ovvero si riorganizza e si riscrive il codice in modo da renderlo migliore assicurandosi però che il test continui ad essere soddisfatto (in questa fase dobbiamo rimanere in uno stato di verde).

Questa ciclo in tre fasi va ripetuto con una cadenza frequente, ogni 2-10 minuti: ciò obbliga a concentrarsi su compiti semplici evitando così di perdersi in costruzioni software complicate che magari non funzionano neanche. Si preferisce invece prima fare qualche piccolo progresso (*increment*) e poi semplificare per migliorare il codice (*simplify*).

È importante inoltre capire perché quel passaggio intermedio, la "bozza" menzionata al secondo punto dell'elenco precedente, è tanto importante: concentrarsi in primo luogo sulla creazione di una base funzionante permette subito di capire se si è scelta la strategia giusta per risolvere il problema corrente. Scrivere direttamente il codice "in bella" impiegherebbe molto più tempo e potrebbe non produrrebbe neanche un codice funzionante, siccome maggiore è la complessità del codice che si scrive più è probabile commettere errori.

In virtù di quanto appena detto, l'uso del TDD come tecnica di progettazione garantisce inoltre due importanti vantaggi:

- Spesso capita di scrivere codice difficilmente testabile: scrivere il test prima e il codice dopo aiuta invece a progettare prodotti la cui correttezza può essere provata.
- Scrivere prima i test aiuta a definire chiaramente le interfacce del programma e come queste comunicano tra di loro, mentre se non dovessimo farlo potremmo avere delle dipendenze complicate da rimuovere.

Durante il testing ci si pone dal **punto di vista del cliente**: la tecnica TDD ci permette dunque di osservare il codice da molteplici prospettive (sviluppatore e cliente), cosa che contribuisce ovviamente alla creazione di un prodotto migliore.

Fondamenti

Ora possiamo iniziare a parlare di Extreme Programming (XP), una tecnica di sviluppo agile nata tra la fine degli anni '90 e l'inizio degli anni 2000 dalla mente di Kent Beck, che la ideò nell'ambito di un progetto Chrysler.

Variabili

Secondo Beck, durante lo sviluppo di software le principali variabili sono:

- **portata**: la quantità di funzionalità da implementare, una variabile delicata dal valore *mutevole* poiché il numero di funzionalità richieste può cambiare nel corso dello sviluppo;
- **tempo**: il tempo che si può dedicare al progetto;
- **qualità**: la qualità del progetto che si vuole ottenere, principalmente relativa a correttezza e affidabilità;
- **costo**: le risorse finanziare che si possono impegnare per il progetto.

Queste 4 variabili **non sono indipendenti** tra di loro, in quanto cambiare una influenza automaticamente le altre, in positivo o in negativo. Ponendo quindi che la qualità non sia negoziabile (il software deve funzionare) bisognerà lavorare sulle altre, specialmente bilanciando costo e tempo.

Nel panorama classico di sviluppo la portata era definita in modo rigido dal cliente, che richiedeva certe funzionalità non negoziabili e pagava lo sviluppatore a progetto completo. Con l'XP si stravolge invece la prospettiva: **il costo è orario**, il tempo disponibile non è fisso ma pari al tempo richiesto per lo sviluppo e la portata viene ricalcolata durante il progetto, essendo così l'unica variabile a variare effettivamente. Si tratta di un approccio *incrementale* che mira ad avere sempre un prodotto consegnabile se il cliente decide di essere soddisfatto dello sviluppo: non si fa aspettare il cliente per dargli tutto il lavoro in un colpo solo, ma questo viene consegnato una parte alla volta. Oltre ad alleggerire la pressione sullo sviluppatore, questo approccio è utile per due motivi:

- Il cliente è certo che lo sviluppatore si sia dedicando al progetto siccome vede il prodotto crescere a poco a poco.
- Dà la possibilità al cliente di avere comunque qualcosa in mano se ad un certo punto vuole interrompere la collaborazione.
- Permette al cliente di cambiare idea sulla portata e sulle funzionalità richieste in corso d'opera, bandendo la rigidità dei documenti di specifica.

Tutti questi aspetti permettono di creare un rapporto molto meno conflittuale tra cliente e sviluppatore, cosa che crea le basi per una maggiore collaborazione tra le due parti.

Principi

Parliamo ora un po' dei fondamenti della filosofia XP, confrontandoli con quanto veniva prescritto nell'ambiente di sviluppo classico. I principi dell'ingegneria del software classica erano infatti i seguenti:

- **Separazione degli interessi** (*aspects* o *concerns*): separare tempi, responsabilità e moduli, ovvero tutte le varie viste o le varie dimensioni su cui si deve affrontare il problema.

- **Astrazione e modularità**: bisogna usare le giuste astrazioni che ci permettono di dominare i problemi complessi (possono essere i diversi linguaggi di programmazione, linguaggi di descrizione o vari altri costrutti).
- **Anticipazione del cambiamento** (*design for change*): in fase di progettazione il programmatore deve pensare a come potrebbe cambiare il prodotto, accomodando la possibile aggiunta di requisiti che il cliente magari non aveva neanche pensato; bisogna stare attenti però, perché spesso questo concetto complica arbitrariamente la progettazione e lo sviluppo, rischiando di far perdere molto tempo su cose che al cliente potrebbero non servire: può essere un'idea migliore partire da qualcosa di semplice ed incrementare man mano.
- **Generalità**: per rendere più semplice la modifica e l'espansione futura è necessario scrivere interfacce molto generali ai sistemi che costruiamo.
- **Incrementalità**: lo sviluppo avviene incrementalmente, un pezzetto alla volta.
- **Rigore e formalità**: è importante essere rigidi e specifici sia nella comunicazione che nella descrizione dei requisiti.

Sebbene non butti via tutti questi principi ma ne erediti invece alcuni per adattarli alle proprie esigenze (specialmente la *separazione degli interessi*, che viene data per scontata), l'XP pone l'accento su altri aspetti, ovvero:

- **Feedback rapido**: bisogna mantenere un costante flusso di feedback; questo viene dato dai test, dai colleghi ma anche dal cliente, che dev'essere continuamente consultato sullo stato dei lavori. Tra le iniziative che favoriscono un veloce ciclo di feedback c'è lo *standup meeting*, una riunione mattutina fatta in piedi in cui ciascuno descrive in poche parole cosa ha fatto il giorno precedente e cosa intende fare oggi.
- **Presumere la semplicità**: non bisogna complicare senza motivo né il codice, che dev'essere scritto con in mente ciò che serve a breve termine e non in un futuro remoto, né le relazioni tra colleghi, che non devono essere eccessivamente gerarchiche (tutti dovrebbero avere compiti molto simili); in generale si dovrebbe semplificare il più possibile in tutti gli ambiti del progetto.
- **Accettare il cambiamento**: non ci si deve aspettare che il software sia immutabile; al contrario, deve essere dato per scontato il concetto di *flessibilità* e *malleabilità*, ovvero che il cliente vorrà fare cambiamenti sia dopo che durante lo sviluppo del prodotto.
- **Modifica incrementale**: ritornando al concetto di baby steps, ogni iterazione di sviluppo dovrebbe essere breve e le funzionalità introdotte piuttosto piccole; questa regola si applica tuttavia a tutti gli ambiti del progetto, tra cui la gestione del team: ovvero non bisognerebbe mai aggiungere più di una persona alla volta al gruppo di lavoro, in quanto aggiungerne di più potrebbe portare a passare più tempo ad istruirle che a sviluppare.
- **Lavoro di qualità**: bisogna ovviamente ottenere un buon prodotto, ma per fare ciò la prospettiva cambia in favore dello sviluppatore, al quale si deve garantire un ambiente di lavoro salutare e un certo benessere; la fidelizzazione dei programmatori è importante perché più si trovano bene e meglio lavorano.

I due punti più in contrasto sono il presumere la semplicità e l'anticipazione del cambiamento: ci sembra infatti più previdente pianificare per il futuro e anticipare eventuali cambiamenti, ma come vedremo nel prossimo paragrafo talvolta questo può essere controproducente.

Presumere la semplicità vs anticipazione del cambiamento

XP mette davanti la semplicità all'anticipazione del cambiamento: non si scrive in anticipo codice che si pensa servirà in futuro. Questo non significa che non si stia progettando per il futuro, ma solo che questo non è il primo aspetto da guardare: il primo aspetto è la semplicità, ovvero fare le cose nella maniera più chiara possibile.

Non pianificare per il futuro sembra rischioso: secondo uno studio condotto da Bohem nel 1976 viene ipotizzata una curva esponenziale per il corso delle modifiche all'aumento dell'avanzamento del progetto; più il progetto avanza più è costoso modificarlo, motivo per cui sembra necessario accomodare il cambiamento futuro in modo da ridurre tale costo.
Al contrario, XP presuppone una curva di tipo logaritmico che tenda ad un asintoto: passato un certo punto nello sviluppo il costo per le modifiche non subisce più cambiamenti sensibili, per cui non ha senso fasciarsi la testa in anticipo in quanto un codice semplice è relativamente facile da modificare.

 Curve di costo: XP vs tradizionale

Va inoltre considerato che Bohem parlava in realtà di cost-to-fix, non del costo per la modifica in sé; inoltre la sua statistica era poco affidabile poiché era stata costruita a partire da pochi dati. La curva esponenziale da lui descritta è stata poi successivamente ritrattata per accomodare il fatto che se un errore avviene in una fase affligge solo le successive, e non le precedenti.

Figure e responsabilità

Al fine di organizzare il lavoro, XP individua diverse figure che partecipano allo sviluppo:

- Cliente: colui che richiede funzionalità e conosce il dominio applicativo.
- Sviluppatore: colui che sviluppa concretamente scrivendo codice.
- Manager: colui che amministra lo sviluppo con uno sguardo generale.

È interessante l'inclusione del cliente nel contesto dello sviluppo: esso non è più soltanto il committente ma ha un ruolo attivo nel lavoro, potendo cioè contribuire alla riuscita del progetto anche e soprattutto in virtù della già citata conoscenza del dominio applicativo.

Ciascuna di tali figure ha responsabilità e diritti riassunti nella seguente tabella (*manager e cliente sono accorpati perché hanno grossomodo gli stessi compiti*):

Soggetto	Ha responsabilità di decidere…	Ha diritto di…
Manager/Cliente	• Portata del progetto, ovvero le funzionalità da realizzare • Priorità tra funzionalità e loro *business value* • Date dei rilasci, anche nel caso di release incrementali	• Sapere cosa può essere fatto, con quali tempi e quali costi • Vedere progressi nel sistema, provati dai test da lui definiti (*trasparenza*) • Cambiare idea, sostituendo funzionalità o cambiandone le priorità a intervalli di tempo fissi (*fine del ciclo di sviluppo incrementale*)
Sviluppatore	• Stime dei tempi per le singole funzionalità (*no deadline imposte dall'alto*) • Scelte tecnologiche e loro conseguenze, ovvero *come* si realizzano le funzionalità richieste • Pianificazione dettagliata delle iterazioni	• Ricevere dei requisiti chiari (*casi d'uso*) con priorità per le varie funzionalità • Cambiare le stime dei tempi man mano che il progetto procede e il contesto di sviluppo cambia • Identificare funzionalità pericolose o troppo difficili da realizzare • Produrre software di qualità, per il quale deve godere di un certo benessere

Come si vede, per migliorare la fiducia tra sviluppatore e cliente sono necessari due requisiti: un certo grado di *trasparenza* da parte di chi sviluppa, ottenuta dall'uso delle contiene release incrementali per mostrare come sta evolvendo il sistema, e una certa dose di *pazienza* da parte del cliente, che deve accettare di lasciare allo sviluppatore la facoltà di decidere come si realizzano le funzionalità e di cambiare le prospettive temporali di sviluppo qualora fosse necessario.

Tecniche

L'eXtreme Programming fornisce una serie di metodologie pratiche per poter garantire tutto ciò che è stato descritto fino ad ora. Lo schema sottostante le descrive mettendole in relazione tra loro in modo che i vari aspetti negativi delle diverse pratiche siano compensati dagli aspetti positivi di quelle in relazione con loro; in sostanza abbiamo un mix perfetto di attività organizzate in modo da garantire i buoni principi di cui sopra.


Relazione delle tecniche XP tra loro

Elenco

1. **Planning game**
2. **Brevi cicli di rilascio**
3. **Uso di una metafora**
4. **Presumere la semplicità**
5. **Testing**
6. **Refactoring**
7. **Pair programming**
8. **Proprietà collettiva**
9. **Integrazione continua**
10. **Settimana da 40 ore**
11. **Cliente sul posto**
12. **Standard di codifica**
13. ***They're just rules***

Planning game

È l'attività di pianificazione che viene fatta all'inizio di ogni iterazione e serve per "congelare" il sottoinsieme di requisiti sul quale il team lavorerà per le prossime ~2 settimane.

Si parte dalle richieste del cliente espresse tramite *user stories*, una versione semplificata degli *use case* degli UML; esse hanno come soggetto sempre un ruolo specifico nell'azienda del cliente e descrivono una funzionalità. Ogni *user story* è dunque composta da tre parti:

- il **soggetto**, ovvero il ruolo dell'utente nell'azienda (può anche essere esterno);
- l'**azione** che vuole eseguire il soggetto;
- la **motivazione** che spinge il soggetto a portare avanti l'azione descritta.

Esempi di *user stories* potrebbero essere:

- *Da bibliotecario, voglio poter visualizzare dove si trova un particolare libro in modo da poterlo reperire per i clienti.*

- *Da utente della biblioteca, voglio poter visualizzare lo stato di un libro per poterlo prendere in prestito.*

Lo scopo del planning game è dunque quello di determinare quali funzionalità saranno presenti nel prossimo rilascio combinando priorità commerciali e valutazioni tecniche: questo richiede una collaborazione da parte del cliente, che come vedremo sarà presente in loco al momento della decisione.

Procedura

Quest'attività di pianificazione si divide fondamentalmente in tre fasi:

1. All'inizio il cliente compila le **carte**, nient'altro che pezzetti di carta volutamente piccoli per impedire di scriverci troppo. Su ogni carta è presente:
    - un identificativo numerico;
    - una breve frase che descrive uno scenario d'uso;
    - un caso di test che funge da test d'accettazione della funzionalità: si tratta in sostanza di un paio di esempi, di solito uno positivo e uno negativo, che devono essere soddisfatti per ritenere completa la feature;
    - il valore di business che la funzionalità ha per il cliente.
2. Per ogni carta il team di sviluppatori fa dunque una **stima** del tempo necessario a realizzarla: raggiunta una stima comune questa viene scritta sulla carta e servirà per confrontare tale previsione con il tempo effettivamente impiegato, di cui si tiene conto sul suo retro. Per ciascuna carta uno sviluppatore assume infatti il ruolo di *tracker*, impegnandosi cioè a tracciare lo stato di avanzamento della relativa funzionalità durante le due settimane (*es. quante feature fatte, quanti bug segnalati, etc.*).
3. Il manager decide quindi sulla base di queste informazioni **quali carte verranno implementate** durante prossima iterazione: per questa operazione prende in considerazione il valore delle feature, le dipendenze tra l'una e l'altra e una serie di altri fattori. Se, come dovrebbe essere, le varie funzionalità rappresentate nelle carte sono indipendenti, il manager può compiere questa scelta calcolando il rapporto tra il valore e

il tempo stimato e usarlo per ordinare le carte: tuttavia l'operazione richiede una certa dose di ragionamento e non è mai così meccanica.

Le stime

Abbiamo detto che le stime dei tempi vengono fatte dall'intero team in accordo; tuttavia il team è composto da persone diverse che quindi faranno stime diverse in funzione dell'esperienza e delle proprie capacità. È tuttavia importante raggiungere una stima accettata da tutti in quanto il team si impegna a rispettarla: se viene deciso che il tempo per una data scheda è di qualche ora e questa viene assegnata a uno sviluppatore che aveva fatto una stima di qualche giorno allora quest'ultimo si troverà in difficoltà nel portare a termine il compito; per questo motivo è importante il contributo anche degli sviluppatori junior o inesperti.

Al di là del problema del raggiungimento di una stima comune, per il quale vedremo delle tecniche specifiche, ci possono essere una serie di problemi di stima legati alla funzionalità in sé. Potremmo infatti avere stime:

- **molto differenti** (ore vs giorni): in questo caso, è possibile che la carta non sia descritta o compresa correttamente; se uno sviluppatore stima poche ore e un altro qualche giorno c'è qualche problema. in conclusione è necessario trovare un punto di incontro.

- quasi uniformi, ma **molto alte**: se la stima supera il tempo di iterazione potrebbe essere che la storia sia troppo ampia. Non si può neanche iniziarla in questo ciclo e continuarla nel prossimo: se alla fine dell'iterazione non ho portato a termine il lavoro prefissato è come se non l'avessi fatto (anche se magari era stato completato all'80%), perché il cliente non lo vede nella release e tale lavoro non è dunque dimostrabile. Per ovviare a questo problema si può fare lo **splitting** delle carte, ovvero scomporre una carta in più carte in modo da dividere il problema in sotto-problemi.

- non uguali ma **simili**: non bisogna prendere la più bassa, alta o la media. Come abbiamo già detto, secondo XP bisogna arrivare ad un accordo in modo tale che chiunque nel team si riconosca nella stima effettuata.

Oltre a ciò, la fase di stima dei tempi si porta dietro diverse problematiche intrinseche, tra cui:

- **perdita di tempo**: per accordarsi su una stima comune si spende molto tempo (troppa comunicazione);

- **effetto àncora** (anchoring effect): si tratta di un effetto che si verifica quando bisogna assegnare un valore ad una quantità ignota. Poiché il cervello umano è più bravo a

ragionare per relazioni piuttosto che per assoluti, una volta che viene fatta la prima stima numerica questa definisce l'ordine di grandezza delle stime successive, facendo cioè da punto di riferimento da cui è molto difficile distanziarsi: nel nostro caso quando il team si riunisce per fare delle stime e il primo membro dà la sua opinione, tutte le stime successive orbiteranno intorno ad essa. Tale effetto impedisce di fare una stima che prenda obiettivamente in considerazione le sensazioni di tutti i membri del team, e va dunque assolutamente evitato.

Per evitare questi problemi e semplificare il processo di stima si sono sviluppati diversi processi, che data la loro natura giocosa aumentano anche l'engagement degli sviluppatori in questa fase di pianificazione.

Planning poker


Planning poker

Una per una vengono presentate brevemente le carte con le user stories facendo attenzione a non fare alcun riferimento alle tempistiche in modo da non creare subito un effetto àncora: in questa fase il team può fare domande, chiedere chiarimenti e discutere per chiarire assunzioni e rischi sulla user story, ma deve stare molto attento a non fare alcuna stima.

Dopodiché ogni componente del team sceglie una carta dal proprio mazzo personale per rappresentare la propria stima e la pone coperta sul tavolo: su queste carte si trovano una serie di numeri senza unità di misura che vanno da 0 a 100 seguendo un andamento non uniforme; il loro scopo è quello di definire un'ordine di grandezza piuttosto che una stima precisa. Ci sono anche delle carte particolari, ovvero:

- il punto di domanda indica che non si è in grado di dare una stima
- la tazza di caffè indica che la riunione è andata troppo per le lunghe ed è necessaria una pausa.

Fatta questa prima stima *blind* le carte vengono girate contemporaneamente: idealmente vi dovrebbe essere l'unanimità sulla stima. Se così non è chi ha espresso le stime più basse e più alte ha ~1 minuto per motivare la propria scelta in modo da cercare di convincere gli altri; si noti che agli altri componenti del team non è concesso parlare per evitare di perdere troppo tempo!
Finito questo momento di consultazione tutti i membri del team fanno una nuova stima e si continua così finché non si raggiunge l'unanimità; solitamente le votazioni convergono dopo un paio di round.

Ma qual'è l'unità di misura su cui si fanno le stime? Dipende: essa può essere scelta prima o dopo aver trovato un accordo; possono essere ore, giorni o pomodori (un pomodoro è formato

da 25 minuti senza alcuna distrazioni,e dopo c'è una pausa). Ovviamente non si può pretendere di lavorare delle ore senza alcuna distrazione, per cui in queste stime si considera anche un certo **slack time**, ovvero un tempo cuscinetto per che comprende il "tempo perso" a causa di distrazioni.

Team Estimation Game

Si tratta di un metodo un po più complesso articolato in 3 fasi e basato sul confronto tra i diversi task piuttosto che sulla stima numerica: esso si basa infatti sull'idea che sia semplice stabilire se un task sia più facile o più difficile di un altro, mentre è molto più complicato capire di quanto sia più facile/difficile. L'idea è dunque quella di splittare in fasi questa cosa di dover dare un valore al task considerandone sempre di più difficili per arrivare a fare una buona stima.

PRIMA FASE


Prima fase del team estimation game

Si fa una pila con le storie e si mette la prima carta al centro del tavolo. I developer si mettono in fila e uno alla volta eseguono queste azioni:

- il **primo della fila estrae una carta della pila**, la legge ad alta voce e la **posiziona** a sinistra (più semplice), a destra (più complicata) o sotto (equivalente) la carta già presente sul tavolo.
- il **prossimo developer** può:
    - **estrarre una nuova carta dalla pila** e **posizionarla** secondo le stesse regole, eventualmente inserendola in mezzo a due colonne già presenti;
    - **spostare una carta precedentemente posizionata** commentando la motivazione della sua scelta; può ovviamente succedere che tale carta venga rispostata nella sua posizione originale, ma dopo un po' si troverà un accordo sulla difficoltà del relativo task.

Terminata la pila avremo le carte disposte sul tavolo in colonne di difficoltà comparabile, ordinate dalla meno difficile (sinistra) alla più difficile (destra). Oltre ad aver ridotto la comunicazione (molte carte non saranno contestate), usando questa tecnica abbiamo evitato anche l'effetto àncora rendendolo relativo: l'assenza di valori precisi evita il rischio di influenzare eccessivamente gli altri. Inoltre a differenza del planning poker si può tornare sulle proprie decisioni, cosa che favorisce un continuo adattamento e ripensamento delle stime.

SECONDA FASE

Si cerca dunque di quantificare le *distanze* tra le carte.


Seconda fase del team estimation game

Ci si mette di nuovo in coda davanti al tavolo con il mazzo di carte del planning poker (uno solo, non uno per persona) e **si cerca di etichettare le colonne in base alle difficoltà**.

Si posiziona la prima carta (solitamente si parte da 2 perchè magari nella prossima iterazione può esserci qualcosa di ancora più facile) sopra la prima colonna.

Quindi:

- il **primo sviluppatore** prende il valore successivo e lo posiziona sulla prima colonna che pensa abbia quel valore (rispetto al 2), oppure lo posiziona tra due colonne se pensa che sia un valore di difficoltà intermedio tra le due.
- lo **sviluppatore successivo** può invece:
    - **estrarre una carta** dal mazzo e **posizionarla** secondo le regole di prima (la prima colonna che pensa abbia un particolare valore di difficoltà);
    - **spostare una carta** con un valore precedentemente posizionato, commentando la motivazione dello spostamento;
    - **passare** il turno, nel caso in cui non ci siano più carte nella pila e non si vogliono spostare altre carte.

È possibile avere delle carte in cui sopra non c'è nessun numero, queste saranno assimilate alla colonna alla loro sinistra.

Al termine di questa fase, la situazione sarà simile alla seguente:


Fine seconda fase del team estimation game

TERZA FASE

Si stima il tempo in ore/uomo di una delle carte più semplici e successivamente si calcolano tutte le colonne in proporzione alla prima. Ma questa fase è davvero cosi utile? Nella pratica si è visto che **è inutile valutare il lavoro fatto in ore/uomo**, anche perchè con il passare del tempo la taratura può variare.

Nella prossima sezione parliamo di come la nozione di **velocity** risolve questo problema.

Velocity

È importante riuscire a stimare la *velocità* con la quale stiamo avanzando. In fisica la velocità è data dal rapporto tra la distanza percorsa e il tempo per percorrerla. Questa proprietà può essere usata anche nella gestione dello sviluppo agile: il numeratore è il punteggio delle storie mentre il denominatore è la lunghezza dell'iterazione (assimilabile in un'unità di tempo).

La ***velocity*** nel mondo agile è quindi il **numero di story point** guadagnati nell'arco dell'iterazione corrente.

Essa riesce quindi a dare un'idea di quanto si è riusciti a fare in termini di complessità astratta. Se per esempio il team è riuscito a fare 50 punti nella iterazione appena finita, è ragionevole prefissarsi di fare almento 50 punti nell'iterazione successiva.

La velocity **non può essere usata** per dare **premi**, per **confrontare** team diversi o **punire** in caso di diminuzione, però si adatta al modo diverso degli sviluppatori di gestire le stime e dal fatto che si tende a sottostimare o sovrastimare carte diverse.

All'atto di aggiungere una persona questa metrica deve inizialmente rimanere invariata, per prevedere la sua formazione; se la rimuovo ci sarà una perdita di produttività.

La velocity **non deve considerare le storie lasciate incompiute**, quindi anche se l'ho completata al 90% devo considerarla come se non l'avessi fatta. Inoltre, **non deve essere** imposta: la velocity di un team è fissa e non può essere aumentata.

Esiste un movimento chiamato ***no estimates***, che evita al team tutta la parte delle stime. Dall'esperienza del prof. Bellettini, però, questa metodologia funziona in team molto maturi che sono in grado di guidare il ciente a formulare storie simili in termini di difficoltà, avendo tutti una misura standard per le storie.

Brevi cicli di rilascio

Per ridurre i rischi, la vita e lo sviluppo dell'applicazione sono scanditi dai rilasci di diversioni del prodotto funzionanti, di solito uno ogni due settimane (come abbiamo visto in scrum con il *freez*, ma con un tempo di rilascio minore). È necessario avere abbastanza tempo per sviluppare qualcosa di concreto, e il cliente per poter pensare alle richieste che ha fatto e stabilire se ha bisogno di modifiche.

Betrand Meyer, nel suo libro *"Agile! The Good, the Hype and the Ugly"*, definisce questa idea *"brillante"*, *"forse l'idea agile con l'influenza e impatto maggiore nell'industria"*.

Uso di una metafora

Definire un **nuovo vocabolario** con cui parlare con l'utente (tecnica *non informatica*) ma anche ai nuovi sviluppatori. Serve per permettere una nominazione di classi e metodi omogenei e fornire una vista d'insieme. Siccome non c'è una vera documentazione in XP, possiamo usare queste metafore come una vista d'insieme, quindi sostituire in parte l'architettura del sistema, e far capire gli elementi fondamentali, il loro scopo e le relazioni fra loro.

Semplicità di progetto

Ovvero l'***arte di massimizzare il lavoro non fatto***, o da non fare. Non è necessario riscrivere cose già esistenti e consolidate.

Uno slogan tipico è **KISS**: **Keep It Simple, Stupid**.

Questo punto si contrappone al *design for change* che viene invece visto come un appesantimento inutile, perchè una feature che aggiungiamo può essere scartata dal cliente.

Testing

È consolidato su due fronti:

- i clienti scrivono i **test di accettazione** (o funzionali) sulle schede per aumentare la loro fiducia nel programmi;
- i programmatori scrivono i **test di unità** perché la fiducia nel codice diventi parte del programma stesso.

Nell'XP ogni aspetto viene massimizzato, ma in particolare il testing viene esasperato di più in quanto, oltre ad essere molto importante, molti altri aspetti si basano su di esso (vedi la figura all'inizio della sezione). Ha il ruolo di **rete di protezione** in tutte le fasi: ogni cambiamento è verificabile tramite i test.

Il testing aiuta molto anche quando non si parte da zero con il programma, ma quando si deve modificare un programma proprietario precedentemente sviluppato anche in modalità non agile. Prima di apportare modifiche al codice scrivo i test e solo successivamente procedo, in modo da non causare problemi.

Un altro concetto importante è che i test dovrebbero **coprire tutte le righe di codice**.

Refactoring

Anche da novizi, non bisogna avere paura di apportare modifiche che semplificano il progetto: bisogna avere coraggio.

Il refactoring è l'operazione che **modifica solo le proprietà interne** del software, **non le funzionalità**. L'obiettivo è eliminare l'entropia generata dalle continue modifiche e aggiunte.

Il refactoring deve essere **graduale e continuo** in modo da poter aggiungere funzionalità in maniera semplice. Chiaramente, in caso di ristrutturazioni architetturali di grosse dimensioni di sistemi legacy non è sempre possibile procedere in questa maniera.

Parti di codice non stimolate da test non sono utili ai fini della soluzione: o si aggiungono test per gestire i casi specifici, altrimenti si possono rimuovere *in toto*.

Il refactoring è una delle tecniche più **importanti** e **fondamentali** dell'XP.

Pair programming

La programmazione a coppie (**pair programming**) è una tecnica controintuitiva: dal punto di vista del manager si pagano due persone per fare il lavoro di una, ma non è così.

Ci sono diversi **vantaggi**:

- in coppia, **ci si controlla a vicenda** su ogni aspetto (codice, rispetto delle regole XP, idee);
- mentre il *pilota* attua le idee, il *navigatore* pensa cosa fare subito dopo: forma di **refactoring**;
- favorisce l'**inserimento di nuovo personale**: piuttosto che lasciare i novizi da soli a studiare libroni, vengono affiancati e incitati a osservare e interagire con persone esperte che stanno lavorando;
- fa ottenere una **proprietà collettiva** (conoscenza osmotica), come descritta da Crystal. Un altro punto importante sono i commenti *naive* (ovvero fatti da programmatori junior) per permettere di chiarire concetti basilari date spesso per scontato.

Raddoppiare il numero di persone raddoppia la produttività? **No**, è stimato invece che la produttività aumenti circa del 50% - quindi non abbastanza per giustificare il costo.

Diversi studi si chiedono se la produttività calcolata puntualmente sia una metrica sensata. Secondo molti no, perché al termine di un'iterazione ciò che sembra poco produttivo in realtà lo è di più: il tempo non successivamente speso in verifica, convalida e refactoring è largamente assorbito dall'**ispezione continua del codice** svoltasi durante le sessioni di pair programming.

Critiche

Betrand Meyer, nel suo libro *“Agile! The Good, the Hype and the Ugly”*, scrive:

> **Applied judiciously, pair programming can unquestionably be useful**. Many developers enjoy the opportunity to program jointly with a peer, particularly to deal with a thorny part of an assignment. The basic techniques, in particular the idea of speaking your thoughts aloud for immediate feedback, are well understood and widely applied. (As a manager I regularly hear, from a developer, “On this problem I would like to engage in a round of pair programming with X ”, and invariably find it a good idea.)
>
> What is puzzling is the insistence of XP advocates that this technique is the only way to develop software and has to be applied at all times. **Such insistence makes no sense**, for two reasons.
>
> The first is the **inconclusiveness of empirical evidence**, noted above. Granted, lack of data is often used as a pretext to block the introduction of new techniques. When an idea is obviously productive, we should not wait for massive, incontrovertible proof. But here there is actually a fair amount of empirical evidence, and it does not show a significant advantage for pair programming. Pair programming may be good in some circumstances, but if it were always the solution the studies would show it. In the absence of scientific evidence, a universal move is based on ideology, not reason.
>
> The second reason, which may also explain why studies’ results vary, is that **people are different**. Many excellent programmers love interacting with someone else when they write programs; and many excellent programmers do not. Those of the second kind want to think in depth, undisturbed. The general agile view is that communication should be encouraged and that the days of the solitary, silent genius are gone. Fine; but if your team has an outstanding programmer who during the critical steps needs peace, quiet and solitude, do you kick him out of the team, or force him to work in a way that for him may be torture?

> It is one thing to require that people explain their work to others; it is another, quite dangerous, to **force a single work pattern**, especially in a highly creative and challenging intellectual endeavor. When Linus Torvalds was writing Linux, he was pretty much by himself; that did not prevent him from showing his code, and, later on, engaging thousands of people to collaborate on it.

Proprietà collettiva

Il codice non appartiene a una singola persona ma al *team*: non devono quindi esistere policy di *"code owners"* a la Microsoft. Tutti i componenti del team sono quindi autorizzati a modificare e sistemare ogni parte del codice, anche se scritta da un altro.

Durante il giorno, più volte al giorno, è comune **cambiare coppia** e saranno quindi possibili situazioni in cui nessuno dei due ha una profonda conoscenza della parte di codice che si sta trattando o che il task non si addica alle competenze della coppia.

In tutti i casi, in XP ci si riferisce al team e non al singolo.

Integrazione continua

Nell'ottica di ricevere feedback rapidi dal cliente è necessario **integrare spesso**, anche **più volte al giorno**. Questo non significa far passare i test d'unità per integrare tutto in un'unica operazione, ma essere graduali: è frequente scoprire che parti testate e funzionanti singolarmente una volta integrate nel prodotto finale non funzionano.

L'integrazione continua e graduale è una tecnica largamente utilizzata in tutti i campi, non solo nello sviluppo software.

Al termine dello sviluppo di una *feature*, è compito della coppia integrarla nella **macchina di riferimento**. L'accesso a tale macchina deve essere regolato in maniera **esclusiva**: in situazioni di lavoro da remoto si può utilizzare un token. La macchina di riferimento si trova, per quanto riguarda le funzionalità, in una situazione **monotona crescente**. Ad ogni integrazione è necessario produrre sempre qualcosa di consegnabile al cliente.

Una *user story* si definisce **completata** solo dopo aver terminato l'integrazione, superato dei test di integrazione e aver mostrato al cliente il risultato della macchina complessiva dopo l'integrazione.

Un'altro punto a favore della continua integrazione è che evita la situazione in cui una coppia modifichi la macchina **dopo molto tempo** dalla propria ultima integrazione, aumentando di molto la probabilità di errori per le altre coppie.

Se una coppia **non riesce ad integrare** blocca anche tutte le altre che non possono andare avanti con le use story, quindi sarà necessario che quella coppia rinunci, ritorni sulla sua macchina e cerchi di risolvere lì - tutte le coppie hanno una propria macchina su cui testano prima di farlo su quella comune.

Settimana di 40 ore

Il mestiere di sviluppatore ha sempre avuto dei **ritmi dettati dalle consegne**: lavorare troppo a lungo porta a un abbassamento della produttività, oltre che a stress e frustrazione.

Nell'XP si cerca di evitare queste situazioni in modo da avere una resa migliore, avere maggior soddisfazione nel lavorare nel team e nell'azienda, avere meno problemi fuori dal lavoro (tante volte questo eccessivo lavoro può causare problemi familiari) e inoltre abbassare la probabilità per l'azienda di perdere dipendenti.

Purtroppo però il mestiere dello sviluppatore non è meccanico e molto spesso si vuole portare a termine quello che si sta facendo perchè magari si è quasi alla soluzione, inoltre si continua a pensare a come risolvere dei problemi anche fuori dall'orario lavorativo.

Cliente sul posto

Dal punto di vista del cliente, il suo inserimento nel luogo fisico di sviluppo è un vantaggio in quanto può essere sicuro che gli sviluppatori stiano lavorando per lui e **può verificare come procede il progetto**.

Dal punto di vista degli sviluppatori, invece, è fondamentale avere il cliente sul posto per potergli **chiedere chiarimenti** in caso di specifiche non chiare. La possibilità di poter far domande è come avere una *documentazione vivente*; il cliente potrà continuare a lavorare per la sua azienda, ma dovrà dare priorità alle richieste degli sviluppatori.

Avere il cliente sul posto ha comunque dei **limiti**: quest'ultimo, infatti, deve essere scelto accuratamente per avere una persona rappresentativa di tutti gli stakeholder; il compito è forse impossibile. Se il cliente del posto non è disponibile, il team deve trovare dei modi per poter comunque avere un *punto di riferimento*: la tecnica Scrum introduce il concetto di **product owner**, un componente interno al team che si comporta come se fosse il cliente.

Il cliente durante le iterazioni può **creare altre storie** che a partire dall'iterazione successiva potrà inserire nel planning game; è inoltre disponibile per **test funzionali**.

Standard di codifica

È necessario prevedere delle regole (**convenzioni comuni**) che specificano come scrivere il codice, per aumentare la leggibilità e quindi la comunicazione attraverso il codice.

Spesso, si utilizzano degli **strumenti** per garantire il rispetto delle convenzioni o autocorreggere il codice auutomaticamente.

Avere uno standard di codifica aiuta inoltre:

- il refactoring;
- la programmazione a coppie;
- la proprietà collettiva.

They're just rules

L'ultima regola *"non è canonica"*, in quanto è stata aggiunta successivamente da alcuni agilisti.

Al termine di un'iterazione si fa un **resoconto** e quindi decidere come comportarsi per l'iterazione successiva. Nel suddetto resoconto si può anche decidere di **sospendere regole** se si pensa che non siano adatte per la situazione o per il team e successivamente possono essere reintrodotte. La decisione di non seguire una regola deve essere sempre fatta a livello di **team**, non dal singolo o dalla coppia.

In conclusione, l'XP non è una tecnica così rigida e rigorosa: ad ogni iterazione, si possono effettuare test per trovare il giusto equilibrio.

Questo punto non però è condiviso da tutti e una motivazione la si può trovare nel fatto che tutti i punti sono interconnessi tra loro, e quindio non è possibile studiarli singolarmente senza considerate anche gli altri perchè non avrebbero senso in quanto hanno una forte dipendenza l'una dall'altra; non a caso nei punti sopra si può notare come si influenzino a vicenda.

Relazione con il modello a cascata

È possibile tentare di raggruppare le diverse tecniche dell'eXtreme Programming nelle macrofasi descritte dal modello a cascata.

- **Requirements**:
  - i *clienti fanno parte del team di sviluppo*: requirements viventi;
  - *consegne incrementali* e pianificazioni continue: evoluzione del progetto.
- **Design**:
  - *metafora* come visione unificante del progetto;
  - *refactoring*: è design puro, molto utile per rendere possibile l'evolvibilità del software;
  - presumere la *semplicità*.
- **Code**:
  - *programmazione a coppie*;
  - *proprietà collettiva*;
  - *integrazione continua*;
  - *standard di codifica*.
- **Test**

    - *test di unità* continuo (da scriversi prima del codice);
    - *test funzionale* scritto dagli utenti nelle user stories.

In XP è inoltre presente la nozione di *prototipo* sotto il nome di ***spike***, ovvero programmi molto semplici creati per esplorare soluzioni potenziali. Sono utili per capire se ho compreso le specifiche, la tecnologia da utilizzare e l'approccio da avere con i componenti esterni con cui bisogna dialogare. Questi prototipi vengono creati, mostrati al cliente e quindi scartati.

Documentazione

La **documentazione** cartacea **non è necessaria**: il cliente, il compagno di peer programming e il codice *sono la documentazione*.

La documentazione è sostituita dal codice in quanto:

- i **test di unità** che sono delle *specifiche eseguibili*, infatti li scrivo prima di fare il codice (prima dico cosa voglio tramite il test);
- il **continuo refactoring** consente di avere un codice estremamente leggibile e quindi elimina il bisogno dei commenti. Scrivere codice semplice tramite refactoring in modo che sia facilmente comprensibile è in realtà molto complesso.

CRC cards

Le **Class Responsibility and Collaboration cards** permettono di rappresentare classi e le relazioni tra di esse. Nate in ambiente didattico per spiegare l'OOP, sono ora utilizzati da alcuni team agile per discutere di design e il modo di utilizzo è simile a quello del planning game.

Le carte CRC sono realizzate su piccole schede di carta o cartoncino. Ciascuna carta descrive una classe (o un oggetto) in modo sommario, indicando:

- Il nome della classe
- Le sue superclassi e sottoclassi (dove applicabile)
- Le sue responsabilità
- Il nome di altre classi con cui questa classe collabora per svolgere i compiti di cui è responsabile
- L'autore

L'uso di schede di piccole dimensioni ha lo scopo di limitare la complessità della descrizione, evitando che vengano riportate troppe informazioni di dettaglio. Serve anche a impedire che a una classe vengano assegnate troppe responsabilità. Il supporto

cartaceo consente una serie di attività gestuali utili in fase di brainstorming, come piazzare le carte su un tavolo e spostarle, riorganizzarle, o anche eliminarle e sostituirle facilmente con altre nel corso della discussione. Il posizionamento delle carte su un tavolo può essere usato intuitivamente per rappresentare informazioni aggiuntive; per esempio, due carte possono essere parzialmente sovrapposte per indicare una relazione di stretta collaborazione, o una carta può essere posta sopra un'altra per indicare una relazione di controllo/supervisione.

Da Wikipedia, l'enciclopedia libera (licenza CC BY-SA 3.0).

Criticità

Quando non utilizzare XP

Back non esclude mai la possibilità di utilizzare l'XP: secondo lui diceva può provare ad utilizzare questo approccio sempre (anche se in realtà non è sempre possibile "provare"), a patto che vengano rispettati i 12 punti elencati sopra.

Da questo possiamo concludere che Agile non si può usare quando:

- l'**ambiente** non permette l'applicazione dei 12 punti, come per esempio succede con i team dislocati in luoghi diversi;
- ci sono **barriere managieriali**, come team troppo numerosi;
- ci sono **barriere tecnologiche**, come quando per esempio non è possibile utilizzare una macchina specifica condivisa da tutte le coppie per i test, ostacolando l'integrazione continua.
- ci sono **troppi stakeholders** diversi e in contrasto tra loro;
- situazioni in cui **la consegna incrementale non ha senso**, come per una *centrale nucleare* (vero Dyatlov?).

Critiche da Meyer

Di seguito sono elencate alcune critiche all'eXtreme Programming fatte da Meyer (già pluricitato in questo documento).

- **Sottovalutazione dell'up-front**, ovvero la progettazione iniziale prima di partire. Per Meyer, a parte in casi eccezionali (sviluppatori o manager particolarmente bravi) la progettazione non può essere fatta in modo totalmente incrementale. Nell'esperienza dei

tesisti e colleghi di dottorando del prof. Bellettini questo problema non è così presente, ma potrebbe trattarsi di *survivorship bias*.
- **Sopravalutazione delle user stories**: secondo Meyer sono troppo specifiche per sostituire i requisiti.
- **Mancata evidenziazione delle dipendenze tra user stories**. Le *user stories* dovrebbero essere indipendenti tra loro, ma questo non è quasi mai possibile; nel design classico si utilizzano i diagrammi di Gantt per chiarire tutte le dipendenze tra i diversi punti del sistema da realizzare.
- Il **TTD può portare ad una visione troppo ristretta**.
- **Cross functional team**: se i team sono troppo disomogenei, ovvero ci sono tante singole figure specializzate in un campo e queste devono collaborare in coppia, ci possono essere dei problemi.

I punti di cui sopra cercano di evidenziare la mancanza di approfondimento e chiarezza dell'XP su alcuni aspetti dell'approccio ad un lavoro fornito da un cliente.

È consigliata la lettura del libro di Meyer.

Mesi uomo

È diffuso tra i manager pensare che la stima in tempo in mesi uomo sia graficata come un ramo di un iperbole, ovvero che il tempo diminuisca simil-esponenzialmente all'aumentare dei mesi uomo; tale stima non considera i tempi di *overhead*, ovvero il tempo impiegato per la comunicazione e tutto ciò che non è l'implementazione. I mesi uomo **non quindi sono una metrica valida**, ma sono utili solo a posteriori per valutare se un approccio ad un problema si è dimostrato valido.

Nella realtà, **all'aumentare delle persone aumenta il bisogno di comunicare**.

Quando il lavoro è strettamente sequenziale e non parallelizzabile (come la *gravidanza*) anche all'aumentare delle persone il tempo non cambia.

Nel mondo dello sviluppatore software spesso c'è un **numero ideale di persone** per un progetto; dopodiché, persone in più causano solo confusione e rallentano i tempi a causa della comunicazione. Il numero può anche essere grande, dipende dall'entità del progetto (esempio: space shuttle).

In generale, le metodologie agili iniziano a **non funzionare più** se il team è **più grande di 8-10 persone**. Quando il progetto non funziona più con un tale numero di persone, è necessario esplorare altre pratiche.

Open source

Apriamo ora le porte a un nuovo modo di sviluppare software che si è molto diffuso negli ultimi decenni ed è ora alla base di progetti applicativi molto importanti e famosi: il processo **open-source**. In due parole, un software open-source è un software il cui codice è liberamente consultabile online e il cui sviluppo si basa non su un singolo team ma su una community di sviluppatori indipendenti.

- **Letteratura**: analisi di quattro brani sulla disciplina
- **Sfide** rispetto allo sviluppo tradizionale

Letteratura

In questa sezione verranno analizzati i seguenti testi visti a lezione.

- ***The Cathedral and the Bazaar*** di Eric Steven Raymond
- ***Care and Feeding of FOSS*** di Craig James
- ***The Emerging Economic Paradigm Of Open Source*** di Bruce Perens
- ***An empirical study of open-source and closed-source software products*** di James Paulson

The Cathedral and the Bazaar

Raymond propone nel suo articolo due immagini per descrivere i due mondi contrapposti closed-source e open-source:

- la **cattedrale** (closed source): si tratta di un ambiente molto gerarchizzato, fortemente legato al progetto iniziale di un unico "architetto" responsabile dei lavori;
- il **bazaar** (open source): è l'ambiente più anarchico possibile, in cui ognuno lavora per sé e costruisce espandendo ciò che gli altri hanno già fatto.

Entrambe le costruzioni risultano splendide e attraenti, ma sono chiaramente legate a modi di pensare la costruzione e la progettazione totalmente opposti.

Vita e vantaggi di un progetto open-source

Nell'articolo Raymond prosegue a descrivere come nasce un progetto open-source, esordendo con la seguente citazione:

> Ogni buon lavoro software inizia dalla frenesia personale di un singolo sviluppatore.

Si delinea dunque la seguente timeline di un progetto open-source:

1. Uno sviluppatore ha un problema e intende risolverlo sviluppando un'applicazione.
2. Lo sviluppatore chiede ad amici o colleghi cosa sanno sull'argomento: alcuni hanno lo stesso problema o problemi simili, ma nessuno ha una soluzione.
3. Le persone interessate cominciano a scambiarsi pareri e conoscenze sull'argomento.
4. Coloro che intendono spendere delle risorse (di fatto il proprio tempo libero) per risolvere il problema danno il via ad un progetto informale.
5. I membri del progetto lavorano sul problema finché non raggiungono dei risultati presentabili.

   Fino a qui però il progetto non è ancora definibile open-source in quanto vi lavora solo un gruppo ristretto di amici e conoscenti: il vero progetto open nasce quando **viene messo online a disposizione di tutti il codice sorgente**. Continuando, dunque:
6. Si rende noto il lavoro online e arrivano i primi suggerimenti esterni al gruppo: questi saranno tanto più frequenti quanto più il progetto pubblicato presenterà errori in quanto la comunità, principalmente composta da altri sviluppatori, sarà motivata a risolverli.
7. Si crea interazione tra i vecchi e i nuovi membri del gruppo di sviluppo.
8. Nuove informazioni e competenze vengono acquisite e si ritorna al punto 5.

Raymond continua esponendo alcune delle caratteristiche e dei vantaggi dei progetti open-source, primo fra tutti il fatto che:

> Se dai a tutti il codice sorgente, ognuno di essi diventa un tuo ingegnere.

---

dove con questo si intende che la possibilità di vedere e commentare il codice sorgente permette a utenti esperti di suggerire modifiche e prendere parte attiva allo sviluppo. Talvolta si tende però a pensare che un progetto di questo tipo sia destinato unicamente ad altri informatici o sviluppatori, ma ciò non è affatto vero: tante attività utili a portare avanti un progetto open-source non richiedono necessariamente competenze informatiche, come per esempio la segnalazione di bug o la moderazione di contenuti nella comunità.

A tal proposito, è importante il seguente concetto:

> Se ci sono abbastanza occhi [che cercano errori], gli errori diventano di poco conto.

Non c'è molto da spiegare: più sono le persone che controllano e leggono il codice più sarà probabile trovare gli errori in esso contenuti; inoltre, gli errori rilevati possono essere risolti più facilmente grazie al supporto della community di sviluppatori, che potrebbe già conoscere una soluzione.
L'accento posto sulla community viene ulteriormente rimarcato dal valore che viene attribuito ai beta-tester, che in un progetto open-source è chiunque utilizzi l'applicazione in qualunque suo stadio vista la sua estrema malleabilità:

> Se tratti i tuoi beta-tester come se fossero la tua risorsa più importante, essi risponderanno diventando la tua risorsa più importante.

Per mantenere attiva la community di sviluppatori è però necessario un costante monitoraggio e cura; per permettere al progetto open-source di sopravvivere anche quando l'interesse dei creatori originali si è spento è fondamentale passarne il controllo a qualcuno di fidato e competente, come ci ricorda Raymond nell'ultima citazione che riportiamo:

> Quando hai perso interesse in un programma, l'ultimo tuo dovere è passarlo a un successore competente.

Spesso questo passaggio di testimone non viene fatto e il progetto muore: occorre invece trovare qualcuno di interessato allo sviluppo, anche perché un programma in uso dovrà necessariamente cambiare ed evolvere in futuro.

Confronto tra modelli

Per capire meglio i concetti fondanti del mondo open-source mostriamo in un modo sintetico un confronto tra lo stesso e i metodi di sviluppo tradizionale e agile nel riguardo di svariati concetti:

Cosa	Tradizionale	Agile	Open source
Documentazione	La documentazione è enfatizzata come strumento	La documentazione è de-enfatizzata.	Tutti i manufatti di sviluppo sono disponibili a chiunque,

Cosa	Tradizionale	Agile	Open source
	di controllo qualità e gestione.		compresi il codice e la documentazione.
Requisiti	Gli analisti traducono le necessità dell'utente in specifiche software.	Gli utenti fanno parte del team.	Gli sviluppatori spesso sono anche gli utenti.
Assegnamento dello staff	Gli sviluppatori sono assegnati ad un unico progetto.	Gli sviluppatori sono assegnati ad un unico progetto.	Gli sviluppatori tipicamente lavorano su più progetti con diversi livelli di partecipazione (*impossibile pianificare lo sviluppo*).
Revisione del codice paritaria	La revisione del codice tra pari è ampiamente accettata ma raramente effettuata.	La *pair programming* introduce una forma di revisione del codice tra pari.	La revisione del codice è una necessità ed è praticata quasi universalmente.
Tempi di rilascio	Tante feature in poche release massicce.	Tante piccole release incrementali.	Gerarchia dei tipi di release: *nightly* (compilazione giornaliera dal branch master), *development* e *stable*.
Organizzazione	I team sono gestiti dall'alto.	I team sono auto-organizzati.	I contributori individuali decidono per sé come organizzare la propria partecipazione.
Testing	Il testing è gestito dallo staff di *Quality Assurance* (QA), che segue le	Il testing è parte integrante dello sviluppo (TDD).	Il testing e la QA possono essere svolti da tutti gli sviluppatori.

Cosa	Tradizionale	Agile	Open source
	attività di sviluppo.		
Distribuzione del lavoro	Parti differenti della codebase sono assegnate a persone differenti.	Chiunque può modificare qualsiasi parte della codebase.	Chiunque può modificare qualsiasi parte della codebase, ma solo i *committer* possono rendere ufficiali le modifiche.

Care and Feeding of FOSS

Ma come si inserisce il modello di sviluppo open-source in un panorama tecnologico sempre più accentrato nelle mani di poche e potenti aziende? Il timore è infatti quello che rendendo il codice disponibile a tutti si potrebbe essere subito vittima di plagio da parte di giganti del settore in grado di realizzare in poco tempo ciò che prenderebbe a un team open-source svariati mesi, accaparrandosi così una larga fetta di mercato. Craig James smentisce tuttavia tale visione nel suo articolo, in cui descrive il ciclo di vita di un software FOSS (Free and Open Source Software) come la sequenza delle seguenti fasi:

1. **Invenzione**: qualcuno ha un'idea e la implementa facendola funzionare. Dal momento in cui l'idea viene resa pubblica si assiste una grossa esplosione di progetti in tal senso finanziati da varie aziende che cercano di diventare leader nel nuovo mercato.
2. **Espansione e innovazione**: il mondo si accorge dell'invenzione e la tecnologia inizia a espandersi in quanto le aziende si 'rincorrono' a vicenda cercando di aggiungere più funzionalità possibili. Questa fase non rappresenta un buon momento per far nascere un progetto open source: un piccolo gruppo non riuscirebbe a prevalere sulle grandi aziende; poiché inoltre le specifiche non sono ancora ben definite si rischierebbe di implementare funzioni inutili.
3. **Consolidamento**: i prodotti di alcune aziende iniziano a dominare il mercato, mentre i competitor vengono assorbiti, falliscono o diventano di nicchia. Diminuisce complessivamente il numero di player e l'innovazione rallenta.
4. **Maturità**: il problema e le specifiche sono ora ben chiare e consolidate. Per un prodotto commerciale è ormai difficile entrare nel mercato, ma per uno open source è paradossalmente il momento migliore: il piccolo team ha uno slancio innovativo che le grosse aziende non hanno più e il loro prodotto può brillare dei vantaggi del mondo open-source, tra cui sicurezza e flessibilità.

5. **FOSS Domination**: lentamente, il prodotto open-source inizia a erodere il vantaggio tecnologico dei competitor commerciali, che d'altra parte non hanno alcun interesse ad innovare ulteriormente: ciascuna loro innovazione potrebbe infatti essere facilmente copiata ed essi sono inoltre già largamente rientrati del loro investimento iniziale. Il prodotto open-source inizia ad accaparrarsi fette di mercato.
6. **The FOSS era**: alla fine il progetto open-source domina completamente il nuovo mercato, mentre le grandi aziende devono ripiegare sulla vendita di servizi più che di software.

Il vantaggio di un progetto open-source non è dunque tanto la rapidità di espansione o l'abilità di intercettare il mercato, quanto il fatto che esso permetta una continua innovazione che *segue* le necessità del mercato, cosa che le grandi aziende faticano a conseguire in quanto ancora legate a un paradigma di investimento in cui una volta fatto il lavoro e guadagnato un tot è più costoso fare manutenzione del software piuttosto che lasciarlo morire.

The Emerging Economic Paradigm Of Open Source

Affrontiamo ora un fenomeno interessante: perché sempre più aziende investono in software open-source? A prima vista parrebbe un controsenso, perché da sempre i produttori di software proteggono il proprio prodotto tramite segreto aziendale: avere codice liberamente consultabile distrugge tale privatezza ed espone all'ascesa di competitor. Per rispondere a questa domanda ci serviamo di un articolo di Bruce Perens sull'argomento.

Perens fa in primo luogo notare che spesso per diverse aziende il software da sviluppare non è il prodotto, ma una **tecnologia abilitante essenziale**: per esempio, Amazon sviluppa molto software per il sito di e-commerce ma il suo prodotto non è il sito. In tali ambiti la scrittura di codice è un *costo*, non il prodotto su cui guadagnare.

Dal punto di vista economico è poi importante stabilire se la tecnologia dà un vantaggio competitivo, ovvero se essa è **differenziante** o **non differenziate**. Per fare ciò è sufficiente rispondere alle seguenti domande:

- **Il cliente si accorge degli effetti del software?** Per esempio, le persone si accorgono dell'esistenza del sistema di raccomandazione dei libri di Amazon?
- **I competitor non hanno accesso allo stesso software?** Se Amazon usasse il sistema di raccomandazione venduto anche a Feltrinelli allora non avrebbe senso mantenerlo privato.

Se la risposta a una delle due domande è no, avere un software proprietario non apporta alcun vantaggio: un modello di sviluppo open-source sparpaglia invece i costi e genera valore, motivo per cui sempre più aziende lo scelgono.

Perens conclude il suo articolo riportando una matrice in cui confronta 4 diversi modelli di sviluppo, ognuno con le sue caratteristiche che ne determinano l'applicabilità in diverse situazioni:

- **Retail**: il software è sviluppato per poter essere venduto a chiunque lo desideri;
- **In-House & Contract**: il software è sviluppato per un singolo committente;
- **Consortium & Non-Open-Source Collaboration**: un modello secondo cui diverse aziende concorrenti si mettono insieme per sviluppare un software comune (molto poco diffuso);
- **Open Source**: il software è disponibile gratuitamente e il suo codice è pubblico.

Tali modelli vengono valutati in base all'efficienza (*quanti soldi vanno agli sviluppatori*), al tasso di fallimento del progetto, ai costi di distribuzione, al rischio di plagio, al fatto di proteggere o meno la differenziazione a livello commerciale del cliente e/o del venditore e alla dimensione richiesta del mercato perché il progetto sia un successo.


Confronto paradigmi

Come si può notare dalla tabella, non tutti i paradigmi proteggono il vantaggio competitivo, con differenze dal punto di vista del cliente e del produttore: è dunque importante scegliere il modello di sviluppo che più si confà alle proprie esigenze.

An empirical study of open-source and closed-source software products

Concludiamo il discorso sul mondo open-source riportando uno studio portato avanti da J.W. Paulson e altri con lo scopo di verificare alcune affermazioni ritenute vere nei riguardi del software open-source.


Risultati studio

Per ognuna di tali affermazioni l'articolo definisce una metrica e la calcola nei riguardi di progetti open-source e closed-source, concludendo così che solo alcune delle dicerie sull'open-source risultano vere mentre molte altre (es. *è più sicuro*, *è più veloce*) si rivelano essere al più circostanziali.

Le sfide del modello open source

Per sua natura il sistema di sviluppo open source pone delle sfide peculiari sconosciute all'ambiente di sviluppo tradizionale; prima di affrontare dunque gli strumenti di design e

organizzazione del lavoro che sono nati per risolvere queste problematiche diamone una breve panoramica.

Difficile integrazione del software

Quella dell'integrazione del software è in realtà una vecchia sfida che viene enormemente amplificata nell'ambito FOSS. Per integrare le nuove feature sviluppate in un software stabile diversi modelli avevano costruito le proprie pratiche:

- Nel modello a cascata l'integrazione era una fase circoscritta e a sé stante.
- A tale struttura molto rigida si contrappone lo schema innovativo *Stabilize & Synchronize* nato in ambiente Microsoft: durante il giorno gli sviluppatori lavorano sul proprio pezzo di codice in cui sono responsabili, e di notte il software viene ricompilato da zero. La mattina dopo, si avevano dunque due possibilità:
    - la compilazione falliva, il responsabile veniva trovato e *"punito"*;
    - la compilazione aveva successo, il software integrato è quindi nella *"versione migliore possibile"*.
- In XP l'integrazione veniva eseguita più volte al giorno in modo esclusivo: un solo sviluppatore alla volta poteva integrare il proprio lavoro sull'unica macchina di integrazione disponibile; questo permetteva di individuare facilmente eventuali problemi di integrazione e risolverli con rapidità.

Ma come organizzare l'integrazione nel mondo open-source? Per sua natura, in questo ambito l'integrazione viene eseguita continuamente e senza coordinazione a priori: è anarchia totale, con lo svantaggio che da un giorno all'altro una enorme parte della codebase potrebbe cambiare in quanto un singolo sviluppatore potrebbe integrare mesi e mesi di lavoro in un'unica botta. Vedremo più avanti che strumenti si sono costruiti per contenere tale problematica.

Sfaldamento del team

Nell'open source nascono inoltre problemi riguardanti la gestione del team. Occorre decidere:

- come comunicare
- come tenersi uniti
- come coordinarsi
- come ottenere nuove collaborazioni

Per comunicare si utilizza di solito **internet**: si potrebbe dire che senza internet non potrebbe esistere il concetto stesso di open source. In particolare si utilizzano spesso dei **forum** per organizzare il lavoro, in modo da tenere la community unita e rispondere dubbi ai nuovi utenti.

Per quanto riguarda il coordinamento del lavoro approfondiremo nelle prossime lezioni vari strumenti per la sincronizzazione del lavoro e di versioning per codice e documentazione (come **git**).

Deve poi essere facile, addirittura banale, poter compilare il codice e ricreare l'ambiente di sviluppo omogeneo per tutti; si utilizzano quindi strumenti di **automatizzazione delle build** (come i **Makefile**) in modo che chiunque voglia partecipare possa farlo indipendentemente dalla propria configurazione software e hardware.

È infine importante *educare* i reporter dei bug e avere un sistema per organizzare per le **segnalazioni di errori**: il sistema dovrebbe essere accessibile a tutti in modo da evitare segnalazioni duplicate e consentire una facile organizzazione delle stesse. Vedremo più avanti come anche una segnalazione d'errore avrà il suo "ciclo di vita".

Sofware Configuration Management

Le soluzioni di Software Configuration Management nascono da problemi complessi purtroppo molto comuni nel mondo dello sviluppo software, come:

- pubblicare un *hotfix* su una versione precedente a quella in cui si sta sviluppando. Può essere difficile localizzare le versioni vecchie, modificarle e rimappare le modifiche sulle versioni nuove;
- condividere lavori con altri gestendo accessi contemporanei e conflitti;
- stabilire la responsabilità di ciascuna linea di codice.

Il Software Configuration Management è l'insieme di pratiche che hanno l'obiettivo di rendere sistematico il processo di sviluppo, tenendo traccia dei cambiamenti in modo che il prodotto sia in ogni instante in uno stato (*configurazione*) ben definito.

Storia

Il Configuration Management negli anni '50 nell'ambito dell'industria aerospaziale. Alla fine degli anni '70 inizia ad essere applicato all'ingegneria del software.

Gli oggetti di cui si controlla l'evoluzione sono detti *configuration item* o *artifact* (manufatti, solitamente file). Dunque l'SMC ci permette di controllare le revisioni degli *artifact* e il risultato di tali revisioni,questo processo è molto utile per la generazione di un prodotto a partire da una configurazione ben determinata.

Manufatti

Gli *"oggetti"* di cui si controlla l'evoluzione sono detti *configuration item* o manufatti; generalmente sono file. Se si cambia nome a un file è come eliminarne uno e partire da zero con uno nuovo. Originariamente i tool tracciavano i file indipendentemente, senza un senso logico (una *configurazione*) comune.

- anni '80: strumenti locali (SCCS, ...)
- anni '90: strumenti client-server centralizzati (CVS, subversion, ...)
- anni '00: strumenti distribuiti peer-to-peer (git, mercurial, bazaar, ...)

git nasce da un'esigenza di Linus Torvalds con il kernel Linux.

Centralizzato vs decentralizzato

Il mondo open source preferisce un approccio decentralizzato al version control. Perché?

- è possibile lavorare offline;

- è molto più veloce, perché la rete non fa più da *bottleneck*;
- supporta diversi modi di lavorare:
    - simil centralizzato: un repository viene considerato “di riferimento”;
    - due peer che collaborano direttamente;
    - gerarchico a più livelli (kernel Linux).

Non c’è sincronizzazione automatica, ma ci sono comandi espliciti per fare *merge* tra repository remote. In git, per via della sua struttura modulare, è possibile utilizzare il proprio algoritmo *merge* rispetto a quelli già inclusi.

Meccanismo di base

Ogni *cambiamento* è regolato da:

- **check-out**: dichiara la volontà di lavorare partendo da una particolare revisione di un manufatto (o di una configurazione di diversi manufatti);
- **check-in** (o **commit**): dichiara la volonta di registrarne una nuova (spesso chiamata change-set).

Queste operazioni vengono attivate rispetto a un *repository*. Scambio di dati tra il repository (che contiene tutte le configurazioni) e il workspace (l’ambiente in cui si trova nel filesystem).

Solitamente ho un repository e n workspace, uno per ogni ambiente dove sto lavorando.

Repository

La repository mantiene:

- date
- etichette
- versioni
- diramazioni (branches)
- ecc…

Per risparmiare spazio, le repository salvano solo le differenze tra una versione e l’altra. In realtà, Git non fa così, perché usa link simbolici: fare il *checkout* di una specifica versione è instantaneo.

Le repository possono essere centralizzate o distribuite.

Nei sistemi di versioning distribuiti c’è il concetto di **hashing**, in modo da identificare file uguali anche se in posizioni diverse. Per confrontare storie diverse si utilizzano gli hash dei file e delle

directory.

Cosa tracciare?

Qualunque sistema si usi, occorre prendere decisioni importanti che influenzano la replicabilità della produzione.

- Si traccia l'evoluzione anche di componenti fuori dal nostro controllo (librerie, compilatori)?
- Si archiviano i file che sostitusicono il prodotto (eseguibile, ecc...)?

La risposta ad entrambe queste domande è **no**, perché è scomodo, anti economico, costoso... Questo porta però problemi, perché non c'è perfetta replicabilità.

Accesso concorrente

Quando il repository è condiviso da un gruppo di lavoro, nasce il problema di gestirne l'accesso concorrente. Esistono due modelli:

- **modello *'pessimistico'*** (RCS): prevedo il possibile conflitto assicurandomi che chi lavora sia l'unico con l'accesso in scrittura. Funziona solo in ambienti centralizzati, nell'open source non può funzionare.
- **modello *'ottimistico'*** (CVS): il sistema si disinteressa del problema e fornisce supporto per le attività di *merge* di change-set paralleli potenzialmente conflittuali.

Il modello ottimistico può essere regolato con i branch: l'attività di *merge* è quindi fondamentale. CVS/Subversion scoraggiava i branch, Git li rende facili e li incoraggia. In Git, l'uso dei branch è talmente comune che a volte è necessario introdurre delle politiche (come GitFlow) sul loro utilizzo.

git


Schema git

In figura, possiamo osservare 4 *livelli* ordinati:

- **working directory** (WD): rappresenta la *configurazione* della directory di lavoro sul filesystem - esiste indipendentemente da git. Può essere vista come l'unione dei *tracked* and *untracked* files;
- **index** (o **area di staging**): insieme dei *tracked* files da git.

- (n?) **local repository**: insieme delle modifiche committate e relativo storico.
- (n) **remote repository**: branch remoto; è possibile avere sia più branch per progetto remoto che più progetti remoti configurati.

Il termine ***repository*** è abbastanza *misleading*, perché è comunemente associato ad un progetto mentre in questa astrazione a livelli corrisponde di fatto a un **branch**.

Il passaggio tra un livello e l'altro non è mai automatico, ma è sempre esplicitato da un'operazione.

Operazioni di base

È consigliata la lettura di git Cheatsheet.

Per ogni branch c'è un puntatore all'ultimo commit di tale branch. L'HEAD punta all'ultimo commit in cui siamo: normalmente corrisponde al puntatore del branch corrente; quando non è così siamo in una situazione di *HEAD scollegato*. È utile potersi spostare tra i commit per controllare revisioni precedenti, ma in caso di nuovi commit è importante creare un nuovo branch per poterci riferire ad esse.

`git commit` — *record changes to the repository*

Il comando git commit ci permette di *salvare* del contenuto dall'index al branch locale.

Dopo aver creato il commit, l'`HEAD` e il puntatore al branch corrente puntano al nuovo commit. Anche il contenuto dell'index equivale al contenuto del commit.

`--amend`

Con l'opzione --amend è possibile rimpiazziare facilmente l'ultimo commit con uno nuovo.

`git switch` — *switch branches*

Il comando git switch ha un sottoinsieme delle funzionalità del comando git checkout ed è più semplice da utilizzare.

Permette di passare a un nuovo branch semplicemente modificando l'HEAD (e di conseguenza il contenuto dell'index e dei file).

`git merge` — *join two or more development histories together*

Il comando git merge è utile per unire branch (o più in generale alberi) insieme.

Se i due branch non sono divergenti, il merge avviene in modo banale con un *fast-forward*: nessun ulteriore commit verrà cambiato, verrà solo modificato il puntatore del branch e l'HEAD. Per forzare la creazione di un merge di commit (in gitFlow è apprezzato) occorre utilizzare l'opzione `--no-ff`.

In tutti gli altri casi, il merge può concludersi con successo oppure possono avvenire conflitti. Per risolverli, git ci proporrà un'interfaccia simile alla seguente.

```
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
=======
git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
```

Una volta risolti tutti i conflitti è sufficiente commitare le modifiche concludendo quindi il merge.

`git reset` — *reset current HEAD to the specified state*

Il comando git reset reimposta il contenuto dei file nell'*index* (e, opzionalmente con l'opzione `--hard` nella WD) all'ultimo commit puntato da HEAD o ad un altro commit.

Git workflow e strumenti

In git, l'utilizzo dei branch è fortemente incentivato dal suo design e dalle sue funzionalità, rendendo praticamente impossibile lavorare senza utilizzarli. I branch consentono di creare versioni separate del codice, permettendo di lavorare su diverse funzionalità o correzioni di bug in modo indipendente e senza interferire con il codice principale. C'è **libertà completa** sul loro utilizzo: tutti i branch hanno pari importanza, chiunque può crearli e nominarli in qualunque modo.

Lavorando in un team, è quindi necessario stabilire delle politiche sui tipi e i nomi di branch, in modo da organizzare il lavoro al meglio.

- **GitFlow**: organizzazione branch
- **Hosting centralizzato**: repository centrali git
- **Gerrit**: meccanismo di review in team
- **Strumenti dell'opensource**: strumenti opensource per build automation e bug tracking

GitFlow

GitFlow è una tecnica di organizzazione dei branch e delle repository che prevede la creazione sia di diversi tipi di branch a vita limitata che il loro *merge* guidato.

È disponibile online una cheatsheet che fornisce una panoramica veloce delle principali operazioni e dei comandi di GitFlow. Si tratta di uno strumento utile per chi è alle prime armi con questa tecnica di organizzazione dei branch, ma non esaustivo: per una comprensione più approfondita del metodo, è meglio integrarlo con altre risorse.

I branch e i tipi di branch previsti da GitFlow sono:

- branch master;
- branch develop;
- feature branches;
- release branches;
- hotfix branches.

`develop` e `master`


GitFlow develop e master

In GitFlow, ci sono due branch che hanno una durata più lunga rispetto ai branch temporanei utilizzati per lavorare su specifiche funzionalità o correzioni di bug:

- `master`: contiene le versioni stabili del codice sorgente, pronte per essere consegnate o rilasciate al cliente o al pubblico. Queste versioni sono considerate *affidabili* e *testate*, quindi possono essere utilizzate in produzione;
- `develop`: è il ramo di integrazione in cui vengono integrati i contribuiti di tutti i gruppi; è anche il punto di partenza degli altri tipi di branch.

Al termine di ogni rilascio, il contenuto del branch `develop` viene integrato nel branch `master`, che rappresenta la versione stabile del codice. Le *versioni notturne*, invece, sono versioni di sviluppo che vengono rilasciate periodicamente e contengono le ultime modifiche apportate al codice. Esse vengono create partendo dal branch `develop`, che rappresenta il punto di integrazione di tutti i contributi dei gruppi di lavoro.

Feature

I ***feature branch*** sono branch temporanei utilizzati per sviluppare nuove funzionalità o per correggere bug. **Possono essere creati solo a partire dal branch `develop`** e vengono utilizzati per isolare il lavoro su una specifica funzionalità o problema dal resto del codice. Quando il lavoro è completato, il feature branch viene integrato di nuovo nel `develop` tramite un merge. In questo modo, è possibile lavorare in modo organizzato su diverse funzionalità o problemi senza interferire tra loro. Per integrare il lavoro svolto in un feature branch nel branch `develop`, è necessario eseguire un merge del feature branch nel `develop`. Ci sono diversi modi di fare ciò, a seconda delle preferenze e delle esigenze specifiche. Un modo semplice di fare il merge è utilizzare il comando `git merge` dalla riga di comando. Se il merge non è possibile a causa di conflitti, sarà necessario risolverli manualmente prima di poter completare l'operazione. Una volta risolti i conflitti, sarà necessario creare un nuovo commit per finalizzare il merge.

Per iniziare una feature...

```
$ git checkout develop                  # entra nel branch develop
$ git branch feature/myFeature          # crea un branch di feature
$ git checkout feature/myFeature        # entra nel branch appena creato
```

Al termine della feature, integro:

```
$ git checkout develop                  # entra nel branch develop
$ git merge --no-ff feature/myFeature   # mergia il branch di feature
$ git branch -d feature/myFeature       # elimina il branch di feature
```

`--no-fast-forward`

Di default, git risolve il merge di due branch con la stessa storia banalmente eseguendo il *fast forward*, ovvero spostando il puntatore del branch di destinazione all'ultimo commit del branch entrante.

In GitFlow è preferibile esplicitamente **disabilitare il fast forward** (con l'opzione `--no-ff`) durante il merge in `develop` in quanto è più facile distinguere il punto di inizio e il punto di fine di una feature.

Squashing

Usando git è anche possibile eseguire in fase di merge lo *squashing* dei commit, ovvero la fusione di tutti i commit del branch entrante in uno solo. Questa operazione è utile per migliorare la leggibilità della history dei commit su branch grossi (come `develop` o `master`) ma il suo uso in GitFlow è opinabile: il prof. Bellettini consiglia di non utilizzarlo, in modo da mantenere i commit originali.

Release

Lo scopo di creare una release è **cristalizzare l'insieme delle funzionalità** presente sul branch `develop` all'inizio di essa dedicandosi solo alla sistemazione degli errori o alle attività necessarie per il deploy (modifica del numero di versione, ...). L'insieme delle funzionalità rilasciate è quello presente sul branch `develop` al momento di inizio di una release.

I bug fix possono essere ri-mergiati in `develop`, anche utilizzando la funzionalità **cherry-pick** di git; essa permette di selezionare un commit specifico da un ramo e applicarlo in un altro ramo. Ad esempio, se si ha un ramo di sviluppo ("`develop`") e un ramo di release ("`release`"), è possibile utilizzare il cherry-pick per selezionare solo i commit che contengono bugfix e applicarli al ramo di release, senza dover fare un merge di tutto il ramo di sviluppo. Ciò può

essere utile in casi in cui si vuole mantenere la stabilità del ramo di release, includendo solo i bugfix considerati essenziali per la release.

Per iniziare una nuova release è sufficiente creare un nuovo branch da `develop`:

```
$ git checkout -b release/v1.2.3 develop
```

Al termine della creazione della release, è necessario fare il merge della release nel branch `master` e nel branch `develop`. Il merge in `master` rappresenta il rilascio della nuova versione del codice, che diventa disponibile per il pubblico o per il cliente. Il merge in `develop`, invece, integra le modifiche apportate durante la creazione della release nel branch di sviluppo, in modo che siano disponibili per le release future. In questo modo, è possibile gestire in modo organizzato il ciclo di vita del codice e il flusso di lavoro.

```
$ git checkout master               # entra nel branch master
$ git merge --no-ff release/v1.2.3  # mergia la feature
$ git tag -a v1.2.3                 # tagga sul branch master il rilascio
$ git checkout develop              # entra nel branch develop
$ git merge --no-ff release/v1.2.3  # mergia la feature
$ git branch -d release/v1.2.3      # elimina il branch della feature
```

In git, i tag sono etichette che possono essere applicate a un commit per segnalarne l'importanza o per marcare un punto specifico dello storico del repository. Un **tag è un puntatore costante al commit** a cui è stato applicato, quindi non cambia mai e permette di fare riferimento in modo stabile a una versione specifica del codice. Al contrario, i branch sono puntatori dinamici che vanno avanti nel tempo, seguendo l'evoluzione del codice attraverso i nuovi commit

In GitFlow, le release sono versioni stabili del codice che vengono rilasciate al pubblico o al cliente. Ogni release viene creata partendo dal branch `develop` e viene gestita come un branch a sé stante, che viene chiuso una volta che tutte le modifiche previste sono state integrate. Al contrario, le feature sono branch temporanei utilizzati per sviluppare nuove funzionalità o per correggere bug. È possibile avere più feature aperte contemporaneamente, ma solo una release rimanere aperta in un dato istante.

Hotfix

Un *hotfix* è una riparazione veloce di difetti urgenti senza dover aspettare la prossima release. È l'unico caso per cui non si parte da `develop`, ma dall'ultima - o una particolare - versione rilasciata su `master`.

```
$ git checkout -b hotfix/CVE-123 master # crea un branch di hotfix
```

Quando lo chiudo:

```
$ git checkout master                   # entra nel branch master
$ git merge --no-ff hotfix/CVE-123      # mergia l'hotfix
$ git tag -a hotfix/CVE-123             # tagga sul branch master il rilascio
$ git checkout develop                  # entra nel branch develop
$ git merge --no-ff hotfix/CVE-123      # mergia l'hotfix
$ git branch -d hotfix/CVE-123          # elimina il branch di hotfix
```

Limiti

Quali sono i limiti di git presentato così?

git e GitFlow come sono stati esposti presentano numerosi vincoli, tra cui:

- la **mancanza di un sistema di autorizzazione granulare**, ovvero la possibilità di assegnare permessi in modo specifico e mirato a diverse funzionalità o risorse. Inoltre, non esiste una distinzione tra diversi livelli di accesso, quindi o si ha accesso completo a tutte le funzionalità o non si ha accesso a niente;
- l'**assenza di code review**, ovvero il processo di revisione del codice sorgente da parte di altri sviluppatori prima che venga integrato nel codice base.

`git request-pull` — *generates a summary of pending changes*

Il tool `git request-pull` è un comando di git che serve per formattare e inviare una proposta di modifiche a un repository tramite una mailing list. Il comando crea un messaggio di posta elettronica che chiede al maintainer del repository di "pullare" le modifiche, ovvero di integrarle nel codice base. Oggi, questa pratica è stata in molti progetti sostituita dalle pull request, che sono richieste di integrazione delle modifiche presentate attraverso un'interfaccia web. Le pull request offrono una serie di vantaggi rispetto alle richieste via email, come una maggiore trasparenza del processo di integrazione, una maggiore efficienza e una maggiore facilità di utilizzo.

La sintassi del comando è la seguente:

```
git request-pull [-p] <start> <URL> [<end>]
```

Per esempio, i contributori Linux usano questo strumento per chiedere a Linus Torvalds di unire le modifiche nella sua versione. L'output generato mostra file per file le modifiche fatte e i commenti dei commit creati, raggruppati per autore.

```
$ git request-pull master git@gitlab.di.unimi.it:silab-gang/sweng.git lezioni/09
The following changes since commit a16f3a0488c062d7b61dc4af15c2ba8081166040:

  Handle '/sweng' path (2022-11-25 19:14:40 +0100)

are available in the git repository at:

  git@gitlab.di.unimi.it:silab-gang/sweng.git lezioni/09

for you to fetch changes up to 4ac534bcd31c8c0353070c3f42eea09737b497b5:

  Refactor notes on Factory method pattern (2022-12-19 09:58:32 +0100)

----------------------------------------------------------------
Daniele Ceribelli (6):
      Add pattern Adapter
      Add notes until decorator
      Finish notes
      Add notes until factory method pattern
      Add notes until Abstract Factory pattern
      Finish notes with all patterns

Marco Aceti (14):
      Add lesson notes
      Add flyweight pattern
      Add observer pattern
      Add more patterns
      Fix typo
      Merge branch 'master' into lezioni/09
      Replace hook UMLs images with PlantUML
      Merge branch 'master' into lezioni/09
      Fix gitLab CI (take 2)
      Refactor diagrams until 'Strategy'
      Refactor all remaining diagrams
      Merge remote-tracking branch 'gitlab/master' into lezioni/09
      Merge remote-tracking branch 'gitlab/master' into lezioni/09
      Make 'Decorator' code samples *readable*

Matteo Mangioni (18):
      Add introduction
      Integrate Singleton notes
      Add images for first patterns
      Integreate Iterator notes
      Integrate Chain of Responsibility notes
      Integrates FlyWeight notes
      Add NullObject pattern
      Merge branch 'lezioni/09' of gitlab.com:silab-gang/sweng into lezioni/09
      Merge branch 'mangio/patterns-appunti' into lezioni/09
      Integrate Strategy section
      Integrate Observer notes
      Refactor notes on pattern Adapter
      Refactor notes on pattern Facade
      Make already reviewed notes follow new style guide
```

```
     Refactor notes on pattern Composite
     Refactor notes on pattern Decorator
     Refactor notes on pattern State
     Refactor notes on Factory method pattern

 _posts/2022-10-26-Patterns.md               | 1301 ++++++++++++++++++++++++++
 assets/09_esempio-abstract-factory.png      |  Bin 0 -> 2362937 bytes
 assets/09_facade.png                        |  Bin 0 -> 73981 bytes
 assets/09_model-view-controller.png         |  Bin 0 -> 2751267 bytes
 assets/09_model-view-presenter.png          |  Bin 0 -> 2118448 bytes
 assets/09_nullObject-valori-non-assenti.png |  Bin 0 -> 67792 bytes
 6 files changed, 1301 insertions(+)
 create mode 100644 _posts/2022-10-26-Patterns.md
 create mode 100644 assets/09_esempio-abstract-factory.png
 create mode 100644 assets/09_facade.png
 create mode 100644 assets/09_model-view-controller.png
 create mode 100644 assets/09_model-view-presenter.png
 create mode 100644 assets/09_nullObject-valori-non-assenti.png
```

Questo modello è molto più *peer to peer* delle pull request proposte dai sistemi semi-centralizzati spiegati in seguito.

Hosting centralizzato

Un hosting centralizzato Git è un servizio che fornisce una repository centrale per i progetti Git dove i contributi vengono integrati e gestiti, garantendo una maggiore trasparenza e controllo del processo di sviluppo e mantenendo molti vantaggi della decentralizzazione, come la possibilità di lavorare in modo asincrono e autonomo.

Gli hosting centralizzati come GitHub e GitLab, nella loro costante evoluzione, spesso inventano nuovi meccanismi e provano a imporre nuovi workflow, come il GitHub Flow o il GitLab Flow, per semplificare e ottimizzare il processo di sviluppo. Tuttavia, è importante valutare attentamente questi nuovi approcci e verificare se si adattano alle esigenze specifiche del progetto e della squadra di sviluppo. Inoltre, molti servizi di hosting centralizzati offrono funzionalità aggiuntive, come la possibilità di eseguire il “fork” di un repository, inviare *pull request* per le modifiche e di utilizzare strumenti di continuous integration (CI) per testare automaticamente le modifiche apportate al codice.

Fork

Il “fork” di un repository Git è una **copia del repository originale** che viene creata su un account di hosting diverso dal proprietario originale. Questo permette a un altro sviluppatore di creare una copia del repository e di lavorare su di essa senza influire sul lavoro del

proprietario originale e **senza la sua autorizzazione**. È possibile quindi mantenere una *connessione* tra i due repository e condividere facilmente le modifiche apportate.

La maggioranza delle piattaforme di hosting centralizzato **ottimizza la condivisione dello spazio degli oggetti**, utilizzando un'unica *repository fisica* per tutti i fork. Tuttavia, questo può comportare alcune problematiche di sicurezza, come ad esempio la difficoltà per la piattaforma di stabilire in quale fork si trova un determinato oggetto in caso di conflitto o la possibilità che un utente malintenzionato possa modificare o eliminare accidentalmente oggetti di altri fork. Per questo motivo, è importante che le piattaforme implementino **misure di sicurezza adeguate** per proteggere i dati dei fork e garantire la tracciabilità delle modifiche (esempio sul kernel Linux).

Review / Pull request

Tra la creazione di una pull request e il suo *merge*, specialmente nei progetti open source (dove chiunque può proporre qualsiasi patch) è fondamentale prevedere un processo di **review**.

La funzionalità di *review/pull request* permette di facilitare le interazioni tra gli sviluppatori utilizzando il sito di hosting come luogo comune per la discussione informale e la revisione delle modifiche.

Continous integration (CI)

Come accennato in precedenza, molti servizi di hosting centralizzati offrono strumenti di **continuous integration** (CI) che possono essere utilizzati per testare automaticamente le modifiche proposte nella pull request. Questi strumenti consentono di verificare che le modifiche non introducano errori o vulnerabilità e di garantire che il codice sia pronto per essere integrato nel repository principale. Possono essere utilizzati anche per eseguire automaticamente la *suite di test* o automatizzare il deployment.

Gerrit

Gerrit è un **sistema di review** del codice sviluppato internamente da Google per gestire i progetti interni; si basa sul concetto di "peer review": tutti gli sviluppatori sono autorizzati a fare review delle proposte di modifica di qualsiasi zona di codice.

Nel processo di review di Gerrit, i **developer** possono sottoporre proposte di cambiamento utilizzando un sistema di "patch" che descrive le modifiche apportate al codice. I **reviewer**, ovvero gli altri sviluppatori del progetto, possono quindi esaminare le patch e decidere se accettarle o rifiutarle. Una volta che una patch ha ricevuto un numero sufficiente di review positivi, viene automaticamente integrata nel **repository principale autoritativo** in cui tutti hanno accesso in sola lettura.

Gerrit obbliga a strutturare le modifiche (*changeset*) in un unico commit (tecnica *squash*) al momento dell'accettazione. Ciò significa che tutte le modifiche apportate devono essere fuse in un unico commit, in modo da rendere più facile la gestione del repository. Al momento della review, invece, le modifiche rimangono separate in versioni singole, ovvero ogni modifica viene presentata come un commit separato, in modo che i reviewer possano esaminarle più facilmente.

Verifier

Il verifier è uno strumento o un processo che viene utilizzato in Gerrit per verificare che le modifiche proposte siano corrette e funzionino come dovrebbero. In particolare, il verifier scarica la patch, la compila, esegue i test e controlla che ci siano tutte le funzioni necessarie. Se il verifier rileva dei problemi, può segnalarli al team di sviluppo perché vengano corretti prima che la patch venga accettata.

Una volta terminato il proprio processo, approva le modifiche votandole positivamente. Solitamente sono necessari 1 o 2 voti per procedere.

Approver

Una volta verificata, una proposta di modifiche deve essere anche approvata. L'approvatore deve determinare la risposta alle seguenti domande riguardo la proposta di modifiche:

- *è valida per lo scopo del progetto?*
- *è valida per l'architettura del progetto?*
- *introduce nuove falle nel design che potrebbero causare problemi in futuro?*
- *segue le best practices stabilite dal progetto?*
- *è un buon modo per implementare la propria funzione?*
- *introduce rischi per la sicurezza o la stabilità?*

Se l'approver ritiene che la proposta di modifiche sia valida, può approvarla scrivendo "LGTM" (acronimo di *"Looks Good To Me"*) nei commenti della pull request.

Strumenti dell'open source

Gli strumenti dell'open source sono una serie di programmi, librerie e servizi che vengono utilizzati per sviluppare progetti open source. Questi strumenti sono pensati per semplificare il processo di sviluppo e gestione di progetti open source, rendendoli accessibili a una comunità di sviluppatori e contribuenti.

- **Build automation**: `make`, Ant e Gradle
- **Bug tracking**: tecniche di bug workflow

Build automation

La build automation è un processo fondamentale nello sviluppo di software open source, che consiste nel creare un sistema automatizzato per compilare il codice sorgente in un eseguibile. Questo processo è importante perché consente di risparmiare tempo e risorse, evitando di dover compilare manualmente il codice ogni volta che si apportano modifiche. Inoltre, la build automation garantisce una maggiore qualità e coerenza del software, poiché il processo di compilazione viene eseguito in modo uniforme ogni volta.

`make`

`make` è uno strumento di build automation che viene utilizzato per automatizzare il processo di compilazione di un progetto. In particolare, `make` viene utilizzato per specificare come ottenere determinati *targets* (obiettivi), ovvero file o azioni che devono essere eseguite, partendo dal codice sorgente. Ad esempio, in un progetto di sviluppo software, un *target* potrebbe essere il file eseguibile del programma, che viene ottenuto compilando il codice sorgente. `make` segue la filosofia *pipeline*, ovvero prevede l'utilizzo di singoli comandi semplici concatenati per svolgere compiti più complessi.

È supportata la *compilazione incrementale*, ovvero il fatto di compilare solo le parti del progetto che sono state modificate dall'ultima volta, al fine di velocizzare il processo. Inoltre, vengono gestite le *dipendenze* tra file, ovvero le relazioni tra i diversi file che compongono il progetto: se un file sorgente dipende da un altro file, make assicura che il file dipendente venga compilato solo dopo che il file da cui dipende è stato compilato. Ciò garantisce che il progetto venga compilato in modo coerente e che le modifiche apportate a un file siano considerate correttamente nella compilazione dei file dipendenti.

```
CC=gcc
CFLAGS=-I.

%.o: %.c $(DEPS)
  $(CC) -c -o $@ $< $(CFLAGS)

hellomake: hellomake.c hellofunc.o
  $(CC) -o hellomake hellomake.o hellofunc.o $< $(CFLAGS)
```

Nell'esempio, se il *target* hellomake (definito dai file `hellomake.c` e `hellofunc.o`) è stato aggiornato, occorre ricompilarlo utilizzando i comandi sotto.

Tuttavia, make lavora a un livello molto basso, il che può rendere facile commettere errori durante la sua configurazione e utilizzo.

Non c'è portabilità tra macchine (ambienti) diverse.

`Makefile`

Un *Makefile* è un file di testo che contiene le istruzioni per il programma make su come compilare e linkare i file sorgente di un progetto. Ogni riga del Makefile definisce un obiettivo o una dipendenza, insieme ai comandi che devono essere eseguiti per raggiungerlo. L'utilizzo del Makefile permette di automatizzare la compilazione e il linkaggio dei file sorgente, semplificando il processo di sviluppo di un progetto. Nell'esempio menzionato, il Makefile definisce il target `hellomake`, che dipende dai file `hellomake.c` e `hellofunc.o`, e fornisce i comandi per compilarli e linkarli insieme.

Generazione automatica

Sono stati creati dei tool (`automake`, `autoconf`, `imake`, ...) che *generano* `Makefile` ad-hoc per l'ambiente attuale.

Il *mantra*:

```
$ ./configure
$ make all
$ sudo make install
```

era largamente utilizzato per generare un Makefile ad-hoc per l'ambiente attuale e installare il software sulla macchina in modo automatico. `automake`, `autoconf`, e `imake` sono strumenti che aiutano a questo scopo, generando Makefile che possono essere utilizzati per compilare e installare il software in modo automatico.

Ant

Ant nasce in Apache per supportare il progetto Tomcat. Data una **definizione in XML** della struttura del progetto e delle dipendenze invocava comandi programmati tramite classi Java per compilare il progetto.

Il vantaggio è che Java offre un livello d'astrazione sufficiente a rendere il sistema di build portabile su tutte le piattaforme.

Nella versione base supporta integrazioni con altri tool come CVS, Junit, FTP, JavaDOCS, JAR, ecc... Non solo compila, ma fa anche deployment. Il deployment consiste nell'installare e configurare un'applicazione o un sistema su uno o più server o ambienti di esecuzione. Nel contesto di Ant, il deployment può includere l'invocazione di comandi per copiare i file del progetto sui server di destinazione, configurare le impostazioni di sistema o dell'applicazione, avviare o fermare servizi o processi, e così via. In questo modo, Ant può essere utilizzato non solo per compilare il progetto, ma anche per distribuirlo e rendere disponibile l'applicazione o il sistema ai suoi utenti.

I target possono avere dipendenze da altri target. I target contengono task che fanno effettivamente il lavoro; si possono aggiungere nuovi tipi di task definendo nuove classi Java.

Esempio di un build file:

```
<?xml version="1.0"?>
<project name="Hello" default="compile">
  <target name="clean" description="remove intermediate files">
    <delete dir="classes" />
  </target>
  <target name="clobber" depends="clean" description="remove all artifact files">
    <delete file="hello.jar">
  </target>
  <target name="compile" description="compile the Java source code to class
files">
    <mkdir dir="classes" />
    <javac srcdir="." destdir="classes" />
  </target>
  <target name="jar" depends="compile" description="create a Jar file for the
application">
    <jar destfile="hello.jar">
      <fileset dir="classes" includes="**/*.class" />
      <manifest>
        <attribute name="Main-Class" value="HelloProgram" />
      </manifest>
    </jar>
  </target>
</project>
```

Gradle

Gradle è uno strumento di build automation che utilizza le repository Maven come punto di accesso alle librerie di terze parti. Maven è una piattaforma di gestione delle dipendenze e della build automation per il linguaggio di programmazione Java. Le repository Maven sono archivi online che contengono librerie Java, plugin e altri componenti utilizzati nella build di progetti Java. Gradle utilizza queste repository per cercare e scaricare le librerie di cui ha bisogno per eseguire la build del progetto.

Gradle, che supporta Groovy o Kotlin come linguaggi di scripting, adotta un approccio dichiarativo e fortemente basato su convenzioni. Ciò significa che tutto ciò che è già stato definito come standard non deve essere ridichiarato. Inoltre, Gradle definisce un linguaggio specifico per la gestione delle dipendenze e permette di creare build multi-progetto.

Gradle scala bene in complessità: permette di fare cose semplici senza usare le funzioni complesse. È estendibile tramite plugin che servono per trattare tool, situazioni, linguaggi legati solitamente al mondo Java.

Plugin

I plugin servono per trattare tool, situazioni, linguaggi definendo task e regole per lavorare più facilmente.

Il plugin *Java* definisce:

- una serie di **sourceSet**, ovvero dove è presente il codice e le risorse. Principalmente sono:
  - `src/main/java`: sorgenti Java di produzione;
  - `src/main/resources`: risorse di produzione;
  - `src/test/java`: sorgenti Java di test;
  - `src/test/resources`: risorse di test.
- dei **task**, anche con dipendenze tra loro.


Task gradle

Altri plugin

- application, per l'esecuzione;
- FindBugs, jacoco: per la verifica e la convalida;
- eclipse, idea: per integrazione con gli IDE;

Bug tracking

Il bug tracking è stato reso necessario nel mondo open source per via della numerosità dei contributi e della alta probabilità di avere segnalazioni duplicate.

Inoltre, per gestire le segnalazioni di bug nell'ambito dello sviluppo open source, esistono diversi strumenti come git-bug, BugZilla, Scarab, GNATS, BugManager e Mantis.

Bug workflow

L'obiettivo del bug tracking è avere più informazioni possibili su ogni bug per saperli riprodurre e quindi arrivare a una soluzione.

È importante verificare i bug una volta che l'*issue* è stato aperto, in modo da poter confermare la sua esistenza e la completezza delle informazioni fornite.

Un *issue* è un problema o una richiesta di funzionalità segnalata all'interno di un progetto di software. Gli issue vengono solitamente utilizzati per tenere traccia dei problemi noti o delle richieste di nuove funzionalità all'interno di un progetto, e possono essere gestiti attraverso un sistema di bug tracking o gestione delle richieste. Gli issue possono essere aperti da qualsiasi membro del team o dalla comunità, e possono essere risolti o chiusi da un membro del team responsabile.

Ci sono diversi modi per cui può essere chiuso un bug:

- **duplicate**: quando è stato già segnalato in precedenza e quindi non rappresenta un problema nuovo. In questo caso, viene solitamente fatto riferimento al numero del bug originale che ha già ricevuto una risoluzione;
- **wontfix**: il bug viene chiuso come "non risolvibile" perché o rappresenta una funzionalità voluta dal progetto o è troppo complesso da risolvere per essere considerato conveniente farlo dal punto di vista dei progettisti;
- **can't reproduce**: non è stato possibile riprodurre il bug, ovvero che non è stato possibile ottenere lo stesso risultato o il comportamento segnalato dal bug. Ciò può essere dovuto a una mancanza di dettagli o a un errore nella segnalazione del bug stesso;
- **fixed**: il bug è stato fixato; vs **fix verified**: il fix è stato integrato in una release passando tutti gli step di verifica.

Progettazione

Durante le lezioni, per discutere di progettazione siamo partiti da un esempio di programma in C che stampa una canzone. Il codice considerato è completamente illegibile:

```
#include <stdio.h>
main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);
}
```

Successivamente abbiamo scomposto il codice per renderlo logicamente più sensato e facilmente modificabile, sono state **estratte le parti comuni** e spostate in una funzione apposita, mentre le **parti mutabili sono state salvate in alcune strutture dati**; la canzone viene così stampata tramite un ciclo. In questo modo scrivendo un codice più semplice siamo stati in grado di creare una soluzione più generale e più aperta ai cambiamenti.

```
public class TwelveDaysOfChristmas {
    static String[] days = {"first", "second", ..., "twelfth"};
    static String[] gifts = { "a partdrige in a pear tree", "two turtle doves",
... };

    static String firstLine(int day) {
        return "On the " + days[day] +
               " day of Christmas my true love gave to me:\n";
    }

    static String allGifts(int day) {
        if (day == 0) {
            return "and " + gifts[0];
        } else {
            return gifts[day] + "\n" + allGifts(day-1);
        }
    }

    public static void main(String[] args) {
        System.out.println(firstLine(0));
        System.out.println(gifts[0]);
        for (int day == 1; day < 12; day++) {
            System.out.println(firstLine(day));
            System.out.println(allGifts(day));
        }
    }
}
```

È importante quindi **adottare la soluzione più semplice** (che **non è quella più stupida**!) e una misura convenzionale per dire quanto una cosa è semplice - almeno in Università - si esprime in termini del tempo dedicato dal programmatore all'implementazione. Tale misura si sposa bene con il **TDD**, che richiede **brevi iterazioni** di circa 10 minuti: se la feature attuale richiede più tempo è opportuno ridurre la portata scomponendo il problema.

- **Refactoring**: modifiche al codice senza nuove funzionalità;
- **Design knowledge**: dove mantenere la conoscenza del design?
- **Conoscenze preliminari**: object orientation, SOLID principles, reference escaping, encapsulation e information hiding, immutabilità, code smell
- **Principio Tell-Don't-Ask**
- **Interface segregation**
- **Esempio** applicando i principi precedenti
- **Analisi del testo naturale** con *noun extraction*

Refactoring

Durante il refactoring è opportuno rispettare le seguenti regole:

- le **modifiche al codice non devono modificare le funzionalità**: il refactoring DEVE essere invisibile al cliente;
- **non possono essere aggiunti test aggiuntivi** rispetto alla fase verde appena raggiunta.

Se la fase di refactoring sta richiedendo troppo tempo allora è possibile fare rollback all'ultima versione verde e **pianificare meglio** l'attività di refactoring, per esempio scomponendolo in più step. Vale la regola del *"do it twice"*: il secondo approccio a un problema è solitamente più veloce e migliore.

Motivazioni

Spesso le motivazioni dietro un refactoring sono:

- precedente **design molto complesso e poco leggibile**, a causa della velocità del passare ad uno *scenario verde*;
- **preparare il design di una funzionalità** che non si integra bene in quello esistente; dopo aver raggiunto uno *scenario verde* in una feature, è possibile che la feature successiva sia difficile da integrare. In questo caso, se il *refactoring* non è banale è bene fermarsi, tornare indietro e evolvere il codice per facilitare l'iterazione successiva (**design for change**).
- presenza di **debito tecnico** su lavoro fatto in precendenza, ovvero debolezze e "scorciatoie" che ostacolano notevolmente evoluzioni future: *"ogni debito tecnico lo si ripaga con gli interessi"*.

Design knowledge

La design knowledge è la **conoscenza del design** architetturale di un progetto. È possibile utilizzare:

- la **memoria**: non è efficace perché nel tempo si erode, specialmente in coppia;
- i **documenti di design** (linguaggio naturale o diagrammi): se non viene aggiornato di pari passo con il codice rimane disallineato, risultando più dannoso che d'aiuto.
- le **piattaforme di discussione** (version control, issue management): possono aiutare ma le informazioni sono sparse in luoghi diversi e di conseguenza difficili da reperire e rimane il problema di mantenere aggiornate queste informazioni.
- gli **UML**: tramite diagrammi UML si è cercato di sfruttare l'approccio ***generative programming***, ovvero la generazione automatica del codice a partire da specificazioni di diagrammi. Con l'esperienza si è visto che non funziona.
- il **codice** stesso: tramite la lettura del codice è possibile capire il design ma è difficile rappresentare le ragioni della scelta.

È bene sfruttare tutte le tecniche sopra proposte **combinandole**, partendo dal codice.
È inoltre importante scrivere documentazione per spiegare le ragioni dietro le scelte effettuate e non le scelte in sé, che si possono dedurre dal codice.

Condivisione

Per condividere tali scelte di design (il *know how*) è possibile sfruttare:

- **metodi**: con pratiche (come Agile) o addirittura l'object orientation stessa, che può essere un metodo astratto per condividere scelte di design.
- **design pattern**: fondamentali per condividere scelte di design, sono utili anche per generare un vocabolario comune (sfruttiamo dei nomi riconosciuti da tutti per descrivere i ruoli dei componenti) e aiutano l'implementazione (i pattern hanno delle metodologie note per essere implementati). I pattern non si concentrano sulle prestazioni di un particolare sistema ma sulla generalità e la riusabilità di soluzioni a problemi comuni;
- **principi**: per esempio i principi SOLID.

Conoscenze preliminari

Prima di proseguire è bene richiamare concetti e termini fondamentali presumibilmente visti durante il corso di Programmazione II.

Object orientation

Per essere definito *object oriented*, un linguaggio di programmazione deve soddisfare tre proprietà:

- **ereditarietà**: ovvero la possibilità di poter definire una classe ereditando proprietà e comportamenti di un'altra classe.
- **polimorfismo**: quando una classe può assumere diverse forme in base alle interfacce che implementa. Il prof fa l'esempio del *tennista scacchista*: in un torneo di tennis è poco utile sostituire una persona che gioca a tennis ed è brava con gli scacchi (quindi una classe che implementa entrambe le interfacce) con una che gioca a scacchi. Il collegamento tra capacità e oggetto è fatto **a tempo di compilazione**: non è importante quindi se la capacità non è ancora definita;
- **collegamento dinamico**: in Java il tipo concreto degli oggetti e quindi il problema di stabilire *quale metodo chiamare* viene risolto durante l'esecuzione. In C++ occorre esplicitare questo comportamento utilizzando la keyword `virtual`.

SOLID principles

Ci sono 5 parti che compongono questo principio:

1. **SINGLE RESPONSIBILITY**: una classe, un solo scopo. Così facendo, le classi rimangono semplici e si agevola la riusabilità.
2. **OPEN-CLOSE PRINCIPLE**: le classi devono essere aperte ai cambiamenti (*opened*) ma senza modificare le parti già consegnate e in produzione (*closed*). Il refactoring è comunque possibile, ma deve essere preferibile estendere la classe attuale.
3. **LISKOV SUBSTITUTION PRINCIPLE**: c'è la garanzia che le caratteristiche ereditate dalla classe padre continuinino ad esistere nelle classi figlie. Questo concetto si collega all'aspetto **contract-based** del metodo Agile: le *precondizioni* di un metodo di una classe figlia devono essere ugualmente o meno restrittive del metodo della classe padre. Al contrario, le *postcondizioni* di un metodo della classe figlia non possono garantire più di quello che garantiva il metodo nella classe padre. Fare *casting* bypassa queste regole.
4. **INTERFACE SEGREGATION**: più le capacità e competenze di una classe sono frammentate in tante interfacce più è facile utilizzarla in contesti differenti. In questo modo un client non dipende da metodi che non usa. Meglio quindi avere **tante interfacce specifiche** e piccole (composte da pochi metodi), piuttosto che poche, grandi e generali.
5. **DEPENDENCY INVERSION**: il codice dal quale una classe dipende non deve essere più **concreto** di tale classe. Per esempio, se il *telaio della FIAT 500* dipende da uno specifico motore, è possibile utilizzarlo solo per quel specifico motore. Se invece il telaio dipende da *un* concetto di motore, non c'è questa limitazione. In conlusione, le classi concrete devono tendenzialmente dipendere da classi astratte e non da altre classi concrete.

Reference escaping

Il *reference escaping* è una violazione dell'incapsulamento.

Può capitare, per esempio:

- quando un getter ritorna un riferimento a un segreto;

```
public Deck {
    private List<Card> cards;

    public List<Card> getCards() {
        return this.cards;
    }
}
```

- quando un setter assegna a un segreto un riferimento che gli viene passato;

```
public Deck {
    private List<Card> cards;

    public setCards(List<Card> cards) {
        this.cards = cards;
    }
}
```

- quando il costruttore assegna al segreto un riferimento che gli viene passato;

```
public Deck {
    private List<Card> cards;

    public Deck(List<Card> cards) {
        this.cards = cards;
    }
}
```

Encapsulation e information hiding

Legge di Parnas (L8).

> *Solo ciò che è nascosto può essere cambiato liberamente e senza pericoli.*

Lo stato mostrato all'esterno non può essere modificato, mentre quello nascosto sì.

Questo principio serve per **facilitare la comprensione del codice** e renderne più facile la modifica parziale senza fare danni.

Immutabilità

Una classe è immutabile quando non c'è modo di modificare lo stato di ogni suo oggetto dopo la creazione.

Per assicurare tale proprietà è necessario:

- **non fornire metodi di modifica** allo stato;
- avere tutti gli **attributi privati** per i tipi potenzialmente mutabili (come `List<T>`);

- avere tutti gli **attributi final** se non già privati;
- assicurare l'**accesso esclusivo** a tutte le parti non mutabili, ovvero non avere reference escaping.

Code smell

I *code smell* sono dei segnali che suggeriscono problemi nella progettazione del codice. Di seguito ne sono elencati alcuni:

- **codice duplicato**: si può fare per arrivare velocemente al verde ma è da togliere con il refactoring. Le parti di codice in comune possono quindi essere fattorizzate.
- **metodi troppo lunghi**: sono poco leggibili e poco riusabili;
- **troppi livelli di indentazione**: scarsa leggibilità e riusabilità, è bene fattorizzare il codice;
- **troppi attributi**: suggerisce che la classe non rispetta la single responsability, ovvero fa troppe cose;
- **lunghe sequenze di *if-else* o *switch***;
- **classe troppo grande**;
- **lista parametri troppo lunga**;
- **numeri *magici***: è importante assegnare alle costanti numeriche all'interno del codice un nome per comprendere meglio il loro scopo;
- **commenti che spiegano cosa fa il codice**: indicazione che il codice non è abbastanza chiaro;
- **nomi oscuri o inconsistenti**;
- **codice morto**: nel programma non deve essere presente del codice irraggiungibile o commentato. Utilizzando strumenti di versioning è possibile riaccedere a codice precedentemente scritto con facilità.
- **getter e setter**: vedi principio di **tell don't ask**.

Principio Tell-Don't-Ask

> Non chiedere i dati, ma dì cosa vuoi che si faccia sui dati

---

Il responsabile di un'informazione è anche responsabile di tutte le operazioni su quell'informazione.

Il principio *Tell-Don't-Ask* sancisce che piuttosto di **chiedere** ad un oggetto dei dati e fare delle operazioni con quei dati è meglio **dire** a questo oggetto cosa fare con i dati che contiene.

Esempio

Se desideriamo stampare il contenuto di tutte le carte in un mazzo possiamo partire da questo codice.

```
class Main {
    public static void main(String[] args) {
        Deck deck = new Deck();
        Card card = new Card();

        card.setSuit(Suit.DIAMONDS);
        card.setRank(Rank.THREE);
        deck.getCards().add(card);
        deck.getCards().add(new Card());    // <-- !!!

        System.out.println("There are " + deck.getCards().size() + " cards:");
        for (Card currentCard : deck.getCards()) {
            System.out.println(
                currentCard.getRank() +
                " of " +
                currentCard.getSuit()
            );
        }
    }
}
```

All'interno del ciclo reperiamo gli attributi della carta e li utilizziamo per stampare le sue informazioni. Inoltre, nella riga evidenziata viene aggiunta una carta senza settare i suoi attributi. La responsabilità della gestione dell'informazione della carta è quindi **erroneamente delegata** alla classe chiamante.

Per risolvere, è possibile trasformare la classe `Card` nel seguente modo:

```
class Card {
    private Suit suit;
    private Rank rank;

    public Card(@NotNull Suit s, @NotNull Rank r) {
        suit = s;
        rank = r;
    }

    @Override
    public String toString() {
        return rank + " of " + suit;
    }
}
```

l'informazione viene ora interamente gestita dalla classe `Card`, che la gestisce nel metodo `toString()` per ritornare la sua rappresentazione testuale.

Interface segregation

Le interfacce possono *"nascere"* tramite due approcci:

- **up front**: scrivere direttamente l'interfaccia;
- **down front**: scrivere il codice e quindi tentare di estrarne un'interfaccia.

L'approccio down-front si adatta meglio al TDD ed è illustrato nel seguente esempio.

Esempio con gerarchia Card / Deck

In questo esempio sono trattati numerosi principi, come l'*interface segreagation*, *linking dinamico/statico*, *implementazione di interfacce multiple* e il *contract based design* vs la *programmazione difensiva*.

- **Interface segregation**
- **Collegamento statico e dinamico**
- **Loose coupling**
- **Interfacce multiple**
- ***Contract-based* design vs programmazione difensiva**
- **Classi astratte**

Interface segregation all'opera

```
public static List<Card> drawCards(Deck deck, int number) {
    List<Card> result = new ArrayList<>();
    for (int i = 0; i < number && !deck.isEmpty(); i++) {
        result.add(deck.draw());
    }
    return result;
}
```

Consideriamo il metodo `drawCards` che prende come parametri un `Deck` e un intero.
Le **uniche competenze** riconosciute a `Deck` sono l'indicazione *se è vuoto* (`isEmpty()`) e il *pescare una carta* dal mazzo (`draw()`). `Deck` può quindi **implementare un'interfaccia** che mette a disposizione queste capacità.

È possibile modificare il metodo in modo da accettare un qualunque oggetto in grado di eseguire le operazioni sopra elencate, ovvero che implementi l'interfaccia `CardSource`.

```java
public interface CardSource {
    /**
     * @return The next available card.
     * @pre !isEmpty()
     */
    Card draw();

    /**
     * @return True if there is no card in the source
     */
    boolean isEmpty();
}
```

```java
public class Deck implements CardSource { ... }
```

```java
public static List<Card> drawCards(CardSource deck, int number) {
    List<Card> result = new ArrayList<>();
    for (int i = 0; i < number && !deck.isEmpty(); i++) {
        result.add(deck.draw());
    }
    return result;
}
```

Collegamento statico e dinamico

Notare come è necessario specificare **staticamente** che `Deck` implementi `CardSource`, ovvero occorre forzare la dichiarazione del fatto che `Deck` sia un *sottotipo* di `CardSource` (Java è strong typed) e quindi sia possibile mettere un oggetto `Deck` ovunque sia richiesto un oggetto `CardSource`.
In altri linguaggi come **Go** c'è una **maggiore dinamicità** perché non c'è bisogno di specificare nel codice che un oggetto è sottotipo di qualcos'altro, è sufficiente solo che implementi un metodo con la stessa signature. Il controllo che l'oggetto passato ad una funzione abbia le capacità necessarie avviene a runtime e non prima.

Un problema della troppa dinamicità (**duck typing**) è che se i metodi di un oggetto non hanno dei nomi abbastanza specifici si possono avere dei problemi. Per esempio, in un programma per il gioco del tennis se una funzione richiede un oggetto che implementa il metodo `play()`, e riceve in input un oggetto che non c'entra nulla con il tennis (per esempio un oggetto di tipo `GiocatoreDiScacchi`) che ha il metodo `play()`, si possono avere degli effetti indesiderati.

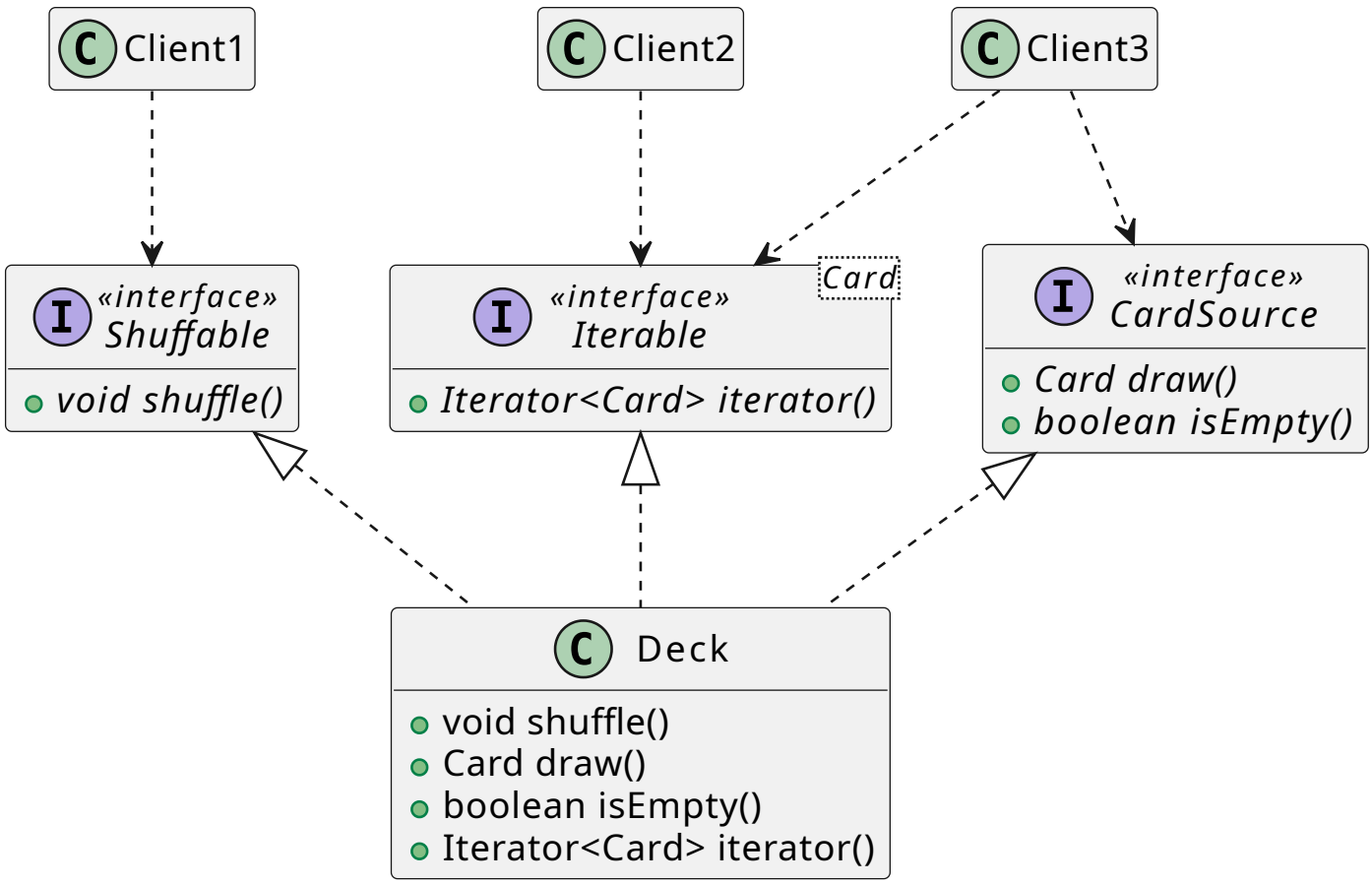
Loose coupling

Il *loose coupling* è la capacità di una variabile o un parametro di accettare l'assegnamento di oggetti aventi tipo diverso da quello della variabile o parametro, a patto che sia un sottotipo.

```
Deck deck = new Deck();
CardSource source = deck;
List<Card> cards = drawCards(deck, 5);
```

Interfacce multiple

Tornando all'esempio, la classe `Deck` (che implementa `CardSource`) **può implementare anche altre interfacce**, come `Shuffable` o `Iterable<Card>`. Al metodo precedente interessa solo che Deck abbia le capacità specificate in `CardSource`, se poi implementa anche altre interfaccie è ininfluente.



Contract-based design vs programmazione difensiva

Tornando alla specificazione dell'interfaccia di `CardSource`, è possibile notare dei commenti in formato Javadoc che specificano le **precondizioni** e le **postcondizioni** (il valore di ritorno) del metodo. Secondo il ***contract-based* design**, esiste un *"contratto"* tra chi implementa un metodo e chi lo chiama.

Per esempio, considerando il metodo `draw()`, **è responsabilità del chiamante** verificare il soddisfacimento delle precondizioni (*"il mazzo non è vuoto"*) prima di invocare il metodo. Se `draw()` viene chiamato quando il mazzo è vuoto ci troviamo in una situazione di **violazione di contratto** e può anche esplodere la centrale nucleare.

Per specificare il contratto si possono utilizzare delle **asserzioni** o il `@pre` nei **commenti**. Le prime sono particolarmenti utili in fase di sviluppo perché interrompono l'esecuzione del programma in caso di violazione, ma vengono solitamente rimosse in favore delle seconde nella fase di deployment.

Un'altro approccio è la **programmazione difensiva** che al contrario delega la responsabilità del soddisfacimento delle precondizioni al *chiamato*, e non al chiamante.

Classi astratte

Una classe astratta che implementa un'interfaccia **non deve necessariamente implementarne** tutti i metodi, ma può delegarne l'implementazione alle sottoclassi impedendo l'istanziamento di oggetti del suo tipo.

Le interfacce diminuiscono leggermente le performance, però migliorano estremamente la generalità (che aiutano l'espandibilità ed evolvibilità del programma), quindi vale la pena di utilizzarle.

È possibile utilizzare le **classi astratte** anche per classi complete, ma che **non ha senso che siano istanziate**. Un buon esempio sono le classi *utility* della libreria standard di Java.

Classe utility della libreria standard di Java

Un esempio è `Collections.shuffle(List<?> list)` che accetta una lista omogenea di elementi e la mischia. Il *tipo* degli elementi è volutamente ignorato in quanto non è necessario conoscerlo per mischiarli.

Per l'**ordinamento**, invece, è necessario conoscere il tipo degli oggetti in quanto bisogna confrontarli tra loro per poterli ordinare. La responsabilità della comparazione è però delegata all'oggetto, che deve aderire all'interfaccia `Comparable<T>`.

`Collections.sort(...)` ha, infatti, la seguente signature:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

La notazione di generico **aggiunge dei vincoli** su `T`, ovvero il tipo degli elementi contenuti nella lista:

- `T extends Comparable<...>` significa che `T` deve estendere - e quindi implementare - l'interfaccia `Comparable<...>`;
- `Comparable<? super T>` significa che tale interfaccia può essere implementata su un antenato di `T` (o anche `T` stesso).

`Comparable` è un altro esempio di *interface segregation*: serve per specificare che un oggetto ha bisogno della caratteristica di essere comparabile.

Digressione: la classe Collections era l'unico modo per definire dei metodi sulle interfacce (es: dare la possibilità di avere dei metodi sulle collezioni, ovvero liste, mappe, ecc), ma ora si possono utilizzare i metodi di default.

Analisi del testo naturale

Come organizzare la partenza del design suddividendo in classi e responsabilità?

I due approcci principali sono:

- **pattern**: riconoscere una situazione comune da una data;
- **TDD**: partendo dalla soluzione più semplice si definiscono classi solo all'occorrenza.

Un'altra tecnica che vedremo è l'**estrazione dei nomi** (noun extraction), per un certo senso *naive* ma adatta in caso di storie complesse.

Noun extraction

Basandosi sulle specifiche — come i commenti esplicativi delle User Stories — si parte dai sostantivi (o frasi sostantivizzate), si *sfolsticono* con dei criteri, si cercano le relazioni tra loro e quindi si produce la gerarchia delle classi.

Per spiegare il procedimento considereremo il seguente esempio:

- *The **library** contains **books** and **journals**. It may have several **copies** of a given book. Some of the books are for **short term loans** only. All other books may be borrowed by any **library member** for three **weeks**.*
- ***Members of the library** can normally borrow up to six **items** at a time, but **members of staff** may borrow up to 12 items at one time. Only member of staff may borrow journals.*
- *The **system** must keep track of when books and journals are borrowed and returned, enforcing the **rules** described above.*

Nell'esempio sopra sono stati evidenziati i sostantivi e le frasi sostantivizzate.

Criteri di *sfoltimento*

I criteri di *sfoltimento* servono per diminuire il numero di sostantivi considerando solo quelli rilevanti per risolvere il problema. In questa fase, in caso di dubbi è possibile rimandare la decisione a un momento successivo.

Di seguito ne sono riportati alcuni:

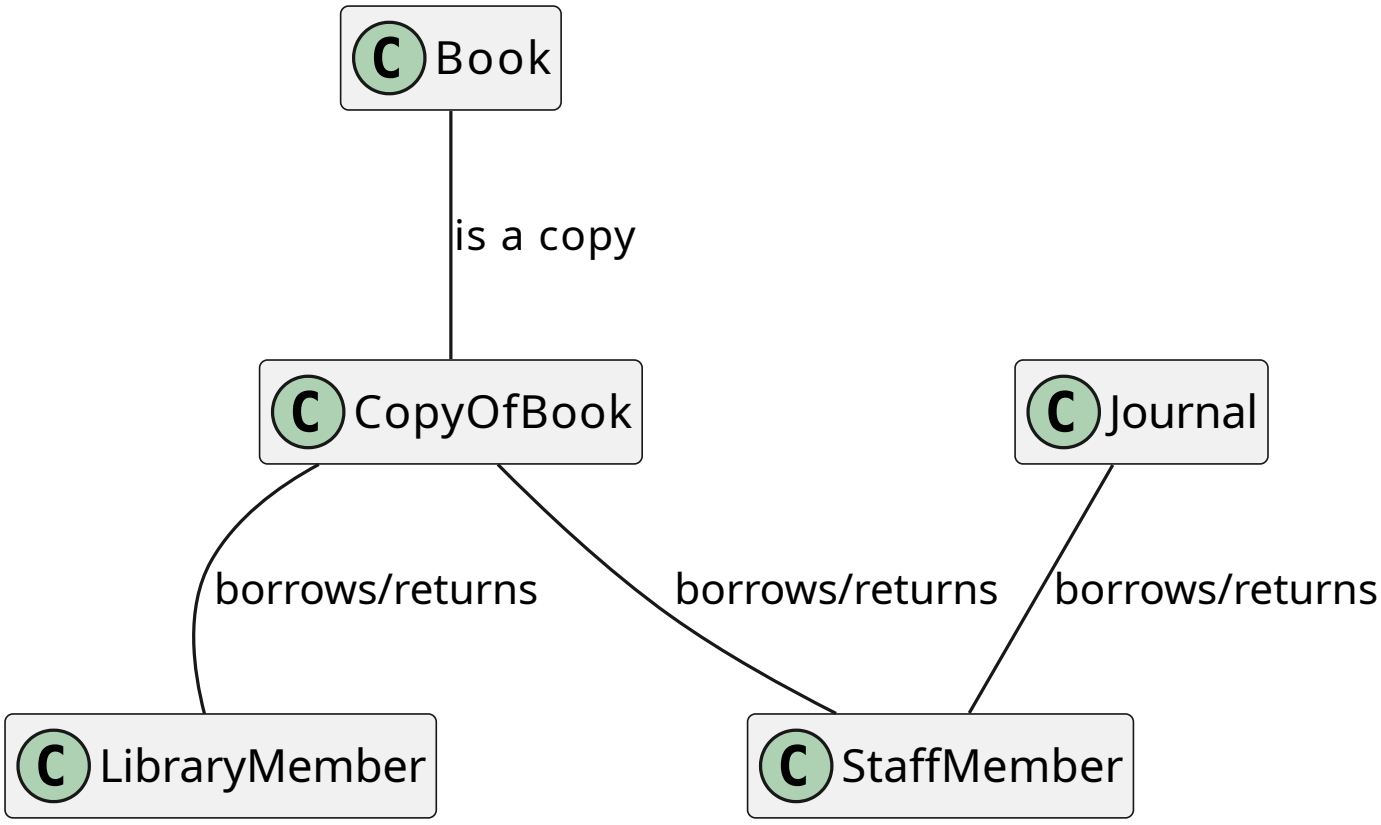
- **Ridondanza**: sinonimi, termini diversi per indicare lo stesso concetto. Anche se è stata utilizzata una locuzione diversa potrebbe essere comunque ridondante, sopratutto in lingue diverse dall'inglese in cui ci sono molti sinonimi.
  Nell'esempio: *library member* e *member of the library*, *loan* e *short term loan*.
- **Vaghezza**: nomi generici, comuni a qualunque specifica; potrebbero essere sintomo di una *classe comune astratta*.
  Nell'esempio: *items*.
- **Nomi di eventi e operazioni**: nomi che indicano azioni e non hanno un concetto di *stato*.
  Nell'esempio: *loan*.
- **Metalinguaggio**: parti *statiche* che fanno parte della logica del programma e che quindi non necessitano di essere rappresentati come classi.
  Nell'esempio: *system*, *rules*.
- **Esterne al sistema**: concetti esterni o verità *"assolute"* al di fuori del controllo del programma.
  Esempio: *library*, *week* (una settimana ha 7 giorni).
- **Attributi**: informazioni atomiche e primitive (stringhe, interi, ...) relative a una classe, che quindi non necessitano la creazione di una classe di per sé.
  Esempio: *name of the member* (se ci fosse stato).

Al termine di questa fase, si avrà una lista di classi *"certe"* e *"incerte"*. In questo esempio, sono soppravvisuti i termini *journal*, *book*, *copy* (of *book*), *library member* e *member of staff*.

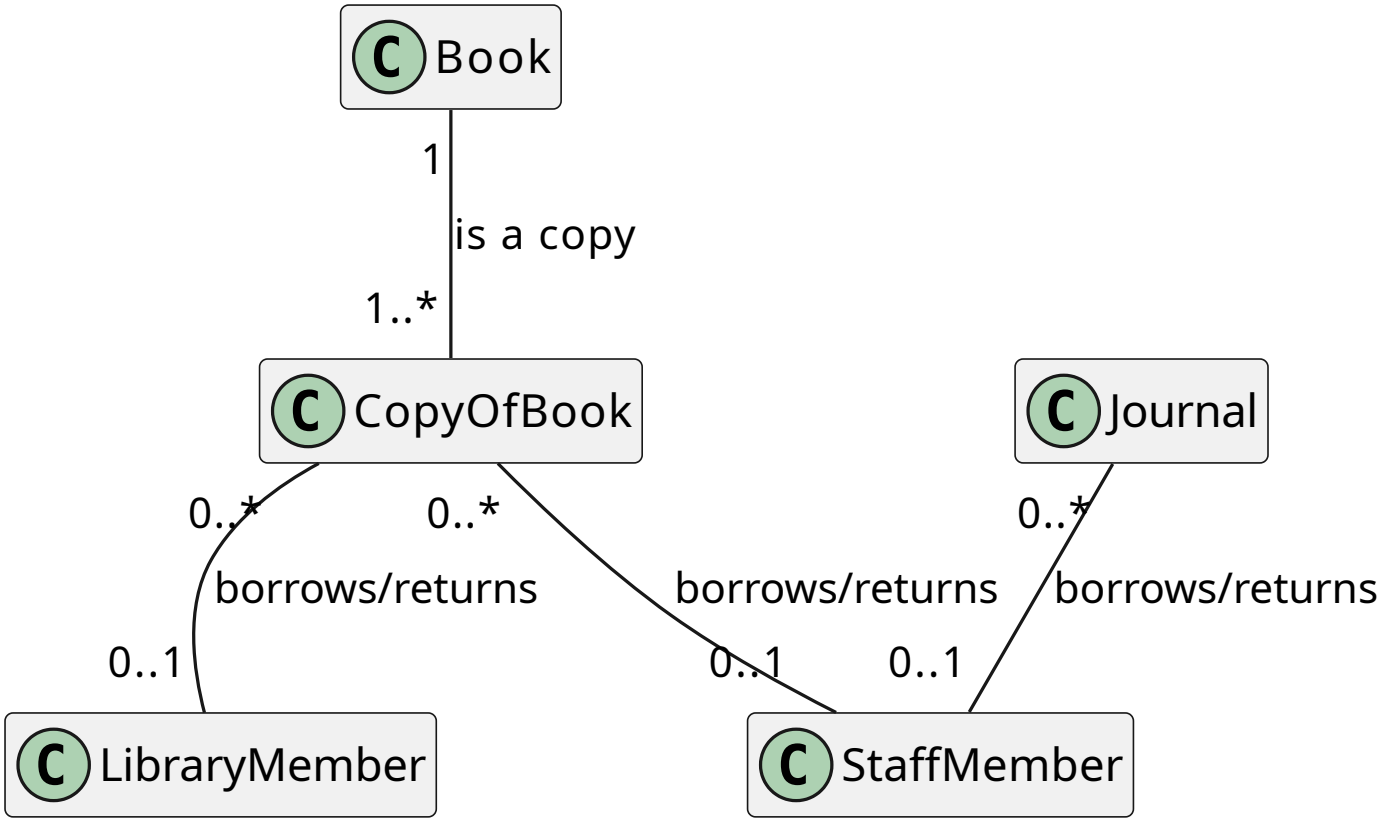
Relazioni tra classi

Il prossimo passo è definire le relazioni tra le classi.

Inizialmente, si collegano con delle *linee* (non frecce) senza specificare la direzione dell'associazione. Parliamo di **associazioni** e non **attributi** perché non è necessariamente vero che tutte le associazioni si trasformino in attributi.

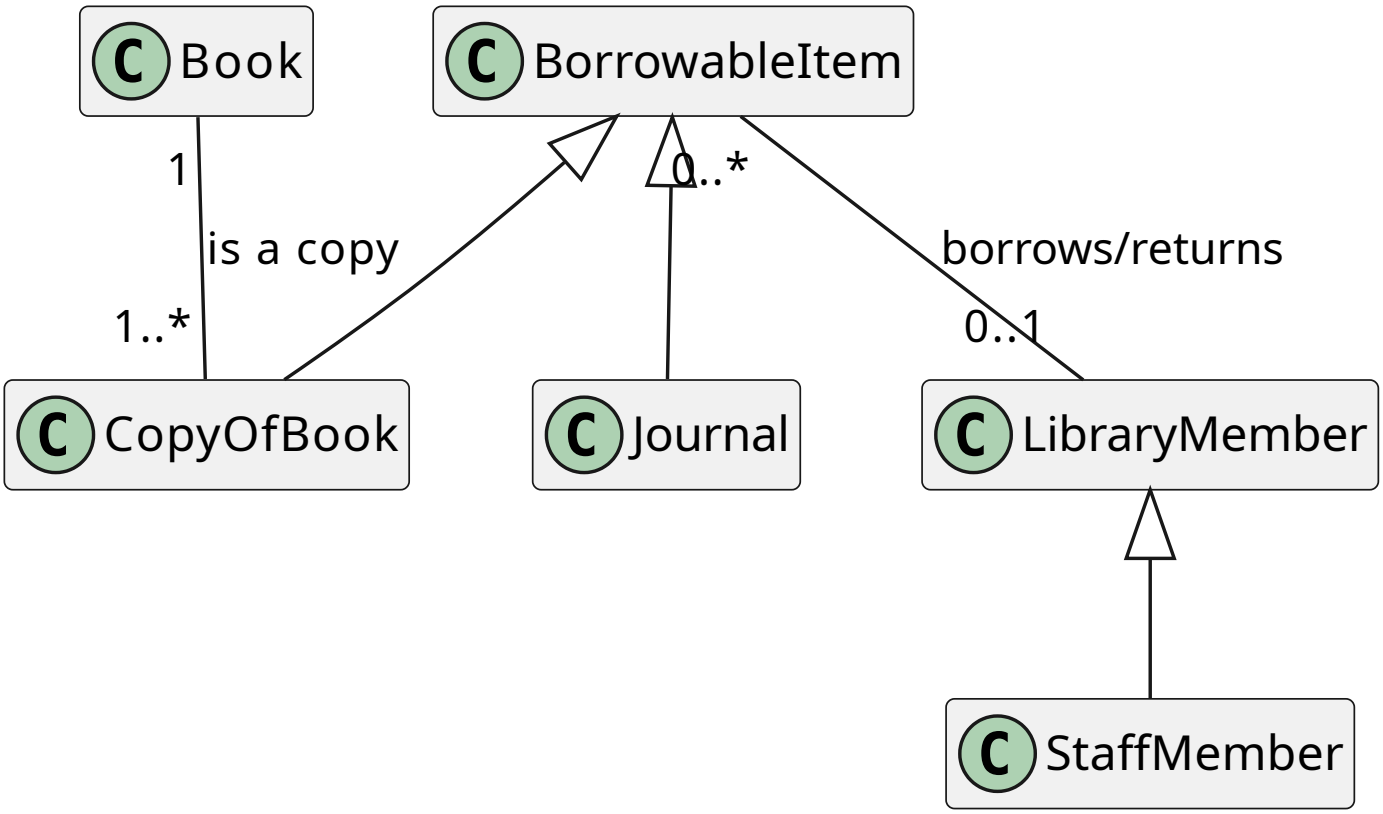


Il prossimo passo è specificare le **cardinalità** delle relazioni, come specificato dal linguaggio UML (opposto in questo aspetto al diagramma ER). La precisione richiesta in questo punto è soggettiva: da una parte, specificare puntualmente il numero massimo di elementi di una associazione può aiutare ad ottimizzare successivamente, dall'altra porta confusione.

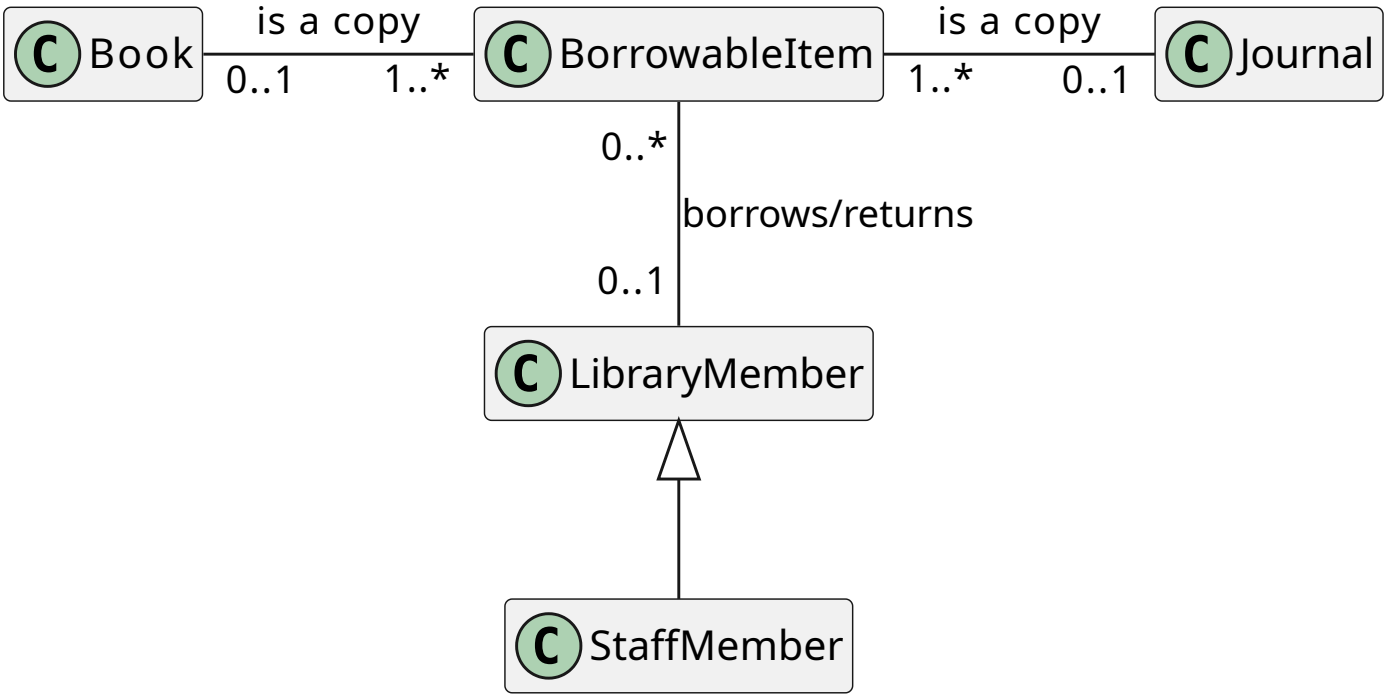


Dopo aver ragionato sulle cardinalità, si iniziano a cercare **generalizzazioni** e fattorizzazioni. In questo caso, notiamo che:

- `StaffMember` *è un* `LibraryMember` con in più la possibilità di prendere `Journal`. Inoltre, un altro indicatore è che hanno la stesso tipo di relazioni con gli altri oggetti.
- `Items` è un termine generico per indicare `CopyOfBook` e `Journal`.



Distinguere `CopyOfBook` e `Journal` è inutile, perché di fatto un `Journal` *è una copia di* un giornale. Si può quindi fattorizzare **rimuovendo la generalizzazione**, come mostrato di seguito.



È imporante però preoccuparsi delle **cardinalità** delle relazioni: è sì vero che un `BorrowableItem` può non *essere una copia di* un `Book` e di un `Journal`, ma deve essere copia

di *esattamente* una delle due opzioni. UML prevede un **linguaggio OCL** (Object Constraint Language) per esprimere vincoli divesamente impossibili da esprimere in un diagramma. È anche possibile scrivere il *constraint* in linguaggio naturale come **nota**.

Patterns

Parlando di progettazione del software e di buone pratiche è impossibile non parlare di **design patterns**, soluzioni universalmente riconosciute valide a problemi di design ricorrenti: si tratta cioè di strumenti concettuali di progettazione che esprimono un'architettura vincente del software catturando la soluzione ad una famiglia di problemi.

Ad ogni pattern sono associati una serie di **idiomi**, implementazioni del pattern specifiche per un certo linguaggio di programmazione che sfruttano i costrutti del linguaggio per realizzare l'architettura dettata dal pattern. Durante questa discussione vedremo alcuni idiomi per Java, che talvolta si discosteranno fortemente dalla struttura descritta dai diagrammi UML dei pattern.

Ma attenzione, esistono anche degli **anti-pattern**, soluzioni che *sembrano* buone ma sono dimostratamente problematiche: dovremo assicurarci di tenerci lontani da questi design truffaldini!

- **Discutere di pattern: i meta-patterns**
- **Singleton**
- **Iterator**
- **Chain of responsibility**
- **Flyweight**
- **NullObject**
- **Strategy**
- **Observer**
- **Adapter**
- **Facade**
- **Composite**
- **Decorator**
- **State**
- **Factory method**
- **Abstract factory**
- **Model view controller**
- **Model view presenter**
- **Builder**

Discutere di pattern: i meta-patterns

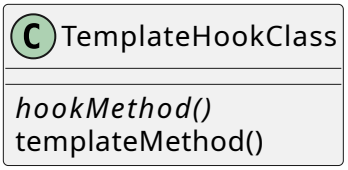
Prima di iniziare a parlare dei principali pattern che un informatico dovrebbe conoscere, possiamo chiederci come possiamo parlare di pattern: semplice, con dei *meta-patterns*, pattern con cui costruire altri pattern!

Nello specifico, i meta-patterns identificano due elementi base su cui ragionare quando si trattano i pattern:

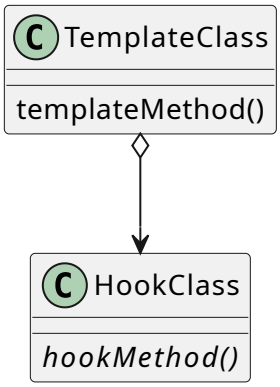
- **HookMethod**: un "metodo astratto" che, implementato, determina il comportamento specifico nelle sottoclassi; è il *punto caldo* su cui interveniamo per adattare lo schema alla situazione.
- **TemplateMethod**: metodo che coordina generalmente più HookMethod per realizzare il design voluto; è l'*elemento freddo* di invariabilità del pattern che ne realizza la rigida struttura.

Ovviamente i metodi *template* devono avere un modo per accedere ai metodi *hook* se intendono utilizzarli per realizzare i pattern. Tale collegamento può essere fatto in tre modi differenti:

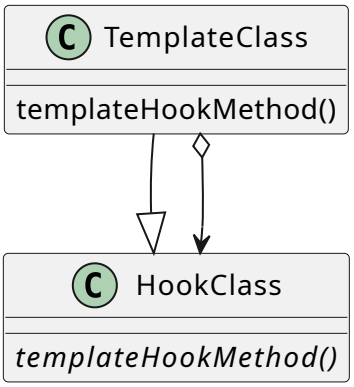
- **Unification**: *hook* e *template* si trovano nella stessa classe astratta, classe da cui erediteranno le classi concrete per implementare i metodi *hook* e, di conseguenza, il pattern; i metodi *template* sono invece già implementati in quanto la loro struttura non si deve adattare alla specifica applicazione.



- **Connection**: *hook* e *template* sono in classi separate, indicate rispettivamente come *hook class* (astratta) e *template class* (concreta), collegate tra di loro da un'aggregazione: la classe template contiene cioè un'istanza della classe hook, in realtà un'istanza della classe concreta che realizza i metodi hook usati per implementare il pattern.



- **Recursive connection**: come nel caso precedente *hook* e *template* sono in classi separate, ma oltre all'aggregazione tali classi sono qui legate anche da una relazione di generalizzazione: la classe template dipende infatti dalla classe hook.



Vedremo a quale meta-pattern aderiranno i pattern che vediamo. A tal proposito,i pattern che vedremo fanno parte dei cosiddetti "**Gang Of Four patterns**", una serie di 23 pattern definiti da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides; oltre ad averli definiti, questi signori hanno diviso i pattern in tre categorie:

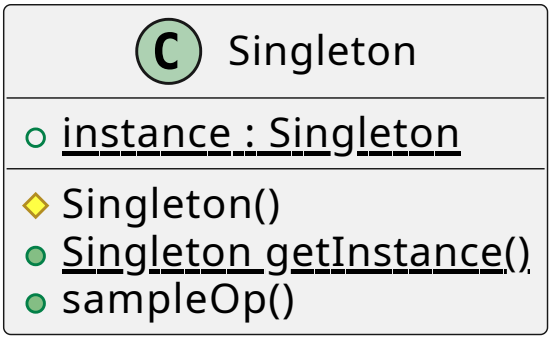
- **Creazionali**: legati alla creazione di oggetti
- **Comportamentali**: legati all'interazione tra oggetti
- **Strutturali**: legati alla composizioni di classi e oggetti

Singleton

Talvolta vorremmo che di un certo oggetto esistesse **una sola istanza** perché logicamente di tale oggetto non ha senso esistano diverse copie all'interno dell'applicazione (es. diverse istanze della classe Gioco in un sistema che gestisce un solo gioco alla volta). Tuttavia i linguaggi Object-Oriented gestiscono solo classi con istanze multiple, per cui la realizzazione di questa unicità può rivelarsi più complessa del previsto.

La soluzione consiste nel rendere la classe stessa responsabile del fatto che non può esistere più di una sua istanza: per fare ciò il primo passo è ovviamente quello di *rendere privato il*

costruttore, o se non privato comunque non pubblico (conviene metterlo protected in modo da poter creare sottotipi).
Bisogna però garantire comunque un modo per recuperare l'unica istanza disponibile della classe: si crea dunque il *metodo statico* `getInstance` che restituisce a chi lo chiama l'unica istanza della classe, creandola tramite il costruttore privato se questa non è già presente. Tale istanza è infatti memorizzata in un *attributo statico* della classe stessa, in modo così da poterla restituire a chiunque ne abbia bisogno.



Con queste accortezze è possibile creare una classe Singleton simile a questa:

```
public class Singleton {
    /* costruttore privato o comunque non pubblico */
    protected Singleton() { ... }

    /* salvo l'istanza per usarla dopo */
    private static Singleton instance = null;

    /* metodo statico */
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    public void metodoIstanza() { ... }
}
```

Tuttavia, per come lo abbiamo scritto questa classe non assicura di non creare più di un'istanza di sé stessa, in quanto non prende in considerazione la **concorrenza**. Se due processi accedono in modo concorrente al metodo `getInstance`, entrambi potrebbero eseguire il controllo sul valore nullo dell'istanza ed ottenere un successo in quanto l'istanza non è ancora stata assegnata al relativo attributo statico nell'altro processo: si ottiene dunque che uno dei due processi ha accesso ad una propria istanza privata, cosa che distrugge completamente il nostro pattern!

Una prima soluzione sarebbe di mettere un lock sull'esecuzione del metodo anteponendovi la direttiva `@Synchronized`: tuttavia, tale approccio comporterebbe un notevole calo di prestazioni del sistema portando vantaggi unicamente alla prima chiamata.
Una soluzione molto più efficiente (non possibile però fino a Java 5) è invece quella che prevede di avere un *blocco sincronizzato* di istruzioni posto all'interno del ramo in cui si pensa che l'istanza sia nulla in cui ci si chiede se effettivamente l'istanza è nulla e solo allora si esegue il costruttore; la presenza del doppio controllo assicura che non vi siano squilibri dovuti alla concorrenza, mentre sincronizzare solamente un blocco e non l'intero metodo fa sì che il calo di prestazioni sia sentito solamente durante le prime chiamate concorrenti.

Idioma Java

Fortunatamente si è sviluppato per il linguaggio Java un idioma molto semplice per il Singleton, in cui al posto di usare una classe per definire l'oggetto si usa un **enumerativo** con un unico valore, l'istanza. Ciascun valore di tali oggetti è infatti trattato nativamente da Java proprio come un Singleton: viene creato al momento del suo primo uso, non ne esiste più di una copia, e chiunque vi acceda accede sempre alla medesima istanza. La possibilità di creare attributi e metodi all'interno degli `enum` completa il quadro.

```
public enum MySingleton {
    INSTANCE;

    public void metodoIstanza() { ... }
}

MySingleton.INSTANCE.sampleOp();
```

Si tratta inoltre di un approccio "thread safe", ovvero che lavora già bene con la concorrenza; l'unico svantaggio è che, se non si conosce l'idioma, a prima vista questa soluzione risulta molto meno chiara rispetto all'approccio precedente.

Iterator

Talvolta gli oggetti che definiamo fanno da **aggregatori** di altri oggetti, contenendo cioè una collezione di questi su cui poi fare particolari operazioni: in questi casi è molto probabile che vorremo poter iterare sui singoli elementi aggregati, ma senza esporre la rappresentazione interna usata per contenerli.

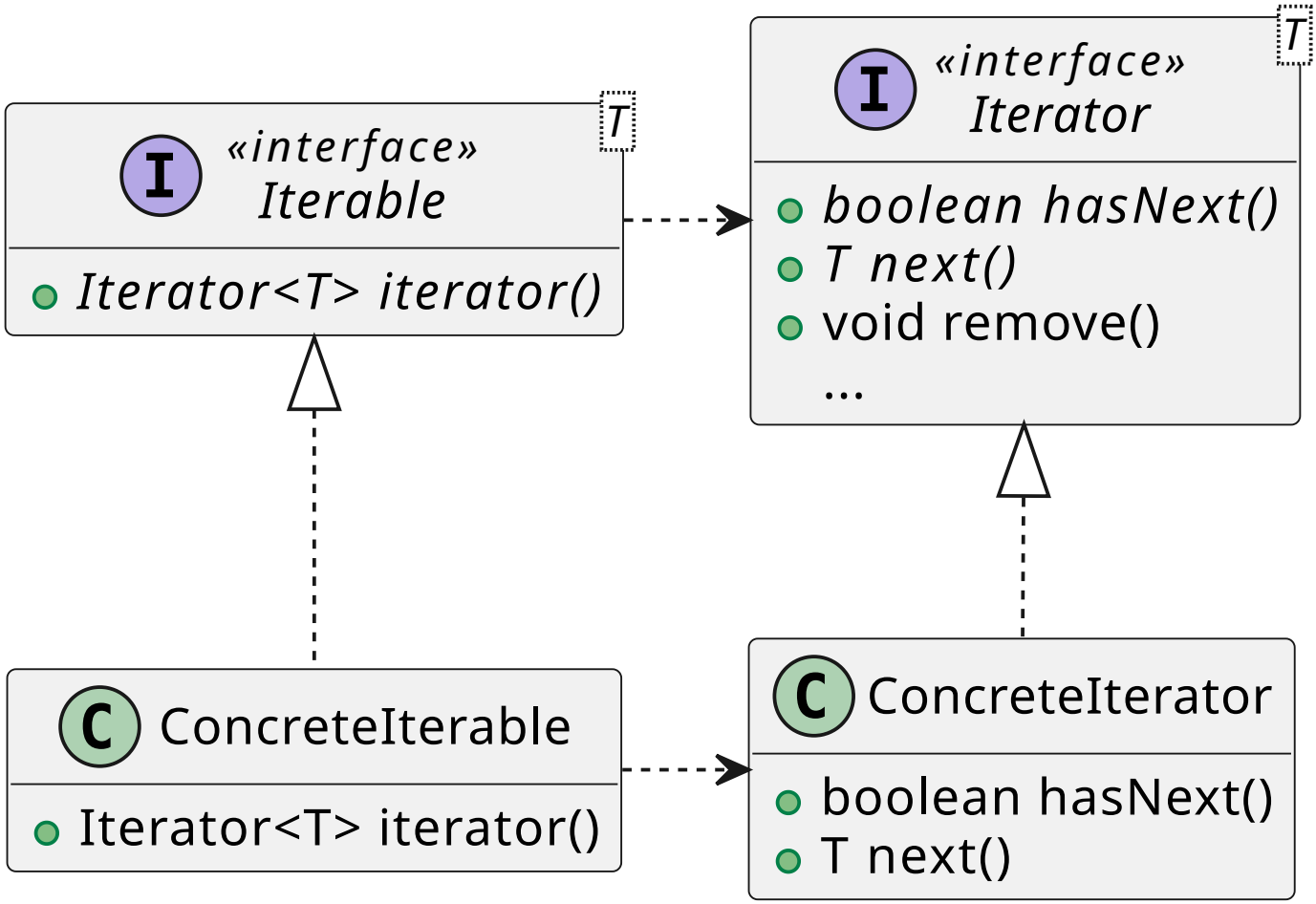
Proprio per risolvere questo tipo di problematiche nasce il pattern Iterator: esso consiste nella creazione di una classe `ConcreteIterator` che abbia accesso alla rappresentazione interna del

nostro oggetto e esponga i suoi elementi in modo sequenziale tramite i metodi `next()` e `hasNext()`; dovendo accedere alla rappresentazione, molto spesso tale iteratore si realizza come una *classe interna anonima*.

Java supporta largamente il pattern Iterator, a tal punto che nella libreria standard esiste un'interfaccia generica per gli iteratori, `Iterator<E>`: all'interno di tale interfaccia sono definiti, oltre ai metodi di cui sopra, il metodo `remove()`, normalmente non supportato in quanto permetterebbe di modificare la collezione contenuta dalla classe, e il metodo `forEachRemaining()`, che esegue una data azione su tutti gli elementi ancora non estratti dell'iteratore.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();

    default void remove() {
        throw new UnsupportedOperationException("remove");
    }

    /* aggiunta funzionale opzionale */
    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

Esiste inoltre un'interfaccia che l'oggetto iterabile può implementare, `Iterable<E>`: essa richiede solamente la presenza di un metodo `iterator()` che restituisca l'iteratore concreto, e una volta implementata permette di utilizzare il proprio oggetto aggregatore all'interno di un costrutto foreach.



Così, per esempio, possiamo passare dal seguente codice:

```
Iterator<Card> cardIterator = deck.getCards();
while (cardIterator.hasNext()) {
    Card card = cardIterator.next();
    System.out.println(card.getSuit());
}
```

... a quest'altro:

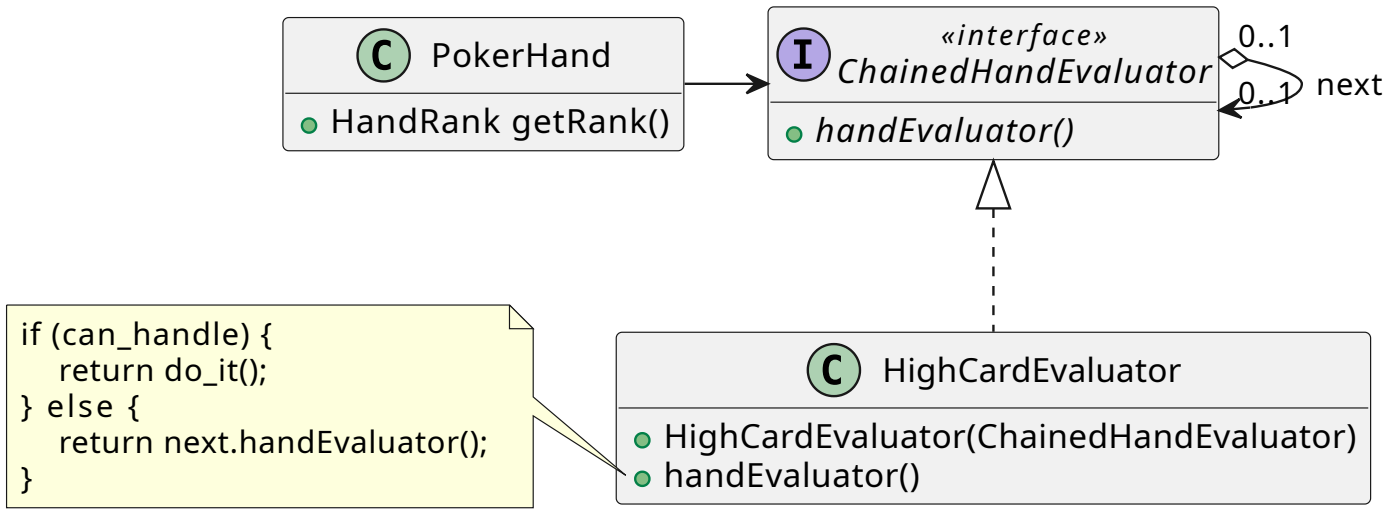
```
for (Card card : deck) {
    System.out.println(card.getSuit());
}
```

Oltre ad essere più stringato il codice è significativamente più chiaro, rendendo palese che la singola `card` sia read-only.

Chain of Responsibility

Talvolta nei nostri programmi vorremmo definire una gestione “a cascata” di una certa richiesta. Pensiamo per esempio a una serie di regole anti-spam: all’arrivo di una mail la prima regola la esamina e si chiede se sia applicabile o meno; in caso affermativo contrassegna la mail come spam, altrimenti *la passa alla prossima regola*, che a sua volta farà lo stesso test passando il controllo alla terza in caso negativo, e così via. Abbiamo cioè un *client* in grado di fare una richiesta, e una **catena di potenziali gestori** di cui non sappiamo a priori chi sarà in grado di gestirla effettivamente.

Il pattern Chain of Responsibility risolve il disaccoppiamento tra client e gestore *concatenando i gestori*. Esso prescrive la creazione di un’interfaccia a cui tutti i gestori devono aderire, contenente solo la dichiarazione di un metodo `evaluate` che implementa la logica descritta prima: si stabilisce se si può gestire la richiesta, e se non si può si chiama lo stesso metodo su *un altro gestore* ottenuto come parametro al momento della creazione.



In questo modo all’interno del client è sufficiente creare una vera e propria catena di gestori e chiamare il metodo `evaluate` del primo: si noti che l’ordine in cui vengono assemblati tali gestori conta, in quanto la valutazione procede sequenzialmente.

```
public interface Gestore {

    /* Il tipo di ritorno dipende dal campo applicativo */
    public ??? evaluate();

}

public class Client {

    private Gestore evaluator =
        new GestoreConcreto1(
            new GestoreConcreto2(
                new GestoreConcreto3(null)));

    public void richiesta() {
        evaluator.evaluate();
    }

}
```

FlyWeight

Talvolta ci troviamo in una situazione simile a quella che aveva ispirato il pattern Singleton: abbiamo una **serie di oggetti immutabili fortemente condivisi** all'interno del programma e per motivi di performance e risparmio di memoria vorremmo che *non esistano istanze diverse a parità di stato*. Se due client devono usare un'istanza con lo stesso stato vorremmo cioè non usino ciascuno un'istanza duplicata ma proprio la *stessa istanza*: essendo le istanze immutabili, tale condivisione non dovrebbe infatti creare alcun tipo di problema.

Il pattern FlyWeight serve a gestire una collezione di oggetti immutabili assicurandone l'unicità: esso consiste nel rendere privato il costruttore e **costruire tutte le istanze a priori con un costruttore statico**, salvandole in una lista privata. I client possono dunque richiedere una certa istanza con un metodo `get` specificando lo stato dell'istanza desiderata: in questo modo, a parità di richiesta verranno restituite le stesse identiche istanze.

Abbiamo visto un'applicazione di questo pattern durante i laboratori parlando di `Card`:

```
public class Card {
    private static final Card[][] CARDS = new Card[Suit.values().length]
[Rank.values().length];

    static {
        for (Suit suit : Suit.values()) {
            for (Rank rank : Rank.values()) {
                CARDS[suit.ordinal()][rank.ordinal()] = new Card(rank, suit);
            }
        }
    }

    public static Card get(Rank pRank, Suit pSuit) {
        return CARDS[pSuit.ordinal()][pRank.ordinal()];
    }
}
```

A differenza del pattern Singleton è difficile definire a priori quante istanze ci sono: abbiamo un'istanza per ogni possibile combinazione dei valori degli attributi che compongono lo stato. Proprio per questo motivo il pattern può risultare un po' inefficiente per oggetti con rappresentazioni grandi: alla prima computazione vengono infatti inizializzati *tutti* gli oggetti, perdendo un po' di tempo e sprecando potenzialmente spazio se non tutte le istanze saranno accedute.

NullObject

Spesso nei nostri programmi avremo bisogno di utilizzare valori "nulli": pensiamo per esempio al termine di una Chain of Responsibilities, dove per fermare la catena di chiamate dobbiamo dare un valore nullo al `next` dell'ultimo gestore. In generale, a una variabile che indica un riferimento ad un oggetto possiamo assegnare il valore speciale `null` per indicare che essa *non punta a nulla*.

Il problema sorge però quando a runtime si prova a dereferenziare tale valore e viene sollevata un'eccezione (`NullPointerException` in Java): questa possibilità ci costringe nel codice ad essere sempre molto titubanti sui valori che ci vengono passati, in quanto non possiamo mai assumere che essi puntino ad un valore reale e dunque dobbiamo sempre controllare che non siano nulli.

C'è però da dire che anche con tali accortezze l'utilizzo di `null` è poco carino, in quanto un valore nullo può indicare cose anche molto diverse:

- un errore a runtime;
- uno stato temporaneamente inconsistente;
- un valore assente o non valido.

Ogni volta che si utilizza `null` il codice diventa un po' meno chiaro, e sarebbe necessario disambiguare con commenti o documentazione per spiegare con che accezione tale valore viene usato. Anche le strategie di gestione del `null` variano drasticamente a seconda del significato assegnato a tale valore: quando non ci sono valori "assenti" e dunque il `null` indica solo un errore è sufficiente controllare che i dati passati non siano nulli con condizioni, asserzioni o l'annotazione `@NotNull`.


null object valori non assenti

Quando invece ci sono **valori "assenti"**, ovvero che indicano situazioni particolari (es. il Joker in un mazzo di carte, che non ha né Rank né Suit), la gestione è più complicata. Se non vogliamo trattarli come `null` per l'ambiguità che tale valore introduce, un'altra opzione è creare un metodo booleano nella classe che restituisce se l'istanza ha il valore nullo (es. `isJoker()`): tuttavia, questo apre le porte a errori da parte dell'utente, che potrebbe dimenticarsi di fare tale controllo e usare l'oggetto come fosse qualunque altro.

Per creare un oggetto che corrisponda al **concetto di nessun valore** o **valore neutro** nasce allora il pattern NullObject: si crea all'interno della classe o dell'interfaccia un *oggetto statico* chiamato `NULL` che fornisce *particolari implementazioni dei metodi* della stessa per realizzare l'idea di valore nullo a livello di dominio. In questo modo tale oggetto mantiene l'identità della classe rimanendo però sufficientemente separato dagli altri valori; inoltre, la presenza di implementazioni specifiche dei metodi evita il lancio di eccezioni ambigue.

```
public interface CardSource {
    Card draw();
    boolean isEmpty();

    public static CardSource NULL = new CardSource() {
        public boolean isEmpty() {
            return true;
        }
        public Card draw() {
            assert !isEmpty();
            return null;
        }
    }
}
```

Quindi possiamo notare che il concetto del NullObject pattern è quello di creare un oggetto in cui viene definito un comportamento specifico per ogni metodo che rispecchia ciò che accadrebbe nel caso in cui il metodo venisse chiamato su null nel normale flusso di istruzioni.

Strategy / Delegation

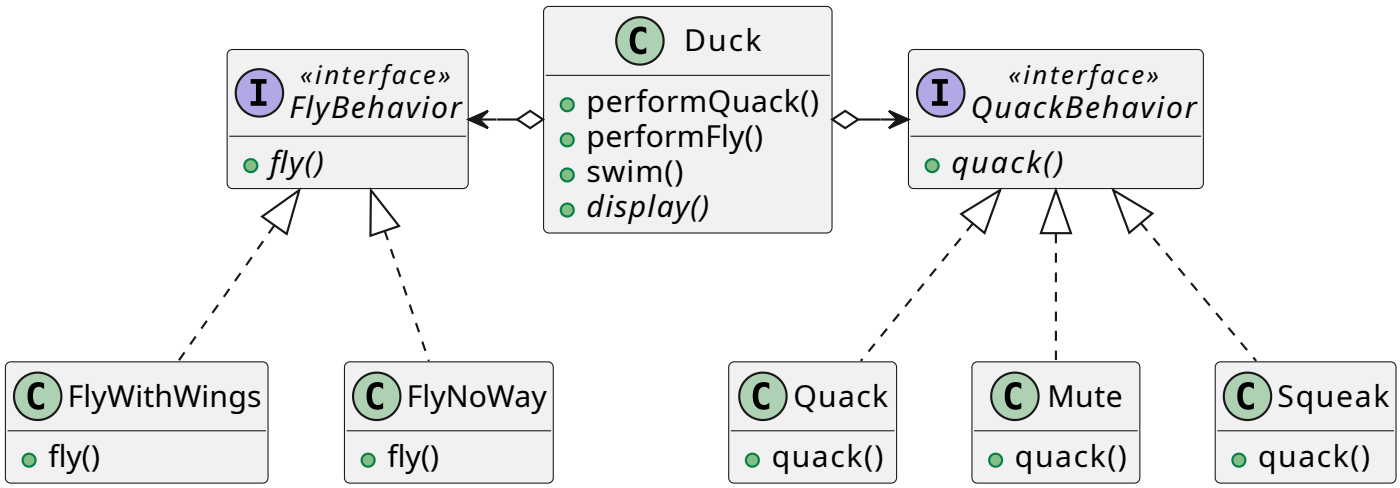
Talvolta nelle nostre classi vogliamo definire **comportamenti diversi per diverse istanze**: la soluzione classica dei linguaggi Object-Oriented è la creazione di una gerarchia di classi in cui le classi figlie sovrascrivano i metodi della classe genitore. Tuttavia, questo espone a delle problematiche: cosa fare se per esempio la classe genitore cambia aggiungendo un metodo che una delle classi figlie non dovrebbe poter implementare (es. `RubberDuck` come figlia di `Duck`, che aggiunge il metodo `fly()`)?

Non volendo violare il principio Open-Close, non siamo intenzionati a rimuovere il metodo incriminato, per cui dobbiamo cercare altre soluzioni. Una prima idea sarebbe quella di sopperire al fatto che la classe genitore non sappia chi sono i suoi figli con costrutti proprietari del linguaggio:

- una classe `Final` non permette di ereditare, ma questo non ci permetterebbe di differenziare il comportamento;
- una classe `Sealed` (aggiunta di Java 17) sceglie esplicitamente chi possano essere i suoi figli: in questo modo si può evitare che la classe figlia problematica non possa ereditare, ma si tratta comunque di una soluzione parziale.

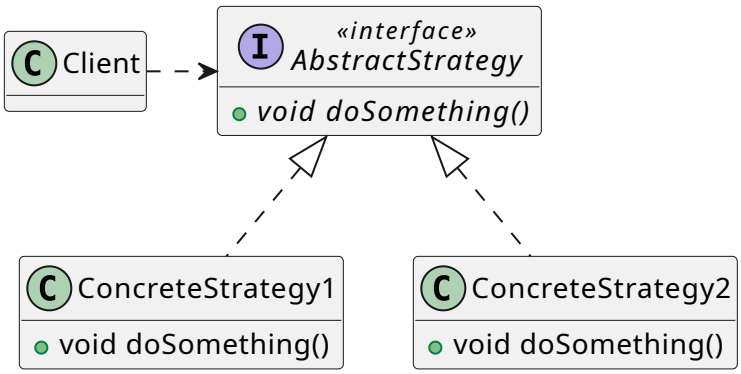
Non si può neanche pensare di fare semplicemente l'override nella classe figlia del metodo aggiunto facendo in modo che lanci un'eccezione: si avrebbe infatti una inaccettabile violazione del principio di sostituzione di Liskov, che afferma sostanzialmente che un'istanza di una sottoclasse deve poter essere usata senza problemi al posto dell'istanza di una classe genitore.

Una soluzione migliore si basa invece sul concetto di **delega**, che sostituisce all'ereditarietà la *composizione*. Fondamentalmente si tratta di individuare ciò che cambia nell'applicazione e separarlo da ciò che rimane fisso: si creano delle *interfacce per i comportamenti da diversificare* e una *classe concreta che implementa ogni diverso comportamento* possibile. All'interno della classe originale si introducono dunque degli **attributi di comportamento**, impostati al momento della costruzione o con dei setter a seconda della dinamicità che vogliamo permettere: quando viene richiesto il comportamento a tale classe essa si limiterà a chiamare il proprio "oggetto di comportamento". Nell'esempio delle `Duck`, per esempio, la struttura è la seguente:



Come si vede, qui non c'è scritto da nessuna parte che una `Duck` deve volare, ma solo che deve definire la sua "politica di volo" incorporando un `FlyBehaviour`.

La differenziazione dei comportamenti si fa dunque *a livello d'istanza* e non di classe: il pattern definisce una famiglia algoritmi e li rende tra di loro intercambiabili tramite *encapsulation*. Per questo motivo tale pattern è usato in situazioni anche molto diversa da quella con cui l'abbiamo introdotto: un altro esempio presente in Java è l'interfaccia `Comparator`.

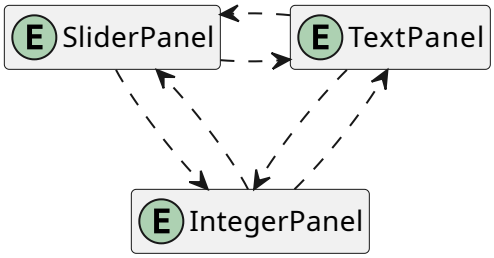


Observer

Molto spesso capita di avere nei nostri programmi una serie di elementi che vanno tenuti sincronizzati: pensiamo per esempio ad una ruota dei colori che deve aggiornare i valori RGB quando l'utente seleziona un punto con il mouse. Abbiamo cioè uno **stato comune** che va mantenuto coerente in tutti gli elementi che lo manipolano.

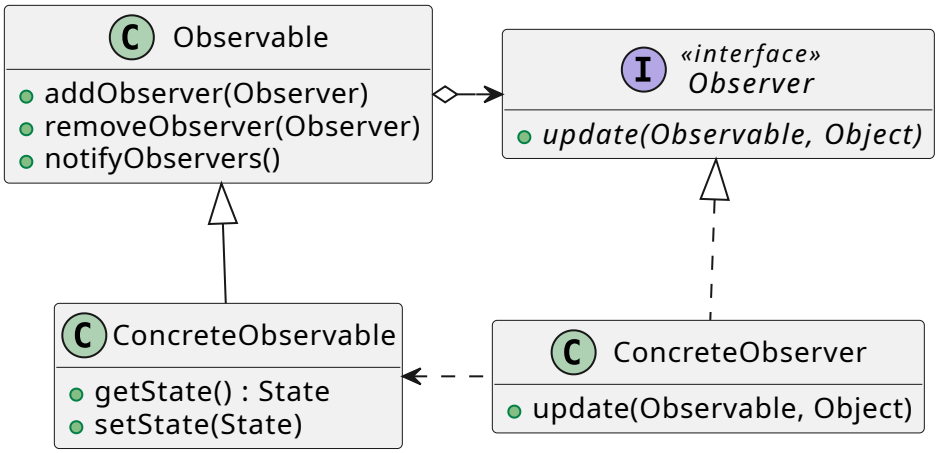
Nella realizzazione di questa funzionalità si rischia di cadere nell'anti-pattern delle *pairwise dependencies* in cui ogni vista dello stato deve conoscere tutte le altre: si ha cioè un forte accoppiamento e una bassissima espandibilità, in quanto per aggiungere una vista dobbiamo modificare tutte le altre. Ovviamente basta avere poco più di due diverse viste perché il numero di dipendenze (e dunque di errori) cresca esponenzialmente: questo anti-pattern è

proprio tutto il contrario del principio di separazione, che predicava forte coesione interna e pochi accoppiamenti esterni.



La soluzione proposta dal pattern Observer è dunque quella di estrarre la parte comune (lo *stato*) e isolarlo in un oggetto a parte, detto **Subject**: tale oggetto verrà osservato da tutte le viste, le cui classi prendono ora il nome di **Observer**. Si sta cioè **centralizzando** la gestione dello stato: abbiamo cioè n classi che osservano una classe centrale e reagiscono ad ogni cambiamento di stato di quest'ultima. Si tratta una situazione talmente comune che in Java erano presenti delle classi (ora deprecate in quanto non *thread-safe*) per realizzare questo pattern: `java.util.Observer` e `java.util.Observable`.

Ma come fanno gli Observer a sapere che il Subject è cambiato? L'idea di fare un continuo *polling* (chiedo "Sei cambiato?" al Subject), non è ovviamente sensata, in quanto bloccherebbe l'esecuzione sprecando tantissime risorse. Invertiamo invece la responsabilità con un'architettura **event-driven**: gli Observer si *registrano* al Subject, che li informerà quando avvengono cambiamenti di stato.



Restano però da capire un paio di cose. Bisogna innanzitutto spiegare *come colleghiamo Observer e Subject*: come si vede in figura, esiste una classe `Observable` che funge da base da cui ereditare per ogni Subject; vi è poi un'interfaccia `Observer` che gli Observer concreti devono ovviamente implementare.

A questo punto gli Observer si possono sottoscrivere al Subject semplicemente attraverso l'uso delle sue funzioni `addObserver()` e `removeObserver()`, venendo così sostanzialmente inseriti o rimossi nella lista interna degli Observer interessati.

Una volta che lo stato del Subject viene cambiato, solitamente attraverso una serie di metodi pubblici che permettano a tutti di modificarlo (`setState()`), esso chiama dunque il suo metodo `notifyObservers()`: questo altro non fa che ciclare su tutti gli Observer sottoscritti chiamandone il metodo `update(Observable, Object)`, dove:

- `Observable` è il Subject di cui è stato modificato lo stato (l'uso di interfacce permette di sottoscrivere un Observer a più Subject tra cui disambiguare al momento dell'update)
- `Object` è la parte di stato che è cambiata (*Object* perché il tipo dipende ovviamente dal Subject in questione)

Sul metodo di notifica del cambiamento di stato esistono però due diverse filosofie, **push** e **pull**, ciascuna con i suoi campi applicativi prediletti: vediamole dunque singolarmente, evidenziando quando e come esse sono utilizzate.

push

In questo caso l'argomento Observable di `update` viene messo nullo, mentre **nell'Object viene passata la totalità dello stato** del Subject:

```
// Observable
@Override
public void notifyObservers() {

    for (Observer observer : observers) {
        observer.update(null, state);
    }

}

// Observer
@Override
public void update(Observable model, Object state) {

    if (state instanceof Integer intValue) {
        doSomethingOn(intValue);
    }

}
```

Come si vede, dovendo definire come reagire al cambiamento di stato in `update` l'Observer dovrà innanzitutto fare un down-casting per ottenere un oggetto della classe corretta. Avendo la responsabilità di tale casting l'Observer dovrà conoscere precisamente la struttura dello stato del Subject, creando una *forte dipendenza* che potrebbe creare problemi di manutenibilità.

Un altro problema di questo approccio è che gli Observer sono solitamente interessati a una piccola porzione dello stato del Subject, quindi passarlo tutto come parametro potrebbe sovraccaricare inutilmente la memoria.

pull

Con questo approccio, invece di mandare lo stato all'`update` **viene passato il Subject stesso**, il quale conterrà uno o più metodi per accedere allo stato (`getState`):

```
// Observable
@Override
public void notifyObservers() {

    for (Observer observer : observers) {
        observer.update(this, null);
    }

}

// Observer
@Override
public void update(Observable model, Object state) {

    if (model instanceof ConcreteObservable cModel) {
        doSomethingOn(cModel.getState());
    }

}
```

Sebbene comporti un passaggio in più poiché l'Observer deve chiamare un metodo del Subject quando riceve la notifica, questo cambio di prospettiva offre due vantaggi: in primo luogo non viene passato tutto lo stato, il che fa risparmiare molta memoria; inoltre, il Subject potrebbe decidere di rendere disponibili sottoinsiemi diversi dello stato con getter diversi, mostrando così ad ogni Observer solo le informazioni per esso rilevanti.

Inoltre, sebbene anche in questo caso sia richiesto un casting (da Observable al Subject), questo approccio rende meno dipendenti dalla rappresentazione interna del Subject: fintanto che la firma dei getter non cambia lo stato interno del Setter può cambiare senza problemi.

Approccio ibrido e dipendenze

Partiamo col dire che molto spesso nei casi reali gli approcci *push* e *pull* sono ibridati tra di loro: ad `update` viene passato sia il Subject che quella parte di stato utile a tutti gli Observer, mentre qualora gli serva qualcosa di più specifico essi se lo andranno a prendere con il getter.

Il vero problema di entrambi gli approcci è però quello delle dipendenze: nel caso *push* dipendiamo dalla rappresentazione interna del Subject, mentre nel caso *pull* dalla sua classe concreta. Poiché tale dipendenza non è facilmente eliminabile, piuttosto che lasciarla nascosta nel casting conviene **esplicitarla**:

- all'interno dell'Observer salvo l'istanza di Observable a cui mi sono sottoscritto, così al momento dell'`update` posso verificare direttamente che l'istanza sia quella al posto di fare un casting;
- creiamo una classe `State` e l'aggreghiamo sia nell'Observer che nell'Observable concreto in modo che essa nasconda la rappresentazione reale dello stato.

Otteniamo dunque un codice simile al seguente:

```java
public class State { /* rappresentazione interna dello stato */ }

public class Observable {
    private State stato;
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(@NotNull Observer obs) { observers.add(obs); }
    public void removeObserver(@NotNull Observer obs) { observers.remove(obs); }
    public void notifyObservers() {
        for (Observer obs: observers) update(this, stato);
    }
}

public class Subject extends Observable {

    public void setState(State nuovoStato) { ... }
    public State getState() { return super.stato; }
    /* Opzionale: altri metodi getter */
}

public interface Observer {
    public void update(Observable subject, Object stato);
}

public class ConcreteObserver {
    private Observable mySubject;

    @Override
    public void update(Observable subject, Object stato) {
        if (subject == mySubject) {
            ...
        }
    }
}
```

ADAPTER

Spesso nei programmi che scriviamo capita di dover **far collaborare interfacce diverse** di componenti non originariamente sviluppati per lavorare insieme. Questo capita in una miriade di situazioni, ma volendone citare alcune:

- in un ambito di sviluppo COTS (*Component Off The Shelf: sviluppiamo solo ciò che non è disponibile tramite librerie o codice open-source*) riutilizziamo tanti componenti presi dal mercato, non pensati per essere compatibili;
- sviluppando ed evolvendo un programma in modo *incrementale* capita di dover integrare componenti nuovi con componenti vecchi (*legacy*) per garantire una certa continuità nell'esperienza utente.

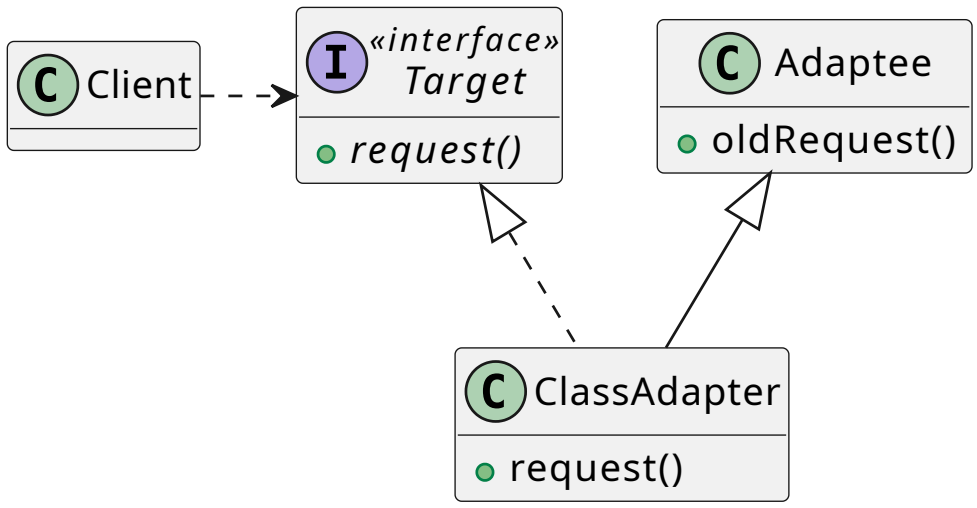
Da tutta una serie di situazioni simili è nato il bisogno di creare delle strutture che permettessero di rendere compatibili componenti già esistenti, ovvero creare della *"colla"* in grado di legare i componenti tra loro per soddisfare le specifiche del sistema. È così ben presto scaturito il pattern **Adapter**, un pattern ormai molto diffuso che consiste nel creare vari moduli che possano essere incollati o adattati ad altre strutture in modo da renderle utilizzabili incrementalmente e in modo controllato.

Sebbene sia già utilizzato molto spesso, talvolta anche inconsciamente, approfondiamo il pattern in questa sede non solo per imparare a usarlo con più criterio, ma anche perché di esso esistono due "versioni":

- **Class Adapter**: adatta una classe.
- **Object Adapter**: adatta un oggetto di una classe.

Come vedremo, questi due pattern sono molto simili a livello di schema UML ma abbastanza differenti da rendere importante capire quale usare in quali contesti, comprendendo vantaggi e svantaggi di entrambi.

Class Adapter



Come si vede dallo schema UML, per permettere a un *Client* di comunicare tramite un'interfaccia *Target* con un componente concreto vecchio detto *Adaptee* il Class Adapter utilizza una classe concreta che **implementa l'interfaccia Target** e **estende la classe Adaptee**, ereditandone così i metodi e la vecchia interfaccia: all'interno di tale classe potremo dunque limitarci a *rimappare le funzionalità* richieste dalla nuova interfaccia su quella vecchia, implementando qualcosa solo se strettamente necessario e comunque sfruttando la logica già presente della classe estesa.

```
public class Adapter extends Adaptee implements Target {
    @Override
    public void request() {
        this.oldRequest();
    }
}
```

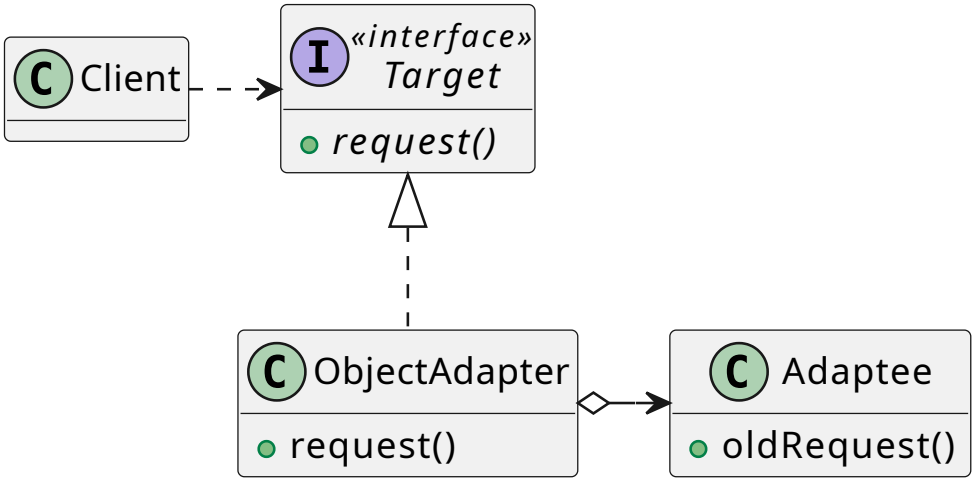
In questo modo il client utilizzerà l'adapter come se fosse l'oggetto completo, non accorgendosi che quando ne chiama un metodo in realtà il codice eseguito è quello appartenente alla vecchia classe già esistente: in un **unica istanza** si sono dunque riunte l'interfaccia vecchia e quella nuova.

Vediamo dunque quali sono i pro e i contro di questo approccio. È utile innanzitutto notare che estendendo l'Adaptee la classe Adapter ha parziale accesso alla sua rappresentazione interna, un vantaggio non da poco quando si considera quanto questo faciliti l'eventuale modifica di funzionalità; inoltre, essa ne eredita le definizioni dei metodi, e se questi non devono cambiare tra la vecchia interfaccia e la nuova si può evitare di ridefinirli totalmente, risparmiando così parecchio codice.

Inoltre, un'istanza della classe Adapter può essere utilizzata attraverso **entrambe le interfacce** in quanto implementa quella nuova ed eredita quella vecchia; questo aspetto può essere considerato sia un vantaggio che uno svantaggio: se infatti da un lato ciò è molto utile in sistemi che evolvono incrementalmente e in cui dunque alcune componenti potrebbero volersi riferire ancora alla vecchia interfaccia, d'altro canto questo aspetto impedisce di imporre tassativamente che l'oggetto sia utilizzato solo tramite l'interfaccia nuova.

Va poi notato che questo approccio perde un po' di senso nel caso in cui si debba adattare un'*interfaccia* e non una classe, in quanto implementare entrambe le interfacce non permette di ereditare codice o funzionalità da quella vecchia. Inoltre, il Class Adapter potrebbe presentare problemi relativi all'ereditarietà multipla, non supportata da alcuni linguaggi a oggetti (es. Java).

Object Adapter



Come abbiamo già detto più volte, spesso conviene prediligere la *composizione* rispetto all'ereditarietà: al pattern del Class Adapter si contrappone dunque l'Object Adapter, che invece di estendere la classe Adaptee **contiene una sua istanza** e **delega** ad essa tramite la vecchia interfaccia le chiamate ai metodi dell'interfaccia nuova, eventualmente operando i necessari rimaneggiamenti.

```
public class Adapter implements Target {
    private final Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        assert adaptee != null;
        this.adaptee = adaptee;
    }

    @Override
    public void request() {
        adaptee.oldRequest();
    }
}
```

Anche in questo caso il client non si accorge di nulla, e in particolare non sarebbe nemmeno in grado di dire con certezza se l'Adapter utilizzato sia un Class Adapter o un Object Adapter: a lui la scelta del paradigma è del tutto trasparente.

Rispetto al Class Adapter l'Object Adapter presenta differenti punti di forza e di debolezza, e il primo di questi ultimi è rappresentato dal fatto che invece di avere un'unica istanza che racchiuda entrambe le interfacce con questo pattern abbiamo invece *due istanze* (Adapter e Adaptee contenuto), cosa che può costituire un notevole spreco di memoria in certe situazioni.

Inoltre, aver sostituito l'ereditarietà con la composizione ha lo sgradevole effetto di non permettere all'Adapter di vedere in alcun modo la rappresentazione protetta dell'Adaptee, che esso dovrà invece manipolare unicamente tramite la sua interfaccia pubblica. Si è poi costretti

a *reimplementare ogni metodo* anche se questo non è cambiato dall'interfaccia vecchia a quella nuova, in quanto è comunque necessario operare la delega all'Adaptee.

Tuttavia, l'Object Adapter si rivela particolarmente utile nel caso ad essere adattata debba essere un'*interfaccia*: non soffrendo di problemi di ereditarietà, un Object Adapter ha la peculiarità di poter adattare chiunque implementi la vecchia interfaccia, ovvero un'intera *gerarchia* di classi potenzialmente non ancora esistenti!

Class Adapter vs Object Adapter

Class Adapter e Object Adapter hanno ciascuno i propri vantaggi e svantaggi che li rendono più adatti ad essere utilizzati in diverse situazioni. Volendo fare un confronto tra i due approcci proponiamo dunque la seguente tabella:

Aspetto	Class Adapter	Object Adapter
Accesso all'Adaptee	L'Adapter può accedere ad attribuiti e metodi protetti dell'Adaptee	L'Adapter può interagire con l'Adaptee solo tramite la sua interfaccia pubblica
Riuso del codice	Non richiede di reimplementare i metodi che non cambiano	Qualunque metodo va reimplementato per fare la delega
Uso della memoria	Un'unica istanza	Due istanze obbligatorie
Adozione delle interfacce	L'istanza può essere usata con entrambe le interfacce	L'istanza può essere usata solo tramite la nuova interfaccia
Problemi di ereditarietà multipla	Possibili	No
Adattamento delle interfacce	Non è indicato	Adattando un'interfaccia può adattare un'intera gerarchia di classi

FACADE

Costruendo un sistema complesso può capitare di dover definire una serie di interfacce molto specifiche e dettagliate per i propri componenti in modo che questi possano lavorare correttamente in concerto tra di loro. Il problema sorge però quando un Client, dovendo accedere al sistema, si ritrova costretto a dover interagire direttamente con i sottosistemi che lo compongono, cosa che lo obbliga a sviscerare i funzionamenti interni dello stesso per ottenere un comportamento tutto sommato semplice.

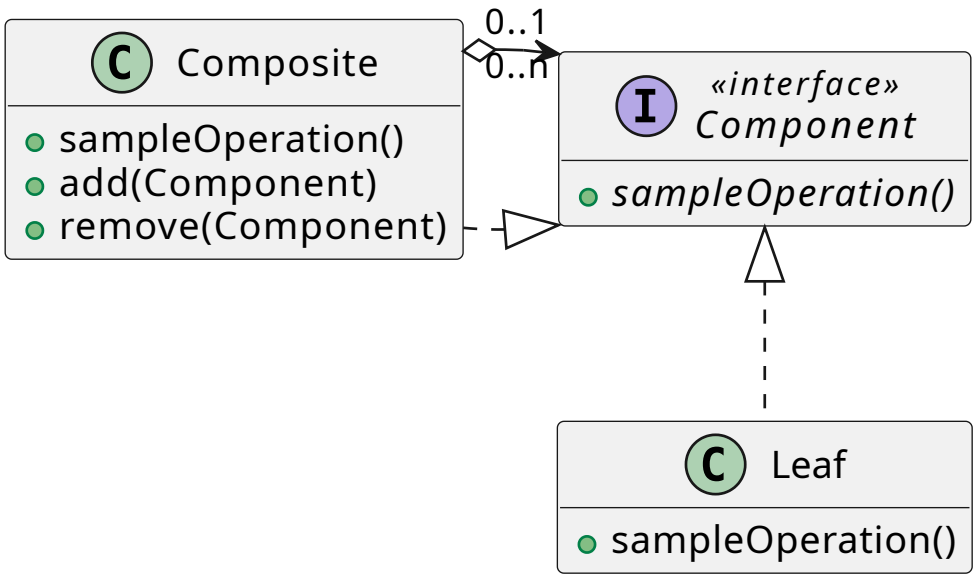
Lo scopo del pattern Facade è allora quello di **fornire un'interfaccia unificata e semplificata a un insieme di interfacce separate**: spesso infatti l'uso comune di un sistema si riduce un paio di operazioni ottenibili combinando varie funzionalità fornite dal package; invece di richiedere al Client di operare tale composizione facciamo ricadere sulle nostre spalle tale compito costruendo una *classe* che faccia da *interfaccia standard* al sistema.

Si noti come questo non impedisca al Client di usare anche le funzionalità più complesse, ma metta solo ulteriormente a disposizione un'interfaccia che gli permetta di sfruttare facilmente quelle più frequentemente utilizzate. Volendo fornire un esempio nella vita reale, un telecomando fornisce un'interfaccia semplice ai controlli della televisione, permettendo di regolare il volume e cambiare canale con semplicità: aprendo però uno sportellino ecco che ci vengono forniti tutti i comandi più specifici.

Composite

Immaginiamo di dover modellare un file system in un'applicazione: esso sarà composto di File e Directory, le quali dovranno essere in grado di contenere al loro interno File e ulteriori Directory; dovremo cioè ottenere una struttura ad albero di Directory avente dei File come foglie. Se però molte funzionalità del file system operano in modo analogo sia sui File che sulle Directory (*es. creazione, cancellazione, ottenimento della dimensione etc.*), come possiamo gestire queste due classi in modo uniforme per evitare di duplicare il codice?

Per gestire simili strutture ad albero che rappresentano *insiemi e gerarchie di parti* viene introdotto il pattern **Composite**: esso mira a gestire oggetti singoli, gruppi e persino gruppi di gruppi in maniera uniforme e trasparente in modo che un client non interessato alla struttura gerarchica possa utilizzarli senza accorgersi delle differenze.



Abbiamo quindi gli oggetti singoli, rappresentati dalla classe *Leaf*, e gli oggetti composti rappresentati dalla classe *Composite*. Per realizzare l'uniformità di gestione dobbiamo introdurre un livello di astrazione, quindi Leaf e Composite implementano una stessa **interfaccia Component** contenente la definizione delle operazioni comuni.
L'uso dell'interfaccia comune permette di definire all'interno di Composite le operazioni di aggiunta e rimozione di oggetti al gruppo in modo generale, permettendo cioè che un *Composite aggreghi sia Leaf che altri Composite*.

A proposito di tale aggregazione, dallo schema UML possiamo notare le relative cardinalità: "0..n" dal lato del Composite e "0..1" da quello del Component. Esse indicano che:

- Un'istanza di Composite aggrega 0 più istanze di Component al suo interno: in questo modo si permette che al momento della creazione il Composite sia totalmente vuoto; se questo non ha alcun senso logico nell'applicazione si può invece modificare la cardinalità in "1..n" imponendo che al costruttore di Composite venga passato un Component iniziale da contenere;

- Un'istanza di Component può essere contenuta in al più un'istanza di Composite: può cioè essere libero o aggregato in un gruppo, ma non può appartenere contemporaneamente a più gruppi, cosa che forza una struttura strettamente ad albero.

Nella maggior parte dei casi un'istanza Composite utilizzerà gli oggetti aggregati per implementare effettivamente i metodi descritti dall'interfaccia comune, delegando a loro l'esecuzione effettiva e limitandosi ad elaborare i risultati. Riprendendo l'esempio di prima, per conoscere la dimensione di una Directory sarà sufficiente sommare le dimensioni dei File e delle altre Directory in essa contenuti.

Il patter Composite presenta numerosi vantaggi, ma non è nemmeno esente da criticità. L'uso di un'interfaccia comune per Leaf e Composite permette al client di non preoccuparsi del tipo dell'oggetto con cui sta interagendo, in quanto ogni Component è in grado di eseguire le operazioni descritte nell'interfaccia in modo indistinguibile; tuttavia, questo implica che non è possibile distinguere tra oggetti singoli e composti.
Inoltre, l'uso dell'interfaccia per l'aggregazione nei Composite rende impossibile imporre dei controlli su cosa possa contenere un certo tipo di Composite: non si può per esempio forzare che raggruppi solo certi tipi di elementi, o che l'albero di composizione abbia profondità al più pari a tre.

Un "dialetto" del pattern tenta di risolvere il problema dell'indistinguibilità tra Leaf e Composite introducendo nell'interfaccia Component un metodo `getComposite` che in un Composite restituisca `this` e in una Leaf restituisca `null`. L'uso di valori nulli e la necessità di strani casting rende però pericolosa l'adozione di questa versione del pattern.

Decorator

Immaginiamo di voler modellare con degli oggetti una grande varietà di pizze differenti sia per la base (*es. normale, integrale, senza glutine...*) che per gli ingredienti che vi si trovano sopra. Per ogni diversa varietà di pizza vorremmo ottenere un oggetto aderente a un'interfaccia comune `Pizza` il cui metodo `toString()` elenchi la base e gli ingredienti che la compongono.

Un primo approccio *statico* a questo problema consiste nel creare una gerarchia di classi che contenga una classe per ogni possibile combinazione di base e ingredienti, che d'ora in avanti chiameremo "**decorazioni**".

```
public interface Pizza {}

public class BaseNormale implements Pizza {
    public String toString() {
        return "Sono una pizza con: base normale";
    }
}

public class BaseIntegrale implements Pizza {
    public String toString() {
        return "Sono una pizza con: base integrale";
    }
}

public class BaseNormaleSalame extends BaseNormale {
    public String toString() {
        return "Sono una pizza con: base normale, salame";
    }
}

public class BaseNormaleSalamePeperoni extends BaseNormaleSalame {
    public String toString() {
        return "Sono una pizza con: base normale, salame, peperoni";
    }
}

...
```

Come è subito ovvio, però, questo approccio risulta assolutamente da evitare per una serie di motivi: in primo luogo l'esplosione combinatoria dovuta all'accoppiamento di ogni possibile base e insieme di decorazioni, e in secondo luogo l'estrema difficoltà che comporterebbe una futura aggiunta di decorazioni.

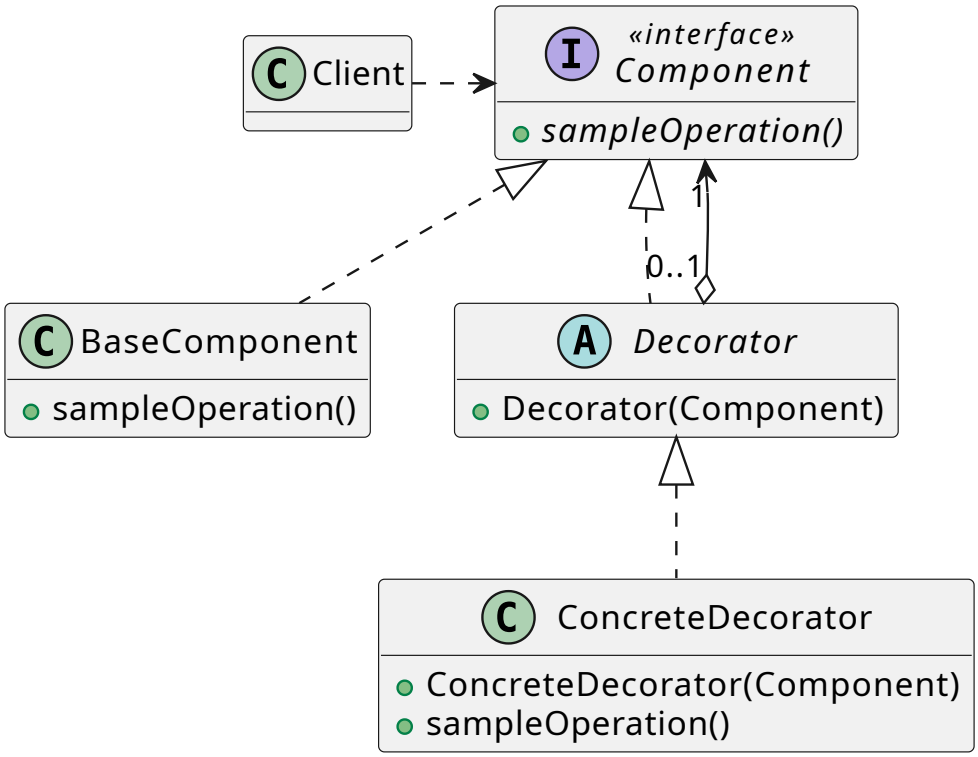
L'ideale sarebbe invece poter **aggiungere funzionalità e caratteristiche dinamicamente**, restringendo la gerarchia ad un'unica classe le cui istanze possano essere "decorate" su richiesta al momento dell'esecuzione.

La soluzione più semplice a questo nuovo problema parrebbe quella che viene definita una GOD CLASS (o *fat class*), ovvero un'unica classe in cui tramite attributi booleani e `switch` vengono attivate o disattivate diverse decorazioni.

```
public class GodPizza {

    boolean baseNormale = false;
    boolean baseIntegrale = false;
    ...

    boolean salame = false;
    boolean pancetta = false;
    boolean peperoni = false;
    ...

    public void setBaseNormale(boolean status) { baseNormale = status; }
    public void setBaseIntegrale(boolean status) { baseIntegrale = status; }
    ...

    public void setSalame(boolean status) { salame = status; }
    public void setPancetta(boolean status) { pancetta = status; }
    public void setPeperoni(boolean status) { peperoni = status; }
    ...

    public String toString() {
        StringBuilder sb = new StringBuilder("Sono una pizza con: ");
        if (baseNormale) sb.append("base normale, ");
        if (baseIntegrale) sb.append("base integrale, ");
        ...
        if (salame) sb.append("salame, ");
        if (pancetta) sb.append("pancetta, ");
        if (peperoni) sb.append("peperoni, ");
        ...
        sb.removeCharAt(sb.length() - 1);
        sb.removeCharAt(sb.length() - 1);
        return sb.toString();
    }
}
```

Si tratta però questo di un chiaro anti-pattern, una soluzione che sebbene invitante e semplice in un primo momento da realizzare nasconde delle criticità non trascurabili. Si tratta infatti di una chiara violazione dell'Open-Close Principle, in quanto per aggiungere un decoratore è necessario modificare la God Class; inoltre, tale classe diventa molto velocemente gigantesca, zeppa di funzionalità tra loro molto diverse (*scarsa separazione delle responsabilità*) e decisamente infernale da leggere, gestire e debuggare in caso di errori.

Introduciamo dunque il pattern **Decorator**, la soluzione più universalmente riconosciuta per questo tipo di situazioni.



A prima vista lo schema UML ricorda molto quello del pattern Composite: abbiamo un'interfaccia *Component* implementata sia da un *ConcreteComponent*, ovvero una base della pizza nel nostro esempio, sia da una **classe astratta Decorator**, la quale è poi estesa da una serie di *ConcreteDecorator*. A differenza del Composite, tuttavia, qui ciascun Decorator aggrega **una e una sola istanza di Component**: tali decoratori sono infatti dei "wrapper", degli oggetti che *ricoprono* altri per aumentarne dinamicamente le funzionalità.
È importante notare che i Decorator ricevono come oggetto da ricoprire al momento della costruzione un *generico Component*, in quanto questo permette ai decoratori di decorare oggetti già decorati. Questo approccio "ricorsivo" permette di creare una catena di decoratori che definisca a runtime in modo semplice e pulito oggetti dotati di moltissime funzionalità aggiunte. I decoratori esporranno infatti i metodi definiti dall'interfaccia **delegando** al Component contenuto l'esecuzione del comportamento principale e aggiungendo la propria funzionalità a posteriori: in questo modo la "base" concreta eseguirà il proprio metodo che verrà successivamente arricchito dai decoratori in maniera del tutto trasparente al Client.

```
public interface Pizza { String toString(); }

public class BaseNormale implements Pizza {
    public String toString() {
        return "Io sono una pizza con: base normale";
    }
}

public class BaseIntegrale implements Pizza {
    public String toString() {
        return "Io sono una pizza con: base integrale";
    }
}

public abstract class IngredienteDecorator implements Pizza {
    private Pizza base;

    public IngredienteDecorator(Pizza base) { this.base = base; }

    public String toString() {
        return base.toString();
    }
}

public class IngredienteSalame extends IngredienteDecorator {
    public IngredienteSalame(Pizza base) { super(base); }

    @Override
    public String toString() { return super.toString() + ", salame"; }
}

public class IngredientePeperoni extends IngredienteDecorator {
    public IngredientePeperoni(Pizza base) { super(base); }

    @Override
    public String toString() { return super.toString() + ", peperoni"; }
}
```

```
public class Client {
    public static void Main() {
        // Voglio una pizza con salame, peperoni e base integrale
        Pizza salamePeperoni =
            new IngredientePeperoni(
                new IngredienteSalame(
                    new BaseIntegrale()
                )
            );
    }
}
```

Vista la somiglianza, inoltre, pattern Decorator e Composite sono facilmente combinabili: si può per esempio immaginare di creare gruppi di oggetti decorati o decorare in un solo colpo gruppi di oggetti semplicemente facendo in modo che Composite, Decorator e classi concrete condividano la stessa interfaccia Component.

Possiamo poi notare una cosa: al momento della costruzione un Decorator salva al proprio interno l'istanza del Component da decorare. Come sappiamo questo darebbe luogo ad un'*escaping reference*, ma in questo caso il comportamento è assolutamente voluto: dovendo decorare un oggetto è infatti sensato pensare che a quest'ultimo debba essere lasciata la possibilità di cambiare e che debba essere il decoratore ad adattarsi a tale cambiamento.

È interessante poi osservare la classe astratta Decorator: in essa viene infatti inserita tutta la logica di composizione, permettendo così di creare nuovi decoratori con estrema facilità. Spesso, inoltre, se i decoratori condividono una certa parte di funzionalità aggiunte queste vengono anch'esse estratte nella classe astratta creando invece un metodo vuoto protetto che i decoratori reimplementeranno per operare la loro funzionalità aggiuntiva.

```
public abstract class IngredienteDecorator implements Pizza {
    private Pizza base;

    public IngredienteDecorator(Pizza base) { this.base = base; }

    public String toString() {
        return base.toString() + nomeIngrediente();
    }

    protected String nomeIngrediente() { return ""; }
}

public class IngredienteSalame extends IngredienteDecorator {
    public IngredienteSalame(Pizza base) {super(base);}

    @Override
    public String nomeIngrediente() { return ", salame"; }
}

public class IngredientePeperoni extends IngredienteDecorator {
    public IngredientePeperoni(Pizza base) {super(base);}

    @Override
    public String nomeIngrediente() { return ", peperoni"; }
}
```

Si noti come l'uso della visibilità `protected` renda l'override del metodo possibile anche al di fuori del package, aumentando così la facilità di aggiunta di nuovi decoratori.

Volendo vedere un esempio concreto di utilizzo di questo pattern è sufficiente guardare alla libreria standard di Java: in essa infatti gli `InputStream` sono realizzati seguendo tale schema.

State

Come sappiamo, le macchine a stati finiti sono uno dei fondamenti teorici dell'informatica: si tratta di oggetti matematici che modellano sistemi in grado di evolvere, ovvero il cui **comportamento varia in base allo stato** in cui si trovano.

Volendo rappresentare un oggetto di questo tipo la prima idea potrebbe essere quella di realizzare il cambio di comportamento con una serie di `if` e `switch`, un approccio che come abbiamo già visto numerose volte diventa presto difficilmente sostenibile.
In alternativa ad esso si introduce invece lo **State pattern** che mantenendo l'astrazione delle macchine a stati finiti permette di modellare facilmente il cambiamento di comportamento di un oggetto al modificarsi dello stato. Si noti che rimanendo legato al concetto di automa a stati finiti uno dei punti di forza di questo pattern è la semplicità di apportare delle modifiche al codice quando le specifiche di ciò che è stato modellato tramite una macchina a stati finiti cambiano.

Un esempio di utilizzo di questo pattern potrebbe essere un software di editing di foto, in cui l'utente ha a disposizione una toolbar con diversi strumenti che gli permettono di compiere operazioni diverse sullo stesso piano di lavoro (*comportamenti diversi dell'azione "tasto sinistro sullo schermo" in base al tool selezionato*).

PlantUML rendering error: Failed to render inline diagram (Failed to generate PlantUML diagrams, PlantUML exited with code 200 (Error line 2 in file: /tmp/.tmpqshJ6N/d02ac340bfb6a18f9490b436c5f5e184a6512bb3.puml Some diagram description contains errors).).

In un automa a stati finiti le componenti fondamentali sono tre:

- gli *stati*, tra cui si distingue lo *stato corrente*;
- le *azioni* che si possono intraprendere in qualunque stato;
- le *transizioni* da uno stato all'altro come effetto ulteriore di un'azione (*es. vim che con 'i' entra in modalità inserimento se era in modalità controllo*).

Come si vede dallo schema UML, il pattern State cerca di modellare ciascuna di queste componenti: un'**interfaccia State** raggruppa la definizione di tutte le *azioni*, rappresentate da metodi, mentre **una classe concreta per ogni stato** definisce che effetto hanno tali azioni quando ci si trova al suo interno con l'implementazione dei suddetti metodi.
Infine, una classe **Context** contiene un riferimento ad uno stato che rappresenta lo *stato corrente* e delega ad esso la risposta alle azioni (che possono essere viste come degli "eventi"); essa espone inoltre un metodo `setState(State)` che permette di modificare lo stato corrente.

```
public class Context {
    private State state;

    public void setState(@NotNull State s) {
        state = s;
    }

    public void sampleOperation() {
        state.sampleOperation(this)
    }
}
```

Rimane dunque solo da definire come si realizzano le *transizioni* di stato: chi ha la responsabilità di cambiare lo stato corrente? Esistono due diversi approcci, ciascuno dei quali presenta delle criticità:

- **gli State realizzano le transizioni**: volendo rimanere aderenti al modello degli automi a stati finiti, possiamo permettere che gli stati concreti chiamino il metodo `setState` del Context all'interno della loro implementazione dei metodi se come effetto di un'azione lo stato corrente cambia. Tuttavia, poiché `setState` chiede in input lo stato a cui transizionare questo approccio richiede che *gli stati si conoscano tra di loro*: si introduce così una *dipendenza tra stati* non chiaramente visibile nello schema UML e si ha uno *sparpagliamento della conoscenza* sulle transizioni che rende questo metodo un po' "sporco".

- **il Context realizza le transizioni**: con questa seconda strategia è compito del contesto eseguire le transizioni di stato, evitando così che gli stati si debbano conoscere; l'unico depositaria della conoscenza sulle transizioni è la classe Context. Ciascuna azione viene dunque intrapresa in due step: il Context richiama il corrispondente metodo dello stato corrente e successivamente ne **intercetta il risultato**; può dunque decidere tramite esso se cambiare stato e eventualmente a quale stato transizionare.
  Si tratta tuttavia di un ritorno al *table-driven design* fatto di `if` e `switch` da cui ci eravamo voluti allontanare: come in quel caso, l'approccio risulta fattibile soltanto finché ci sono poche possibili transizioni. Inoltre, se una transizione non dipende dal risultato di un'azione ma da *come* questa è stata eseguita questo approccio è totalmente impossibile in quanto tale tipo di conoscenza non è presente nella classe Context.

Per via delle difficoltà poste dal secondo approccio si sceglie spesso di effettuare le transizioni all'interno degli stati: questo permette di rendere esplicito e atomico il passaggio di stato.
A tal proposito, è interessante notare come le istanze degli stati concreti non posseggano alcuna informazione di stato in quanto il Context a cui si riferiscono viene passato loro al momento della chiamata dei rispettivi metodi: al di là della loro identità essi sono completamente **stateless**. Si tratta di un approccio molto utile in caso si debbano modellare

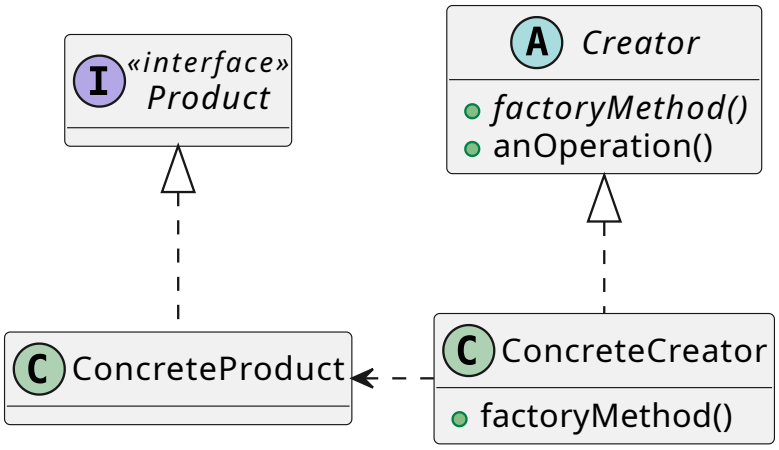
più macchine a stati finiti dello stesso tipo, in quanto l'assenza di stato rende le stesse istanze degli stati concreti **condivisibili** tra diversi Context, in una sorta di pattern Singleton.

Volendo trovare ulteriori analogie con altri pattern, il pattern State ricorda nello schema il pattern Strategy: la differenza sta però nel fatto che i diversi stati concreti sono a conoscenza l'uno dell'altro, mentre le strategie erano tra di loro completamente indipendenti.

Factory method

Talvolta capita che un certo Client sia interessato a creare un oggetto non in base al suo tipo quanto all'*interfaccia* che esso implementa: ad esso non importa conoscere la classe di cui l'oggetto è un'istanza perché essa non ha alcuna rilevanza nel suo contesto. Tuttavia, la normale creazione di un oggetto tramite la keyword `new` richiede di esplicitare la classe a cui esso appartiene, costringendo così il Client ad approfondire inutilmente la sua conoscenza sui tipi che implementano l'interfaccia a cui è interessato.

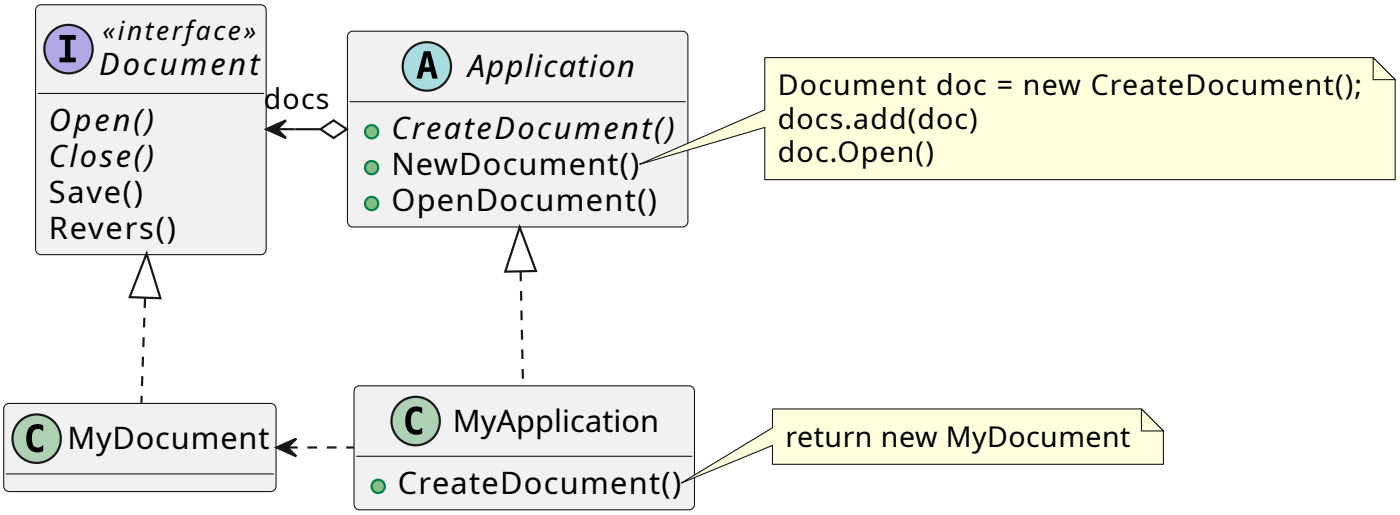
Per evitare questo tipo di situazione introduciamo uno dei cosiddetti **pattern creazionali**, ovvero legati alla creazione di oggetti: stiamo parlando del pattern dei **Factory methods**. Esso definisce una classe astratta *Creator* dotata di *metodi fabbrica* astratti che restituiscono un'istanza di un tipo aderente all'interfaccia *Product* a cui il Client è interessato: a quale classe appartenga effettivamente tale istanza (*Product concreto*) è però lasciato ad un *Creator concreto* tra i tanti che estendono la classe astratta; idealmente dovrebbe esserci un creatore concreto per ogni tipo di prodotto concreto che implementa l'interfaccia Product.



Questo pattern definisce dunque un'**interfaccia per creare un Product ma lascia al Creator concreto la scelta di cosa creare effettivamente**: in questo modo all'interno della classe astratta Creator è possibile scrivere dei metodi che richiedono la creazione di un Product pur senza sapere di preciso il tipo dell'oggetto che verrà creato, in quanto questo sarà determinato dall'implementazione di `factoryMethod` del creatore concreto. Si sfruttano dunque al massimo grado **polimorfismo** e **collegamento dinamico**, in quanto il tipo dell'oggetto da creare viene deciso a runtime: poiché nemmeno il Creator conosce il tipo concreto dei Product creati risulta

dunque subito chiaro perché i factory methods non possano essere metodi statici di tale classe.
I factory methods rappresentano un esempio dell'utilità delle astrazioni permesse dai linguaggi ad oggetti: in un contesto in cui normalmente non è possibile fare overriding, come un costruttore, la soluzione è quella di virtualizzare il tutto con la creazione di metodi che possono essere esportati in classi concrete. Per questo motivo i factory method vengono talvolta detti anche *virtual constructors*, "costruttori virtuali".

Per capire meglio il funzionamento del pattern, vediamo un esempio di come esso può essere utilizzato. Consideriamo un software capace di aprire contemporaneamente più documenti di tipo differente in diverse pagine, come per esempio Microsoft Word o Excel: al loro interno, quando viene creato un nuovo file vengono fatte una serie di operazioni generiche (creare la nuova pagina, mostrare vari popup...), ma ad un certo punto è necessario creare un oggetto che rappresenti il nuovo documento e il cui tipo dipende dunque dal documento creato. Il codice di creazione del nuovo oggetto `Documento` non può dunque trovarsi in un metodo della classe astratta `Application` (Creator) insieme con il resto delle operazioni generiche in quanto specifico della tipologia di documento creato: è dunque necessario virtualizzare la creazione dell'oggetto in un metodo `createDocument()` implementato da una serie di sottoclassi concrete `MyApplication` (ConcreteCreator) ciascuna specifica per un tipo di documento.



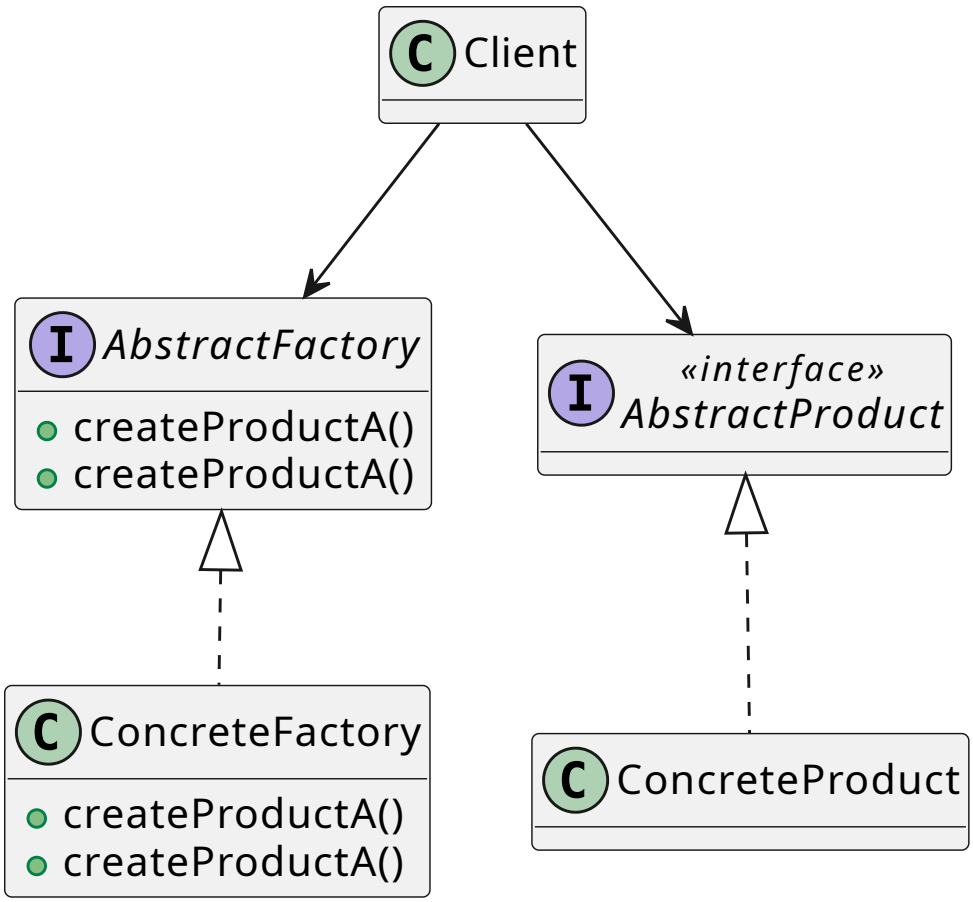
Abstract factory

Vediamo ora una generalizzazione del Factory method pattern che si utilizza quando, al posto di creare un solo oggetto aderente ad un'interfaccia, è necessario creare *più oggetti aderenti a varie interfacce i cui tipi concreti siano però compatibili tra di loro*.

Immaginiamo per esempio di aver progettato un'applicazione cross-platform e di doverne creare la User Interface: essa dovrà avere stili diversi in base al sistema operativo sui cui si sta eseguendo. Non conoscendo su quale os si starà operando, il resto dell'applicazione gestirà gli

elementi dell'UI tramite delle opportune interfacce che nascondano il tipo concreto delle istanze, il quale determinerà lo stile con cui esse verranno rappresentate: sarà però fondamentale che *tutti gli elementi dell'UI condividano lo stesso stile* in modo da non creare un'orrendo arlecchino.

Ecco dunque che introduciamo il pattern delle **Abstract Factory**, un metodo in grado di fornire un'**interfaccia per creare famiglie di oggetti compatibili tra loro senza specificare la loro classe concreta** così da garantire una certa **omogeneità** all'insieme.
Per fare ciò il pattern propone di creare un'interfaccia *AbstractFactory* contenente la definizione di un factory method per ogni tipo di prodotto astratto (*Product*) e una serie di *ConcreteFactory* che restituiranno dei *ConcreteProduct* in uno specifico stile: in questo modo, interagendo con una Factory concreta un Client potrà ottenere in modo a lui trasparente una serie di prodotti concreti coerenti in stile tra di loro.



Tornando al problema della User Interface, volendo sfruttare l'Abstract Factory pattern dobbiamo creare un'interfaccia `GUIFactory` che contenga la dichiarazione di due metodi creazionali, `createButton()` e `createCheckbox()`: questi permetteranno al client di creare un bottone e una checkbox nello stile specificato dalla classe concreta della factory; per ciascuno di tali elementi dell'UI dobbiamo dunque creare un'interfaccia prodotto, ovvero rispettivamente le interfacce `Button` e `Checkbox`. All'interno delle classi factory concrete tali metodi creazionali restituiranno però dei prodotti concreti nello stile specifico della factory da cui sono prodotti: così, per esempio, una `MacFactory` (per lo stile di MacOs) creerà `MacButton` e `MacCheckbox`,

mentre una `WinFactory` (per lo stile di Windows) creerà `WindowsButton` e `WinCheckbox`.
In questo modo la nostra applicazione dovrà possedere al suo interno unicamente un riferimento alla factory adatta al sistema operativo su cui sta girando e potrà creare tramite essa tutti gli elementi di UI di cui avrà bisogno senza preoccuparsi di specificare ogni volta lo stile: la factory concreta glielo restituirà sempre nello stile selezionato inizialmente.

 Esempio abstract factory

MODEL VIEW CONTROLLER

Spesso nelle applicazioni capita che uno stesso dato sia riportato tramite diverse **viste** all'interno dell'interfaccia utente: il colore di un testo, per esempio, potrebbe essere rappresentato contemporaneamente da una terna di valori RGB, dal suo valore esadecimale e da uno slider di colori. Si tratta di modi differenti di rappresentare la medesima **informazione condivisa**, che viene replicata più volte per dare all'utente diversi modi in cui visualizzarla.
La condivisione di un medesimo valore porta però con sé un problema: se tale dato viene modificato dall'utente interagendo con una delle viste è necessario che tale *modifica venga propagata a tutte le altre viste* in modo da mantenere l'informazione **coerente**.

Abbiamo dunque bisogno di un framework che ci permetta di mantenere un'informazione condivisa in modo efficiente e pulito e che permetta di rappresentarla facilmente sotto diversi punti di vista: l'invitante soluzione di fare semplicemente sì che le viste comunichino direttamente i cambiamenti del dato l'una con l'altra si rivela infatti velocemente impraticabile.
Il pattern **Model View Controller** (MVC) propone invece di suddividere la gestione del dato e dell'interazione con l'utente in tre tipologie di classi:

- **Model**: un'unica classe contenente lo **stato condiviso**; si tratta dell'unico depositario dell'informazione con cui tutte le viste dovranno comunicare per aggiornare i dati mostrati.
- **View**: una serie di classi che costituiscono l'**interfaccia con l'utente**; esse mostrano il dato secondo il loro specifico punto di vista e permettono all'utente di interagire con l'applicazione.
- **Controller**: ciascuna vista possiede infine una classe di controllo collegata che si occupa della **logica dell'applicazione**; ogni volta che l'utente interagisce con una vista tale interazione viene passata al relativo Controller, che si occuperà di rispondere all'input eventualmente modificando lo stato condiviso nel Model.

Abbiamo dunque una suddivisione dell'applicazione in tre tipi di componenti differenti che cooperano tra di loro senza però essere *strettamente* dipendenti l'uno dall'altro. Un tipico ciclo di interazione tra le tre componenti funziona infatti come mostrato in figura:

1. Una View riceve un'interazione da parte dell'utente e comunica tale evento al proprio Controller;
2. Il Controller gestisce l'interazione e se essa richiede un cambiamento dello stato comune chiede al Model di modificare il proprio contenuto;
3. Come ulteriore passaggio, il Controller aggiorna il dato mostrato dalla View ad esso associata prima ancora che il modello sia cambiato;
4. Ricevuta la richiesta, il Model aggiorna l'informazione condivisa e notifica *tutte* le View del cambiamento: in questo modo esso non avrà effetto solo nella vista che ha ricevuto l'input dell'utente ma in tutte;
5. Le View ricevono la comunicazione del fatto che il Model è cambiato e aggiornano la propria informazione mostrata recuperando il dato aggiornato dal modello (politica *pull*).

Questo modello di interazione circolare permette di separare l'interfaccia utente (*view*) dall'interfaccia dello stato comune (*model*) e dalla logica del cambiamento di stato (*controller*): grazie alla mediazione del Controller le View non hanno bisogno di conoscere direttamente la struttura dei dati contenuti nel Model, cosa che ci permette di riutilizzare le stesse View, e dunque le stesse interfacce utente, per dati diversi (es. una casella di testo è una View e non dipende dal dato che ci si inserisce).
È inoltre interessante notare come un Controller potrebbe voler comunicare dei *cambiamenti virtuali* alla View da cui è partito un input prima ancora che al Model venga chiesta un eventuale modifica dello stato. Nel caso ci siano errori nell'input inserito dall'utente, infatti, esso va informato in qualche modo: il Controller non cambierà dunque lo stato condiviso ma solo lo stato dalla relativa View in modo da mostrare un qualche messaggio d'errore. Similmente, se i dati inseriti sono già presenti nel Model (cosa che il Controller non può sapere a priori) quest'ultimo potrebbe avvisare il Controller di tale evenienza al momento della richiesta di cambiamento: esso dovrà dunque nuovamente notificare l'utente che l'inserimento dei dati non è andato a buon fine aggiornando la propria View.

Portiamo ora attenzione su un altro aspetto: nell'insieme dei meccanismi che realizzano il pattern Model View Controller si possono riscontrare una serie di altri pattern che abbiamo già trattato. Per agevolare la comprensione del funzionamento di questo nuovo "mega-pattern", vediamo quindi quali sono i pattern utilizzati al suo interno:

- **Observer**, poiché *le View sono Observer del Model*: ogni vista si registra come Observer del modello in modo che il Model, in pieno stile Observable, le notifichi dei suoi cambiamenti di stato. Spesso la strategia di aggiornamento delle viste è qui quella **pull**, ovvero quella secondo cui agli Observer viene passato un riferimento all'oggetto Observable in modo che siano loro stessi a recuperare i dati di cui hanno bisogno tramite opportuni metodi getter: questo permette infatti di memorizzare nello stesso Model i dati di diverse View.
  Va inoltre fatto notare che se l'interfaccia esposta dalle View è un'*interfaccia a eventi*, come per esempio un'interfaccia grafica (es. un click sullo schermo genera un evento), *anche la*

comunicazione tra View e Controller può avvenire tramite il pattern Observer: ciascun Controller si registra infatti come Observer degli eventi che avvengono sulla View.
- **Strategy**, poiché *i Controller sono Strategy per le View*: poiché ad ogni vista è collegato uno e un solo Controller che regola come la vista reagisca agli input dell'utente, i Controller possono essere visti come strategie di gestione degli eventi generati dalle viste. Poiché le viste sono componenti sostanzialmente "stupidi" che risolvono le interazioni dell'utente delegando al proprio Controller la loro gestione, questo approccio permette per esempio di gestire viste identiche in modi diversi semplicemente cambiando il Controller ad esse associato: così, per esempio, è possibile rendere una casella di testo read-only oppure modificabile senza modificare in alcun modo la classe della relativa vista e rispettando così l'Open-Close Principle.
- **Composite**, poiché *le View sono spesso composte da più Component*: quando le View rappresentano interfacce grafiche (GUI) esse sono spesso realizzate componendo diversi elementi tra di loro (es. aree di testo, bottoni, etc...). Per questo motivo è spesso prevalente il pattern Composite nella loro implementazione, utile specialmente per quanto riguarda la creazione su schermo dell'interfaccia, che viene disegnata pezzo per pezzo.

In conclusione, il Model è in grado di interagire con tutte le viste che l'osservano tramite un unico comando (*update*), mentre le View comunicano con il Model passando attraverso il Controller, che fa da una sorta di "Adapter" tra i due. Questo permette allo stesso dato di avere interfacce disomogenee senza alcun tipo di problema riguardante la coerenza dello stesso.

Tuttavia, il problema principale del pattern Model View Controller è la *dipendenza circolare* tra le tre componenti: le view comunicano ai rispettivi controller gli eventi, questi li elaborano e aggiornano il modello il quale a sua volta avvisa le view dei cambiamenti di stato. Questa struttura fortemente interconnessa rende difficoltoso lo sviluppo e il testing in quanto non esiste un chiaro punto da cui partire a costruire: si potrebbe pensare di fare mocking delle view e iniziare a sviluppare il resto, ma questo approccio porta comunque a una serie di inutili complicazioni; bisogna inoltre considerare che il testing delle view è spesso particolarmente complesso coinvolgendo varie funzioni di libreria o funzioni grafiche.
Come vedremo nel prossimo paragrafo, per ovviare a questo problema si decide spesso di spezzare il circolo vizioso di Model, View e Controller modificando lievemente le rispettive dipendenze.

Model view presenter

Come preannunciato esiste una variante del Model View Controller chiamata **Model View Presenter** che fornisce una soluzione al problema del testing delle viste e delle relative interfacce grafiche. Questo nuovo pattern eleva il ruolo del Controller, ora chiamato *Presenter*, a completo *intermediario tra View e Model* in entrambi i sensi di comunicazione: non solo dunque

le View delegano ai rispettivi Presenter la gestione delle interazioni con l'utente, ma al momento del cambiamento dell'informazione condivisa il Model notifica non direttamente le viste ma i Presenter stessi, i quali avranno dunque il compito di aggiornare la propria View per mostrare il dato modificato.


MVP

Model e View perdono dunque alcun legame diretto, facendo apparire sempre più i Presenter come Adapter tra stato concreto (*model*) e stato virtuale mostrato all'utente (*view*). La rottura di tale legame facilita il testing delle viste poiché invece di verificare che una vista e la rispettiva controparte grafica abbiano ricevuto e processato correttamente un aggiornamento del dato da parte del Model è sufficiente verificare che un update del Model provochi nei Presenter un aggiornamento del dato mostrato dalla propria View: siamo dunque riusciti a **isolare l'interfaccia logica da quella grafica**, rendendo più semplice il testing di entrambe e sfoggiando un esempio importante del cosiddetto *design for testing*.

In ultimo, utilizzando questo pattern è importante fare attenzione di mantenere segreta la rappresentazione interna del Model ai Presenter e viceversa, evitando in particolar modo eventuali *escaping reference*: la separazione delle responsabilità costruita con la suddivisione dei dati dalla loro logica di gestione perderebbe infatti alcuna valenza se si legassero troppo strettamente Model e Presenter.

Builder

Può talvolta capitare che l'inizializzazione di un'istanza di una classe richieda un numero molto grande di *parametri*, alcuni dei quali *obbligatori* e altri *facoltativi*. Come si realizzano i costruttori della classe in questo tipo di situazioni?

Telescoping constructor pattern

L'approccio più immediato a questo problema è quello dei **costruttori telescopici** (*telescoping constructor pattern*): all'interno della classe si realizza *un costruttore completo* che richiede tutti i parametri e una serie di *costruttori secondari* che invece prendono i parametri obbligatori e *diverse combinazioni dei parametri opzionali*, rimappando poi spesso la propria esecuzione sul costruttore completo tramite l'assegnamento di valori di default ai parametri non ricevuti.

```
public class MyClass {
    private final T0 optionalField1;
    private final T1 mandatoryField;
    private final T2 optionalField2;

    public MyClass(T1 mf) {
        this(defaultValue1, mf, defaultValue2);
    }

    public MyClass(T1 mf, T0 of) {
        this(of, mf, defaultValue2);
    }

    public MyClass(T1 mf, T2 of) {
        this(defaultValue1, mf, of);
    }

    public MyClass(T1 mf, T0 of1, T2 of2) {
        this.optionalField1 = of1;
        this.optionalField2 = of2;
        this.mandatoryField = mf;
    }
}
```

Questa tecnica si rivela però presto molto poco funzionale: innanzitutto il numero di costruttori da realizzare cresce esponenzialmente nel numero di parametri opzionali, rendendo la classe estremamente confusionaria.
Sorgono inoltre dei problemi nel caso di *parametri opzionali dello stesso tipo*, in quanto è impossibile disambiguare tra di essi al momento della definizione dei costruttori: con due parametri opzionali dello stesso tipo, per esempio, non sarebbe possibile distinguere il costruttore che assegni il primo ma non il secondo e viceversa (si noti come non si può nemmeno distinguere tramite il nome del costruttore in quanto questo deve necessariamente essere lo stesso della classe). Se linguaggi come Python risolvono questo problema imponendo che il chiamante di un costruttore espliciti il nome del parametro opzionale che sta assegnando, questo tipo di meccanismo non esiste in Java: ciò ci costringerebbe quindi a far sì che nei costruttori vengano passati o tutti i parametri dello stesso tipo o nessuno di essi.

JavaBeans pattern

Per risolvere i problemi appena visti la prossima soluzione che viene in mente è dunque quella di fornire un *unico costruttore* che prenda in input *solamente i parametri obbligatori* e creare poi una serie di *setter per i parametri opzionali*: si tratta del cosiddetto **pattern JavaBeans**.

```
public class MyClass {
    private T0 optionalField1;
    private T1 mandatoryField;
    private T2 optionalField2;

    public MyClass(T1 mf) {
        this.mandatoryField = mf;
    }

    public void setOptionalField1(T0 of) {
        this.optionalField1 = of;
    }

    public void setOptionalField2(T2 of) {
        this.optionalField2 = of;
    }
}
```

Anche questo approccio presenta tuttavia delle notevoli difficoltà. In primo luogo, un oggetto costruito con il pattern JavaBeans *non può essere immutabile* in quanto richiede la presenza di setter per i propri attributi opzionali (che dunque non possono essere `final`): possiamo dunque creare solo oggetti mutabili.
Un problema forse più grave è inoltre che questo pattern ammette la presenza di *momenti nella vita di un oggetto in cui esso non è stato ancora costruito completamente*: tra la creazione e l'assegnamento tramite setter dei parametri opzionali, infatti, l'istanza si trova in uno stato non finito e dunque non consistente che potrebbe creare numerosi problemi in sistemi di tipo concorrente o multi-thread.

Builder pattern

Gli autori del libro Effective Java propongono dunque un nuovo pattern che prende gli aspetti migliori della prima e della seconda soluzione finora proposta risolvendo al tempo stesso i problemi di entrambe: essa permetterà infatti di creare oggetti immutabili (rendendo gli attributi `final`) e di assegnare solo alcuni dei parametri opzionali senza generare problemi di inconsistenza o di sovrapposizione dei tipi degli attributi. Questo pattern creazionale prende il nome di **Builder**.

PlantUML rendering error: Failed to render inline diagram (Failed to generate PlantUML diagrams, PlantUML exited with code 200 (Error line 2 in file: /tmp/.tmpqshJ6N/eed9b855eac2b441a4a0ce8c16d6fcd95026f49e.puml Some diagram description contains errors).).

Data una classe da costruire `MyClass` avente parametri obbligatori e opzionali il primo passo è quello di rendere **privato** il suo costruttore, il quale prenderà in input non più una lista di

parametri ma un'istanza di una **nuova classe** `Builder`. Tale classe viene definita come una *classe statica, pubblica e interna* a `MyClass`, con la quale condivide il tipo e il numero di attributi obbligatori e opzionali (questi ultimi subito inizializzati al loro valore di default).
Seguendo il pattern JavaBeans, la classe Builder esporrà un costruttore pubblico contenente solo i parametri obbligatori e una serie di setter per i parametri opzionali. Ma a che pro costruire un oggetto della classe Builder quando quella che volevamo ottenere era un'istanza di `MyClass`? La risposta sta nella definizione **metodo** `build()`: tramite esso, il Builder restituirà un'istanza di MyClass inizializzata con propri i parametri obbligatori e opzionali; essendo una classe interna, infatti, il Builder sarà l'unico in grado di accedere al costruttore privato di `MyClass`.

```
public class MyClass {
    private final T0 optionalField1;
    private final T1 mandatoryField;
    private final T2 optionalField2;

    private MyClass(Builder builder) {
        mandatoryField = builder.mandatoryField;
        optionalField1 = builder.optionalField1;
        optionalField2 = builder.optionalField2;
    }

    public static class Builder {
        private T1 mandatoryField;
        private T0 optionalField1 = defaultValue1;
        private T2 optionalField2 = defaultValue2;

        public Builder(T1 mf) {
            mandatoryField = mf;
        }

        public Builder withOptionalField1(T0 of) {
            optionalField1 = of;
            return this;
        }

        public Builder withOptionalField2(T2 of) {
            optionalField2 = of;
            return this;
        }

        public MyClass build() {
            return new MyClass(this);
        }
    }
}
```

Questo pattern è particolarmente intelligente per una serie di motivi: innanzitutto, rendendo privato il costruttore di `MyClass` ci si assicura che le sue istanze siano costruite unicamente

tramite il Builder. A tal proposito, il fatto che `Builder` sia una classe **statica** è di non poca importanza: questo permette infatti di creare una sua istanza senza prima istanziare la classe che la contiene, cosa che come abbiamo visto sarebbe impossibile essendo il costruttore di MyClass privato. Per creare un'istanza di Builder è dunque sufficiente la seguente sintassi:

```
MyClass.Builder = new MyClass.Builder(...);
```

Si potrebbe notare che essendo statica la classe Builder potrà accedere solamente agli elementi statici di `MyClass`, ma questo non costituisce un problema: come abbiamo visto, essa dovrà solamente richiamarne il costruttore, che per sua stessa natura è sempre statico.
È importante notare che non vale però il contrario: `MyClass`, una volta ricevuta un'istanza di Builder come parametro del costruttore, può benissimo accedere ai suoi campi privati e sfrutta questa possibilità per copiare i valori dei parametri obbligatori e opzionali che il Builder ha ricevuto all'interno dei propri attributi. Assegnando tali valori al momento della creazione, gli attributi di `MyClass` potranno quindi anche essere `final`, permettendo così la creazione di oggetti immutabili.

Un altro particolare da sottolineare è che i setter degli attributi opzionali del Builder sono setter un po' "spuri", in quanto invece di non ritornare nulla *ritornano il Builder stesso*: questo permette infatti di concatenare più setter l'uno con l'altro ottenendo così una notazione più fluente. È possibile infatti creare inline un'istanza di Builder, settare direttamente i suoi parametri opzionali e infine richiamare il metodo `build()` per ottenere facilmente un'istanza di `MyClass`:

```
MyClass inst = (new
MyClass.Builder(mandatoryField).withOptionalField1(optionalField1)).build();
```

L'utilizzo di un Builder risolve inoltre eventuali problemi dovuti alla concorrenza: quando viene chiamato il metodo `build()` l'istanza di `MyClass` viene restituita già completa, ovvero con tutti i parametri obbligatori e opzionali al valore desiderato (o di default se nessun setter è stato chiamato). Abbiamo così eliminato la possibilità di inconsistenze e creazioni parziali delle istanze di `MyClass`.

UML

UML (*Unified Modeling Language*) è un linguaggio di *modeling* il cui scopo è determinare uno standard comune nella rappresentazione visuale del software.

UML rappresenta in realtà una famiglia di formalismi, che si concretizzano nei concetti di **diagrammi**.

- **Class diagram**

Class diagram

Concetto e struttura

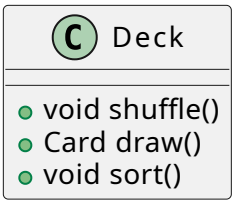
Lo scopo del **diagramma delle classi** è fornire una vista statica del software (una sorta di "fotografia") tramite la rappresentazione delle sue classi, corredate di metodi, attributi e relazioni.

I componenti identificabili in un diagramma delle classi sono:

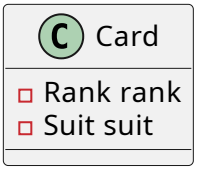
- **oggetti** (*Classi* e *Interfacce*), rispettivamente riconoscibili per le lettere "C" e "I" nella parte superiore di ogni blocco.

Esiste anche il marcatore "A", che rappresenta una classe astratta. Inoltre, per i diagrammi UML relativi a Java si può usare la lettera "E" per rappresentare le classi enum;

- **metodi**: preceduti da un cerchio e dal tipo di valore ritornato;



- **attributi**: preceduti da un quadrato, corrispondono agli attributi dell'oggetto;



- **relazioni**: frecce che connettono gli oggetti.

È possibile rappresentare il *cerchio* dei metodi e il quadrato degli attributi con colori diversi in base alla visibilità. In Java, ad esempio, si può usare il verde per la visibilità `public`, l'arancio per `protected` e il rosso per `private`.

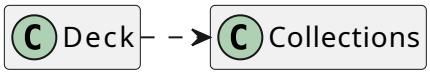
Valgono anche due regole sintattiche generali:

- se una scritta è in *corsivo* vuol dire che all'elemento corrispondente manca qualche definizione ed è dunque da considerarsi **astratto**;
- se una scritta è sottolineata vuol dire che l'elemento corrispondente (tipicamente metodo o attributo) è **statico**, ovvero ha una visibilità a livello di classe e non a livello di istanza (*i.e* è possibile riferircisi ad esso anche senza avere precedentemente istanziato la classe).

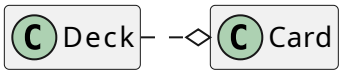
Relazioni

Nel diagramma delle classi UML esistono relazioni di diversi tipi. Ogni relazione viene rappresentata tramite una specifica forma di freccia:

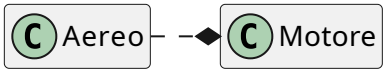
- **frecce tratteggiate** (***associazione***): sono le più generiche e indicano una relazione "gerarchica" tra classi. Ciò che c'è scritto nella classe da cui parte la freccia dipende dal codice che c'è nella classe a cui arriva la freccia (*e.g.* `Deck` dipende da `Collections`);



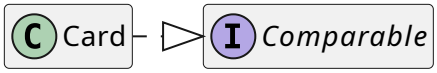
- **frecce con rombo bianco** (***aggregazione***): indica che all'interno della classe (*e.g.* `Deck`) è presente una collezione (in questo caso una lista) di (n) oggetti (`Card` nell'esempio). Questa relazione non è più tra classi, bensì tra *istanze* delle classi (*e.g.* un'istanza di `Deck` aggrega da 0 a 52 carte);



- **frecce con rombo nero** (***composizione***): è utilizzata quando si hanno degli elementi che sono *fisicamente* collegati tra loro (non solo virtualmente come nel caso delle carte). Senza l'uno l'altro non può vivere e viceversa.
  Un esempio può essere la rappresentazione del concetto di *aereo*: senza il *motore* l'aereo non può esistere, poichè il primo è un oggetto indispensabile per funzionamento del secondo. Specularmente, non accadrà mai che il motore passi ad un altro aereo (a differenza delle carte che possono passare a più mani).



- **frecce con la punta a triangolo** (***implementazione***): una classe può *implementare* una classe astratta o un'interfaccia.



La direzione delle frecce è importante perché indica il *senso* della relazione.
Per esempio, il mazzo *conosce* le carte che contiene, ma le carte *non conoscono* i mazzi di cui fanno parte – è per questo che la direzione della freccia va da `Deck` a `Card` e non viceversa.

Sequence diagram

Concetto e struttura

Lo scopo del **diagramma di sequenza** è rappresentare il flusso di interazione tra attori all'interno di un software nel tempo. Di seguito ne è indicato un esempio.

Si compone di alcuni elementi chiave:

- **attori**: rappresentano le entità coinvolte nel processo; spesso sono *oggetti*;
- **invocazioni**: identificano chiamate di metodo su un attore da parte di un altro e sono rappresentate con una freccia che va da *sinistra verso destra*.
  La parte a sinistra è il *chiamante* e la parte a destra è il *chiamato*;
- **valori di ritorno**: visualizzati tramite una freccia tratteggiata che va da *destra verso sinistra*;
- **cicli**: aree rettangolari etichettate con il termine `loop` che specificano la presenza di un ciclo in una certa zona del diagramma;
- **condizioni**: aree rettangolari etichettate con il termine `opt` che specificano la necessità di verificare alcune condizioni prima di entrare nella zona corrispondente.

State diagram

Concetto e struttura

L'obiettivo del **diagramma di stato** è fornire un'astrazione di comportamento significativa che sia comune all'intera classe.

La sua struttura deriva dai classici *State Charts*, dei quali costituisce un'ulteriore astrazione.

Al fine di comprendere meglio i diagrammi di stato, può essere utile ricordare che:

Negli *State Charts*, un automa è una sestupla $\langle S,\ I,\ U;\ \delta,\ t,\ s_0 \rangle$.

- S: insieme finito e non vuoto degli stati;
- I: insieme finito dei possibili ingressi;
- U: insieme finito delle possibili uscite;
- δ: funzione di transizione;
- t: funzione di uscita;
- s_0: stato iniziale.

Negli *State Diagram*, ogni *stato* è rappresentato da un rettangolo e lo *stato iniziale* è indicato da un pallino nero.

Nel diagramma le frecce possono avere diversi significati:

- **evento/azione**: semplice e immediata transizione da uno stato ad un altro;
- **guardie**: disambiguano le transazioni in uscita da uno stato che sono legate ad uno stesso evento;
- ***time event***: rappresentano eventi temporizzati.
  - *After(duration)*: indicano un tempo massimo di permanenza nello stato destinazione. Allo scadere del timer, lo stato cambia.

- ***change event***: rappresentano eventi che si innescano al verificarsi di un cambiamento.
  - *When(condition)*: indicano eventi espressi in termini di valori degli attributi.

Il verificarsi di eventi non esplicitamente marcati da un arco deve portare alla terminazione dell'esecuzione e al sollevamento di un errore.

Superstate

Ulteriore evoluzione dello State Diagram, il **Superstate** consente di rappresentare più facilmente una "gerarchia" di stati.

La transizione in uno stato può quindi condurre ad un'altra FSM concettualmente "innestata".

 Esempio superstate

Nel caso d'esempio, lo stato `acceso` possiede al suo interno un ulteriore diagramma di stato.

Ulteriori aggiunte

- è possibile associare al diagramma uno stato ***history***, il cui scopo è memorizzare lo stato storico prima dell'interruzione dell'FSM;
- è possibile rendere il diagramma capace di rappresentare il concetto di **concorrenza** tramite la divisione in **regioni** (ognuna regolata da una propria FSM). Le regioni possono essere attive contemporaneamente. I confini tra regioni, come mostrato nell'esempio, sono identificati da linee tratteggiate.


Esempio concorrenza

Use cases diagram

Concetto e struttura

I **diagrammi dei casi d'uso** rappresentano l'astrazione di un insieme di scenari tra loro correlati.
Essi adottano un linguaggio che verte alla risoluzione di esigenze comunicative tramite un lessico potenzialmente meno tecnico. Tale natura "informale" li rende ottimi mezzi di comunicazione col *cliente*.

Possono essere utilizzati, ad esempio, per:

- eplicitare differenti modalità di fare un compito;
- stabilire quale dovrebbe essere la normale interazione nello scenario e le eccezioni che possono verificarsi.

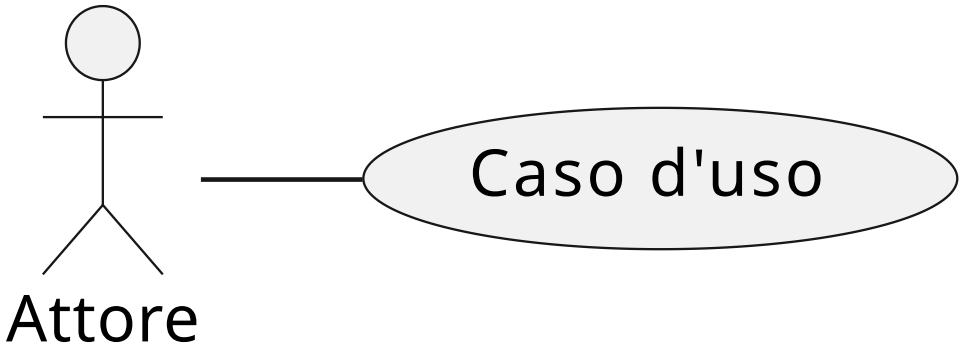
Infatti, ogni *scenario* è corredato di:

- **pre e post condizioni** da rispettare;
- **flusso di esecuzione** da percorrere in condizioni normali;
- eventuali **eccezioni** e loro possibili trattamenti.

Infine, parte della versatilità degli *Use Case diagrams* risiede nella loro capacità di collegarsi, eventualmente, ad altri tipi di diagrammi (*Sequence, Activity, etc*) che possono essere impiegati per descriverne in modo più approfondito il flusso.

Scenari

I componenti di ogni scenario si dividono in **Attori** e **Casi d'Uso**.



In generale il *collegamento* tra un attore e un caso d'uso rappresenta una **relazione di partecipazione**, *i.e* "Questo attore partecipa a questo caso d'uso".

L'interazione può comunque essere denominata.

Sono contemplati anche collegamenti fra un caso d'uso e un altro (vedi paragrafo dedicato).

Identificazione degli attori

Gli *attori* non sono necessariamente persone fisiche. Possono corrispondere anche a dei **ruoli** o addirittura ad un **sistema esterno**.

Ogni attore è un'entità esterna al sistema ed interagisce direttamente con esso, fungendo allo stesso tempo da *fonte* e *destinatario* di informazione.

Ci sono due attori particolari:

- **attore beneficiario**: colui che trae beneficio dall'interazione con lo use case, *i.e.* chi è **interessato** a quella funzionalità.
  Gli altri attori possono cambiare, ma il beneficiario rimmarrà probabilmente lo stesso;
- **attore primario**: colui che avvia l'interazione con lo use case.

Identificazione use case

Il miglior modo di identificare i casi d'uso è interrogarsi su due fronti:

- **sistema**: *"quali funzionalità si desidera che il sistema possieda?"*;

- **attori beneficiari**: *“cosa vogliono?”*, *“come agiscono?”*, *“perchè si interfacciano col sistema?”* e *“cosa si aspettano?”*.

Associazioni

Ogni diagramma dei casi d’uso deve seguire due convenzioni per quanto riguarda le associazioni.

- Ogni attore deve avere almeno un’interazione con un caso d’uso.

Un attore che non dovesse possedere alcuna associazione con un caso d’uso sarebbe impossibilitato a interagire col sistema e rappresentarlo nel diagramma non avrebbe alcun senso.

- Ogni caso d’uso deve essere associato ad almeno un attore.

Un caso d’uso che non coinvolge alcun attore è un caso d’uso che, per definizione, non ha senso di esistere, poichè nessuno è in grado di interagirvi.

Relazioni *use case - use case*

Esistono due tipologie di relazioni tra use case:

- **inclusione (*include*)**: relazione che esprime il predicato *“far parte di”*.
  Chi include conosce sempre gli inclusi, ma non viceversa. La parte inclusa *deve* essere eseguita;
- **estensione (*extend*)**: relazione che viene utilizzata per rappresentare casi eccezionali che specificano comportamenti particolari in alcuni use case.

Generalizzazione

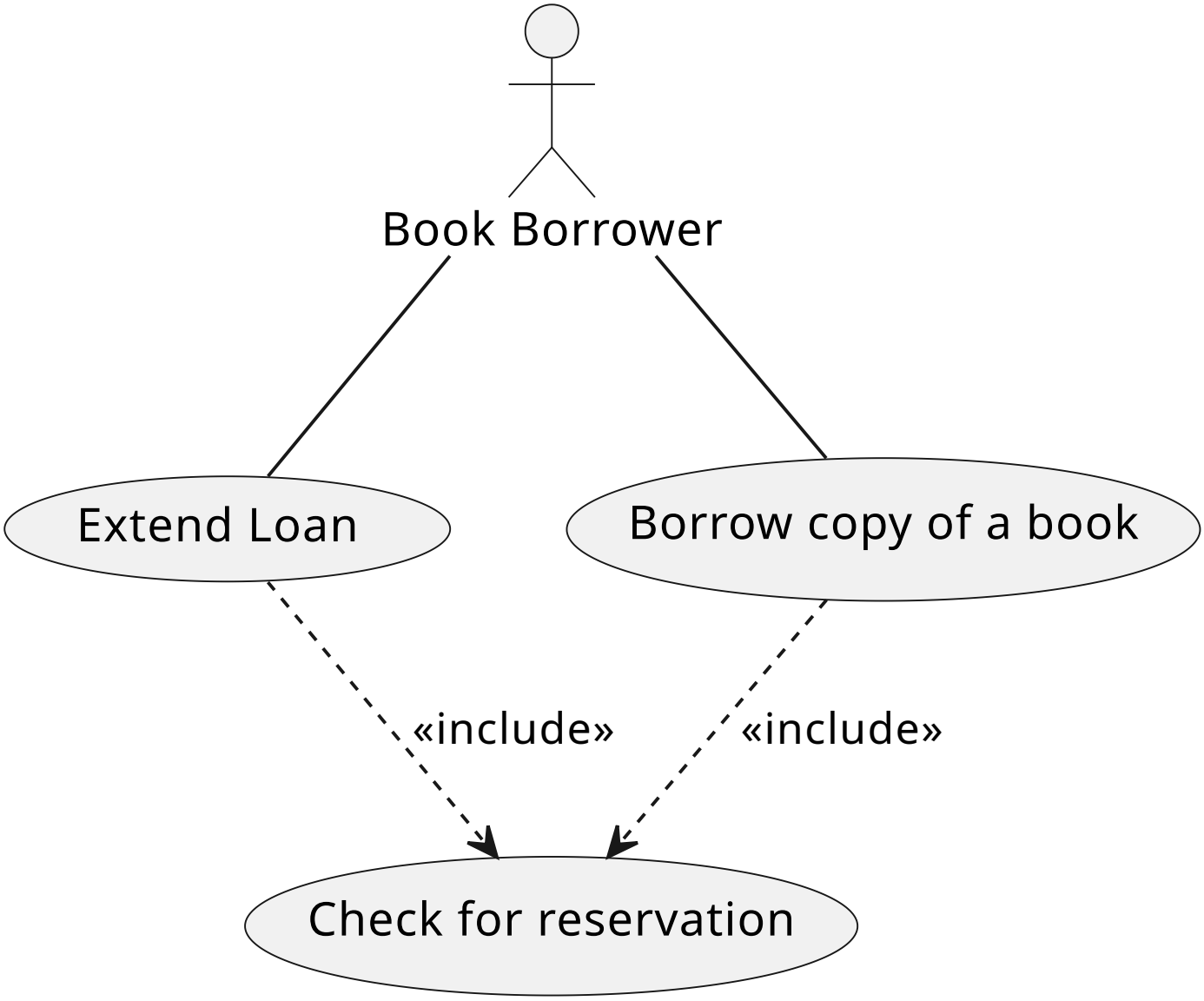
L’associazione di **generalizzazione** rappresenta un particolare tipo di relazione, applicabile sia ad una coppia *attore - attore* che ad una coppia *use case - use case*.

La sua semantica dipende dal contesto a cui viene applicata:

- **tra attori**: permette di esplicitare eventuali relazioni tra ruoli.
  Ad esempio un ruolo potrebbe includerne un altro.
- **tra use case**: la semantica è simile all'*extend*, ma senza punti d'estensione. Infatti, alcune parti della descrizione vengono *ereditate* e altre vengono *sostituite*. Non si applica il secondo principio della Liskov.

Esempi di utilizzo

Nel seguente diagramma,

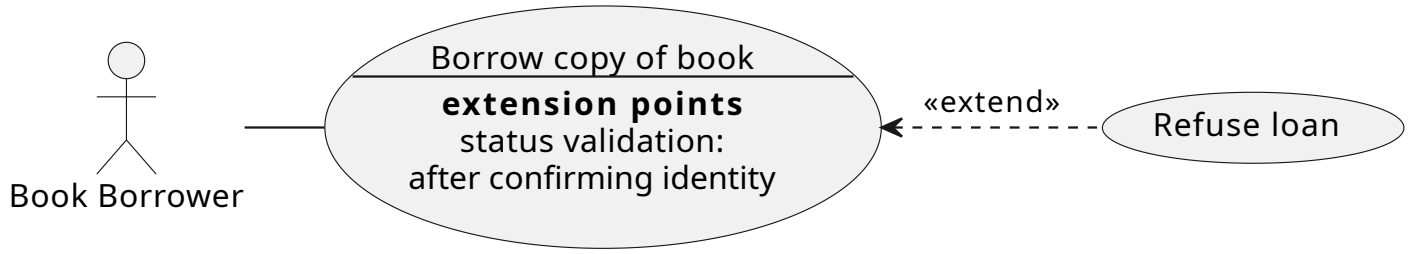


l'attore *Book Borrower* è associato alle seguenti operazioni:

- prendere in prestito un libro;
- chiedere l'estensione del prestito di un libro.

In entrambi i casi, il bibliotecario deve controllare l'esistenza di una richiesta di prenotazione per il libro.

Il prossimo diagramma è differente:



rifiuta il prestito può essere l'**estensione** di un comportamento normale come *prendi in prestito il libro*. In quest'ultimo ci sono dei punti di estensione in cui vengono fatti dei controlli, come la verifica dello stato di prestito del libro o dell'identità del richiedente.

Activity diagram

Concetto e struttura

Gli *activity diagram* presentano una conformazione simile agli *state diagram*, ma con svariate differenze:

- al posto degli stati **vi sono le *attività***;
- non si usano più le **transizioni** etichettate tramite eventi – queste sono quasi tutte **implicitamente temporizzate**;
- possono esserci ***azioni* dentro le attività**;
- le *attività* possono rappresentare **elementi esterni** al sistema.

La peculiare capacità di collegarsi con attività esterne fa sì che sia possibile utilizzare gli activity diagram come **collante** con e tra diversi casi d'uso.

Inoltre la **visuale parzialmente informale**, eppure leggermente più tecnica e profonda rispetto ai diagrammi dei casi d'uso, rende gli activity diagram un **ottimo mezzo di comunicazione interna** (*e.g.* con un manager).

Livelli di astrazione

Si possono utilizzare i diagrami delle attività per:

- descrivere la logica interna di un **business process** (caso più comune);
- descrivere il **flusso interno di un metodo**, con eventuali indicazioni di (pseudo)concorrenza;
- dettagliare il **flusso di un caso d'uso**, ovvero chiarire meglio il suo flusso di esecuzione rispetto ad altri diagrammi (*e.g.* Sequence Diagram). Questa rappresentazione è assai utile nei casi in cui, ad esempio, la concorrenza è un fattore rilevante.

Sincronizzazione


Esempio activity diagram

Attraverso l'uso di barre si possono stabilire dei punti di sincronizzazione (JOIN).
I JOIN, se non diversamente specificato, vengono considerati in ***AND***.

È però possibile porre dei vincoli diversi per stabilire i criteri di soddisfacimento della barra di sincronizzazione (come una ***OR***).

Decisioni


Esempio activity decision diagram

È possibile specificare nel flusso di esecuzione dei momenti di **decisione**. I corsi d'azione intraprendibili in questi frangenti sono rappresentati tramite degli archi.

Le decisioni devono rispettare due proprietà:

- gli archi collegati alla decisione devono essere **mutualmente esclusivi**;
- l'**unione** delle condizioni di decisione deve essere sempre vera.

È bene puntualizzare che i punti di decisione sono *veri* momenti di decisione umana, non banali valutazioni di una condizione logica.

Swim lane


Swim lane esempio

Si può partizionare il diagramma al fine di rappresentare, sulle singole *activity*, delle particolari responsibilità. Queste vengono visualizzate tramite delle *"corsie"* verticali che identificano *chi*

svolge una determinata attività.

Component diagram

Concetto e struttura

Lo scopo del **diagramma dei componenti** è rappresentare e raggruppare i componenti del sistema.
"Componente" è un termine trasversale che include file, librerie, documenti *etc.* (ma che è diverso dal concetto di *classe*!).

Il diagramma include:

- **componenti**: rettangoli che rappresentano una funzione di sistema ben determinata. Possono essere *annidati*;
- **interfacce**: cerchi che indicano le interfacce implementate o utilizzate dai componenti. I collegamenti con i componenti indicano la presenza di una dipendenza;
- **stereotipi**: racchiusi tra i caratteri `<<>>`, etichettano e identificano una serie di funzionalità appartenenti ad uno stesso "gruppo".


Esempio component diagram

Si noti che un componente può usarne un altro conoscendone solo l'interfaccia.

Identificare i componenti

Alcune linee guida per identificare e rappresentare correttamente i componenti sono:

- capire quali parti del sistema sono rimpiazzabili facilmente e/o sono versionate separatamente;
- identificare quali parti del sistema svolgono una funzione ben determinata;
- pensare in termini di "gerarchia" dei componenti;
- chiarire l'esistenza di dipendenze con altri componenti e di dipendenze con le interfacce.

Deployment diagram

Il *Deployment diagram* permette di rappresentare la **dislocazione fisica** delle risorse.
Più precisamente, specifica la dislocazione fisica delle *istanze dei componenti*.

La conformazione del diagramma è quindi molto simile a quella del diagramma dei componenti, ma con qualche differenza:

- i **nodi** del sistema indicano macchine fisiche;
- i **collegamenti** tra nodi eplicitano le modalità di comunicazione tra gli stessi (*e.g.* RMI, HTTP).


Esempio deployment diagram

Il Deployment diagram risulta di particolare utilità per il *deployer*, *i.e.* la figura che si occupa dell'installazione fisica del sistema.

Mocking

Un aspetto importante da considerare durante la scrittura dei test è la chiarezza del loro *scopo*.

Chiunque li legga deve essere in grado di determinare rapidamente quale comportamento si sta testando. Tuttavia, questo può risultare molto difficile se i test non sono strutturati in modo ottimale. L'obiettivo di un test può essere molto confusionario se, ad esempio, vengono invocati senza alcun criterio diversi comportamenti del **system under test** (SUT, ciò che *sta venendo testato*), *e.g.* alcuni per impostare lo stato pre-test (fixture) del SUT, altri per utilizzare il SUT e altri ancora per verificare lo stato post-test del SUT.

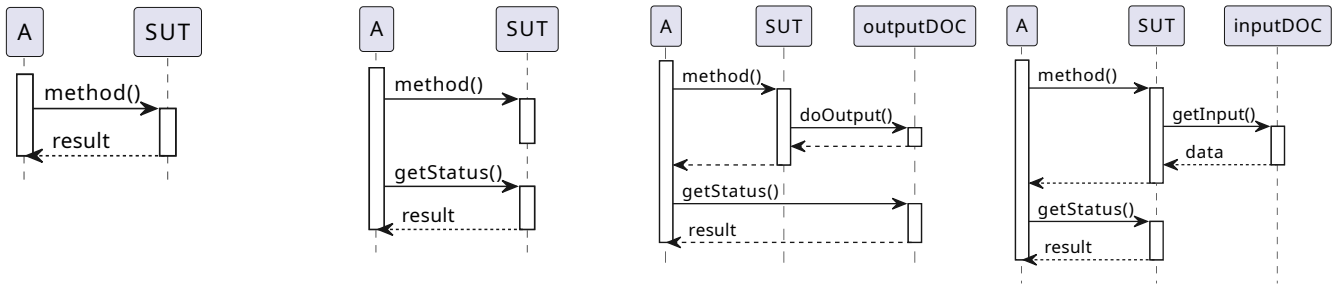
Un modo per rendere evidente ciò che si sta testando è strutturare ogni test in modo che abbia quattro fasi distinte, eseguite in sequenza:

1. **SET UP**: si inizializza tutto il necessario affinché il SUT *esibisca il comportamento atteso* e il test, successivamente, sia in grado di *osservare il risultato effettivo* (ad esempio, creare i vari Test Double).
2. **EXERCISE**: si interagisce con il SUT, facendo dunque eseguire il codice che si vuole effettivamente testare.
3. **VERIFY**: si fa tutto il necessario per determinare se il risultato atteso è stato ottenuto o meno (*e.g.* tramite asserzioni di vario tipo).
4. **TEARDOWN**: fase di pulizia atta a riportare l'ambiente nello stato in cui è stato trovato.

Per aumentare ulteriormente la leggibilità dei nostri test è desiderabile anche fare in modo che ogni metodo di test verifichi una e una sola funzionalità. Ciò non significa che un metodo di test che verifica più funzionalità sia scorretto, ma fornirà sicuramente una minore localizzazione delle anomalie rispetto a un gruppo di test che testano singole funzionalità; in altre parole, sarà meno leggibile e contraddistinto da una logica più complessa.

- **Test Double**
- **Mockito**

Test Double



Può risultare assai difficile testare un SUT che dipende da componenti software non utilizzabili per un motivo o per l'altro. Questi componenti prendono il nome di **depended-on component** (DOC) e i problemi che questi possono far emergere durante la stesura di un test sono molteplici. Ad esempio, i DOC potrebbero non essere disponibili in quel momento, non restituire i risultati che servono a un determinato test (o restituirli solo tramite artifici troppo complessi) oppure perché la loro esecuzione avrebbe effetti collaterali indesiderati. In altri casi ancora, la strategia di testing adottata richiede un maggiore controllo o più visibilità sul comportamento interno del SUT e l'utilizzo di un DOC reale rende l'operazione complessa.

Quando si scrive un test in cui non si può (o si sceglie di non) usare il vero componente da cui si è dipendenti, si può sostituire quest'ultimo con un Test Double, durante la fase di set up.

Test Double è un termine generico utilizzato per indicare un qualunque oggetto con cui si sostituisce un DOC reale a scopo di test.
Ovviamente, a seconda del tipo di test che si sta eseguendo, si può codificare diversamente il comportamento del Test Double. Non è necessario che questo si comporti come il DOC reale: il suo scopo è solo quello di fornire le stesse API in modo che la sua presenza risulti essere trasparente al SUT. In altre parole, per il SUT interagire con il DOC reale o con il Test Double deve essere esattamente la stessa cosa. L'utilizzo di Test Double rende possibile la scrittura di test che precedentemente risultavano troppo articolati, complicati o dispersivi da realizzare.

Il **mocking** è la tecnica di testing che ci permette di sostituire i DOC reali con i vari Test Double. Effettuare mocking permette di ottenere test più efficienti, affidabili e puliti, consentendo agli sviluppatori di isolare il SUT in un ambiente più controllato.

Come si può osservare dall'immagine sottostante, vi sono diversi tipi di Test Double:


Test doubles

- **Dummy objects**
- **Stub objects**
- **Mock objects**

Dummy Objects

Nella maggior parte dei test è necessario fare in modo che il SUT si trovi in uno stato opportuno prima di quella che è la fase di exercise; a tale scopo, durante la fase di set up, vengono effettuate chiamate ad alcuni suoi metodi. Questi possono prendere come argomenti degli oggetti che, a volte, vengono solo memorizzati in variabili d'istanza e non sono di fatto mai utilizzati nel codice testato. É dunque necessario crearli unicamente per conformarsi alla firma di qualche metodo che si pianifica di chiamare nella fase di set up. La costruzione di questi oggetti può essere non banale e aggiunge una complessità del tutto superflua al test.

In questi casi, si può passare come argomento un **dummy object**. Questi oggetti sono una forma degenere di Test Double in quanto esistono solo per poter essere passati da un metodo all'altro e non vengono mai realmente utilizzati. La loro utilità risiede nell'eliminare la necessità di costruire gli oggetti reali.

Si noti che un dummy object non è la stessa cosa di un *null object*. Un dummy object non viene utilizzato dal SUT, quindi il suo comportamento è irrilevante. Al contrario, un null object viene utilizzato dal SUT, ma è progettato per non fare nulla o produrre un risultato sempre "innocuo".

Senza Mockito

```
@Test
public void testDummy() {
    MyClass dummy = ??;

    List<MyClass> SUT = new
ArrayList<MyClass>();

    SUT.add(dummy);


assertThat(SUT.size()).isEqualTo(1);
}
```

Con Mockito

```
@Test
public void testDummy() {
    MyClass dummy =
mock(MyClass.class);

    List<MyClass> SUT = new
ArrayList<MyClass>();

    SUT.add(dummy);


assertThat(SUT.size()).isEqualTo(1);
}
```

Stub Objects

Altre volte risulta difficile testare il SUT perché il suo comportamento dipende dai cosiddetti *input indiretti*: valori restituiti da altri componenti software (DOC) con i quali interagisce. Gli **input indiretti** possono essere valori di ritorno dei metodi del DOC, parametri aggiornati, errori o eccezioni sollevate dal DOC.


Stub object

In presenza di input indiretti la verifica del comportamento del SUT richiede di sostituire i DOC reali con Test Double che immettano gli input desiderati nel SUT. Test Double con questo scopo prendono il nome di **stub object**: sostituiscono un componente reale, da cui dipende il SUT, e forniscono risposte (input) "preconfezionate" alle sole chiamate fatte durante il testing. L'utilizzo di stub consente al test di forzare la realizzazione di determinati scenari particolari o di interesse specifico.

Senza Mockito

```
@Test
public void testConStub() {
    MyClass stub = ??;
    MyList<int> SUT = new
MyList<int>();

    SUT.add(stub.getValue(0));
// deve ritornare 4
    SUT.add(stub.getValue(1));
// deve ritornare 7
    SUT.add(stub.getValue(1));
// deve ritornare 3

    res = SUT.somma();


assertThat(res).isEqualTo(14);
}
```

Con Mockito

```
@Test
public void testConStub() {
    MyClass stub =
mock(MyClass.class);

when(stub.getValue(0)).thenReturn(4);

when(stub.getValue(1)).thenReturn(7,
3);

    MyList<int> SUT = new MyList<int>
();
    SUT.add(stub.getValue(0));  //
deve ritornare 4
    SUT.add(stub.getValue(1));  //
deve ritornare 7
    SUT.add(stub.getValue(1));  //
deve ritornare 3

    res = SUT.somma();

    assertThat(res).isEqualTo(14);
}
```

Mock Objects

Il comportamento del SUT può includere azioni che non possono essere osservate attraverso la sua API pubblica, ma che sono osservate o sperimentate da altri sistemi o componenti dell'applicazione. Tali attività ricadono sotto il nome di **output indiretti** del SUT. Gli output indiretti possono includere chiamate a metodi di un altro componente, record inseriti in un database, record scritti su un file *etc*.


Mock object

Testare il comportamento del SUT può voler dire anche verificare che gli output indiretti siano quelli corretti e a tale scopo servono punti di osservazione appropriati. Un **punto di osservazione** è un modo con cui il test può ispezionare lo stato post-exercise del SUT. I punti di osservazione utili a verificare gli output indiretti sono costituiti da Test Double che prendono il nome di **mock object**. Questi intercettano gli output indiretti del SUT nella fase di exercise e permettono di confrontarli con gli output attesi in un momento successivo (*i.e.* la fase di verifying).

Un mock object è dunque utilizzato per instrumentare e controllare le chiamate fatte dal SUT. In genere, l'oggetto Mock include anche la funzionalità di uno Stub; deve infatti essere in grado di restituire valori al SUT, anche se l'enfasi è posta sulla *verifica* delle chiamate effettuate e non dal loro risultato.

Senza Mockito

```
@Test
public void testConMock() {
    MyClass mock = ??;

    MyList<int> SUT = new
MyList<int>();

    res = SUT.somma(mock);


assertThat(res).isEqualTo(14);
    // assert che getValue è
stata chiamata 3 volte
    // prima una volta con
parametro 0 e poi...
}
```

Con Mockito

```
@Test
public void testConMock() {
     MyClass mock =
mock(MyClass.class);


when(mock.getValue(0)).thenReturn(4);

when(mock.getValue(1)).thenReturn(7,3);

     MyList<int> SUT = new MyList<int>
();

     res = SUT.somma(mock);

     assertThat(res).isEqualTo(14);
     InOrder io = inOrder(mock);
     io.verify(mock).getValue(0);
     io.verify(mock,
times(2)).getValue(1);
}
```

Spy objects

Un altro modo per implementare punti di osservazione che controllino e instrumentino le chiamate effettuate dal SUT su determinati DOC sono gli **spy object**. A differenza dei mock, questi sono costruiti a partire da oggetti reali.
Successivamente alla fase d'interazione con il SUT (exercise), durante la fase di verifica dei risultati (verify), il test confronta le chiamate effettuate dal SUT sul Test Spy con il comportamento desiderato (expected).

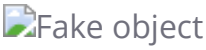
Senza Mockito

```
@Test
public void testConSpy() {
        MyClass spy = ??;

        MyList<int> SUT = new
MyList<int>();

        res = SUT.somma(spy);


assertThat(set).isEqualTo(14);
        // assert che getValue è
stata chiamata 3 volte
        // prima una volta con
parametro 0 e poi...
        }
```

Con Mockito

```
@Test
public void testConSpy() {
     MyClass spy = spy(new
MyClass());

     MyList<int> SUT = new
MyList<int>();

     res = SUT.somma(spy);


assertThat(res).isEqualTo(14);
     InOrder io = inOrder(spy);
     io.verify(spy).getValue(0);
     io.verify(spy,
times(2)).getValue(1);
}
```

Fake Objects



Un **fake object** è un oggetto reale che implementa a tutti gli effetti le funzionalità del DOC, ma per farlo impiega una qualche "scorciatoia" in una maniera che non risulterebbe applicabile ad un contesto di produzione *e.g.* database in memoria invece di un database reale, soluzioni inefficienti, parti di codice open source utilizzabili solo in fase di testing.

Riepilogo

La tabella sottostante fornisce un riepilogo di ciò che rappresenta ciascuna variante dei Test Double.

Test Double	Purpose	Has behavior?	Injects Indirect Inputs into SUT	Handles Indirect Outputs of SUT	Values Provided by Test(er)
Dummy Object	Utilizzato come segnaposto quando è necessario passare un argomento a un metodo	NO	NO, mai usato	NO, mai usato	Nessuno
Stub Object	Fornisce risposte preconfezionate alle sole chiamate fatte durante il testing	SI	SI	NO, li ignora	Input indiretti per il SUT
Mock Object	Instrumentare e controllare le chiamate	SI	Opzionale	Verifica la correttezza rispetto alle aspettative.	Input indiretti per il SUT (opzionali) e output indiretti attesi dal SUT
Spy Object	Instrumentare e controllare le chiamate ad oggetti reali	SI	Opzionale	Li cattura per una verifica successiva	Input indiretti per il SUT (opzionali)
Fake Object	Permette di eseguire test che altrimenti sarebbero impossibili o avrebbero effetti collaterali indesiderati (es test molto lenti)	SI	NO	Li utilizza	Nessuno

Mockito

Mockito è un framework di testing open source per Java rilasciato sotto la licenza MIT. Il framework facilita di gran lunga la creazione di mock objects e in generale di tutti i tipi di Test

Double, permettendo quindi di concentrarsi sulla scrittura della logica di testing.
Inoltre, l'impiego di mockito aumenta notevolmente la leggibilità dei test.

Creare Test Double

Mockito mette a disposizione principalmente due metodi per creare Test Double: il metodo `mock()` e il metodo `spy()`.

Il metodo `mock()` è usato per creare **Test Double** (dummy, stub o mock objects) a partire da una determinata classe o interfaccia: l'oggetto creato si presenterà con la stessa interfaccia (*metodi e firme di questi ultimi*) del tipo specificato in fase di costruzione. Di default, per ogni metodo dell'oggetto reale, il Test Double creato fornisce **un'implementazione minimale**. Questo si limiterà a restituirà dei valori di default per il tipo di ritorno del metodo oppure a non fare nulla se il metodo ritorna `void`.
Ad esempio, se si crea un oggetto con `mock()` a partire da una classe che ha un metodo `getValue()` che restituisce un `int`, il metodo `getValue()` del Test Double restituirà 0, che è il valore predefinito per un `int` in Java.
Il Test Double può essere configurato, mediante opportuna operazione di *stubbing* anche per restituire valori specifici o lanciare eccezioni qualora vengano chiamati determinati metodi.

Il metodo `spy()` viene utilizzato per creare spy objects a partire da oggetti reali. Quello che si ottiene è un oggetto che ha le stesse funzionalità dell'oggetto originale, ma che può essere utilizzato per fare il "tracciamento" delle chiamate ai suoi metodi e per verificare che esse vengano portate a termine come previsto. A differenza degli oggetti creati con il metodo `mock()`, uno spy continuerà a chiamare il metodo reale, a meno che non si specifichi il contrario.

Stubbing

```
when(mockedObj.methodname(args)).thenXXX(values);
```

- args: values, matchers, argumentCaptor
- matchers: anyInt(), argThat(is(closeTo(1.0, 0.001)))
- thenXXX: thenReturn, thenThrows, thenAnswer, thenCallRealMethod

Il metodo `when()` insieme ai vari `thenXXX()` (es `thenReturn()`, `thenThrow()`) è usato per specificare il comportamento di un Test Double (stub mock o spy obj) quando viene chiamato un suo determinato metodo. Si supponga, ad esempio, di avere una classe *Foo* con un metodo

`getValue()` che restituisce un `int`. Per fare in modo che restituisca un valore diverso dallo 0 (default per int), è possibile scrivere:

```
Foo foo = mock(Foo.class);
when(foo.getValue()).thenReturn(42);
```

Da questo momento in poi, il metodo `getValue()` del Test Double restituirà l'intero 42 ogni volta che verrà chiamato.
Ovviamente il metodo `when()` può essere usato per specificare il comportamento di qualsiasi metodo del Test Double. Se quindi viene scritto:

```
Foo foo = mock(Foo.class);
when(foo.someMethod()).thenThrow(new SomeException());
```

il Test Double lancerà un'eccezione di tipo `SomeException` quando viene chiamato il suo metodo `someMethod()`. Questo permette, per esempio, di testare il comportamento del nostro codice quando viene lanciata un'eccezione senza dover implementare l'oggetto reale.

Anche i metodi del tipo `doXXX()` (es. `doReturn()`, `doThrow()`, `doAnswer()`, `doNothing()`) sono usati per specificare il comportamento del Test Double quando viene chiamato un suo metodo. Tuttavia, a differenza del metodo `when()`, questi possono essere usati anche per specificare il comportamento di un metodo che ha come tipo di ritorno `void`; è consigliabile usarli solo in questo caso, oppure in tutti i casi in cui utilizzare il metodo `when()` risulterebbe difficile (*e.g.* metodi con tipi di ritorno non banali come `Optional`). Questi metodi si utilizzano come segue:

```
doXXX(values).when(mockedObj).methodname(args)
```

Verifying

Con oggetti di tipo mock o spy si desidera spesso verificare l'occorrenza di una chiamata con certi parametri. Mockito permette di farlo con il metodo `verify()`: è possibile verificare che un metodo sia stato chiamato, con quali parametri e per quante volte.

```
verify(mockedclass, howMany).methodname(args)
```

Il parametro *howMany* del metodo verify specifica il numero di volte che il metodo associato all'oggetto mockato deve essere chiamato durante l'esecuzione del test.

Le possibili opzioni sono:

- `times(n)`: verifica che `methodname()` sia stato chiamato esattamente `n` volte;
- `never()`: verifica che `methodname()` non sia mai stato chiamato;
- `atLeastOnce()`: verifica che `methodname()` sia stato chiamato almeno una volta;
- `atLeast(n)`: verifica che `methodname()` sia stato chiamato almeno `n` volte;
- `atMost(n)`: verifica che `methodname()` sia stato chiamato al massimo `n` volte.

Se si desidera verificare l'ordine delle occorrenze delle chiamate ai metodi di un oggetto, si può utilizzare il metodo `inOrder()`:

```
InOrder inO = inOrder(mock1, mock2, ...)
inO.verify...
```

È possibile anche catturare un parametro per farci delle asserzioni.

```
ArgumentCaptor<Person> arg = ArgumentCaptor.forClass(Person.class);
verify(mock).doSomething(arg.capture());
assertEquals("John", arg.getValue().getName());
```

Argument Matchers

Quando si esegue un'operazione di stubbing oppure quando si verifica la chiamata a un metodo, al posto di specificare i valori precisi (values) si può utilizzare quello che è un **argument matcher**. \ Questo agisce come un segnaposto che corrisponde a qualsiasi valore corretto (*i.e.* che soddisfa la condizione di match), consentendo di specificare il comportamento senza dover conoscere il valore esatto dell'argomento che sarà passato al metodo. Alcuni possibili matcher sono:

- `any()`, `anyInt()`, `anyString()`, etc.: questi metodi sono usati per creare degli argument matcher, che permettono di specificare che un particolare argomento del metodo può essere qualsiasi valore di un particolare tipo. Per esempio, si può utilizzare `anyInt()` per specificare che un argomento può essere un qualsiasi valore int. Questo risulta utile qualora si desideri fare lo stub di un metodo che restituisca un valore indipendentemente dagli argomenti passatigli.

- `eq()`: questo metodo viene utilizzato per creare un argument matcher che corrisponde a un valore specifico. Per esempio, si può utilizzare `eq(42)` per specificare che un argomento deve avere il valore 42 per poter essere confrontato. Ciò è utile quando si vuole fare lo stub di un metodo in modo che questo restituisca un valore solo quando viene chiamato con argomenti specifici.

- Il metodo `argThat()` è un modo più generale per specificare argument matchers. Permette di creare matcher personalizzati implementando l'interfaccia `ArgumentMatcher`. Questa definisce un metodo `matches()` che può essere utilizzato per determinare se un particolare argomento corrisponde al matcher. Tale metodologia risulta utile quando si vuole abbinare gli argomenti in maniere più complesse o articolate rispetto ai matcher di argomenti di tipo `any()`.

Reset a Test Double

Infine, il metodo `reset()` è usato per ripristinare un Test Double al suo stato iniziale, cancellando qualsiasi metodo che era stato precedentemente ridefinito. È utile quando si vuole riutilizzare un Test Double in più test.

Esempio di testing con pattern OBSERVER (PULL)

```
@Test
void modelTest {
    // setup
    Model model = new Model();
    Observer obs = mock(Observer.class);
    Observer obs1 = mock(Observer.class);

    // exercise
    model.addObserver(obs);
    model.addObserver(obs1);
    model.setTemp(42.0, scale);

    // verify
    verify(obs).update(eq(model), eq("42.0"));
    verify(obs1).update(eq(model), eq("42.0"));
}
```

Test di un observer con un modello non generico, ma di cui si ha solo interfaccia di cui viene fornita una versione dummy:

```
@Test
void observerTest {
    abstract class MockObservableIModel extends Observable implements Model {};
    MockOBservableIModel model = mock(MockObservableIModel.class);
    when(model.getTemp()).thenReturn(42.42);

    observer.update(model, null);

    verify(model).getTemp();
    assertThat(val).isCloseTo(42.42, Offset.offset(.01));
}
```

Verifica e convalida

Terminologia

Verifica e convalida

Verifica e convalida sono due termini con un significato apparentemente molto simile ma che celano in realtà una differenza non banale tra loro:

- per *verifica (della correttezza)* si intende l'attività di confronto del software con le **specifiche** (formali) prodotte dall'analista;
- per *convalida (dell'affidabilità)* si intende l'attività di confronto del software con i **requisiti** (informali) posti dal committente.

Ci sono quindi due punti critici che vanno a sottolineare maggiormente questa differenza:

- requisiti e specifiche sono spesso **formulati diversamente**. Solitamente i *requisiti*, essendo scritti dal committente, sono formulati in un linguaggio più vicino al dominio di quest'ultimo. Diversamente, le *specifiche* sono scritte in un linguaggio più vicino al dominio dello sviluppatore, spesso in maniera formale e poco ambigua;
- è facile che i requisiti **cambino** in corso d'opera mentre le specifiche rimangano congelate; questo aspetto dipende molto dai contratti tra committente e il team di sviluppo.

La definizione dei *requisiti* forniti dal cliente è immediata ma informale: scrivere dei test che li *convalidano* può risultare molto complicato. Invece, è più semplice validare le *specifiche*

attraverso test in quanto sono scritte dal team di sviluppo e sono quindi più formali e complete.

Ad ogni modo, nelle attività di verifica e convalida si cercano degli **errori**, ma la parola "errore" stessa può assumere molti significati a seconda del contesto. È quindi importante capire di quale *errore* si sta parlando, introducendo termini diversi, come *malfunzionamento* e *difetto*.

N.B: Esistono dei glossari e vocabolari di terminologia comune redatti dalla IEEE, ad esempio Systems and software engineering — Vocabulary che possono essere addottati dagli sviluppatori come standard in modo da snellire la comunicazione tra di loro.

Malfunzionamento (guasto/failure)

Un malfunzionamento è uno **scostamento** dal corretto funzionamento del programma.

Non dipende dal codice e in generale non ci si accorge di esso osservando il codice ma solo da un punto di vista più esterno, utilizzando il programma. Il malfunzionamento potrebbe riguardare sia le specifiche (quindi relativo alla fase di *verifica*) che i requisiti (fase di *convalida*, ovvero "non rispetta le aspettative"). Secondo il vocabolario citato in precedenza:

---

failure:

1. termination of the ability of a product to perform a required function or its inability to perform within previously specified limits.
   ISO/IEC 25000:2005, Software Engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE.4.20.
2. an event in which a system or system component does not perform a required function within specified limits.

NOTE: A failure may be produced when a fault is encountered

Esempio

Di seguito è illustrato un esempio di malfunzionamento.

```java
static int raddoppia(int par) {
    int risultato;
    risultato = (par * par);
    return risultato;
}

static void main(String[] args) {
    int risultato = raddoppia(3);
    System.out.println(risultato);  // 9
}
```

La funzione dovrebbe ritornare il doppio del numero in ingresso, ma se passiamo 3 in argomento verrà ritornato 9.

Difetto (anomalia/fault)

Un difetto è il **punto del codice** che causa il verificarsi del malfunzionamento.

È **condizione necessaria** (ma non sufficiente) per la verifica di un malfunzionamento.

> **fault:**
>
> 1. a manifestation of an error in software.
> 2. an incorrect step, process, or data definition in a computer program.
> 3. a defect in a hardware device or component. Syn: bug
>
> *NOTE: A fault, if encountered, may cause a failure.*

Nell'esempio di codice precedente, il difetto è in `risulato = (par * par)`.

Il *difetto* è condizione *non sufficiente* per il verificarsi di un *malfunzionamento*: ad esempio, **non si verificano malfunzionamenti** in caso l'argomento passato sia 0 oppure 2. Il raddoppio in quei casi avverrebbe in maniera corretta.

Un altro esempio di tale è proprietà è il caso in cui esistono *"più anomalie che si compensano"*: se si sta utilizzando una libreria per operazioni su temperature in gradi Fahrenheit, ponendo il caso che stia partendo da gradi Celsius, dovrà essere effettuata una conversione. Se in questa conversione è presente un'anomalia che però si riflette allo stesso modo in fase di riconversione per restituire il risultato, le due anomalie combinate non si manifestano in un malfunzionamento.

Spesso le anomalie si annidano nella gestione di casi particolari o eccezionali del programma in quanto il flusso di controllo ordinario è solitamente il più testato.

Sbaglio (mistake)

Uno sbaglio è la causa di un'anomalia. Si tratta in genere di un errore umano.

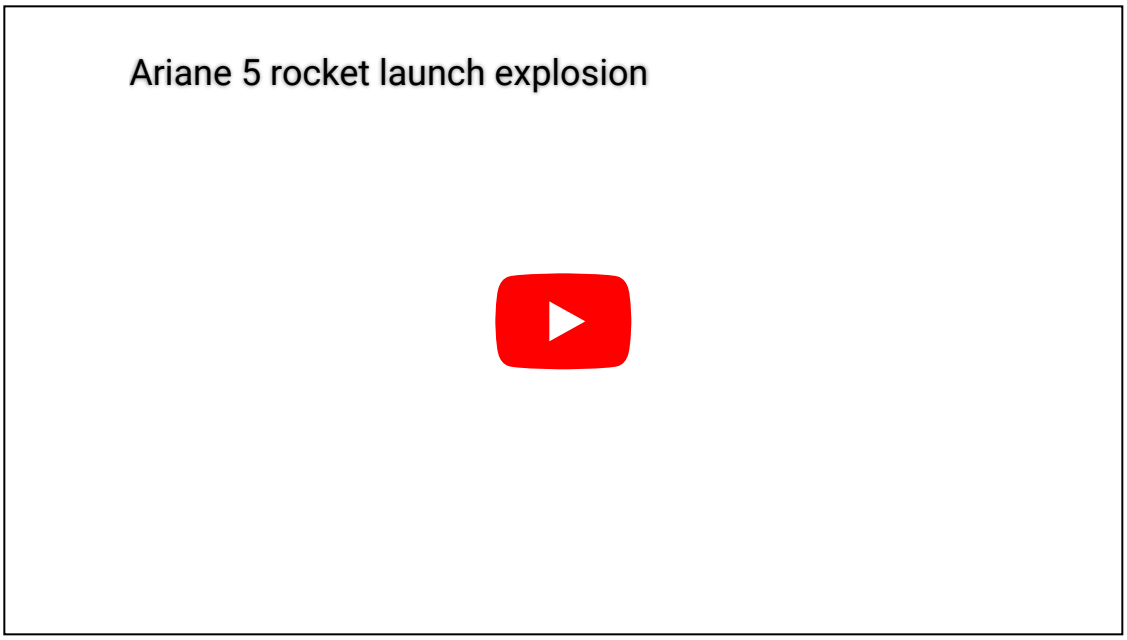
> **mistake:**
>
> 1. a human action that produces an incorrect result
>
> *NOTE: The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).*

Relativamente all'esempio precedente, possibili sbagli possono essere:

- **errori di battitura** (scrivere `*` invece di `+`);
- **concettuali** (non sapere cosa vuol dire *raddoppiare*);
- relativi alla **padronanza del linguaggio** (credere che `*` sia il simbolo dell'addizione).

È importante capire quale sia la *causa* di uno sbaglio in modo da poter intraprendere azioni correttive per il futuro (*es. studiare meglio la sintassi del linguaggio*).

Esempio notevole: *il caso Ariane 5*



Wikipedia: Ariane 5 notable launches

Il 4 giugno 1996 il primo volo di prova del razzo Ariane 5 è fallito a causa di un problema al software di controllo che ha portato all'autodistruzione del missile.

Il *malfunziamento* è palese: il razzo è esploso e chiaramente non era il comportamento richiesto.

Qual era l'*anomalia*? Il malfunziamento si è verificato per una eccezione di overflow, sollevatosi durante una conversione da un 64 bit float a un 16 bit signed int che indicava il valore della velocità orizzontale. Questo ha bloccato sia l'unità principale che il backup dell'unità stessa.

Lo *sbaglio*? Tale eccezione non veniva gestita perché questa parte del software era stata ereditata da Ariane 4, modello di razzo antecedente a Ariane 5, la cui traiettora faceva sì che non si raggiungessero mai velocità orizzontali non rappresentabili con int 16 bit. La variabile incriminata non veniva protetta per gli *"ampi margini di sicurezza"* (a posteriori, non così ampi).

Il comportamento della variabile non era mai stato analizzato con i dati relativi alla traiettoria da seguire.

Tecniche

Classificazione delle tecniche

Nell'ambito della *verifica e convalida* è possibile classificare le tecniche di analisi in due categorie:

- **tecniche statiche**, basate sull'analisi degli elementi sintattici del codice.
  Ad esempio: metodi formali, analisi del dataflow e modelli statistici;
- **tecniche dinamiche**, basate sull'esecuzione del programma.
  Ad esempio: testing e debugging.

In generale, **è più facile determinare tecniche dinamiche** rispetto alle tecniche statiche. Per contro, una volta ideate e a patto di avere dimensioni del codice ragionevoli e costrutti sintattici non troppo complessi le **tecniche statiche sono più veloci** nell'analizzare il codice e, soprattutto, più complete dato che le tecniche dinamiche lavorano sui possibili stati del programma - che possono essere infiniti.

Ovviamente diverse metodologie di verifica e convalida avranno i rispettivi pro e contro. Come si possono dunque confrontare queste tecniche?


Classificazione tecniche verifica e convalida

Nell'immagine sopra è possibile osservare una *piramide immaginaria a 3 dimensioni* che riassume dove si posizionano le tecniche di verifica e convalida relativamente le une con le altre. La cima della piramide rappresenta il **punto ideale** a cui tendere, nel quale è possibile affermare di esser riusciti a verificare perfettamente una proprietà arbitraria attraverso una prova logica (dal lato statico) o una ricerca esaustiva su tutti gli stati del problema (dal lato dinamico).

Tale punto ideale è **praticamente impossibile** da raggiungere per la stragrande maggioranza dei problemi che siamo interessati a risolvere. Bisogna scegliere da quale versante iniziare la scalata della piramide: **lato verde** (approccio statico) o **lato blu** (approccio dinamico)?

Più ci si posiziona verso il basso, più si degenera in:

- **estrema semplificazione delle proprietà** (in basso a sinistra): si stanno in qualche modo *rilassando* eccessivamente gli obiettivi che si vogliono raggiungere.

  Ad esempio, se si vuole dimostrare che si sta usando un puntatore in maniera corretta e nel farlo si sta semplicemente controllando che non valga `null`, è *cambiata* la proprietà

che si vuole come obiettivo (controllare che un puntatore non valga `null` **non significa che** lo si stia usando nel modo corretto);

- **estrema inaccuratezza pessimistica** (in basso al centro): è dovuta all'approccio pessimistico che ha come mantra:

  > *"Se non riesco a dimostrare l'assenza di un problema assumo che il problema sia presente"*

  Ad esempio, si manifesta nei compilatori quando non riescono a dimostrare che una determinata funzione che deve ritornare un valore ritorni effettivamente un valore per tutti i possibili cammini `if` / `else if` / eccetera. La mancanza di capacità nel dimostrare l'assenza di un problema non ne implica la presenza di uno.

- **estrema inaccuratezza ottimistica** (in basso a destra): è dovuta all'approccio ottimistico che ha come mantra:

  > *"Se non riesco a dimostrare la presenza di un problema assumo che questo non sia presente"*

  È una possibile deriva degli approcci legati al testing: con esso si cercano malfunzionamenti, se a seguito dei test non ne vengono trovati allora si assume che il programma funzioni correttamente.

A metà strada tra questi estremi inferiori e l'estremo superiore ideale si posizionano quindi le varie tecniche di verifica e convalida, ciascuna più o meno legata ai tra approcci sopra descritti. Tra queste evidenziamo le dimostrazioni con metodi formali, il testing e il debugging.

Metodi formali

L'approccio dei metodi formali tenta di dimostrare l'*assenza* di anomalie nel prodotto finale.
Si possono utilizzare diverse tecniche (spiegate nelle lezioni successive), come:

- analisi di dataflow;
- dimostrazione di correttezza delle specifiche logiche.

Questo approccio segue la linea dell'*inaccuratezza pessimistica*.

Testing

Il testing è l'insieme delle tecniche che si prefiggono di rilevare **malfunzionamenti**.
Attraverso il testing non si può dimostrare la correttezza ma solo aumentare la *fiducia* dei clienti rispetto all'affidabilità del prodotto.

Le tecniche di testing possono essere molto varie e si raggruppano in:

- **white box**: si ha accesso al codice da testare e si possono cercare anomalie guardandolo da un punto di vista interno;
- **black box**: non si ha accesso al codice ma è possibile testare e cercare malfunzionamenti tramite le interfacce esterne;
- **gray box**: non si ha accesso al codice ma si ha solo un'idea dell'implementazione ad alto livello.
  Per esempio, se sappiamo che il sistema segue il pattern MODEL VIEW CONTROLLER ci si può aspettare che certe stimolazioni portino a chiamate al database mentre altre no.

Come è chiaro, questo approccio segue una logica di *inaccuratezza ottimistica*.

È inoltre interessante notare che il Test Driven Development (TDD) adotta una filosofia di testing completamente black box: imponendo che venga scritto prima il test del codice questo non può assumere niente sul funzionamento interno dell'oggetto di testing.

Debugging

Dato un *programma* e un *malfunzionamento noto e riproducibile*, il debugging permette di localizzare le **anomalie** che causano i malfunzionamenti. A differenza del testing, infatti, è richiesta la conoscenza a priori di un malfunzionamento prima di procedere con il debugging.

Molto spesso viene usato il debugging al posto del **testing**, almeno a livello di terminologia: questo è un problema perché il debugging non è fatto per la "grande esecuzione" ma al contrario per esaminare in maniera granulare (a volte anche passo passo per istruzioni macchina) una determinata sezione di codice in esecuzione con lo scopo di trovare l'anomalia che provoca un malfunzionamento. Se si usassero le tecniche di debugging per effettuare il testing il tempo speso sarebbe enorme: il debugging osserva infatti *stati interni* dell'esecuzione e per rilevare un malfunzionamento in questo modo sarebbe necessario osservare tutti i possibili - e potenzialmente infiniti - stati del programma.

Due possibili approcci al debugging sono:

- partendo da una malfunzionamento *noto* e *riproducibile* si avvia una procedura di analisi basata sulla **produzione degli stati intermedi** dell'esecuzione del programma: *passo*

passo (a livello a piacere, da istruzione macchina a chiamata di funzione) si controllano tutti gli stati di memoria alla ricerca di uno inconsistente;
- ***divide-et-impera***: il codice viene smontato sezione per sezione e componente per componente in modo da poter trovare il punto in cui c'è l'anomalia. Si possono mettere breakpoint o *"print tattiche"*.

Testing

Definizioni

Correttezza

La maggior parte dei problemi che si verificano durante lo sviluppo di un progetto sono causati da *problemi di comunicazione*. Ci possono essere incomprensioni quando le informazioni passano da una figura all'altra, come quando ci si interfaccia tra cliente, analista e programmatore. Il programmatore dovrà adattare il proprio linguaggio per farsi comprendere dal cliente prestando maggiore attenzione alla formalità e alla chiarezza della comunicazione con il passare del tempo. Più i concetti sono spiegati chiaramente, più è difficile incorrere in problemi successivi: è quindi necessario fare attenzione alla **terminologia** utilizzata.

Partiamo quindi dalle basi: quando un programma si definisce ***corretto***?

Considerando un generico programma P come una funzione da un insieme di dati D (dominio) a un insieme di dati R (codominio) allora:

- $P(d)$ indica l'**esecuzione** di P su un certo input $d \in D$,
- il risultato $P(d)$ è **corretto** se soddisfa le specifiche, altrimenti è scorretto,
- $\mathrm{ok}(P, d)$ indica la **correttezza** di P per il dato d

quindi

$$P \text{ è } corretto \iff \forall d \in D\,,\ \mathrm{ok}(P, d)$$

A parole, *un programma **è corretto** quando **per ogni dato** del dominio vale* $\mathrm{ok}(P,\, d)$.

Per indicare la correttezza di programma P si utilizza la notazione $\mathrm{ok}(P,\, D)$, che appunto indica che P è *corretto* per qualunque $d \in D$.

Definizione di test

Durante l'attività di testing ciò che viene fatto è sottoporre il programma a una serie di stimolazioni per saggiarne il comportamento in tali circostanze. Eseguire un test vuole quindi dire eseguire il programma con una serie di input appartenenti al suo dominio e confrontare i risultati ottenuti con il risultato atteso secondo le specifiche.

Volendone dare una definizione più rigorosa, *un **test** è un sottoinsieme del dominio dei dati* e *un singolo **caso di test** è un elemento di esso*. Un test sono quindi **più stimolazioni**, mentre un caso di test è una **singola stimolazione**.
Matematicamente:

- un test T per un programma P è un sottoinsieme del suo dominio D;
- un elemento t di un test T è detto *caso di test*;
- l'esecuzione di un test consiste nell'esecuzione del programma $\forall t \in T \subseteq D$.

Un programma P supera (o *passa*) un test T se:

$$\mathrm{ok}(P,\, T) \iff \forall t \in T\,,\ \mathrm{ok}(P,\, t)$$

Quindi, *un programma è **corretto per un test** quando **per ogni caso di test** esso è **corretto***.

Lo scopo dei test è però ricercare comportamenti anomali nel programma per permetterci di correggerli. Diciamo quindi che *un test T ha **successo** se rileva uno o più malfunzionamenti presenti nel programma P*:

$$\mathrm{successo}(T,\, P) \iff \exists t \in T \mid \neg\, \mathrm{ok}(P,\, t)$$

Test ideale

Se un test non rileva alcun malfunzionamento **non significa che il programma sia corretto**: come visto nella lezione precedente, il test è un'attività ottimistica e normalmente il passaggio di un test non garantisce l'assenza di anomalie. Questo smette però di essere vero nel caso di *test ideali*.

*Un test T si definisce **ideale** per P se e solo se*

$$\text{ok}(P, T) \Rightarrow \text{ok}(P, D)$$

*ovvero se il superamento del test **implica la correttezza del programma**.*

Purtroppo però in generale è ***impossibile* trovare un test ideale**, come ci suggerisce la seguente ipotesi universalmente accettata:

> **Tesi di Dijkstra**:
>
> *Il test di un programma può rilevare la presenza di malfunzionamenti ma non dimostrarne l'assenza.*
>
> *Non esiste quindi un algoritmo che dato un programma arbitrario P generi un test ideale **finito***
> *(il caso $T = D$ non va considerato).*

Notiamo come la tesi escluda esplicitamente il *test esaustivo* $T = D$, restringendosi a considerare i test finiti (mentre il dominio D potrebbe anche essere infinito). Per capire il perché di questa distinzione è sufficiente osservare il seguente esempio:

```
static int sum(int a, int b) {
    return a + b;
}
```

In Java un int è espresso su 32 bit, quindi il dominio di questa semplice funzione somma ha cardinalità $2^{32} \cdot 2^{32} = 2^{64} \sim 2 \cdot 10^{19}$. Considerando quindi un tempo di esecuzione ottimistico di 1 nanosecondo per ogni caso di test, un test esaustivo che provi tutte le possibili combinazioni di interi impiegherebbe più di 600 anni per essere eseguito per intero.

*Il **test esaustivo** è quindi **impraticabile**.*

Proprietà

Affidabilità

*Un criterio di selezione C si dice **affidabile** se presi due test T_1 e T_2 in base al criterio allora o entrambi hanno successo o nessuno dei due ha successo.*

$$\boxed{\text{affidabile}(C, P) \iff (\forall T_1 \in C, \forall T_2 \in C\,,\ \text{successo}(T_1, P) \Leftrightarrow \text{successo}(T_2, P))}$$

Validità

*Un criterio di selezione si dice **valido** se qualora P non sia corretto, allora esiste almeno un test T selezionato in base al criterio C che ha successo e quindi rileva uno o più malfunzionamenti per il programma P:*

$$\text{valido}(C, P) \iff (\neg\, \text{ok}(P, D) \Rightarrow \exists T \in C \,|\, \text{successo}(T, , P))$$

Esempio

Si consideri il seguente codice.

```
static int raddoppia(int par) {
    int risultato;
    risultato = (par * par);
    return risultato;
}
```

Un criterio che seleziona:

- *"i sottoinsiemi di $0, 2$"* è **affidabile**, perché il programma funziona sia con 0 sia con 2, ma **non valido**, perché sappiamo che il programma non è corretto e non esiste un test che trovi malfunzionamenti;
- *"i sottoinsiemi di $0, 1, 2, 3, 4$"* è **non affidabile**, perché i risultati dei casi di test non sono tutti coerenti (e quindi il test $T1 = 0, 1$ non ha successo mentre $T2 = 0, 3$ sì), ma **valido** perché esiste un test che rileva i malfunzionamenti.
- *"i sottoinsieme finiti di D con almeno un valore maggiore di 18"* è **affidabile**, perché i risultati dei casi di test sono tutti coerenti, e **valido** perché rileva i malfunzionamenti.

In questo caso la ricerca di un criterio valido e affidabile era semplice perché conoscevamo già l'anomalia. Tuttavia, lo stesso non si può dire di un qualunque programma P in quanto **non si conoscono i malfunzionamenti a priori** e dunque è molto più difficile trovare criteri validi e affidabili.

Conclusione

L'obiettivo sarebbe quindi quello di trovare un *criterio valido e affidabile* sempre. Tuttavia ciò è purtroppo impossibile in quanto un criterio di questo tipo selezionerebbe test ideali, che sappiamo non esistere.

Immaginiamo infatti di avere un *criterio valido e affidabile* e che un test selezionato da esso **non abbia successo**. Sapendo che:

- non avendo successo allora non sono stati trovati errori,
- essendo il criterio affidabile allora tutti gli altri test selezionati da quel criterio non troveranno errori,
- essendo il criterio valido allora se ci fosse stato un errore almeno uno dei test lo avrebbe trovato

allora il programma è **corretto**, ovvero abbiamo trovato un test che quando non ha successo implica la correttezza del programma: in poche parole, un *test ideale*. Esiste quindi un altro modo per implicare la correttezza di un programma:

$$\text{affidabile}(C, P) \land \text{valido}(C, P) \land T \in C \land \neg\, \text{successo}(T, P) \implies \text{ok}(P, D)$$

In conclusione, trovare un criterio che sia **contemporaneamente** affidabile e valido significherebbe trovare un criterio che selezioni **test ideali** che sappiamo non esistere per la *tesi di Dijkstra*. Dovremo dunque accontentarci di criteri che garantiscano solo una delle due caratteristiche.

Utilità di un test

Abbandonata la vana speranza di un criterio di selezione universalmente valido che permetta di testare alla perfezione qualunque programma vediamo ora cosa significa *utilizzare* un criterio di selezione per costruire un test. Come sappiamo un test altro non è che un insieme di casi di test, specifici input appartenenti al dominio del programma: un criterio di selezione governa dunque quanti e quali casi di test vengono aggiunti al test che si sta costruendo.

Possiamo quindi ora farci una domanda: quali sono le **caratteristiche** che **rendono utile** un caso di test, ovvero che rendono “possibile” o “probabile” che il caso di test evidenzi un malfunzionamento causato da un’anomalia? Ebbene, un caso di test utile deve:

- **eseguire il comando che contiene l’anomalia** – non è altrimenti possibile che il malfunzionamento si manifesti;
- l’esecuzione del comando che contiene l’anomalia deve portare il sistema in uno **stato scorretto**, o per meglio dire **inconsistente**;
- lo stato inconsistente deve propagarsi fino all’uscita del codice in esame in modo da **produrre un output diverso da quello atteso**;

Un buon criterio di selezione dovrà quindi selezionare test contenenti casi di test utili: ma quanti dovrebbe contenerne? Per capire ciò si può utilizzare un metro di misura legato alle caratteristiche del codice: a ogni criterio è infatti possibile associare una **metrica** che misuri la **copertura** del test che si sta costruendo e che ci permetta di decidere *quando smettere di*

aggiungere casi di test, *quali casi di test è opportuno aggiungere* e di *confrontare la bontà di test diversi*. Aggiungeremo infatti solo casi di test che permettano di aumentare la metrica di copertura, e test in grado di garantire una copertura maggiore saranno inerentemente migliori di test con una copertura minore.

Criteri di selezione

Assodato che un test ideale è impossibile da realizzare, come possiamo scegliere un *sottoinsieme del dominio* che approssimi il più possibile un *test ideale*?
Esistono una serie di **criteri di selezione** che hanno proprio lo scopo di guidare la selezione dei casi di test all'interno del dominio in modo da massimizzare la probabilità che il test abbia successo. Prima però di illustrarne alcuni, vediamo quali caratteristiche dovrebbero avere questi criteri.

Esploriamo quindi ora una serie di criteri di selezione, elencandone pro e contro, esplicitandone la metrica di copertura utilizzata e infine confrontandoli tra di loro per comprenderne le relazioni.

- **Criterio di copertura dei comandi**
- **Criterio di copertura delle decisioni**
- **Criterio di copertura delle condizioni**
- **Criterio di copertura delle decisioni e condizioni**
- **Criterio di copertura delle condizioni composte**
- **Criterio di copertura delle condizioni e decisioni modificate**
- **Implicazioni tra criteri di copertura**

Criterio di copertura dei comandi

*Un test T soddisfa il **criterio di copertura dei comandi** se e solo se ogni comando eseguibile del programma è eseguito in corrispondenza di almeno un caso di test $t \in T$.*
La metrica è dunque la frazione di **comandi eseguibili su quelli eseguiti** dall'intero test.

Consideriamo per esempio il seguente programma in pseudocodice:

Esempio 1: copertura dei comandi

Pseudocodice

```
01  void main(){
02      float x, y;
03      read(x);
04      read(y);
05      if (x != 0)
06          x = x + 10;
07      y = y / x;
08      write(x);
09      write(y);
10  }
```

Diagramma di flusso di esecuzione

È possibile ricostruire un **diagramma di flusso di esecuzione** del codice trasformando ogni comando in un nodo del diagramma: *coprire tutti i comandi* significa quindi visitare tutti i nodi raggiungibili.

Applicare il *criterio di copertura dei comandi* significa quindi trovare un insieme di casi di test in cui *per ogni nodo esiste un caso di test che lo attraversa*.

Nel nostro esempio il singolo caso di test $\langle 3,\ 7 \rangle$ risulterebbe quindi sufficiente, dato che la sua esecuzione permette di *coprire* tutti i comandi del programma. Tuttavia, pur massimizzando la metrica di copertura dei comandi tale test non è in grado di rilevare l'anomalia alla riga 7, in cui viene fatta una divisione senza prima controllare che il divisore sia diverso da zero.

Soddisfare il criterio di copertura dei comando **non garantisce** dunque la correttezza del programma. Come sappiamo infatti un'anomalia non sempre genera un malfunzionamento, per cui eseguire semplicemente tutte le righe di codice raggiungibili non assicura di rilevare eventuali errori.

Criterio di copertura delle decisioni

*Un test T soddisfa il **criterio di copertura delle decisioni** se e solo se ogni decisione (effettiva) viene resa sia vera che falsa in corrispondenza di almeno un caso di test $t \in T$.*
La metrica è quindi la frazione delle **decisioni totali possibili** presenti nel codice che sono state rese **sia vere che false** nel test.

Dovendo attraversare ogni possibile flusso di controllo il criterio di copertura delle decisioni **implica il criterio di copertura dei comandi**. Estraendo il codice in un diagramma di flusso, infatti, è possibile coprire tutte le decisioni se e solo se ogni arco (*e quindi ogni nodo*) viene attraversato. Non è invece vero l'inverso.

Esempio 2: copertura delle decisioni

Pseudocodice **Diagramma di flusso di esecuzione**

```
01  void main(){
02      float x, y;
03      read(x);
04      read(y);
05      if (x != 0 && y > 0)
06          x = x + 10;
07      else
08          y = y / x
09      write(x);
10      write(y);
11  }
```

Dall'esempio sopra, un test che soddisfi il suddetto criterio potrebbe includere $\langle 3,\ 7\rangle$, $\langle 3,\ -2\rangle$.
Nonostante sia un criterio *"migliore"* del precedente, la copertura delle decisioni **non garantisce** la correttezza del programma: nell'esempio il caso $\langle 0,\ 5\rangle$ eseguirebbe comunque una divisione per zero.

Criterio di copertura delle condizioni

*Un test T soddisfa il **criterio di copertura delle condizioni** se e solo se ogni singola condizione (effettiva) viene resa sia vera che falsa in corrispondenza di almeno un caso di test $t \in T$.*
Similmente ai criteri precedenti, la metrica è quindi la percentuale delle **condizioni** che sono state rese **sia vere che false** su quelle per cui è possibile farlo.

Sebbene simile, si tratta di un criterio diverso da quello di copertura delle decisioni: in caso di condizioni composte, come per esempio `x != 0 && y < 3`, la copertura delle decisioni imporrebbe che l'*intera condizione* sia resa sia vera che falsa, mentre la copertura delle condizioni richiede di rendere vere e false le singole *condizioni atomiche* `x != 0` e `y < 3` in almeno un caso di test.
Come vedremo nell'esempio, ciò non impone quindi di seguire tutti i percorsi sul diagramma di flusso e fa sì che questo criterio **non implica** il soddisfacimento di nessuno dei precedenti.

Esempio 3: copertura delle condizioni

Pseudocodice **Diagramma di flusso di esecuzione**

```
01  void main(){
02      float x, y;
03      read(x);
04      read(y);
05      if (x != 0 || y > 0)
06          y = y / x;
07      else
08          y = (y + 2) / x
09      y = y / x;
10      write(x);
11      write(y);
12  }
```

Nell'esempio sopra, il test $\langle 0,\ 5\rangle$, $\langle 5,\ -5\rangle$ **soddisfa il criterio di copertura della condizioni** \ (`x != 0` è falsificato da $\langle 0,\ 5\rangle$ e verificato da $\langle 5,\ -5\rangle$, mentre `y > 0` è verificato da $\langle 0,\ 5\rangle$ e falsificato da $\langle 5,\ -5\rangle$), ma **la decisione è sempre vera**.

Sono infatti presenti anomalie alla riga 6 (possibile divisione per zero) e alla riga 8 (overflow e divisione per zero), ma i comandi contenuti nella riga 8 non sono coperti. In questo caso più che mai, quindi, la copertura delle condizioni **non garantisce** la correttezza del programma.

Criterio di copertura delle decisioni e condizioni

*Un test T soddisfa il **criterio di copertura delle decisioni e delle condizioni** se e solo se **ogni decisione** vale sia vero che falso e **ogni condizione** che compare nelle decisioni del programma vale sia vero che falso per diversi casi di test $t \in T$.*

È – intuitivamente – l'**intersezione** del criterio di copertura delle decisioni con il criterio di copertura delle condizioni, per cui il soddisfacimento di questo criterio **implica** sia il criterio di copertura delle condizioni che quello di copertura delle decisioni (e quindi dei comandi).

Nell'esempio 3, il test $\langle 0,\ -5\rangle$, $\langle 5,\ 5\rangle$ soddisfa il criterio di copertura delle decisioni e condizioni e rileva l'anomalia alla riga 8 ma non quella alla riga 6. **Non garantisce** quindi neanche in questo caso la correttezza del programma.

Criterio di copertura delle condizioni composte

*Un test T soddisfa il **criterio di copertura delle condizioni composte** se e solo se ogni possibile composizione delle condizioni base vale sia vero che falso per diversi casi di test $t \in T$.*

Viene cioè testata ogni possibile combinazione di valori delle condizioni atomiche quando queste sono aggregate in condizioni composte: riprendendo per esempio la condizione `x != 0 && y < 3`, vengono testati separatamente i casi $\langle V, V \rangle$, $\langle V, F \rangle$, $\langle F, V \rangle$ e $\langle F, F \rangle$.
È quindi facile notare che **questo criterio implica il precedente** (criterio di copertura delle decisioni e condizioni), implicando a sua volta il criterio di copertura delle decisioni, delle condizioni e dei comandi.

Data la **natura combinatoria** di questo criterio, all'aumento del numero di condizioni di base *il numero di casi di test* cresce però troppo rapidamente, motivo per cui il soddisfacimento di questo criterio è considerato **non applicabile** in pratica. Inoltre, dato che le condizioni di base potrebbero non risultare indipendenti tra loro, potrebbero esistere **combinazioni non fattibili** che non avrebbe alcun senso testare.

Criterio di copertura delle condizioni e delle decisioni modificate

Non volendo testare tutte le combinazioni di condizioni, ci si rende presto conto che certe combinazioni sono **più rilevanti** di altre: se modificando una sola condizione atomica si riesce a modificare l'esito della decisione, allora è molto significativa – indipendentemente dalla sua dimensione. Se invece l'esito della decisione non varia, allora la modifica può essere considerata neutra o meno significativa.
Il criterio così ottenuto prende il nome di **criterio di copertura delle condizioni e delle decisioni modificate**.

Si dà quindi importanza nella selezione delle combinazioni al fatto che la modifica di una singola condizione base porti a **modificare l'esito della decisione**. Per ogni condizione base devono quindi esistere due casi di test che modificano il valore di una sola condizione base e che portino a un diverso esito della decisione: in questo modo, inoltre, il criterio **implica quello di copertura delle condizioni e delle decisioni**.

Si può dimostrare che se si hanno N condizioni base **sono sufficienti** $N + 1$ **casi di test** per soddisfare questo criterio, decisamente meno di quelli richiesti dal criterio delle condizioni composte.

Implicazioni tra criteri di copertura

Ecco dunque uno schema delle implicazioni tra i vari criteri di copertura. Come si vede, il criterio delle condizioni composte va considerato troppo oneroso e quindi non applicabile, mentre gli altri criteri possono invece essere utilizzati anche nell'ambito di progetti di dimensioni reali.

Altri criteri

I criteri visti finora **non considerano i cicli** e possono essere soddisfatti da test che percorrono ogni ciclo al più una volta. Molti errori però si verificano durante **iterazioni successive alla prima**, come per esempio quando si superano i limiti di un array.

Occorre quindi sviluppare dei criteri che tengano conto anche delle iterazioni e stimolino i cicli un numero di volte sufficiente.

Esempio 4: copertura delle iterazioni

Pseudocodice

```
01  void main() {
02      float a, b, x, y;
03      read(x);
04      read(y);
05      a = x;
06      b = y;
07      while (a != b) {
08          if (a > b)
09              a = a - b;
10          else
11              b = b - a;
12      }
13      write(a);
14  }
```

Diagramma di flusso di esecuzione

- **Criterio di copertura dei cammini**
- **Criterio di n-copertura dei cicli**

Criterio di copertura dei cammini

*Un test T soddisfa il **criterio di copertura dei cammini** se e solo se ogni cammino del grafo di controllo del programma viene percorso per almeno un caso di $t \in T$.*
La metrica è quindi il rapporto tra i **cammini percorsi** e **quelli effettivamente percorribili**.

Questo criterio è molto generale ma è spesso impraticabile, anche per programmi semplici: la presenza di cicli imporrebbe infatti di testare tutti gli infiniti cammini che li attraversano un numero arbitrario di volte. Il criterio è quindi considerato **non applicabile** in pratica.

Criterio di n-copertura dei cicli

*Un test T soddisfa il **criterio di n-copertura** se e solo se ogni cammino del grafo contenente al massimo un numero d'iterazioni di ogni ciclo non superiore a n viene percorso per almeno un caso di test $t \in T$.*

La definizione sopra non significa che il test deve eseguire n volte un ciclo, ma che per ogni numero k compreso tra 0 e n deve esistere un caso di test che esegue tale ciclo k volte. Si sta quindi **limitando il numero massimo di percorrenze** dei cicli.
Di conseguenza, al crescere di n il numero di test aumenta molto rapidamente. Inoltre, fissare n a livello di programma può non essere un'azione così semplice: il numero d'iterazioni che necessita un ciclo per essere testato a fondo può essere **molto differente** a seconda del caso.

Per cercare di minimizzare il numero di test spesso il criterio applicato è quello di **2-copertura dei cicli**. Si tratta infatti del numero minimo che permette comunque di testare tutte le casistiche principali:

- zero iterazioni;
- una iterazione;
- *più di una* iterazione.

Il caso $n = 2$ è cioè il minimo per considerare casistiche non banali: dando uno sguardo all'esempio sopra, infatti, con $n = 1$ il ciclo (`while`) sarebbe stato indistinguibile da una semplice selezione (`if`); testando due iterazioni si incominciano a testare le vere caratteristiche del ciclo. Esso permette cioè di testare non solo i comandi che compongono il ciclo, ma anche sue le pre/post-condizioni ed eventuali invarianti.

A differenza del criterio di copertura dei cammini, il criterio di n-copertura è considerato **applicabile** a programmi reali.

Mappa finale delle implicazioni tra criteri di selezione

Aggiungendo i criteri di copertura che considerano esplicitamente i cicli si ottiene il seguente schema di implicazione tra tutti i criteri di selezione.


Mappa implicazioni criteri

Analisi statica

Come abbiamo detto nella lezione precedente, il testing dell'esecuzione del programma non è però l'unica cosa che possiamo fare per aumentare la fiducia nostra e del cliente nella correttezza del programma. Un'altra importante iniziativa in tal senso è l'ispezione tramite varie tecniche del *codice* del programma, attività che prende il nome di **analisi statica**.

L'analisi statica si basa cioè sull'esame di un **insieme finito di elementi** (*le istruzioni del programma*), contrariamente all'analisi dinamica che invece considera un insieme infinito (*gli stati delle esecuzioni*). È un'attività perciò **meno costosa del testing**, poiché non soffre del problema dell'*"esplosione dello spazio degli stati"*.

Considerando solamente il codice "statico" del programma, questa tecnica non ha la capacità di rilevare anomalie dipendenti da particolari valori assunti dalle variabili a runtime. Si tratta nondimeno di un'attività estremamente utile, che può aiutare a individuare numerosi errori e inaccortezze.

Compilatori

Prima di trasformare il codice sorgente in eseguibile, i compilatori fanno un'attività di analisi statica per identificare errori sintattici (o presunti tali) all'interno del codice.

Il lavoro dei compilatori si può dividere solitamente in **quattro tipi di analisi** (gli esempi sono presi dal compilatore di Rust, caratteristico per la quantità e qualità di analisi svolta durante la compilazione):

- **analisi lessicale**: identifica i token appartenenti o meno al linguaggio, permettendo di individuare possibili errori di battitura;

```
error: expected one of `!`, `.`, `::`, `;`, `?`, `{`, `}`, or an operator,
found `ciao`
--> src/main.rs:2:9
  |
2 |     BRO ciao = "mondo";
  |           ^^^^ expected one of 8 possible tokens
```

- **analisi sintattica**: controlla che i token identificati siano in relazioni *sensate* tra di loro in base alla grammatica del linguaggio, scovando così possibili errori di incomprensione del linguaggio;

```
error: expected `{`, found keyword `for`
--> src/main.rs:2:14
  |
2 |     if !expr for;
  |              ^^^ expected `{`
  |
```

- **controllo dei tipi**: nei linguaggi tipizzati, individua violazioni di regole d'uso dei tipi ed eventuali incompatibilità tra tipi di dati;

```
error[E0308]: mismatched types
--> src/main.rs:2:24
  |
2 |     let name: String = 42;
  |               ------   ^^- help: try using a conversion method:
`.to_string()`
  |               |        |
  |               |        expected struct `String`, found integer
  |               expected due to this

For more information about this error, try `rustc --explain E0308`.
```

- **analisi flusso dei dati**: si cercano di rilevare problemi relativi all'*evoluzione dei valori* associati alle variabili, come per esempio porzioni di codice non raggiungibili.

```
error: equal expressions as operands to `!=`
--> src/main.rs:2:8
  |
2 |     if 1 != 1 {
  |        ^^^^^^
  |
```

Se i primi tre tipi di analisi sono abbastanza facili da comprendere, l'ultimo merita una maggiore attenzione, motivo per cui gli dedichiamo il prossimo paragrafo.

Analisi Data Flow

Nata nell'ambito dell'**ottimizzazione dei compilatori**, che per migliorare le proprie performance ricercavano porzioni di codice non raggiungibile da non compilare, l'**analisi del flusso di dati** è stata più avanti imbracciata dall'ingegneria del software per ricercare e prevenire le cause di errori simili.
Parlando di flusso dei dati si potrebbe pensare a un'analisi prettamente dinamica come il testing, ma l'insieme dei controlli statici che si possono fare sul codice per comprendere come vengono *utilizzati* i valori presenti nel programma è invece particolarmente significativo.

È possibile infatti analizzare staticamente il tipo delle operazioni eseguite su una variabile e l'**insieme dei legami di compatibilità** tra di esse per determinare se il valore in questione viene usato in maniera *semanticamente sensata* all'interno del codice.
Nello specifico, le operazioni che possono essere eseguite su un **dato** sono solamente di tre tipi:

- d (**definizione**): il comando **assegna un valore** alla variabile; anche il passaggio del dato come parametro ad una funzione che lo modifica è considerata un'operazione di (ri)definizione;
- u (**uso**): il comando **legge il contenuto** di una variabile, come per esempio l'espressione sul lato destro di un assegnamento;
- a (**annullamento**): al termine dell'esecuzione del comando il valore della variabile **non è significativo/affidabile**. Per esempio, dopo la *dichiarazione senza inizializzazione* di una variabile e al termine del suo *scope* il valore è da considerarsi inaffidabile.

Dal punto di vista di ciascuna variabile è possibile ridurre una qualsiasi sequenza d'istruzioni (*ovvero un cammino sul diagramma di flusso*) a una sequenza di definizioni, usi e annullamenti.

- **Regole**

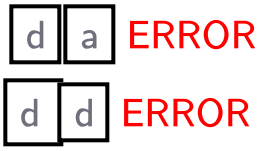
- **Sequenze**

Regole

Fatta questa semplificazione è allora possibile individuare la presenza di anomalie nell'uso delle variabili definendo alcune **regole di flusso**: alcune di queste devono essere necessariamente rispettate in un programma corretto (1 e 3), mentre altre hanno più a che fare con la semantica dell'uso di un valore (2).

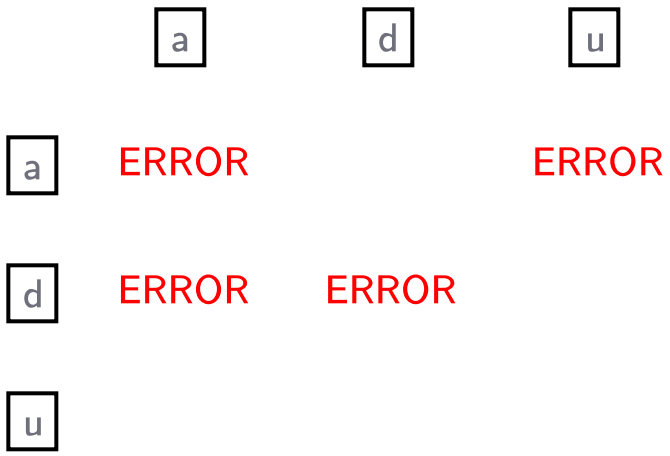
1. L'**uso di una variabile** deve essere **sempre preceduto** in ogni sequenza **da una definizione senza annullamenti intermedi**.

2. La **definizione di una variabile** deve essere **sempre seguita** da **un uso**, **prima** di un suo **annullamento** o nuova **definizione**.



3. L'**annullamento di una variabile** deve essere **sempre seguito** da **una definizione**, **prima** di un **uso** o **altro annullamento**.

Riassumendo, a u, d a, d d e a a sono sequenze che identificano **situazioni anomale**, anche se non necessariamente dannose: se per esempio usare una variabile subito dopo averla annullata rende l'esecuzione del programma non controllabile, un annullamento subito dopo una definizione non crea nessun problema a runtime, ma è altresì indice di un possibile errore concettuale.



Esempio

Consideriamo la seguente funzione C con il compito di scambiare il valore di due variabili:

```
void swap(int &x1, int &x2) {
    int x1;
    x3 = x1;
    x3 = x2;
    x2 = x1;
}
```

Analizzando il codice, le sequenze per ogni variabile sono le seguenti:

Variabile	Sequenza	Anomalie
`x1`	a u u a	`x1` viene usata 2 volte senza essere stata prima definita
`x2`	... d u d ...	Nessuna
`x3`	... d d d ...	`x3` viene definita più volte senza nel frattempo essere stata usata

Come si vede, in un codice sintatticamente corretto l'analisi Data Flow ci permette quindi di scovare un errore semantico osservando le sequenze di operazioni sulle sue variabili.

Sequenze

Abbiamo accennato più volte al concetto di "sequenza" di operazioni su una variabile. Più formalmente, definiamo **sequenza** di operazioni per la variabile $\mathtt{a}$ secondo il cammino p la concatenazione della tipologia delle istruzioni che coinvolgono tale variabile, e la indichiamo con $\mathrm{P}(p, \mathtt{a})$.

Considerando per esempio il seguente programma C:

```
01  void main() {
02      float a, b, x, y;
03      read(x);
04      read(y);
05      a = x;
06      b = y;
07      while (a != b)
08          if (a > b)
09              a = a - b;
10          else
11              b = b - a;
12      write(a);
13  }
```

possiamo dire che:

$$\mathrm{P}([1,2,3,4,5,6,7,8,9,7,12,13],\,\mathtt{a}) = \boxed{\mathsf{a}_2}\boxed{\mathsf{d}_5}\boxed{\mathsf{u}_7}\boxed{\mathsf{u}_8}\boxed{\mathsf{u}_9}\boxed{\mathsf{d}_9}\boxed{\mathsf{u}_7}\boxed{\mathsf{u}_{12}}\boxed{\mathsf{a}_{13}}$$

Eseguendo questo tipo di operazione su tutte le variabili e per tutti i cammini del programma si potrebbe verificare la presenza eventuali anomalie, ma come sappiamo **i cammini sono potenzialmente infiniti** quando il programma contiene cicli e decisioni: per scoprire quali percorsi segue effettivamente l'esecuzione del programma dovremmo eseguirlo e quindi uscire dal campo dell'analisi statica.

Espressioni regolari

Tuttavia non tutto è perduto: un caso di cammini contenenti **cicli** e **decisioni** è possibile rappresentare un insieme di sequenze ottenute dal programma P utilizzando delle **espressioni regolari**. Con $\mathrm{P}([1\,\rightarrow],\,\mathtt{a})$ si indica infatti l'espressione regolare che rappresenta **tutti i cammini** che partono dall'istruzione 1 per la variabile $\mathtt{a}$.

Questo perché nelle espressioni regolari è possibile inserire, oltre che una serie di parentesi che isolano sotto-sequenze, anche due simboli molto particolari:

- la **pipe** (|), che indica che i simboli (o le sotto-sequenze) alla propria destra e alla propria sinistra si *escludono* a vicenda: *una e una sola* delle due è presente;
- l'**asterisco** (*), che indica che il simbolo (o la sotto-sequenza) precedente può essere *ripetuto da 0 a n volte*.

Grazie a questi simboli è possibile rappresentare rispettivamente decisioni e cicli. Prendendo per esempio il codice precedente, è possibile costruire $\mathrm{P}([1\,\rightarrow],\,\mathtt{a})$ come:

$$\boxed{\mathsf{a}_2}\boxed{\mathsf{d}_5}$$

$$\boxed{\mathsf{a}_2}\boxed{\mathsf{d}_5}\quad\boxed{\mathsf{u}_7}\Big(\quad \textit{while body} \quad\Big)*\quad\boxed{\mathsf{u}_{12}}\boxed{\mathsf{a}_{}}$$

$$\boxed{\mathsf{a}_2}\boxed{\mathsf{d}_5}\quad\boxed{\mathsf{u}_7}\Big(\quad\boxed{\mathsf{u}_8}\quad \textit{if body} \quad\Big)*\quad\boxed{\mathsf{u}_{12}}\boxed{\mathsf{a}_{}}$$

$$\boxed{\mathsf{a}_2}\boxed{\mathsf{d}_5}\quad\boxed{\mathsf{u}_7}\Big(\quad\boxed{\mathsf{u}_8}\quad\Big(,\boxed{\mathsf{u}_9}\boxed{\mathsf{d}_9}\,\Big|\,:\boxed{\mathsf{u}_{11}}\Big)\quad\Big)*\quad\boxed{\mathsf{u}_{12}}\boxed{\mathsf{a}_{}}$$

$$\boxed{\mathsf{a}_2}\boxed{\mathsf{d}_5}\quad\boxed{\mathsf{u}_7}\Big(\quad\boxed{\mathsf{u}_8}\quad\Big(,\boxed{\mathsf{u}_9}\boxed{\mathsf{d}_9}\,\Big|\,:\boxed{\mathsf{u}_{11}}\Big)\quad\boxed{\mathsf{u}_7}\quad\Big)*\quad\boxed{\mathsf{u}_{12}}\boxed{\mathsf{a}_{}}$$

Osserviamo come $\boxed{\mathsf{u}_7}$ si ripeta due volte: questo può rendere *fastidioso* ricercare errori, per via della difficoltà di considerare cammini multipli. Comunque sia, una volta ottenuta un'espressione regolare è facile verificare l'eventuale presenza di errori applicando le solite regole (nell'esempio non ce n'erano).

Bisogna però fare attenzione a un'aspetto: le espressioni regolari così costruite rappresentano **tutti i cammini** possibili del programma, ma **non tutti e i soli**! Trattandosi di oggetti puramente matematici, infatti, le espressioni regolari sono necessariamente *più generali* di qualunque programma: esse non tengono infatti conto degli *effetti* che le istruzioni hanno sui dati e delle relative proprietà che si possono inferire.
Riprendendo a esempio l'espressione regolare di cui sopra, essa contiene la sequenza nella quale il ciclo viene eseguito *infinite volte*, ma osservando il programma è facile indovinare che tale comportamento non sia in realtà possibile: diminuendo progressivamente `a` e `b` a seconda di chi sia il maggiore si può dimostrare che prima o poi i due convergeranno allo stesso valore permettendo così di uscire dal ciclo.

In effetti, uno stesso programma può essere rappresentato tramite **un numero infinito di espressioni regolari** valide. Si potrebbe addirittura argomentare che l'espressione regolare

$$\left(\boxed{\mathrm{u}} \mid : \boxed{\mathrm{d}} \mid : \boxed{\mathrm{a}}\right)*$$

possa rappresentare qualsiasi programma.
Allontanandosi però dai casi estremi, si dimostra essere impossibile scrivere un algoritmo che dato un qualsiasi programma riesca a generare un'espressione regolare che rappresenti **tutti e soli** i suoi cammini possibili senza osservare i valori delle variabili. Bisogna dunque accontentarsi di trovare espressioni regolari che rappresentino **al meglio** l'esecuzione del programma, ovvero con il minor numero di cammini impossibili rappresentati.

Nell'analisi Data Flow tramite espressioni regolari è quindi necessario tenere conto che il modello generato è un'**astrazione pessimistica**: se viene notificata la presenza di un errore non si può essere certi che esso ci sia veramente, in quanto esso potrebbe derivare da un cammino non percorribile.

Testing

Oltre ad essere un processo utile di per sé per il rilevamento di potenziali errori, l'**analisi statica** può anche contribuire a guidare l'attività di **testing**.
Per capire come, osserviamo che a partire dall'analisi statica è possibile fare le seguenti osservazioni:

- perché si presenti un malfunzionamento dovuto a una anomalia in una *definizione*, deve esserci un *uso* che si serva del valore assegnato;
- un ciclo dovrebbe essere ripetuto (di nuovo) se verrà *usato* un valore *definito* alla iterazione precedente.

L'analisi statica può quindi aiutare a **selezionare i casi di test** basandosi sulle *sequenze definizione-uso* delle variabili, costruendo cioè dei nuovi criteri di copertura.

Terminologia

Per rendere più scorrevole la spiegazione dei prossimi argomenti introduciamo innanzitutto un po' di terminologia.

Dato un nodo i del diagramma di flusso (*un comando/riga del programma*), chiamiamo $\mathrm{def}(i)$ l'**insieme delle variabili definite in $\mathbf{i}$**.

Data invece una variabile x e un nodo i, chiamiamo $\mathrm{du}(x, i)$ l'insieme dei nodi j tali che:

- $x \in \mathrm{def}(i)$, ovvero la variabile x è **definita** in i;
- x è **usata** nel nodo j;
- **esiste un cammino** da i a j **libero da definizioni** di x, ovvero che se seguito non sovrascrive il valore di x.

Si tratta cioè dell'**insieme di nodi $\mathbf{j}$ che *potrebbero* usare il valore di $\mathbf{x}$ definito in $\mathbf{i}$**.

Criteri di copertura derivati dall'analisi statica

- **Criterio di copertura delle definizioni**
- **Criterio di copertura degli usi**
- **Criterio di copertura dei cammini DU**

Criterio di copertura delle definizioni

*Un test T soddisfa il **criterio di copertura delle definizioni** se e solo se per ogni nodo i e ogni variabile $x \in \mathrm{def}(i)$, T include un caso di test che esegue un cammino libero da definizioni da i ad **almeno uno** degli elementi di $\mathrm{du}(i, x)$.*

Ci si vuole cioè assicurare di testare tutte le definizioni, assicurandosi che funzionino osservando almeno un uso del valore da loro assegnato. Matematicamente si può dire che:

$$T \in C_{def} \iff \forall i \in P,\ \forall x \in \mathrm{def}(i),\ \exists j \in \mathrm{du}(i, x),\\ \exists t \in T \text{ che esegue un cammino da } i \text{ a } j \text{ senza ulteriori definizioni di } x.$$

Riconsideriamo l'esempio già visto in precedenza, considerando la variabile $\mathbf{a}$:

```
01  void main() {
02      float a, b, x, y;
03      read(x);
04      read(y);
05      a = x;
06      b = y;
07      while (a != b)
08          if (a > b)
09              a = a - b;
10          else
11              b = b - a;
12      write(a);
13  }
```

Partiamo definendo gli insiemi dei nodi degli usi $\mathrm{du}(i, \mathtt{a})$:

1. $\mathrm{du}(5, \mathtt{a}) = 7, 8, 9, 11, 12$;
2. $\mathrm{du}(9, \mathtt{a}) = 7, 8, 9, 11, 12$.

È solo **un caso** il fatto che in questo esempio tali insiemi siano uguali.
Comunque sia, l'obiettivo è *per ognuna delle due definizioni* ottenere un **uso** di tale definizione:

1. Per la prima definizione la soluzione è banale, a riga 7 la variabile `a` viene letta sempre: $\boxed{\mathsf{d}_5}\boxed{\mathsf{u}_7}$.
2. Per la seconda, invece, è necessario scegliere un valore tale per cui il flusso di esecuzione entri almeno una volta nel ciclo ed esegua almeno una volta la riga 9: $\boxed{\mathsf{d}_9}\boxed{\mathsf{u}_7}$.

Un test che soddisfa totalmente il criterio può essere il seguente:

$$T = \langle 8, 4 \rangle.$$

Come si vede, il criterio di copertura delle definizioni non copre tutti i comandi e di conseguenza **non implica il criterio di copertura dei comandi**.

Criterio di copertura degli usi

*Un test T soddisfa il **criterio di copertura degli usi** se e solo se per ogni nodo i e ogni variabile x appartenente a $\mathrm{def}(i)$, T include un caso di test che esegue un cammino libero da definizioni da i ad **ogni elemento** di $\mathrm{du}(i, x)$.*

Sembra simile al precedente, con la differenza che ora bisogna coprire **tutti** i potenziali usi di una variabile definita. Questo appare ancora più chiaro osservando la formula matematica:

$$\begin{aligned} T \in C_{path} \iff & \forall i \in P,\ \forall x \in \mathrm{def}(i),\ \forall j \in \mathrm{du}(i, x), \\ & \exists t \in T \text{ che esegue un cammino da } i \text{ a } j \text{ senza ulteriori definizioni di } x \end{aligned}$$

Si noti però che il criterio di copertura degli usi **non implica il criterio di copertura delle definizioni**, perché nel caso in cui non esistano $j \in \mathrm{du}(i, x)$ l'uso del $\forall$ è più *"permissivo"* del $\exists$ del criterio precedente: quest'ultimo richiedeva infatti che per ogni definizione esistesse almeno un uso, mentre il criterio di copertura degli usi non pone tale clausola (*se non ci sono usi il $\forall$ è sempre vero*). Viene quindi da sé che questo criterio non copre neanche il criterio di copertura dei comandi.

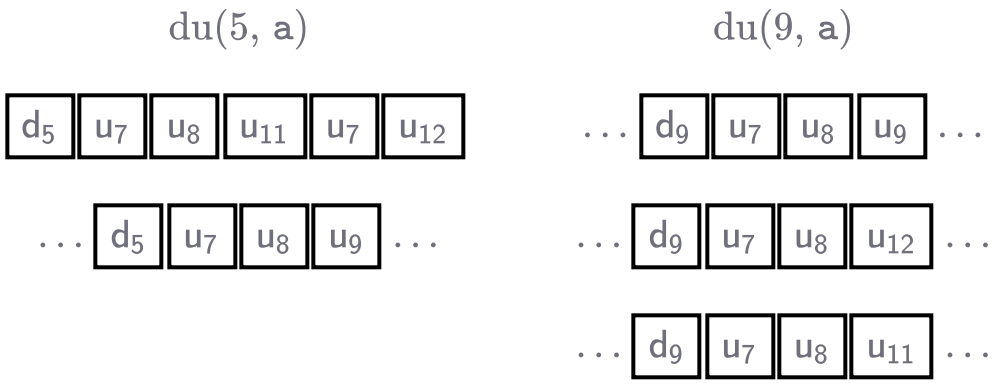
Riconsideriamo nuovamente il programma in C visto in precedenza come esempio:

```
01  void main() {
02      float a, b, x, y;
03      read(x);
04      read(y);
05      a = x;
06      b = y;
07      while (a != b)
08          if (a > b)
09              a = a - b;
10          else
11              b = b - a;
12      write(a);
13  }
```

Come prima, consideriamo la variabile `a` e i relativi insieme dei nodi degli usi per ogni sua definizione:

1. $\mathrm{du}(5, \mathtt{a}) = 7, 8, 9, 11, 12$;
2. $\mathrm{du}(9, \mathtt{a}) = 7, 8, 9, 11, 12$.

Per ogni definizione occorre coprire **tutti gli usi**:



Un test che soddisfa totalmente il criterio può essere il seguente:

$$T = \langle 4, 8\rangle, \langle 12, 8\rangle, \langle 12, 4\rangle.$$

Questo esempio permette di notare qualcosa sulla natura dei cicli: dovendo testare ogni percorso al loro interno è necessario fare almeno due iterazioni. Può quindi sorgere un dubbio:

è meglio che le due iterazioni siano fatte nello stesso caso di test o in casi test separati? Ovvero, è meglio **minimizzare** i **casi di test** o le **iterazioni per caso**?
Opinione diffusa è quella secondo cui è preferibile **minimizzare le iterazioni**: partizionando le casistiche in diversi casi di test è possibile rilevare con più precisione gli errori, riducendo il tempo di debug. In alcune situazioni però aumentare il numero di iterazioni può diminuire il tempo di esecuzione totale dei test, in quanto dovendo riavviare il programma per ciascun caso di test la somma dei tempi di startup può diventare significativa per software molto massicci.

Criterio di copertura dei cammini DU

Nel criterio precedente si richiedeva di testare *un* cammino da ogni definizione ad ogni suo uso, ma come sappiamo i cammini tra due istruzioni di un programma possono essere molteplici. Potrebbe dunque sorgere l'idea di testarli *tutti*: da questa intuizione nasce il **criterio di copertura dei cammini DU**.

$$\begin{aligned} T \in C_{pathDU} \iff & \forall i \in P,\ \forall x \in \mathrm{def}(i),\ \forall j \in \mathrm{du}(i, x), \\ & \forall \text{ cammino da } i \text{ a } j \text{ senza ulteriori definizioni di } x \\ & \exists t \in T \text{ che lo esegue.} \end{aligned}$$

Questo criterio può essere **utile da ipotizzare**, ma a causa dell'esplosione combinatoria del numero dei cammini è considerato **impraticabile** (*"sopra la barra rossa"*).

Oltre le variabili

L'analisi del flusso dati si può estendere anche su altri *"oggetti"*, non solo variabili.
Per esempio, è possibile prevedere le seguenti operazioni su un **file**:

- a (**apertura**): specializzata in *per lettura* o *per scrittura*;
- c (**chiusura**);
- l (**lettura**);
- s (**scrittura**).

Date queste operazioni si possono individuare una serie di regole, come per esempio:

1. l, s e c devono essere precedute da a senza c intermedie;
2. a deve essere seguita da c prima di un'altra a;
3. legami tra tipo di apertura (per lettura o per scrittura) e relative operazioni.

È interessante notare il **legame** tra l'attività di analisi del flusso di dati e i diagrammi UML a stati finiti: un *oggetto* risponde a una certa *tipologia di eventi*, può essere in diversi *stati* e in certi *stati*

non sono ammesse alcune *operazioni*. Si noti come nessuna delle due discipline entra comunque nel merito del valore delle variabili, relegato ad un'analisi a runtime.

Criterio di *copertura del budget*

Molto spesso nei contesti reali l'unico criterio applicato è quello di **copertura del budget**: si continuano a creare casi di test finché non sono finite le risorse (tempo e soldi). Questa tecnica ovviamente non fornisce alcuna garanzia sull'efficacia dei test, ed è sicuramente sconsigliata.

Tecniche di review

Finora abbiamo esplorato tecniche più o meno *automatiche* per la ricerca di errori, che stimolavano il programma con specifici input o ne analizzavano il codice per individuare potenziali anomalie.
Tuttavia, alcuni tipi di errori non possono essere rilevati con questi metodi: si tratta soprattutto errori legati a *incomprensione delle specifiche*. Del resto, attività come il testing richiedono che il programmatore fornisca l'output "corretto" che si aspetta dal programma che viene confrontato con quello effettivo per scovare eventuali differenze: se chi scrive il codice non comprende in primo luogo *cosa* dovrebbe fare il suo software non c'è modo di individuare l'errore.

Per questo motivo molto spesso il codice viene sottoposto ad un'attività di **review**, in cui un operatore umano ne analizza la struttura e il funzionamento: egli sarà chiaramente in grado di cogliere una serie di **errori semantici** che sfuggono alla comprensione dei tool automatici di test. Spesso questa mansione viene svolta da un **team di testing** separato dal team di sviluppo: non solo questo promuove l'effettiva ricerca di errori (*mentre gli sviluppatori avrebbero tutto l'interesse di non trovarne nessuno*), ma sottopone il software a uno sguardo esterno più critico e imparziale.

Anche per la review esistono una serie di tecniche: vediamone quindi le principali.

- **Bebugging**
- **Analisi mutazionale**
- **Object oriented testing**
- **Testing funzionale**
- **Software inspection**
- **Modelli statistici**
- **Debugging**

Bebugging

Talvolta può capitare che il team di testing **non trovi errori** nel programma sotto osservazione. Oltre ad essere scoraggiante per chi esegue la review questo è spesso indice del fatto che tale attività non viene svolta in maniera corretta, poiché raramente un programma è effettivamente corretto al 100% dopo la prima scrittura.

Un metodo efficace per risolvere questo problema è il cosiddetto **bebugging**, una tecnica secondo la quale gli sviluppatori **inseriscono deliberatamente n errori** nel codice prima di mandarlo in analisi al team di testing, a cui viene comunicato il numero n di errori da trovare. L'ovvio vantaggio di questa tecnica è l'**incentivo** per il team di testing a continuare a cercare errori, facendo sì che durante la ricerca ne vengano scovati molti altri non ancora noti.

La metrica utilizzata per valutare l'efficacia del testing tramite questa tecnica è dunque la **percentuale di errori trovati** tra quelli inseriti artificialmente, che può fornire un'indicazione della frazione di errori che il team di testing è in grado di trovare. Se per esempio il team di sviluppo ha aggiunto 10 bug *"artificiali"* e durante il testing ne vengono trovati 8 più 2 non noti, si può supporre che il team di review riesce a trovare l'*80% degli errori* e che quindi ce ne è ancora un altra porzione di errori *reali* da scovare.
Bisogna però essere molto cauti nel fare considerazioni di questo tipo: è possibile che gli errori immessi artificialmente siano **troppo facili** o **troppo difficili** da trovare, per cui conviene sempre prendere tutto con le pinze.

Analisi mutazionale

Una evoluzione del bebugging è l'**analisi mutazionale**. Dato un programma P e un test T (*insieme di casi di test*), viene generato un insieme di programmi Π *simili* al programma P in esame: tali programmi prendono il nome di **mutanti**.
Si esegue poi il test T su ciascun mutante: se P era corretto i programmi in Π **devono essere sbagliati**, ovvero devono produrre un **risultato diverso** per almeno un caso di test $t \in T$. Se così non fosse, infatti, vorrebbe dire che il programma P non viene opportunamente testato nell'aspetto in cui si discosta dal mutante che non ha sollevato errori, per cui non si può essere sicuri della sua correttezza. Non viene cioè testata la correttezza del programma, ma piuttosto **quanto il test è approfondito**.

Si può quindi valutare la capacità di un test di rilevare le differenze introdotte nei mutanti tramite un nuovo criterio di copertura, che prende il nome di **criterio di copertura dei mutanti**.

- **Criterio di copertura dei mutanti**
- **Generazione dei mutanti**
- **Automazione**

Criterio di copertura dei mutanti

*Un test T soddisfa il **criterio di copertura dei mutanti** se e solo se per ogni mutante $\pi \in \Pi$ esiste almeno un caso di test $t \in T$ la cui esecuzione produca per π un risultato diverso da quello prodotto da P.*

La metrica di valutazione di questo criterio è la **frazione di mutanti π riconosciuta come diversa** da P sul totale di mutanti generati. Se non tutti i mutanti vengono scovati sarà necessario aggiungere dei casi di test che li riconoscano.

I tre passi da seguire per costruire un test tramite l'analisi mutazionale sono quindi:

1. **analisi** delle classi e generazione dei mutanti;
2. **selezionare** dei casi di test da aggiungere a T, in base alla metrica di cui sopra;
3. **esecuzione** dei casi di test sui mutanti, pensando anche alle performance;

Analizziamo ciascuno di tali step in maggior dettaglio.

Generazione dei mutanti

Idealmente i mutanti generati dovrebbero essere il **meno differenti possibile** dal programma di partenza, ovvero dovrebbe esserci **un mutante per ogni singola anomalia** che sarebbe possibile inserire nel programma.

Questo richiederebbe però di generare **infiniti** mutanti, mentre per mantenere la suite di test *eseguibile in tempi ragionevoli* il numero di mutanti non dovrebbe essere troppo elevato: un centinaio è una buona stima, ma un migliaio sarebbe auspicabile.
Visto il numero limitato è necessario dunque concentrarsi sulla "**qualità**" dei mutanti generati, dove i mutanti sono tanto più buoni quanto più permettono di scovare degli errori. Per questo motivo vengono creati degli specifici *operatori* che dato un programma restituiscono dei mutanti *utili*.

Operatori mutanti

Come già accennato, gli **operatori mutanti** sono delle funzioni (*o piccoli programmi*) che dato un programma P generano un insieme di mutanti Π. Essi operano eseguendo piccole **modifiche sintattiche** che modifichino la **semantica del programma** senza però causare errori di compilazione.

Tali operatori si distinguono in **classi** in base agli oggetti su cui operano:

- **costanti** e **variabili**, per esempio scambiando l'occorrenza di una con l'altra;
- **operatori** ed **espressioni**, per esempio trasformando `<` in `<=`, oppure `true` in `false`;
- **comandi**, per esempio trasformando un `while` in `if`, facendo così eseguire il ciclo una sola volta.

Alcuni operatori possono essere anche specifici su alcuni tipi di applicazioni, come nel caso di:

- operatori per **sistemi concorrenti**: operano principalmente sulle primitive di sincronizzazione – come eseguire una `notify()` invece che una `notifyAll()`;
- operatori per **sistemi object-oriented**: operano principalmente sulle interfacce dei moduli.

Poiché la generazione dei mutanti è un'attività tediosa, il compito di applicare questi operatori viene spesso affidato a tool automatici. Esistono però numerosi **problemi di prestazioni**, in quanto per ogni mutante occorre modificare il codice, ricompilarlo, controllare che non si sovrapponga allo spazio di compilazione delle classi di altri mutanti e fare una serie di altre operazioni che comportano un pesante overhead. Per questo motivo i tool moderni lavorano spesso sull'**eseguibile** in sé (*sul bytecode nel caso di Java*): sebbene questo diminuisca il lavoro da fare per ogni mutante è possibile che il codice eseguibile così ottenuto sia un programma che non sarebbe possibile generare tramite compilazione. Si espande quindi l'universo delle possibili anomalie anche a quelle *non ottenibili*, un aspetto che bisognerà tenere in considerazione nella valutazione della metrica di copertura.

High Order Mutation

Normalmente i mutanti vengono generati introducendo una *singola modifica* al programma originale. Nella variante **HOM** (**High Order Mutation**) si applicano invece modifiche a **codice già modificato**.

La giustificazione per tale tecnica è che esistono alcuni casi in cui trovare errori dopo aver applicato più modifiche è *più difficile* rispetto ad applicarne solo una. Può essere che un errore mascheri parzialmente lo stato inconsistente dell'altro rendendo più difficile il rilevamento di malfunzionamenti, cosa che porta a generare test ancora più approfonditi.

Automazione

Generalmente nel testing gli unici due *outcomes* sono *risultato corretto* o *non corretto* e la metrica è una misura della correttezza del programma. Il discriminante delle tecniche di analisi mutazionale è invece il numero di casi di test che forniscono un risultato ***diverso*** da quello di P, indipendentemente dalla correttezza (di entrambi).

Come già detto, trovare errori con queste tecniche (specialmente l'HOM) misura quindi il **livello di approfondimento** dei casi di test e **non** la **correttezza** del programma di partenza. Prescindere dalla *correttezza* dei risultati ha però un aspetto positivo: per eseguire l'analisi mutazionale non è necessario conoscere il comportamento corretto del programma, eliminando la necessità di un *oracolo* che ce lo fornisca. Si può quindi misurare la bontà di un insieme casi di test **automatizzando la loro creazione**: come già detto precedentemente, occorre però vigilare sulla **proliferazione del numero di esecuzioni** da effettuare per completare il test – un caso di test dà infatti origine a $n + 1$ esecuzioni, dove n è il numero di mutanti.

Il seguente diagramma di flusso visualizza quindi l'attività **facilmente automatizzabile** di analisi mutazionale:

Benché semplice, questo algoritmo **non garantisce la terminazione** per una serie di motivi:

- quando si estrae un caso di test casuale, c'è sempre il rischio di **estrarre sempre lo stesso**;
- si potrebbe essere *particolarmente sfortunati* e **non trovare un caso di test utile** in tempo breve;
- **esistono infinite varianti** di programmi **funzionalmente identici** ma **sintatticamente diversi**, ovvero che svolgono la stessa funzione anche se sono diversi: una modifica sintattica potrebbe non avere alcun effetto sul funzionamento del programma, come per esempio scambiare `<` con `<=` in un algoritmo di ordinamento. In tal caso, nessun nuovo caso di test permetterebbe di coprire il mutante, in quanto esso restituirebbe sempre lo stesso output del programma originale.

Spesso viene quindi posto un timeout sull'algoritmo dipendente sia dal tempo totale di esecuzione, sia dal numero di casi di test estratti.

Per verificare la validità del test, è necessario controllare il **numero di mutanti generati**: se questo numero è elevato, il test non era affidabile. In alternativa, è possibile *"nascondere"* i mutanti, a patto che non sia richiesta una copertura totale. In questo modo, è possibile **analizzare programmi** che sono **funzionalmente uguali ma sintatticamente diversi**, al fine di dimostrarne l'equivalenza o scoprire casi in cui essa non è valida.

Object-oriented testing

Finora abbiamo trattato i programmi come funzioni matematiche da un dominio di input a un dominio di output. Questo è tutto sommato vero per quanto riguarda i **linguaggi procedurali**: un programma in tale paradigma è composto da un insieme di funzioni e procedure che preso un dato in ingresso ne restituiscono uno in uscita. A meno di eventuali variabili globali

condivise (il cui uso è comunque sconsigliato), tali funzioni sono indipendenti l'una dall'altra, e possono quindi essere *testate indipendentemente* come fossero dei piccoli sotto-programmi.

La situazione cambia per quanto riguarda invece i **linguaggi object oriented** (**OO**), che introducono i concetti di classe e istanza: in tali linguaggi gli oggetti sono l'**unione di metodi e stato**. Le tecniche di testing viste finora smettono quindi di funzionare: la maggior parte delle volte testare i metodi isolatamente come funzioni da input ad output perde di senso, in quanto non si considera il contesto (lo *stato* dell'oggetto associato) in cui essi operano.

Bisogna dunque sviluppare una serie di tecniche di test specifiche per i linguaggi orientati agli oggetti, in cui l'**unità testabile** si sposti dalle procedure alle **classi**.

Ereditarietà e collegamento dinamico

Prima di capire *come* è possibile testare un'intera classe, affrontiamo due punti critici che derivano dal funzionamento intrinseco dei linguaggi a oggetti: l'**ereditarietà** e il **collegamento dinamico**.

Partiamo dal primo e immaginiamo di avere una classe già completamente testata. Creando ora una sottoclasse di tale classe originale può sorgere un dubbio: *visto che i metodi ereditati sono già stati testati nella classe genitore ha senso testarli nella classe figlia?* Un quesito simile sorge nel caso di metodi di default appartenenti a un'interfaccia: *ha senso testare i metodi di default direttamente nell'interfaccia o è meglio testarli nelle classi concrete che implementano tale interfaccia?*
Il consenso degli esperti è di **testare nuovamente tutti i metodi ereditati**: nelle sottoclassi e nelle classi che implementano delle interfacce con metodi di default tali metodi opereranno infatti in **nuovi contesti**, per cui non vi è alcuna certezza che funzionino ancora a dovere. Inoltre, a causa del collegamento dinamico non è nemmeno sicuro che eseguire lo stesso metodo nella classe base significa eseguire le stesse istruzioni nella classe ereditata.
In generale dunque non si eredita l'attività di testing, ma si possono invece ereditare i casi di test e i relativi valori attesi (*l'oracolo*): è perciò opportuno **rieseguire** i casi di test anche nelle sottoclassi.

Un altro motivo per cui il testing object-oriented differisce fortemente da quello per linguaggi funzionali è la preponderanza del **collegamento dinamico**, attraverso il quale le chiamate ai metodi vengono collegate a runtime in base al tipo effettivo degli oggetti. Dal punto di vista teorico, infatti, tale meccanismo rende difficile stabilire staticamente tutti i possibili cammini di esecuzione, complicando la determinazione dei criteri di copertura.

Testare una classe

Entriamo ora nel vivo della questione. Per **testare una classe**:

- la **si isola** utilizzando più *classi stub* possibili per renderla eseguibile indipendentemente dal contesto;
- si implementano eventuali **metodi astratti** o non ancora implementati (stub);
- si aggiunge una funzione per permettere di estrarre ed esaminare lo stato dell'oggetto e quindi **bypassare l'incapsulamento**;
- si costruisce una classe driver che permetta di istanziare oggetti e chiamare i metodi secondo il **criterio di copertura** scelto.

Ebbene sì, sono stati progettati dei criteri di copertura specifici per il testing delle classi. Vediamo dunque di cosa si tratta.

Copertura della classe

I **criteri classici** visti precedentemente (comandi, decisioni, ...) continuano a valere ma **non sono sufficienti**. Per testare completamente una classe occorre considerare lo **stato dell'oggetto**: in particolare, è comodo utilizzare una **macchina a stati** che rappresenti gli *stati possibili* della classe e le relative *transazioni*, ovvero le chiamate di metodi che cambiano lo stato.

Tale rappresentazione potrebbe esistere nella documentazione o essere creato specificatamente per l'attività di testing. Il seguente diagramma rappresenta per esempio una macchina a stati di una classe avente due metodi, `m1` e `m2`.


Grafo criteri di copertura

Ottenuta una rappresentazione di questo tipo, alcuni criteri di copertura che si possono ipotizzare sono:

- **coprire tutti i nodi**: per ogni **stato** dell'oggetto deve esistere almeno un caso di test che lo raggiunge;
- **coprire tutti gli archi**: per ogni stato e per ogni metodo deve esistere almeno un caso di test che esegue tale metodo trovandosi in tale stato;
- **coprire tutte le coppie di archi input/output**: per ogni stato e per ogni coppia di archi entranti e uscenti deve esistere almeno un caso di test che arriva nello stato tramite l'arco entrante e lo lascia tramite l'arco uscente (consideriamo anche *come* siamo arrivati nello stato);
- **coprire tutti i cammini identificabili nel grafo**: spesso i cammini in questione sono infiniti, cosa che rende l'applicazione di questo criterio infattibile (*"sopra la linea rossa"*).

Tipo di testing: white o black box?

Abbiamo assunto che il diagramma degli stati facesse parte delle specifiche del progetto. Se così fosse, allora il testing appena descritto assume una connotazione **black box**: il diagramma rappresenta sì la classe ma è ancora una sua **astrazione**, che non considera il codice effettivo che rappresenta lo stato o che implementa uno specifico metodo ma solo le relazioni tra i vari stati.

In caso il diagramma degli stati non sia però fornito, il testing delle classi è comunque possibile! Attraverso tecniche di **reverse engineering** guidate da certe euristiche (che operano ad un livello di astrazione variabile) è possibile ad **estrarre informazioni sugli stati** di una *classe già scritta*; spesso tali informazioni non sono comprensibili per un essere umano, motivo per cui esse vengono piuttosto utilizzate da vari tool di testing automatico. In questo caso, però, il testing assume caratteristiche **white box**, in quanto il codice che implementava la classe era già noto prima di iniziare a testarlo.

Testing funzionale

Introduciamo ora una nuova attività di testing che parte da presupposti completamente diversi rispetto a quelli del test strutturale.

Il **test funzionale** è infatti un tipo di test che si concentra sulla verifica del comportamento del programma dal punto di vista dell'**utente finale**, senza considerare il suo funzionamento interno. In altre parole, il test funzionale è un approccio **black box** in cui non si ha (o non comunque non si sfrutta) la conoscenza del codice sorgente.

Talvolta questo può essere l'**unico approccio possibile** al testing, come nel caso di validazione del lavoro di un committente esterno; altre volte invece si decide volontariamente di farlo, concentrandosi sul **dominio delle informazioni** invece che sulla struttura di controllo.
Il test funzionale, che prende in considerazione le **specifiche** (e non i requisiti) del progetto per discriminare un comportamento corretto da uno scorretto, permette infatti di identificare errori che **non possono essere individuati** con criteri strutturali, come per esempio funzionalità non implementate, flussi di esecuzione dimenticati o errori di interfaccia e di prestazioni.

Le tecniche di test funzionale si possono raggruppare in:

- **metodi basati su grafi**: oltre alle tecniche già viste in precedenza, si può per esempio lavorare anche sui diagrammi di sequenza;
- **suddivisioni del dominio in classi di equivalenza**: si possono raggruppare i valori del dominio che causano lo stesso comportamento in classi d'equivalenza, così da testare tutti i *comportamenti distinti* piuttosto che tutti i possibili valori del dominio. Occorre fare

attenzione a non fare l'inverso, ovvero a concentrarsi sui soli valori appartenenti ad una classe di equivalenza ignorando il resto;
- **analisi dei valori limite (test di frontiera)**: si testano, tra tutti i possibili valori del dominio, quelli *"a cavallo"* tra una categoria e l'altra, in quanto essi possono più facilmente causare malfunzionamenti;
- **collaudo per confronto**: si confronta la nuova versione del programma con la vecchia, assicurandosi che non siano presenti regressioni. Non solo si possono confrontare gli eseguibili, ma anche *specifiche formali eseguibili* che rappresentino le caratteristiche importanti del software;

Non tutte le metodologie di testing funzionale ricadono però in una di queste categorie, e la più notevole è sicuramente il **testing delle interfacce**, di cui diamo un assaggio prima di passare a parlare di classi di equivalenza.

Testing delle interfacce

Questa tecnica mira a testare come i vari sotto-sistemi del programma dialoghino e **collaborino** tra loro: per "interfacce" non si intendono infatti le `interface` Java o le *signature*, ma l'insieme di funzionalità che permettono l'interoperabilità dei componenti.
Esistono in particolare diversi tipi di interfacce:

- a **invocazione di parametri**;
- a **condivisione di memoria**;
- a **metodi sincroni**;
- a **passaggio di messaggi**.

Le interfacce aderenti a ciascuna categoria possono essere analizzate in modi diversi alla ricerca di anomalie. Gli sbagli più comuni sono per esempio **errori nell'uso dell'interfaccia**, come il passaggio di parametri in ordine o tipo errato oppure assunzioni sbagliate circa ciò che le funzionalità richiedono (*precondizioni*), ed **errori di tempistica o di sincronizzazione** tra componenti.

Classi di equivalenza

La tecnica delle classi di equivalenza si pone l'obiettivo di dividere il dominio del programma in **classi di dati**, ovvero gruppi di valori di input che *dovrebbero* **stimolare il programma nella**

stessa maniera. Non si tratta quindi di classi di equivalenza degli output, ovvero valori che dati in pasto al programma forniscono lo stesso risultato, quanto piuttosto valori che dati in pasto al programma forniscono un risultato diverso ma *prodotto nello stesso modo*.

Una volta individuate le classi di dati l'obiettivo sarebbe quindi di estrarre da esse casi di test in modo da testare il funzionamento del programma in tutti suoi funzionamenti standard.
In realtà, dunque, si cercano di individuare casi di test che rivelino eventuali **classi di equivalenza di errori**, ovvero insiemi di valori che generano malfunzionamenti per lo stesso motivo. Classi di equivalenza di questo tipo sono solitamente più *"stabili"* rispetto alle normali classi di equivalenza in quanto il risultato ottenuto, ovvero l'errore, è spesso lo stesso o molto simile.

Volendo dare una definizione più formale, *una **classe di equivalenza** rappresenta un insieme di **stati validi o non validi** per i dati in input e un insieme di stati validi per i dati di output*, dove per dato si intendono valori, intervalli o insiemi di valori correlati.
È importante comprendere anche i possibili *stati non validi* in quanto bisogna testare che il programma reagisca bene all'input mal formattato. Ogni dominio avrà quindi almeno due classi di equivalenza:

- la classe degli **input validi**
- la classe degli **input non validi**

Per fare un esempio si può considerare un programma che chiede in input un **codice PIN di 4 cifre**. Il suo dominio può quindi essere suddiviso in due semplici classi di equivalenza:

1. PIN corretto;
2. tutti i numeri di 4 cifre diversi dal PIN.

Volendo fare un altro esempio, se ci si aspetta che i valori in input ricadano in un intervallo, per esempio $[100,\ 700]$), si possono definire la classe di equivalenza valida $x \in [100, 700]$ e la classe di equivalenza non valida $x \notin [100, 700]$. Per voler aumentare la granularità si può però spezzare la classe degli input non validi in due, ottenendo una classe valida e due non valide:

1. $x \in [100, 700]$;
2. $x < 100$;
3. $x > 700$.

Come si vede, la scelta delle classi di equivalenza da considerare non è univoca, e richiede un minimo di conoscenza di dominio. Alternativamente esistono delle tecniche standard di individuazione delle classi di equivalenza a partire dalle specifiche che prendono il nome di **category partition**.

Test di frontiera

La tecnica dei **test di frontiera** è *complementare* a quella delle classi di equivalenza. Partendo dal presupposto che gli errori tendono ad accumularsi sui **casi limite**, ovvero quelli la cui gestione è più particolare, questa tecnica suggerisce di selezionare come casi di test non valori a caso all'interno delle classi di equivalenza, ma i valori presenti **al confine** tra di loro.

Category partition

La tecnica di **category partition** è un metodologia che permette di caratterizzare e identificare le classi di equivalenza del dominio di un problema a partire dalle sue specifiche. Può essere utilizzata a **vari livelli** a seconda che si debbano realizzare test di unità, test di integrazione e o test funzionali.

Il metodo è composto da una serie di passi in sequenza:

1. **analisi delle specifiche**: in questa fase vengono identificate le *unità funzionali individuali* che possono essere verificate singolarmente; non necessariamente sono un'unica classe, è sufficiente che siano componenti facilmente separabili dal resto, sia a livello di testing che concettuale. Per ogni unità vengono quindi identificate delle caratteristiche (**categorie**) dei parametri e dell'ambiente in cui opera;
2. **scegliere dei valori**: per ogni categoria, occorre scegliere quali sono i *valori sensati* su cui fare riferimento;
3. **determinare eventuali vincoli tra le scelte**, che non sono sempre indipendenti;
4. **scrivere test e documentazione**.

Per capire meglio ciascuna di tali fasi vediamo un'esempio di utilizzo della tecnica di *category partition* prendendo come soggetto il comando `find` della shell Linux.

PASSO 1 – analizzare le specifiche

Per prima cosa analizziamo le specifiche del comando:

> **Syntax**: `find <pattern> <file>`
>
> The find command is used to locate one or more instances of a given pattern in a file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occur in it.

> The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes (`"`). To include a quotation mark in the pattern, two quotes (`""`) in a row must be used.

Vista la relativa semplicità, `find` è un'unità funzionale individuale che può essere verificata separatamente. Bisogna dunque individuarne i parametri: come è chiaro dalla sintassi essi sono due, il `pattern` da cercare e il `file` in cui farlo.

Ora, ciascuno di tali parametri può possedere determinate caratteristiche, ed è nostro compito in questa fase comprenderle ed estrarle.
Tali caratteristiche possono essere di due tipi: **esplicite**, ovvero quelle ricavabili direttamente dalla lettura specifiche, e **implicite**, ovvero quelle che provengono dalla nostra conoscenza del dominio di applicazione e che quindi non vengono specificate.

Tornando al nostro caso di studio possiamo per esempio ottenere la seguente tabella:

Oggetto	Caratteristiche esplicite	Caratteristiche implicite
`pattern`	• lunghezza del pattern; • pattern tra doppi apici; • pattern contenente spazi; • pattern contenente apici.	• pattern tra apici con/senza spazi; • più apici successivi inclusi nel pattern.
`file` **(nome)**	*(nessuna)*	• caratteri nel nome ammissibili o meno; • file esistente (con permessi di lettura) o meno.
`file` **(contenuto)**	• numero occorrenze del pattern nel file; • massimo numero di occorrenze del pattern in una linea; • massima lunghezza linea.	• pattern sovrapposti; • tipo del file.

È importante *esplicitare le caratteristiche implicite* dei parametri dell'unità funzionale perché **le specifiche non sono mai complete** e solo così possiamo disporre di tutti gli elementi su cui ragionare nelle fasi successive.

Si presti poi attenzione alla distinzione fatta tra il *nome* del file e il suo *contenuto*: il primo infatti è un **parametro** che viene passato al comando per iniziarne l'esecuzione, mentre il secondo fa parte dell'**ambiente** in cui il comando opera ed è dunque soggetto ad una sottile distinzione concettuale.

ALPHA E BETA TESTING

Spesso, però, analizzare le specifiche non basta per comprendere tutte le variabili che entrano in gioco durante l'esecuzione di un programma. Bisogna infatti ricordare che ci sono moltissime altre caratteristiche d'ambiente che ancora **non sono state considerate**: la versione del sistema operativo, del browser, il tipo di architettura della macchina su cui gira il programma eccetera.

Spesso, quindi, la fase di testing funzionale si divide in due fasi:

- **alpha testing**: l'unità funzionale viene testata in-house, ovvero su una macchina all'interno dello studio di sviluppo. In questa fase si considerano soprattutto le caratteristiche legate alle specifiche di cui sopra;
- **beta testing**: per testare varie *configurazioni d'ambiente* una versione preliminare del programma viene distribuito in un *ambiente variegato* per osservare come esso si comporta sulle macchine di diversi utenti.

Per il momento, però, consideriamo solo la fase di alpha testing e le categorie ad essa relative.

PASSO 2 – scegliere dei valori

Individuate le caratteristiche dei parametri e delle variabili d'ambiente da cui l'unità funzionale dipende, che prendono il nome di **categorie**, si passa quindi alla seconda fase.

In questa fase si devono identificati **tutti e i soli** *casi significativi* per ogni categoria, ovvero quei valori della stessa che si ritiene abbia senso testare; poiché si tratta di un compito molto soggettivo è importante in questa fase avere **esperienza** (*know-how*) nel dominio d'applicazione.

Per capire meglio di cosa stiamo parlando ritorniamo al nostro esempio e consideriamo il parametro `pattern`. Per ciascuna delle sue categorie possono essere individuati vari casi significativi:

- **lunghezza del pattern**: vuoto, un solo carattere, più caratteri, più lungo di almeno una linea del file;
- **presenza di apici**: pattern tra apici, pattern non tra apici, pattern tra apici errati;
- **presenza di spazi**: nessuno spazio nel pattern, uno spazio nel pattern, molti spazi nel pattern;
- **presenza di apici interni**: nessun apice nel pattern, un apice nel pattern, molti apici nel pattern.

È interessante notare il *mantra* già visto del "**nessuno**, **uno**, **molti**", spesso molto utile in questa fase.

PASSO 3 – determinare i vincoli tra le scelte

Trovati tutti i valori significativi delle categorie dell'unità funzionale come possiamo costruire i casi di test da utilizzare per verificarne la correttezza?

Si potrebbe pensare di testare **tutte le combinazioni** di valori significativi, facendo cioè il prodotto cartesiano tra le categorie. Nella pratica, però, ciò risulterebbe in un numero esagerato di casi di test: già solo nel nostro semplice esempio questi sarebbero ben 1944, decisamente **troppi**.

Nel tentativo di evitare quest'esplosione combinatoria ci si accorge però che spesso le anomalie sorgono dall'interazione di **coppie** di caratteristiche indipendentemente dal valore assunto da tutte le altre: per esempio, un problema potrebbe presentarsi se si usa il browser *Edge* sul sistema operativo *Linux*, indipendentemente da caratteristiche quali la dimensione dello schermo, l'architettura del processore eccetera.
Per ridurre il numero di casi di test si sviluppa quindi la tecnica del ***pairwise testing***, che riduce l'insieme delle configurazioni da testare a tutte le combinazioni di coppie di valori. È quindi presente almeno un caso di test *per ogni coppia ipotizzabile* di valori: in rete e in Java sono presenti diversi **strumenti** che permettono di creare casi di test combinati con il metodo *pairwise*.

Un'ulteriore tentativo di ridurre il numero di casi di test prevede di definire una serie di ***vincoli*** per la generazione delle coppie, escludendo particolari combinazioni di caratteristiche: così, per esempio si potrebbe escludere la coppia "OS == MacOs" e "browser == Edge" perché sfruttando la conoscenza di dominio sappiamo che tale browser non è disponibile sul suddetto sistema operativo.
Volendo essere più precisi, la creazione di vincoli prevede un passaggio intermedio: vengono definite una serie di **proprietà** (es. *NotEmpty* o *Quoted* per l'esempio su `find`) e si creano dei vincoli logici a partire da esse. I vincoli seguono poi una struttura tra le seguenti:

- **se**: si può limitare l'uso di un valore solo ai casi in cui è definita una proprietà. Per esempio, è inutile testare il caso *"il file non esiste"* se la proprietà *NotEmpty* si manifesta;
- **single**: alcune caratteristiche prese singolarmente anche se combinate con altre generano lo stesso risultato. Per esempio, se il file non contiene occorrenze del pattern cercato il risultato del programma è indipendente dal tipo di pattern cercato;
- **error**: alcune caratteristiche generano semplicemente errore, come per esempio se si omette un parametro.

PASSO 4 – scrivere i test

Fissati i vincoli e fatti i calcoli combinatori si procede ad enumerare iterativamente tutti i casi di test generati continuando ad aggiungere vincoli fino ad arrivare ad un **numero ragionevole**.

Ovviamente, i casi di test avranno poi bisogno di **valori specifici** per le caratteristiche: non basta dire "pattern con apici all'interno", bisogna creare un pattern aderente a questa descrizione! Fortunatamente questa operazione è solitamente molto facile, anche con tool automatici.

Conclusioni

Per quanto intuitiva e utile, la tecnica di category partition presenta due criticità:

- individuare i **casi significativi** delle varie caratteristiche può essere difficile e si può sbagliare, anche utilizzando mantra come "*zero*, *uno*, *molti*";
- una volta generati i casi di test serve comunque un "**oracolo**" che fornisca la risposta giusta, ovvero quella che ci si attende dall'esecuzione sul caso di test. L'attività non è dunque *completamente* automatizzabile.

Va però detto che esistono delle tecniche di **property-based testing** che cercano di eliminare la necessità di un oracolo considerando particolari proprietà che dovrebbero sempre valere durante l'esecuzione (invarianti) piuttosto che analizzare il risultato dell'esecuzione dei casi di test per determinare la correttezza del programma.

Object orientation e testing funzionale

Trattandosi di un approccio **black box** che ragiona sulle **funzionalità** e non sui dettagli implementativi, l'introduzione del paradigma a oggetti **non dovrebbe cambiare nulla** per quanto riguarda il testing funzionale. Se questa affermazione è vera per quanto riguarda la verifica di singole unità funzionali, lo stesso non si può dire nel caso di **test di integrazione**.

Nei linguaggi procedurali i test di integrazione sono infatti scritti secondo logiche alternativamente **bottom-up** o **top-down**: esiste cioè un punto di partenza dal quale partire ad aggregare le componenti, seguendo cioè una qualche forma di **albero di decomposizione** del programma.
Per quanto riguarda la programmazione a oggetti, invece, la situazione è **molto più caotica**: le relazioni tra le classi sono spesso cicliche e non gerarchiche (tranne per l'ereditarietà — la relazione meno interessante), in una serie di *inter-dipendenze* che rendono difficoltoso individuare un punto da cui partire a integrare.

Relazioni interessanti in questa fase sono infatti *associazioni*,*aggregazioni* o *dipendenze*, ma rendono complicato identificare il **sottoinsieme di classi da testare**. Per fare ciò si possono comunque utilizzare alcuni strumenti già visti:

- si può partire dai diagrammi degli **use cases e scenari** per testare i componenti citati;
- si possono osservare i **sequence diagram** per testare le classi protagoniste delle interazioni a scambio di messaggi descritte;
- si possono infine usare gli **state diagram** nella modalità che abbiamo già descritto.

Software inspection

Un'altra classe di tecniche di verifica e convalida è la **software inspection**, ovvero tecniche manuali per individuare e correggere gli errori basati su una attività di gruppo in cui si analizza il codice insieme passo passo: si pensi per esempio alla tecnica di *pair programming* già ampiamente citata parlando di XP.

Le tecniche di software inspection sono molto interessanti in quanto hanno **pochi requisiti** e l'unico **tool** richiesto è un essere **umano** che si prenda la briga ispezionare il codice, spesso in riunioni di gruppo da 5-6 persone.

Trattandosi di una tecnica umana essa è molto **flessibile**: l'**oggetto sotto ispezione** può essere una porzione di codice non funzionante, una serie di specifiche formali o informali o direttamente l'eseguibile compilato. La software inspection può quindi essere eseguita durante tutte le fasi del ciclo di vita di un programma.

Fagan code inspection

La **Fagan code inspection** è una metodologia sviluppata da Michael Fagan alla IBM negli anni '70. La metodologia prevede che un **gruppo di esperti** esegua una serie di passi per verificare la correttezza del codice sorgente al fine di individuare eventuali errori, incongruenze o altri problemi.
È **la più diffusa** tra le tecniche di ispezione, nonché la più rigorosa e definita.

Ruoli

Essendo un'attività di gruppo, nella Fagan code inspection vengono identificati diversi ruoli:

- **Moderatore**: è colui che coordina i meeting, sceglie i partecipanti e ha la responsabilità di far rispettare le regole di cui parleremo tra poco. È di solito una persona che lavora ad un progetto diverso da quello in esame in modo da evitare conflitti di interessi.
- **Readers e Testers**: non sono persone diverse, semplicemente a seconda dei momenti i partecipanti possono coprire uno di questi due ruoli: i primi leggono il codice al gruppo, mentre i secondi cercano difetti al suo interno. La lettura del codice è una vera e propria *parafrasi* di esso, ovvero un'interpretazione del codice nella quale si spiega quello che fa ma seguendo comunque la sua struttura.
- **Autore**: è colui che ha scritto il codice sotto ispezione; è un partecipante passivo che risponde solo a eventuali domande. È simile al ruolo del *cliente* nell'eXtreme Programming: pronto a rispondere a qualsiasi domanda per accelerare il lavoro degli altri.

Processo

Definiti i ruoli, secondo la tecnica **Fagan** di ispezione del codice il processo si articola come segue:

1. **Planning**: in questa prima fase il moderatore sceglie i partecipanti, si definiscono i loro ruoli e il tempo da dedicare alla ispezione, pianificando anche i vari incontri.
2. **Overview**: viene fornito a tutti i partecipanti materiale sul progetto per permettere loro di farsi un'idea del contesto in cui l'oggetto dell'ispezione si inserisce in ottica della riunione vera e propria.
3. **Preparation**: i partecipanti *"offline"* comprendono il codice e la sua struttura autonomamente sulla base anche del materiale distribuito nella fase precedente;
4. **Inspection**: la vera e propria fase di ispezione. In questa fase si verifica che il codice soddisfi le regole definite in precedenza e si segnalano eventuali problemi o anomalie. Durante l'ispezione, il gruppo di esperti esamina il codice riga per riga, confrontandolo con le specifiche e cercando di individuare errori, incongruenze o altri problemi.
5. **Rework**: una volta individuati i problemi, l'autore del codice si occupa di correggere i difetti individuati.
6. **Follow-up**: possibile re-ispezione del nuovo codice ottenuto dopo la fase precedente.

Se la maggior parte delle fasi è abbastanza autoesplicativa, è bene dare uno sguardo più approfondito all'attività di ispezione vera e propria.

Ispezione

Durante la fase di ispezione, l'obiettivo è **trovare e registrare** i difetti **senza correggerli**: la tentazione di correggere i difetti è sicuramente fortissima ma non è compito dei partecipanti alla riunione farlo. Ciascuno di loro potrebbe infatti avere le proprie idee e preferenze e metterli d'accordo potrebbe non essere facile: si preferisce quindi che sia l'autore del codice a correggere successivamente i problemi trovati.

Per evitare ulteriormente di perdere tempo sono previste *al massimo* 2 sessioni di ispezione di 2 ore al giorno, durante le quali lavorare approssimativamente a **150 linee di codice all'ora**. Quest'ultimo vincolo è **molto variable** in quanto cambia in base al linguaggio, al progetto, all'attenzione ai dettagli richiesta e alla complessità.

Una possibilità prevista in questa fase è anche quella di fare *"test a mano"*: si analizza il flusso di controllo del programma su una serie di casi di test così da verificarne il funzionamento.

Ancora più prominente è però l'uso di una serie di **checklist**, di cui parliamo nel prossimo paragrafo.

Checklist

Rispetto all'attività di testing, che a partire dai malfunzionamenti permetteva di risalire ai difetti e dunque agli sbagli commessi, il *thought-process* per le **checklist** è inverso: **si parte dagli *sbagli*** che più frequentemente hanno portato ad inserire determinate anomalie nel codice e si controlla che nessuno di questi sia stato commesso nuovamente.

In letteratura è reperibile la **conoscenza** di tutto ciò che è meglio evitare poiché in passato ha portato più volte ad avere anomalie nel codice. Tale conoscenza è raccolta in **checklist comuni** per i vari linguaggi.

Inoltre, l'ispezione del codice funziona così bene anche perché tali checklist possono essere **redatte internamente** all'azienda, in base all'esperienza passata e alla storia di un determinato progetto. \ Man mano che il progetto va avanti, l'**individuazione di un nuovo sbaglio** si traduce in un'**evoluzione della checklist**: dalla prossima ispezione si controllerà di non aver commesso lo stesso errore.

Esempio NASA

La NASA nel suo *Software Formal Inspections Guidebook* (1993) ha formalizzato circa **2.5 pagine di checklist** per C e 4 per FORTRAN.

Sono divise in *functionality*, *data usage*, *control*, *linkage*, *computation*, *maintenance* e *clarity*.

Di seguito sono elencati alcuni esempi dei punti di tali checklist:

- ☐ Does each module have a single function?
- ☐ Does the code match the Detailed Design?
- ☐ Are all constant names upper case?
- ☐ Are pointers not `typecast` (except assignment of `NULL`)?
- ☐ Are nested `#include` files avoided?
- ☐ Are non-standard usage isolated in subroutines and well documented?
- ☐ Are there sufficient comments to understand the code?

Struttura di incentivi

Perché l'ispezione del codice come è stata descritta funzioni bene, occorre prevedere una serie di **dinamiche positive** di incentivi al suo interno.

In particolare, è importante sottolineare che i difetti trovati **non devono essere utilizzati** per la valutazione del personale: questo evita che i programmatori nascondano i difetti nel proprio codice, minando la qualità del prodotto.

Dall'altro canto si possono invece considerare per la **valutazione di readers e tester** i difetti trovati durante l'ispezione, in modo che questi siano incentivati a fare l'ispezione più accurata possibile.

Variante: *active* design reviews

Purché il processo di ispezione funzioni al meglio **le persone** coinvolte **devono partecipare**, ma per come abbiamo descritto l'attività di Fagan Code Inspection nulla vieterebbe ai revisori non preparati di essere presenti ma non partecipare, rimanendo in silenzio e pensando ad altro.

Innanzitutto, per sopperire a questo problema i partecipanti andrebbero **scelti** tra persone di adeguata esperienza e sopratutto assicurando che nel team vi siano revisori per diversi aspetti nel progetto.

Qualora questo non bastasse, una variante del processo che prende il nome di **active design review** suggerisce che sia l'**autore** a leggere le checklist e sollevare questioni all'attenzione dei revisori, chiedendo diverse domande. Essendo presi direttamente in causa, i revisori saranno quindi costretti a partecipare.

Automazione

Sebbene l'ispezione del codice sia una tecnica manuale esistono diversi **strumenti di supporto automatici** in grado di velocizzare notevolmente il lavoro. Alcuni di questi tool possono aiutare con:

- **controlli banali**, come la formattazione; in fase di ispezione manuale si controllerà poi il risultato del controllo automatico;
- **riferimenti**: checklist e standard in formati elettronici facilmente consultabili e compilabili;
- **aiuti alla comprensione del codice**: tool che permettono di navigare e leggere il codice con maggiore facilità e spesso utilizzati durante attività di *re-engineering*;
- **annotazione e comunicazioni** del team, come l'email;
- **guida al processo e rinforzo**: non permettere di chiudere il processo se non sono stati soddisfatti alcuni requisiti (*come la necessità di approvazione prima del merge di una pull request*).

Pro e contro

Finito di descrivere il processo di ispezione del software possiamo chiederci: funziona? Prove empiriche parrebbero suggerire che la risposta sia **sì** e evidenziano anche che tale tecnica è particolarmente *cost-effective*.

I vantaggi dell'uso di questa tecnica di verifica e convalida sono infatti numerosi:

- Esiste un **processo rigoroso e dettagliato**;
- Si basa sull'**accumulo dell'esperienza**, auto-migliorandosi con il tempo (vd. *checklist*);
- Il processo integra una serie di **incentivi sociali** che spingono l'autore del codice ad analizzarlo in modo critico;
- A differenza del testing è possibile per la mente umana **astrarre il dominio completo** dei dati, considerando quindi in un certo senso tutti i casi di test;
- È applicabile anche a **programmi incompleti**.

La software inspection funziona così bene che è spesso utilizzata come *baseline* per valutare altre tecniche di verifica e convalida.

Questo non significa però che essa sia esente da **limiti**.
Innanzitutto il test può essere fatto **solo a livello di unità** in quanto la mente umana ha difficoltà a lavorare in situazioni in cui sono presenti molte informazioni contemporaneamente in assenza di astrazioni e indirettezze. Inoltre la software inspection **non è incrementale**: spesso infatti la fase di follow-up non è così efficace, in quanto il codice è cambiato talmente tanto che è necessario ricominciare l'ispezione da capo.

Ciò non toglie però che, come afferma la **Legge di Fagan (L17)**:

> *Le ispezioni aumentano in maniera significativa la produttività, qualità e la stabilità del progetto.*

Confronto tra tecniche di verifica e convalida

Numerosi studi hanno provato a confrontare l'efficacia di varie tecniche di testing, con particolare riferimento a testing strutturale, testing funzionale e software inspection. Un articolo del 2004 riporta in una **tabella di confronto** i risultati di alcuni di questi studi, considerando come metrica di valutazione delle tecniche di verifica e convalida la *percentuale media di difetti individuati*.


Confronto tecniche verifica e convalida

Come si può notare, a seconda dello studio appare più efficace l'una o l'altra tecnica; inoltre, la somma delle percentuali per ogni riga non è 100%, il che significa che alcuni difetti non possono essere rilevati da nessuna delle tre tecniche.
Nonostante ciò, si possono fare una serie di osservazioni: innanzitutto, l'efficacia di una o dell'altra tecnica dipende dalla **tipologia del progetto** su cui viene esercitata. Inoltre, **non è detto** che tecniche diverse trovino **gli stessi errori**: l'ispezione potrebbe aver trovato una certa tipologia di errore mentre il testing funzionale un'altra; le diverse tecniche controllano infatti diversamente aspetti differenti del programma, osservandolo da **diversi punti di vista**.

Confrontare le varie tecniche non è dunque necessariamente una perdita di tempo, mentre lo è sicuramente **confrontare solo i numeri**, come la varietà di risultati diversi ottenuti dai parte di studi diversi. Tra l'altro, dal riassunto della tabella si **perdono** informazioni sulle **modalità di rilevazione** dei dati, attribuendole ad espressioni generiche (come *comunemente*, *in media*, *progetti junior*, ...).

In conclusione, non c'è una risposta semplice al confronto e **non esiste una tecnica *sempre* migliore** rispetto alle altre.

Combinare le tecniche

Una domanda che sorge spontanea è chiedersi quindi cosa può succedere se si **combinano insieme** diverse tecniche di verifica e convalida.

Diversi studi mostrano che applicando tutte e quattro le tecniche qui descritte — anche se solo in modo superficiale — il risultato è sicuramente **più performante** delle tecniche applicate singolarmente.

Anche se una certa percentuale di errori può essere rilevata senza alcuna tecnica formale di verifica e convalida, semplicemente usando il software, si può infatti notare ciascuna tecnica presa singolarmente migliora tale percentuale e che la **combinazione** di tecniche diverse la incrementa ulteriormente. Questo perché tendenzialmente **ogni tecnica controlla aspetti differenti** e le rispettive aree di controllo si sovrappongono poco: è dunque conveniente applicare superficialmente ciascuna tecnica piuttosto che una sola tecnica in modo molto approfondito.

In conclusione, come afferma la **Legge di Hetzel-Meyer (L20)**:

> *Una combinazione di diversi metodi di V/C supera qualsiasi metodo singolo.*

Gruppi di test autonomi

È convinzione comune che colui che ha sviluppato un pezzo di codice sia la persona meno adatta a testarlo, come afferma la **Legge di Weinberg (L23)**:

> *Uno sviluppatore non è adatto a testare il suo codice.*

Di conseguenza, si preferisce spesso che il testing sia affidato ad un **gruppo di tester autonomi**. Questo implica infatti una serie di vantaggi, sia **tecnici** che e **psicologici**:

- **Aspetti tecnici**:
  - **maggiore specializzazione**: si evita così di richiedere che i propri sviluppatori siano anche esperti di testing;
  - **maggiore conoscenze delle tecniche di verifica e convalida** e dei relativi tool: chi fa il *tester* di lavoro acquisisce competenze specifiche sui tool e sugli strumenti di testing (spesso complessi), oltre che sui concetti di copertura e mutazioni.
- **Aspetti psicologici**:
  - **distacco dal codice**: a causa dell'assenza di modelli mentali precedenti su come il software dovrebbe operare, un tester esterno pone maggiore attenzione agli aspetti spesso trascurati o dimenticati;
  - **indipendenza nella valutazione**: una persona che testa il proprio codice è incentivata a *non* trovare molti errori in quanto potrebbe suggerire un lavoro di

dubbia qualità in fase di sviluppo. Un gruppo specializzato nel testing è invece incentivato a trovarne il più possibile per giustificare il loro impiego.

Ci sono tuttavia anche una serie di **svantaggi** legati all'avere un gruppo di tester autonomo. Innanzitutto, i problemi più ovvi sono legati all'**aspetto tecnico**: il fatto che i tester diventino specializzati nel testing significa che **perderanno** con il tempo la **capacità di progettare** e **codificare**, oltre a possedere una **minore conoscenza dei requisiti** del progetto.

Nell'analisi di Elisabeth Hendrickson denominata "**Better testing — worse quality?**" viene analizzata poi la tecnica sotto un **punto di vista psicologico**: come è possibile che un maggior investimento nel team di testing porti a un calo delle prestazioni in termini di numero di errori nel codice?

La risposta pare dipendere dal concetto di ***responsabilità***: seppur vero che l'attività di testing è compito del tester, è anche vero che è lo sviluppatore stesso che ha il compito di fare **test di unità** del proprio codice — il team di testing dovrebbe occuparsi solo di quello funzionale o di integrazione. Spesso però, a fronte di un aumento del personale nel team di testing e specialmente quando una deadline è vicina, il team di sviluppo tende a **spostare la responsabilità** di trovare gli errori ai tester, **abbassando la qualità del codice**.
Il team di testing troverà sì gli errori, riconsegnando il codice agli sviluppatori per correggerli, ma questo passaggio ulteriore implica una notevole **perdita di tempo** e risorse.

Inoltre, la presenza di un team di testing dedicato può generare **pressioni negative** sul team di sviluppo: ogni sviluppatore potrebbe sentirsi sotto costante valutazione da parte del team di testing.

Possibili alternative

Una possibile soluzione alle criticità appena evidenziate consisterebbe nella **rotazione del personale**: una stessa persona potrebbe ricoprire il ruolo di sviluppatore per un progetto e di tester per un altro. Questo approccio mostra diversi vantaggi, tra cui:

- **evitare pressioni negative**: ricoprendo diversi ruoli in diversi progetti, il personale non si dovrebbe sentire *giudicato* o *giudicante*;
- **evitare il progressivo depauperamento tecnico** dovuto ad all'eccessiva specializzazione;
- **evitare lo svuotamento dei ruoli**.

C'è però da considerare un certo **aumento dei costi di formazione** per via del raddoppio delle responsabilità individuali e un parallelo **aumento della difficoltà di pianificazione**: potrebbe succedere che la stessa persona debba lavorare a più progetti contemporaneamente, dovendo quindi dividere il proprio tempo e le proprie competenze.

Un'altra possibile alternativa consiste nella **condivisione del personale**, che prevede che siano gli stessi sviluppatori a occuparsi del testing: ciò permette di **sopperire** al problema di **scarsa conoscenza del software** in esame e del relativo dominio applicativo ma, oltre a far riemergere le **criticità** individuate precedentemente, aumenta le **difficoltà nella gestione dei ruoli**.

Modelli statistici

Negli ultimi tempi si stanno sviluppando una serie di **modelli statistici** sulla distribuzione degli errori nel codice che dovrebbero teoricamente aiutare l'attività di testing guidandola verso le porzioni di sorgente che *più probabilmente* potrebbero presentare difetti.

Tali modelli propongono infatti una **correlazione statistica** tra una serie di **metriche** quali la lunghezza del codice, il tipo di linguaggio, il grado massimo di indentamento etc. e:

- **la presenza di errori** per categoria di errore;
- **il numero di errori** per categoria di errore.

L'idea sarebbe quindi di **predire la distribuzione e il numero di errori** all'interno di uno specifico modulo del software in esame.

Occorre però **fare attenzione** alle conclusioni di queste statistiche. Utilizzare i risultati di tali modelli statistici come indicazioni sul fatto che su determinati moduli vada fatta più attività di testing rispetto ad altri potrebbe inizialmente sembrare la **soluzione più logica**. Tuttavia, tali risultati non considerano l'attività di testing già effettuata e le correzioni successive e quindi **non cambiano**: codice inizialmente *"scritto male"* secondo il modello rimarrà per sempre scritto male, anche se testato estensivamente.

Con ciò in mente, si cita spesso la **Legge di Pareto/Zipf (L24)**:

> *Circa l'80% dei difetti proviene dal 20% dei moduli.*

Sebbene tale affermazione è indubbiamente probabilmente vera, è difficile sfruttare questa nozione in quanto non sono conosciuti in principio i **moduli particolarmente problematici**, e il testing è comunque necessario anche in tutti gli altri.

Debugging

Il debugging è l'insieme di tecniche che mirano a **localizzare** e **rimuovere** le anomalie che sono la causa di malfunzionamenti riscontrati nel programma. Come già detto, esso non è invece utilizzato per *rilevare* tali malfunzionamenti.

Il debugging richiede una **comprensione approfondita del codice** e del funzionamento del programma e può essere un processo complesso e articolato. Tuttavia, può contribuire in modo significativo a migliorare la qualità e la stabilità del codice, oltre che a *risolvere* malfunzionamenti.

Trattandosi di ricerca delle anomalie che generano malfunzionamenti noti, l'attività è definita per un **programma** e **un insieme di dati che causano malfunzionamenti**. Essa si basa infatti sulla **riproducibilità** del malfunzionamento, verificando prima che non sia dovuto in realtà a specifiche in errate.

Si tratta di un'attività molto complicata, come fa notare Brian W. Kernighan nella sua famosa citazione:

> *"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it".*

È dunque importante scrivere codice **più semplice possibile** in modo tale da poterne fare un altrettanto semplice debugging laddove necessario.

Perché è così difficile?

L'attività di debugging è particolarmente complessa soprattutto perché non è sempre possibile individuare con precisione la **relazione anomalia-malfunzionamento**. Non è un legame banale, in quanto potrebbero esserci anomalie che prima di manifestarsi sotto forma di malfunzionamenti abbiano avuto molte evoluzioni.

Inoltre, **non esiste una relazione biunivoca** tra anomalie e malfunzionamenti: non è detto che un'anomalia causi un unico malfunzionamento, ma nemmeno che un malfunzionamento sia causato da un'unica anomalia.

Un altro problema è dovuto al fatto che la **correzione di anomalie** non garantisce affatto un software migliore o con meno errori: per correggere un'anomalia è necessario per forza di cose anche modificare il codice sorgente, ma ogni volta che viene fatto si apre la possibilità di introdurre **nuove anomalie** nel codice stesso.

Tecnica *naïve*

La tecnica di debugging maggiormente utilizzata dai programmatori consiste nell'introdurre nel modulo in esame una serie di **comandi di output** (es. *print*) che stampino su console il valore intermedio assunto dalle sue variabili. Questo permetterebbe di osservare l'evoluzione dei dati e, si spera, di comprendere la causa del malfunzionamento a partire da tale storia.

Nonostante sia **facile da applicare**, si tratta in realtà di una tecnica **molto debole**: non solo essa **richiede la modifica del codice** (e quindi una *rimozione* di tali modifiche al termine), ma è **poco flessibile** in quanto richiede una nuova compilazione per ogni stato esaminato.
Bisogna inoltre considerare che questa tecnica testa un **programma diverso** da quello originale che presenta delle *print* aggiuntive solo *apparentemente* innocue e senza effetti collaterali.

L'unico scenario (irrealistico) in cui la tecnica potrebbe essere considerata sufficiente sarebbe nel caso in cui il codice sia progettato talmente bene e il modulo così ben isolato che basterebbe scrivere un'unica *print* per risalire all'anomalia.

Tecnica *naïve* avanzata

Un miglioramento parziale alla tecnica appena descritta si può ottenere sfruttando le **funzionalità del linguaggio** oppure alcuni tool specifici per il debug, come per esempio:

- `#ifdef` e `gcc -D` per il linguaggio C;
- **librerie di logging** (con diverso livello), che permettono peraltro di rimuovere i messaggi di log in fase di produzione del codice;
- **asserzioni**, ovvero check interni al codice di specifiche proprietà: possono essere visti anche come *"oracoli" interni* al codice che permettono di segnalare facilmente stati illegali.

Ciò non toglie che la tecnica sia comunque **naïve**, in quanto si sta ancora modificando il codice in modo che fornisca informazioni aggiuntive.

Dump di memoria

Una tecnica lievemente più interessante è quella dei **dump di memoria**, che consiste nel produrre un'**immagine esatta** della **memoria** del programma dopo un passo di esecuzione: si scrive cioè su un file l'intero contenuto della memoria a livello di linguaggio macchina (*nei sistemi a 32 bit, la dimensione dei dump può arrivare fino a 4GB*).

```
Segmentation fault (core dumped)
```

Sebbene questa tecnica non richieda la modifica del codice, essa è spesso **difficile da applicare** a causa della differenza tra la rappresentazione astratta dello stato (legata alle strutture dati del linguaggio utilizzato) e la rappresentazione a livello di memoria di tale stato. Viene inoltre prodotta una **enorme mole di dati** per la maggior parte inutili.

Debugging simbolico

Il prossimo passo è invece il cosiddetto **debugging simbolico**, un'attività che utilizza **tool** specifici di debugging per semplificare la ricerca delle anomalie che causano il malfunzionamento in esame. Tali strumenti permettono di **osservare in tempo reale l'esecuzione del programma**, sollevando una cortina e rendendo possibile analizzare l'evoluzione del valore delle variabili passo per passo: questi tool non alterano il codice ma *come* esso è eseguito.

A tal proposito, i debugger simbolici forniscono informazioni sullo stato delle variabili utilizzando lo **stesso livello di astrazione** del linguaggio utilizzato per scrivere il codice: gli stati sono cioè rappresentati con **stessi simboli** per cui le locazioni di memoria sono state definite (*stesse strutture dati*), rendendo quindi utile e semplice l'attività di **ispezione dello stato**.

In aggiunta, i debugger simbolici forniscono **ulteriori strumenti** che permettono di visualizzare il comportamento del programma in maniera selettiva, come per esempio *watch* e *spy monitor*.

Per regolare il flusso del programma è poi possibile inserire **breakpoint** e **watchpoint** su certe linee di codice che ne arrestino l'esecuzione in uno specifico punto, eventualmente rendendoli dipendenti dal valore di variabili. Volendo poi riprendere l'esecuzione si può invece scegliere la granularità del successivo passo:

- **singolo**: si procede alla linea successiva;
- **dentro una funzione**: si salta al codice eseguito dalle funzioni richiamate sulla riga corrente;
- **drop/reset del frame**: vengono scartate le variabili nel frame d'esecuzione ritornando ad una situazione precedente.

Debugging per prova

Molti debugging simbolici permettono non solo di visualizzare gli stati ottenuti, ma anche di **esaminarli automaticamente** in modo da verificarne la correttezza.

In particolare, utilizzando **watch condizionali** è possibile aggiungere **asserzioni a livello di monitor**, verificando così che certe proprietà continuino a valere durante l'intera esecuzione.

Così, per esempio, è possibile chiedere al *monitor* (l'*esecutore* del programma) di controllare che gli indici di un array siano sempre interni all'intervallo di definizione.

Altre funzionalità dei debugger

Ma non finisce qui! I debugger moderni sono strumenti veramente molto interessanti, che permettono per esempio anche di:

- **modificare il contenuto di una variabile** (o zona di memoria) a runtime;
- **modificare il codice**: nonostante non sia sempre possibile, può essere comodo per esempio dopo tante iterazioni di un ciclo;
- ottenere **rappresentazioni grafiche** dei dati: strutture dinamiche come puntatori, alberi e grafi possono essere rappresentate graficamente per migliorare la comprensione dello stato.

Automazione

Visti tutti questi fantastici tool può sorgere una domanda: **l'attività di debugging può essere automatizzata?**

Andreas Zeller tratta questo argomento in maniera approfondita nel suo Debugging Book, proponendo alcune direzioni di sviluppo di ipotetici strumenti di debugging automatico.
I due concetti principali della trattazione sono i seguenti:

- **shrinking input**: dato un **input molto grande** e complesso che causa un malfunzionamento, strumenti automatici possono aiutare a ridurlo il più possibile in modo da semplificare il debugging;
- **differential debugging**: dato lo stesso input, in maniera automatica vengono esplorati gli stati del programma mutando ad ogni iterazione piccole porzioni di codice per individuare dove è più probabile che si trovi l'anomalia.

Purtroppo per il momento la prospettiva di debugger automatici è ancora lontana.
Tuttavia, esiste già qualcosa di simile, vale a dire il comando `git bisect` di Git: data una versione vecchia in cui il bug non è presente, una versione nuova in cui esso si è manifestato e un oracolo che stabilisce se il bug è presente o meno, Git esegue una **ricerca dicotomica** per trovare la versione che ha introdotto il problema. Sebbene non sia proprio la stessa cosa, si tratta sicuramente di uno strumento utile.

Reti di Petri

In questa lezione verranno mostrate le reti di Petri come esempio di **linguaggio formale**: fin dall'inizio del corso è stato possibile apprendere come l'ingegneria del software si occupi di linguaggi e comunicazione.

Partendo infatti dai processi che sfruttano un linguaggio poco formale e con poca terminologia tecnica (ad esempio le *user story*) e passando per la progettazione in cui è stato utilizzato un linguaggio più rigoroso, si arriva infine a un vero linguaggio formale utile a **raccogliere delle specifiche**.

Esiste un modello standard di rete di Petri e delle possibili estensioni di quest'ultimo: ad esempio nelle prossime lezioni saranno illustrati alcuni possibili dialetti come le **reti temporizzate**, utili a descrivere sistemi real time che necessitano di requisiti formali per ridurne le criticità.

In generale utilizzare linguaggi complessi e formali per descrivere le specifiche può essere costoso: vengono infatti utilizzati perlopiù in **contesti critici** dove i fallimenti provocano conseguenze molto gravi e in cui la **sicurezza deve essere garantita** prima di mettere in funzione il software.

Le reti di Petri sono in parte simili agli **automi a stati finiti** (FSM), ma nascono specificatamente per descrivere sistemi concorrenti. Tra gli altri aspetti, i concetti di *stato* e *transizione* per le reti di Petri differiscono rispetto a quelli già conosciuti per le FSM:

- lo **stato** nelle reti di Petri non è più un'informazione atomica osservata a livello di sistema ma è frammentata in **parti diverse** la cui composizione avviene tramite la loro visione generale;
- di conseguenza le **transizioni** non operano sullo stato globale ma si limitano a variarne una parte.

Nelle FSM esiste un unico stato attivo e gli stati disponibili sono dati dal prodotto cartesiano di tutti i possibili valori delle diverse entità. Per contro nelle reti di Petri ci sono **diversi stati attivi** in un dato momento, cosa che permette di semplificarne notevolmente la rappresentazione e l'analisi.

Definizioni

Definizione informale

Un vantaggio delle reti di Petri è che possono essere viste in maniera informale dal cliente. È infatti facile rappresentare una rete di Petri come un grafo in cui ogni nodo rappresenta o un **posto** o una **transizione** e gli archi i collegamenti presenti tra le transizioni e i posti. Il grafo è **bipartito**, ovvero un grafo in cui nodi di un tipo sono messi in relazione **solo** con nodi dell'altro tipo: in questo caso *i posti possono essere collegati soltano a transizioni e viceversa*.

Ad ogni posto è assegnato un certo numero di ***gettoni*** (o **token**) – sarà successivamente approfondito il senso dell'assegnamento di un numero infinito di gettoni a un posto.

La **disposizione dei gettoni nei posti** in un dato momento all'interno della rete di Petri ne determina il suo **stato complessivo**.


Rete Petri informale

Per far evolvere lo stato della rete, l'**assegnamento dei gettoni deve poter variare**.
La trasformazione dello stato è effettuata dallo scatto di una transizione:

- una transizione si dice ***abilitata*** (*enabled*) quando la somma dei gettoni dei posti collegati ingresso è maggiore di un certo numero;
- una transizione ***scatta*** (*fire*) quando, dopo essere stata abilitata, consuma i gettoni dei posti collegati in ingresso e ne genera altri all'interno dei posti collegati in uscita. È importante notare come i gettoni **non si spostano** da un posto a un altro conseguentemente a uno scatto, ma vengono **distrutti** nei posti in ingresso alla transizione e **generati** nei posti in uscita. Quest'ultima considerazione è importante per capire che i gettoni *non sono necessariamente sempre dello stesso numero in ingresso e in uscita*.

Tramite questo **modello operativo** è facile mostrare al cliente quando qualcosa cambia all'interno del sistema, perché risulta più intuitivo rispetto a un linguaggio logico e descrittivo.

Lo **svantaggio** è che fornisce informazioni parziali su *come* il sistema compie le azioni che dovrebbe eseguire, rischiando di essere una via di mezzo tra *specifica* e *documento di design*. Si

può comunque chiamare specifica perché *viene definito totalmente e inequivocabilmente il comportamento del sistema*.

La rete descritta è quindi una **macchina di riferimento** da utilizzare come confronto per stabilire la validità del sistema sotto esame, come se fosse un *oracolo*.

Definizione matematica

Come già detto, esistono numerosi dialetti di reti di Petri. In questo caso vediamo le **PT net** (reti con posti e transizioni) che sono le più classiche, successivamente verranno descritte delle estensioni e riduzioni di queste reti.

Una rete di Petri classicamente è una 5-tupla $[P, T;\ F, W, M_0]$ in cui:

- P è l'insieme degli **identificatori dei posti**;
- T è l'insieme degli **identificatori delle transizioni**;
- F è l'insieme delle **relazioni di flusso**;
- W è una funzione che associa un **peso ad ogni flusso**;
- M_0 è la **marcatura iniziale**, ovvero l'assegnamento iniziale dei *gettoni*.

In generale definiamo come **marcatura** una *particolare configurazione dell'assegnamento dei gettoni all'interno della rete di Petri*, sia essa *iniziale* o una sua *evoluzione*.

Da notare che P e T a livello matematico sono degli insiemi di **identificatori** che non si sovrappongono (dato che si tratta di entità differenti) a cui poi verrà assegnato un significato, quindi precedentemente sono stati associati a posti e transizioni, ma di fatto sono tutti **identificatori**.

Data la 5-tupla appena descritta esistono le seguenti proprietà:

- $P \cap T = \varnothing$;
- $P \cup T \neq \varnothing$ (una rete in cui non c'è nulla non è una rete: almeno un posto o una transizione ci devono essere);
- $F \subseteq (P \times T) \cup (T \times P)$;
- $W :: F \to \mathbb{N} \setminus 0$;
- $M_0 : P \to \mathbb{N}$.

Utilizziamo inoltre alcune *scorciatoie*:

- $\mathrm{Pre}(a) = d \in (P \cup T) \quad \langle d, a \rangle \in F$.
  Il **preset** di un nodo a è l'insieme degli elementi d appartenenti all'unione degli insiemi degli identificatori di posti e transizioni tali che esiste una relazione di flusso tra d e a

appartenente a F. \ In sostanza questo insieme rappresenta l'insieme degli **identificatori antecedenti** ad a;
- $\mathrm{Post}(a) = d \in (P \cup T) \quad \langle a, , d \rangle \in F$.
  Il **postset** di un nodo a è l'insieme degli elementi d appartenenti all'unione degli insiemi degli identificatori di posti e transizioni tali che esiste una relazione di flusso tra a e d appartenente a F. \ In sostanza questo insieme rappresenta l'insieme degli **identificatori successivi** ad a.

Tutto questo rappresenta la parte statica delle reti di Petri, ovvero quando vengono osservate in un preciso istante di tempo, senza considerare i cambiamenti che potrebbero avvenire al suo interno.

Comportamento dinamico

Una transizione $t \in T$ è **abilitata** in una particolare marcatura M se e solo se

$$\boxed{\forall p \in \mathrm{Pre}(t) \quad M(p) \geq W(\langle p, t \rangle)}.$$

In notazione, $\boxed{M\ [\ t >}$ significa che *t è abilitata in M.*

Significa che per ogni elemento collegato in ingresso a t esiste un numero di gettoni maggiore del peso dell'arco che collega p a t. Un aspetto interessante di questa definizione è che non si sta ragionando su tutti i posti della rete, ma solo su quelli collegati in ingresso a t. Di conseguenza, non è necessario conoscere l'intera rete per poter affermare che una transizione sia abilitata o meno, ma è sufficiente controllare la zona che comprende i posti appartenenti a $\mathrm{Pre}(a)$. Questa proprietà è chiamata **località dell'analisi**.

Lo **scatto** di una transizione $t \in T$ in una particolare marcatura M produce nel momento successivo una nuova marcatura M' tale per cui

$$\begin{aligned}
\forall p \in \mathrm{Pre}(t) \setminus \mathrm{Post}(t) \quad & M'(p) = M(p) - W(\langle p, t \rangle); \\
\forall p \in \mathrm{Post}(t) \setminus \mathrm{Pre}(t) \quad & M'(p) = M(p) + W(\langle t, p \rangle); \\
\forall p \in \mathrm{Post}(t) \cap \mathrm{Pre}(t) \quad & M'(p) = M(p) - W(\langle p, t \rangle) + W(\langle t, p \rangle); \\
\forall p \in P - (\mathrm{Post}(t) \cup \mathrm{Pre}(t)) \quad & M'(p) = M(p).
\end{aligned}$$

Specificando in modo descrittivo le notazioni precedenti:

- per ogni identificatore p appartenente al preset ma non al postset della transizione t, il numero di gettoni della nuova marcatura M' sarà uguale al numero di gettoni della marcatura precedente M meno il peso dell'arco che collega p a t;
- per ogni identificatore p appartenente al postset ma non al preset della transizione t, il numero di gettoni della nuova marcatura M' sarà uguale al numero di gettoni della

marcatura precedente M più il peso dell'arco che collega t a p;
- per ogni identificatore p appartenente sia al preset sia al postset della transizione t, il numero di gettoni della nuova marcatura M' sarà uguale al numero di gettoni della marcatura precedente M meno il peso dell'arco che collega p a t più il peso dell'arco che collega t a p;
- per ogni identificatore p appartenente all'insieme dei posti meno l'unione tra preset e postset di p la marcatura non cambia.

In notazione, $\boxed{M \boxed{[t>} M'}$ significa che lo scatto di t in M produce M'.

È importante notare come una transizione può scattare nel caso in cui non abbia alcun elemento nel suo preset; questo significa che la transizione in questione **non possiede prerequisiti** per scattare.

Macchine a stati finiti

È *meccanicamente* possibile trasformare una macchina a stati finiti in una rete di Petri.


Produttore

Riferendosi all'esempio del produttore, l'unico problema è l'**esistenza di collegamenti diretti tra posti**: come è stato detto in precedenza questo non è possibile in una rete di Petri. Sarà quindi necessario interporre tra i posti delle transizioni per avere una rete di Petri valida. Immaginando di mettere un solo gettone in uno dei due posti della rete appena creata, questo indicherà lo **stato attivo** presente nella macchina a stati finiti.
Seguendo questi passaggi diventa banale mappare una macchina a stati finiti su una rete di Petri: di seguito è possibile osservare l'operazione analoga eseguita sulle FSM di un consumatore e di un buffer.


Consumatore buffer

Componendo le reti di Petri di *produttore*, *consumatore* e *buffer* appena create, si crea la seguente.

Produttore consumatore buffer

In termini di automi a stati finiti, per trovare gli **stati raggiungibili** da questa composizione sarebbe stato necessario eseguire il prodotto cartesiano tra gli stati delle tre macchine a stati finiti combinate tra loro. Trattandosi invece di una rete di Petri, è sufficiente unire tutti gli identificatori uguali in un unico identificatore (ad esempio la transizione deposita della rete *produttore* e della rete *buffer*) e aggiungere a quest'ultimo tutti collegamenti posseduti dagli identificatori uniti.

ATTENZIONE: nell'esempio della rete composta le coppie di transizioni *"preleva"* e *"deposita"* dovrebbero avere due nomi differenti, ma siccome sono indicate con due rettangoli diversi è stato omesso questo particolare. In termini matematici **devono avere nomi differenti**.

Precedentemente è stato detto che, *nel caso di una rappresentazione di una FSM in termini di una rete di Petri*, si rappresenta lo stato attivo nella FSM con un gettone: di conseguenza, portando all'interno della rete composta tutti i gettoni delle varie reti si arriva ad ottenere il risultato descritto dall'immagine precedente, in cui tutte le "entità" (*consumatore*, *produttore* e *buffer*) mantengono la propria individualità (è infatti presente un gettone per ogni entità).

In questo caso si può quindi notare che il produttore è pronto a produrre, il buffer è vuoto e il consumatore è pronto a consumare una volta che il buffer avrà al suo interno qualcosa.

Come evolve questa rete?

Per rispondere a questa domanda la prima cosa da considerare è quali sono le **transizioni abilitate**: in questo caso si tratta solo della transizione *produci* sotto a p_0, in quanto è l'unica ad avere tutti gli elementi del suo preset con un numero di gettoni sufficienti a farla scattare; p_0 possiede infatti un gettone e l'arco ha peso 1 (*quando non è specificato il peso è 1*).
Una rete di Petri *non forza lo scatto di alcuna transizione*, quindi volendo si potrebbe rimanere nello stato corrente all'infinito senza far mai scattare *produci*. Se però *produci* scatta, il gettone in p_0 viene distrutto e in p_1 viene generato un nuovo gettone.


Primo scatto

Dopo questo scatto la rete di Petri si trova in una situazione in cui il produttore ha prodotto qualcosa ed è pronto a depositarlo nel buffer: a questo punto non resta che porsi nuovamente la domanda *"quali transizioni sono abilitate?"* per capire come può procedere l'evoluzione della rete.
È facile notare come la transizione *deposita* sotto b_0 sia l'unica abilitata e di conseguenza, *se dovesse scattare*, il risultato sarebbe il seguente:


Secondo scatto

Ora è possibile identificare una situazione particolare, ovvero quella in cui le transizioni pronte a scattare sono due. Sorge spontanea la domanda: *"quale delle due transizioni scatta prima?"*.
Nelle reti di Petri descritte fino ad ora non è stato presentato lo **scatto simultaneo** delle transizioni, ma nulla vieta che possa avvenire in un contesto reale. In tal caso si tratterebbe di un'istanza di **non determinismo**, ovvero *non si può dire quale transizione deve scattare*. Sono quindi 3 le situazioni che si possono verificare:

- scatta la prima transizione;

- scatta la seconda transizione;
- non scatta nessuna transizione (la *non evoluzione* è comunque un'evoluzione).

Nel caso in cui fosse stato necessario definire che una delle due transizioni scatti prima dell'altra, ci si troverebbe di fronte ad una rete **non corretta**: è infatti possibile modificare la rete in modo tale che imponga un ordine di scatto alle transizioni.

Sfruttare le reti di Petri

A questo punto è possibile chiedersi se si stiano sfruttando realmente tutte le potenzialità delle reti di Petri, siccome la rete dell'esempio precedente è stata ricavata da un automa a stati finiti. Per capire ciò è possibile osservare un secondo esempio in cui è presentata una rete alternativa alla precedente, ma con lo stesso scopo.


Rete alternativa

La differenza che salta subito all'occhio è il numero di gettoni presenti all'interno di b_0 che indicano il numero di posizioni libere nel buffer. Questo è un vantaggio perchè se il buffer dovesse cambiare la sua capienza, sfruttando questa rete è sufficiente modificare la marcatura di b_0 e il problema sarebbe risolto; la rete precedente avrebbe invece bisogno di una pesante modifica per essere adattata.
Di conseguenza si può applicare lo stesso concetto per il consumatore e per il produttore: aumentandone il numero dei gettoni (rispettivamente in p_0 e c_0) aumenterebbe il numero di entità in grado di produrre e consumare.


Rete alternativa diverse entità

È possibile affermare quindi che cambiando il **numero di gettoni** è possibile moltiplicare gli elementi del sistema di cui si vuole tracciare l'evoluzione. Si sottolinea ancora che questo risulterebbe molto oneroso in termini di dimensioni se fosse stato riadattato in una macchina a stati finiti.

Per definizione le macchine a stati finiti **non** possono rappresentare **situazioni infinite**, se si volesse quindi modificare ulteriormente l'esempio appena visto imponendo una capienza illimitata al buffer, non sarebbe possibile utilizzando una macchina a stati finiti. Sfruttando le reti di Petri invece è sufficiente eliminare l'identificatore del posto b_0: in questo modo abbiamo una situazione in cui i produttori possono depositare senza limiti all'interno del buffer, mentre i consumatori non possono prelevare più elementi di quelli presenti nel buffer. Questo vincolo è imposto dalla marcatura di b_1, infatti la transizione "preleva" può scattare al massimo n volte consecutivamente, dove n è la marcatura di b_1 — assumendo che nel mentre non avvengano depositi da parte dei produttori.

Un altra modifica applicabile all'esempio sfrutta i pesi degli archi: ponendo un peso di 3 all'arco che collega *deposita* a b_1 si può dire che il produttore crea e deposita tre prodotti, occupando tre posizioni nel buffer. Ponendo invece un peso di 2 all'arco che collega b_1 a *preleva* si specifica che è possibile prelevare dal buffer due elementi alla volta. Questo esempio, in parte forzato, è utile per chiarire il fatto che nelle reti di Petri *gli archi non sono semplici collegamenti, ma è possibile attribuirgli un significato*.

Vengono infatti informalmente chiamati *archi*, rifacendosi alla terminologia dei grafi, ma in realtà indicano una relazione più profonda che coinvolge due identificatori: in questo esempio esiste infatti una relazione per cui ogni elemento prodotto occupa tre posizioni all'interno del buffer e un'altra relazione in cui ogni consumatore può prelevare obbligatoriamente due elementi alla volta. Tramite il peso degli archi è possibile creare delle situazioni ambigue: ad esempio se la relazione che coinvolge *deposita* e p_0 avesse un peso di 2, ogni volta che viene prodotto qualcosa i produttori si moltiplicherebbero e ovviamente questa situazione indicherebbe che la rete è sbagliata, quindi è necessario fare attenzione ad evitare questo tipo di situazioni.


Archi con pesi

È da sottolineare che è possibile ridurre una rete P/T avente pesi sugli archi in una rete P/T senza pesi sugli archi: successivamente verrà illustrato come ciò è possibile.

Relazioni

Di seguito verranno elencati le tipologie di relazioni che possono coinvolgere i diversi identificatori e cosa comporta la loro presenza.

- **Sequenza**
- **Conflitto**
- **Concorrenza**

Sequenza

Una transizione t_1 è **in sequenza** con una transizione t_2 in una marcatura M se e solo se

$$\boxed{M\,[\,t_1>} \wedge \neg \boxed{M\,[\,t_2>} \wedge \boxed{M\,[\,t_1 t_2>}\,.$$

Questa formula indica che:

- t_1 è abilitata in M;
- t_2 NON è abilitata in M;
- t_2 viene abilitata dallo scatto di t_1 in M.

Si può notare una **relazione d'ordine non simmetrica** in cui *lo scatto di* t_1 è una condizione sufficiente per cui t_2 possa scattare: questo tipo di relazione permette quindi di creare un **ordine di scatto** delle transizioni. È condizione sufficiente e non necessaria perchè osservando l'esempio sottostante è facile capire che lo sacatto di t_0 non è necessario per far si che t_2 scatti: infatti anche se dovesse avvenire lo scatto di t_2, la transizione t_1 diventerebbe comunque abilitata.

Conflitto

Due transizioni (t_1, t_2) sono in:

- **conflitto strutturale** $\iff \mathrm{Pre}(t_1) \cap \mathrm{Pre}(t_2) \neq \varnothing$;
- **conflitto effettivo** in una marcatura $M \iff$:
  - $\boxed{M\,[\,t_1>}\ \wedge\ \boxed{M\,[\,t_2>}$;
  - $\exists p \in \mathrm{Pre}(t_1) \cap \mathrm{Pre}(t_2) \mid M(p) < W(\langle p, t_1\rangle) + W(\langle p, t_2\rangle)$.

Analizzando i due tipi di conflitto è possibile notare che:

- due transizioni sono in **conflitto strutturale** se l'intersezione dei due preset non è vuota e quindi hanno posti in ingresso in comune: possono quindi interferire tra loro. Il conflitto strutturale dipende solo dalla topologia dela rete, infatti non vengono citate le marcature;
- due transizioni sono in **conflitto effettivo** se sono entrambe abilitate in una marcatura M ed esiste un posto in ingresso in comune ai due preset tale per cui il numero di gettoni in quel posto è minore della somma dei pesi dei due flussi che vanno dal posto alla transizione (quindi il posto in ingresso non ha abbastanza gettoni per far scattare entrambe le transizioni). Entrano quindi in conflitto sulla disponibilità di gettoni nel preset.

Esiste una versione **rilassata** della definizione di conflitto esplicitata dalla seguente formula:

$$\boxed{M\,[\,t_1>}\ \wedge\ \boxed{M\,[\,t_2>}\ \wedge\ \neg\,\boxed{M\,[\,t_1 t_2>}.$$

Questa proposizione indica che il conflitto è presente se t_1 e t_2 sono abilitate in una marcatura M e non è possibile la sequenza $t_1\ t_2$ a partire da M. Ma cosa vuol dire che è una *versione rilassata*? Per capirlo si osservi questo l'esempio sottostante:

Secondo le definizioni di conflitto che sono state date, in questa rete di Petri è presente un conflitto sia per la prima definizione che per la seconda. È possibile però fare in modo che

rimanga in conflitto per la prima definizione data ma non più per la definizione rilassata introducendo una piccola modifica:

Aggiungendo una relazione tra t_1 a p_1 si può notare che dopo lo scatto di t_1 quest'ultima è ancora abilitata e quindi non rientra più sotto la definizione rilassata di conflitto.

Lasciando da parte la definizione rilassata, è facile osservare a questo punto che la definizione per il conflitto strutturale si basa solo sui preset, ignorando quindi qualsiasi arco in uscita, mentre la quella per il conflitto effettivo ragiona anche sugli effetti dello scatto delle transizioni. Si noti che la presenza di un conflitto strutturale **non implica** obbligatoriamente la presenza di un conflitto effettivo in quanto quest'ultimo per esistere necessita che venga soddisfatta una condizione in più. Al contrario invece un conflitto effettivo **implica** la presenza di un conflitto strutturale in qunato le condizioni di quest'ultimo sono comprese in quelle del conflitto effettivo.
Di seguito viene mostrato un esempio di conflitto *effettivo* e *strutturale*.

Concorrenza

È in qualche modo intuitivo considerare la relazione di concorrenza come la relazione opposta alla relazione di conflitto: due transizioni (t_1, t_2) sono in:

- **concorrenza strutturale** $\iff \mathrm{Pre}(t_1) \cap \mathrm{Pre}(t_2) = \varnothing$;
- **concorrenza effettiva** in una marcatura $M \iff$
  - $\boxed{M\,[\,t_1\,>}\cap\boxed{M\,[\,t_2\,>}$;
  - $\forall p \in \mathrm{Pre}(t_1) \cap \mathrm{Pre}(t_2) \quad M(p) \geq W(\langle p, t_1 \rangle) + W(\langle p, t_2 \rangle)$.

Quest'ultima formula indica che due identificatori delle transizioni sono in concorrenza effettiva se e solo se per tutti i posti che hanno in comune c'è un numero di gettoni sufficienti per farle scattare entrambe.

In questo caso non esiste alcun legame tra concorrenza strutturale ed effettiva, diversamente da quanto abbiamo visto in precedenza per le relazioni di conflitto. Se si verificano le condizioni per avere una concorrenza strutturale è **possibile** che le due transizioni non siano abilitate, oppure se si verificano le condizioni per avere concorrenza effettiva è **possibile** che t_1 e t_2 abbiano posti in comune che posseggano abbastanza gettoni per entrambe.

Questo però non esclude il fatto che sia possibile avere concorrenza strutturale ed effettiva contemporaneamente, infatti di seguito sono riportati degli esempi che confermano ciò:

Ovviamente è anche possibile che non ci sia alcun tipo di concorrenza: è sufficiente che due transizioni abbiano in comune un posto e una delle due non sia abilitata.

Insieme di raggiungibilità

L'insieme di raggiungibilità R di una rete $\mathcal{P}/\mathcal{T}$ a partire da una marcatura M è il più piccolo insieme di marcature tale che:

- $M \in R(\mathcal{P}/\mathcal{T}, M)$;
- $M' \in R(\mathcal{P}/\mathcal{T}, M) \land \exists t \in T \quad \boxed{M' \ [t>} \ M'\text{`} \implies M'' \in R(\mathcal{P}/\mathcal{T}, M).$

Questa definizione induttiva viene interpretata nel seguente modo:

- **passo base**: la marcatura M appartiene all'insieme di raggiungibilità $R(\mathcal{P}/\mathcal{T}, M) \setminus (M$ indica la marcatura iniziale mentre $\mathcal{P}/\mathcal{T}$ indica la rete posti-transizioni);
- **passo induttivo**: se M' appartiene all'insieme di raggiungibilità (quindi si dice che *è raggiungibile*) ed esiste una transizione della rete tale per cui è abilitata in M' e porta in $M'\text{`}$ — per cui con uno scatto è possibile passare dalla marcatura M' alla marcatura M'' — *allora* anche quest'ultima è **raggiungibile**.

Procedendo ricorsivamente con questa definizione è possibile ottenere tutte le marcature raggiungibili.

Limitatezza

Una proprietà importante delle reti di Petri è la **limitatezza**, che indica se le possibili evoluzioni della rete possono essere limitate o illimitate, quindi se gli stati raggiungibili sono in numero finito oppure infiniti. Volendo dare una definizione più formale è possibile dire che una rete posti-transizioni ($\mathcal{P}/\mathcal{T}$) con marcatura M si dice **limitata** se e solo se:

$$\exists k \in \mathbb{N}, \forall M'! \in R(\mathcal{P}/\mathcal{T}, M), \forall p \in P \quad M'(p) \leq k$$

cioè se esiste un numero naturale k tale per cui per ogni marcatura M' raggiungibile da M, per ogni posto p all'interno della rete il numero di gettoni in quella marcatura *raggiungibile* è minore o uguale di k — ovvero se è possibile porre un numero finito tale per cui dopo qualsiasi evoluzione non esista alcun posto che possiede un numero di gettoni maggiore di k — allora è possibile affermare che **la rete è limitata**.
Se ciò non si verifica esiste almeno un posto in cui è possibile aumentare tendenzialmente all'infinito il numero di gettoni, tramite una certa evoluzione della rete. È importante sottolineare che la limitatezza di una rete può dipendere dalla sua **marcatura iniziale**.

Da reti di Petri a automi

Precedentemente è stato mostrato come a partire da un automa a stati finiti sia possibile ricavare una rete di Petri, ma è possibile fare **il contrario**? Se la rete è limitata allora l'insieme di raggiungibilità è finito, di conseguenza è possibile definire un corrispondente automa a stati finiti che prende ogni marcatura raggiungibile come un proprio *stato* e ne traccia le transizioni di stato dell'automa conseguenti alla transizione scattata nella rete di Petri. Due considerazioni:

- gli **stati** sono le possibili marcature dell'insieme di raggiungibilità;
- le **transizioni** sono gli eventi che permettono il passaggio da una configurazione alla successiva.

Riuscire a passare dalle reti di Petri agli automi ci permette di modellare un problema in modo più sintetico, ma allo stesso tempo rimane possibile utilizzare i **tool di analisi** che sfruttano proprietà già consolidate per gli automi. L'unico problema è che questo approccio vale solo per **reti limitate**.

Vitalità di una transizione

Una transizione t in una marcatura M si può dire *viva* con un certo **grado**:

- **grado 0** (o **morta**): non è abilitata in nessuna marcatura appartenente all'insieme di raggiungibilità, quindi qualunque evoluzione avvenga nella rete, la transizione non portà mai scattare (non è sempre un aspetto negativo);
- **grado 1**: esiste almeno una marcatura raggiungibile a partire da M in cui la transizione è abilitata;
- **grado 2**: per ogni numero n naturale escluso lo zero esiste almeno una sequenza di scatti ammissibile a partire da M in cui la transizione scatta n volte, ovvero è possibile far scattare la transizione un numero n grande a piacere di volte;
- **grado 3**: esiste una sequenza di scatti ammissibile a partire da M per cui la transizione scatta *infinite* volte;
- **grado 4** in *qualunque marcatura raggiungibile* esiste una sequenza ammissibile in cui è possibile far scattare la transizione almeno una volta, di conseguenza può scattare infinite volte in qualunque situazione ci si trovi (ovvero in qualunque marcatura).
  In questo caso si dice che la transizione è **viva** *in maniera assoluta*.

Si noti come il concetto di n *grande a piacere* presente nel grado 2 sia differente dal concetto di *infinite volte* nel grado.

Gli esempi seguenti rappresentano delle situazioni verosimili riguardanti la vitalità delle transizioni:

- **grado 0**: qualunque cosa accada *la centrale nucleare non può esplodere*;
- **grado 1**: in un certo momento se si assume il controllo di tutto ciò che avverrà *è possibile portare la centrale nucleare allo spegnimento*;
- **grado 2**: Duccio, ingegnere della centrale nucleare che si trova in coffee break, è in grado di interagire con la macchinetta del caffé appena accesa in modo da avere un numero di caffé *grande a piacere*, almeno finchè qualcuno non inserisce una moneta nella macchinetta;
- **grado 3**: Biascica, guardia giurata della centrale, è in grado di fare alzare la sbarra per il parcheggio *un numero infinito di volte*;
- **grado 4**: se succede qualcosa fuori dal controllo all'interno della centrale si può comunque riuscire ad eseguire lo spegnimento (René urla *"chiudi tutto, Duccio!"*).

Una rete viene chiamata **viva** quando tutte le sue transizioni sono vive.

Esempio


Esempio vitalità tranisizioni

- Da questo esempio pratico è possibile notare come la transizione t_0 è di **grado 0** in quanto non potrà mai scattare, perchè è impossibile che abbia i gettoni necessari nel preset per scattare (al massimo o in p_0 o in p_1).
- La transizione t_1 è di **grado 1** perchè esiste almeno una marcatura raggiungibile per cui essa scatti, infatti la marcatura corrente è quella che ne *permette* lo scatto (ricordando ancora che se una transizione è abilitata allo scatto non significa che debba scattare).
- Osservando la transizione t_3 è possibile notare che essa scatti infinite volte (e non n grande a piacere, quindi non si tratta di una transizione di grado 2), ma nel caso avvenga lo scatto di t_1 la transizione t_3 non potrà mai più essere abilitata (quindi esiste una marcatura in cui non sarà possibile il suo scatto) garantendo che non si tratta di una transizione di grado 4, ma bensì di **grado 3**.
- Il caso più particolare è quello della transizione t_2: è noto che t_3 può scattare infinite volte e quindi in p_2 possono esserci infiniti gettoni; inoltre, conseguentemente allo scatto di t_1 il posto p_1 conterrà un gettone, ma comunque la transizione t_2 non può scattare infinite volte. Questo perchè è vero che all'infinito posso generare gettoni in p_2, ma dal momento che scatta t_1 si perde questa possibilità, permettendo a t_2 di scattare tante volte quanti sono i gettoni in p_2. La transazione è quindi di **grado 2**.
- Infine t_4 è una transizione viva (di **grado 4**), perchè qualunque sia la marcatura raggiungibile dalla marcatura corrente è possibile prendere il controllo e sicuramente esiste una sequenza di scatti tale per cui t_4 diventi abilitata.

Capacità dei posti

Inizialmente è stato detto che esistono diversi dialetti riguardanti le reti di Petri. Una possibile estensione consiste infatti nel fissare una **capacità massima** rispetto al numero di gettoni ammissibili in un posto. Un esempio potrebbe essere quello in cui in un sistema possono essere presenti k lettori contemporaneamente e non più di k. Avendo la possibilità di definire una capacità dei posti, è facile intuire che diventa possibile *forzare la limitatezza della rete*.

Tale estensione aumenta la potenza espressiva oppure è semplicemente una scorciatoia? Tramite l'esempio sottostate si può notare che questa estensione non è altro che una tecnica per facilitare la scrittura della rete.


Simulazione capacità posti

Nella rete con capacità dei posti limitata per far sì che ad esempio la transizione t_0 scatti, è necessario sia che i posti nel suo preset abbiano gettoni sufficienti sia che dopo il suo scatto il posto p_0 non superi il limite assegnatogli. Volendo scrivere la stessa rete utilizzando il metodo classico visto fino ad ora basta aggiungere un **posto complementare** che quindi rende le reti **equipollenti**, ossia aventi lo stesso valore espressivo.

Fino a che nel posto complementare esistono dei gettoni, la transizione t_0 può infatti scattare; dal momento però che tutti i gettoni di $p_0(\mathrm{compl})$ vengono bruciati, t_0 non sarà più abilitata e nel posto p_0 ci sarà il numero massimo di gettoni possibili. Notare come la somma dei gettoni del posto considerato sia esattamente la capacità massima scelta in precedenza.

Questa proprietà vale solo per le reti **pure**, ovvero *le reti che **per ogni transizione** hanno preset e postset disgiunti*.

Posto complementare

Un posto complementare è un posto avente in uscita verso ognuna delle transizioni del posto considerato un **arco di ugual peso** ma di **direzione opposta**.

Matematicamente è possibile scivere questa definizione nel seguente modo:
un posto pc è *complementare* di p se e solo se

$$\begin{aligned}\forall t \in T \left[\exists \langle p, t\rangle \in F \iff \exists \langle t, pc\rangle \in F \quad W(\langle p, , t\rangle) = W(\langle t, pc\rangle)\right] \\ \wedge \forall t \in T \left[\exists \langle t, p\rangle \in F \iff \exists \langle pc, t\rangle \in F \quad W(\langle pc, t\rangle) = W(\langle t, p\rangle)\right].\end{aligned}$$

Per ogni transizione appartenente a T in uscita da p, quindi tale per cui esiste una relazione di flusso dal posto p alla transizione t deve esistere un flusso che va dalla transizione t al posto complementare pc avente lo stesso peso.
Inoltre, per le transizioni in ingresso al posto p (quindi per ogni transizione t appartenente a T in ingresso a p) tali per cui esista un flusso da t al posto p, deve esistere un flusso che va dal posto complementare pc a t di direzione opposta e avente lo stesso peso.

Questa formula garantisce che la **somma** del numero di gettoni tra il posto e il suo complementare sia costante, permettendo quindi di formulare la **condizione di abilitazione** (lavorando sul preset della transizione) in modo da dipendere anche dal numero di gettoni presenti nel posto in arrivo.

Abilitazione con capacità

Come è possibile definire la condizione di abilitazione nel caso di **reti con capacità sui posti**?

La definizione di *abilitazione* per reti con capacità sui posti è la seguente:
$t \in T$ è **abilitata** in M se solo se:

$$\begin{aligned} \forall p \in \mathrm{Pre}(t) \quad & M(p) \geq W(\langle p,t \rangle) \\ \forall p \in \mathrm{Post}(t) \setminus \mathrm{Pre}(t) \quad & M(p) + W(\langle t,p \rangle) \leq C(p) \\ \forall p \in \mathrm{Post}(t) \cap \mathrm{Pre}(t) \quad & M(p) - W(\langle p,t \rangle) + W(\langle t,p \rangle) \leq C(p). \end{aligned}$$

Considerando l'immagine seguente, infatti, possiamo notare come la rete di sinistra abbia ancora una transazione abilitata, mentre quella di destra no. Nella seconda rete è come se **lo scatto venisse spezzato in due fasi**: la prima in cui vengono generati i gettoni nel posto (in questo caso p_3), la seconda invece in cui vengono tolti tanti gettoni quanto è il peso dell flusso da p_3 a t_1.
Nella prima rete invece questo non accade, è come se si verificasse tutto nello stesso istante.


Esempio abilitazione reti con capacità

A questo punto, ci si potrebbe chiedere se fosse possibile generare la situazione equivalente nel caso di una rete $\mathcal{P}/\mathcal{T}$ classica: la risposta è **no**, ad eccezione del caso in cui si usano delle reti con posti complementari. Utilizzando i **posti complementari** è infatti possibile rappresentare **solo le *reti pure equivalenti***, ma *non tutte le reti in generale*: finché non sono presenti archi in entrata e uscita allo stesso posto dalla stessa transizione non sorge alcun tipo di problema.

Come è possibile superare questa limitazione? Si possono pensare due approcci:

- si trova un altro approccio diverso dai posti complementari;

- si cerca di dimostrare che una rete non pura ha sempre una equivalente rete pura; quindi, si procede a rimuovere la capacità utilizzando i posti complementari.

Entrambe le soluzioni non sono così immediate.

Archi inibitori

Esiste un'altra estenzione delle reti di petri in cui si utilizzano gli **archi inibitori**, ovvero degli archi che indicano la situazione in cui una transizione ha bisogno che **non siano presenti gettoni nel posto** in modo che possa essere abilitata. Un *arco inibitore di peso* n indica che la transazione collegata è abilitata se nel posto collegato sono presenti **meno di** n gettoni.

In caso di **rete limitata** la **potenza espressiva** di una rete che sfrutta gli archi inibitori **non cambia**, perché esistendo un limite massimo k di gettoni all'interno della rete sarà sufficiente creare un posto complementare contente un numero di gettoni tali per cui la somma tra quest'ultimi e i gettoni presenti nel posto considerato sia minore di k.
A questo punto è necessario che siano presenti due archi (uno in ingresso e uno in uscita) di peso k, in modo da permettere lo scatto della transizione solo nel caso in cui tutti i gettoni siano all'interno del posto complementare.
In realtà **non è necessario** che tutta la rete sia limitata, è sufficiente che il singolo posto lo sia: è necessario garantire che qualunque sia lo *stato generale* della rete, in quel preciso posto non ci siano più di k gettoni.

Nel caso di una rete **non limitata** invece non è sempre possibile avere una traduzione equivalente della rete di Petri: la **potenza espressiva** delle reti con gli archi inibitori **aumenta**.

Il problema degli archi inibitori è che rendono **inutilizzabili** alcune **tecniche di analisi** che verranno affrontate successivamente.

Eliminazione pesi sugli archi

In precedenza è stato accennato che per ogni rete avente dei pesi sugli archi è possibile crearne una **equivalente** senza pesi sugli archi (ovvero avente tutti gli archi di peso 1).
Per fare ciò è necessario considerare due casi distinti, ovvero quello con peso sugli archi in **ingresso ad una transizione** e quello con peso sugli archi **in uscita** ad una traniszione.

Pesi su archi in ingresso

Per poter effettuare questa modifica è necessario avere lo **scatto di una nuova transizione** (in quanto ovviamente non è possibile collegare due archi dallo stesso posto alla stessa

transizione), ma non basta. Dopo lo scatto di t_0 è infatti possibile che t_0^{BIS} non scatti e la rete evolva senza che in p_1 ci sia il giusto numero di gettoni (problema di concorrenza).

Per risolvere questo problema si sfrutta una sorta di **lock**, ovvero un posto collegato bidirezionalmente con tutte le transizioni della rete tranne per t_0, a cui è collegato solo in ingresso, e per t_0^{BIS}, a cui è collegato solo in uscita. In questo modo è come se lo scatto di t_0 sia scomposto logicamente in due parti: quando t_0 scatta viene attivato il lock in modo tale che nessun'altra transizione sia abilitata e, successivamente, lo scatto di t_0^{BIS} lo rilascia. Questo ovviamente non obbliga t_0^{BIS} a scattare immediatamente, però è certo che la rete non potrà evolvere in alcun altro modo e quindi non si creeranno marcature non esistenti nella rete originale. Questa soluzione non è molto elegante perchè esiste un posto avente in ingresso un arco per ogni transizione della rete.


Eliminazione pesi archi ingresso

Pesi su archi in uscita

In questo caso il peso da rimuovere è su un arco che esce da un posto ed entra in una transizione, quindi è necessario che vengano **distrutti due gettoni** dallo stesso scatto.
L'approccio da utilizzare è simile: è infatti presente un **posto globale** che fa da **lock** in modo da risolvere il problema di concorrenza tra t_8 e t_1. In questo caso però è presente un ulteriore problema, ovvero al momento dello scatto di t_8 il gettone in p_0 viene consumato, di conseguenza t_1 non può scattare. Inoltre il resto della rete rimane bloccata, in quanto all'interno del posto globale non è più presente il gettone che è stato consumato sempre dallo scatto di t_8.
Questo **deadlock** può essere risolto aggiungendo un controllo sul posto p_0, in modo tale che possa scattare solo quando possiede due o più gettoni: in questo modo non può verificarsi la situazione in cui t_8 scatti senza un successivo scatto di t_1.

Il meccanismo della rete inizia ad essere **molto complesso**; nell'esempio viene mostrato solo il caso in cui devono essere consumati due gettoni. In altri casi con più gettoni, o con situazioni differenti, la rete aumenterebbe ulteriormente di complessità. Risulta quindi più facile pensare la rete in modo differente.

La tenica descritta sopra non è infatti l'unica esistente per modellare il sistema: nonostante possa essere adatta per questo particolare esempio, è comunque possibile trovarne un'altra per modellare una rete senza fruttare i pesi o una loro **traduzione meccanica**.


Eliminazione pesi archi uscita

Reti $\mathcal{C}/\mathcal{E}$

Le reti $\mathcal{C}/\mathcal{E}$ (condizioni eventi) sono delle particolari reti **più semplici**, in cui tutti gli archi hanno **peso uno** e tutti i posti hanno capacità massima uno. A prima vista, questo tipo di rete può risultare poco modellabile, ma è in realtà più semplice ed immediata da capire: infatti *i posti rappresentano delle condizioni* che possono essere **vere** o **false** ed in base ad esse è possibile il verificarsi di certi eventi, rappresentati dalle transizioni. Ogni rete $\mathcal{P}/\mathcal{T}$ **limitata** è **traducibile** in un'equivalente rete $\mathcal{C}/\mathcal{E}$.
Per le reti illimitate non è invece possibile trovare una traduzione, siccome non si possono rappresentare infiniti stati con un tipo di rete che per definizione è limitata.

Conservatività

La conservatività è una proprietà di una rete rispetto ad una funzione H che assegna un peso ad ogni posto della rete, e ognuno di questi pesi è positivo.
Esiste quindi una **funzione di pesi** $H : P \to \mathbb{N} \setminus 0$ tale per cui una rete $\mathcal{P}/\mathcal{T}$ con una marcatura M si dice **conservativa rispetto ad** H se e solo se:

$$\forall M' \in R(\mathcal{P}/\mathcal{T}, M) \quad \sum_{p \in P} H(p)M'(p) = \sum_{p \in P} H(p)M(p).$$

Ovvero, per ogni marcatura M' raggiungibile dalla marcatura iniziale data una certa marcatura e una funzione H, si dice che la rete è conservativa se la sommatoria dei gettoni di ogni posto (quest'ultimi pesati attraverso la funzione H) è ***costante*** *per qualunque marcatura raggiungibile*.

Conservatività $\Rightarrow$ limitatezza

Esiste inoltre un **legame** tra **conservatività** e **limitatezza**, ovvero *una rete che garantisce la conservatività è limitata, ma non è detto il viceversa* (quindi *la limitatezza è una condizione necesaria ma non sufficiente per la conservatività*).

Dimostrazione

Assumendo che $\sum_{p \in P} H(p)M(p) = k$, allora

$$\forall M' \in R(\mathcal{P}/\mathcal{T}, M) \quad \sum_{p \in P} H(p)M'(p) = k.$$

Sapendo inoltre che $\forall p \in P \quad H(p) > 0$, allora ogni elemento della sommatoria ha un **contributo nullo o positivo**. Infatti, se non ci sono gettoni all'interno del posto il contributo della sommatoria sarà un numero positivo ($H(p)$) moltiplicato per 0, quindi nullo.

Quindi, se esiste almeno una marcatura di p cui numero di gettoni è diverso da 0, il suo contributo è positivo ma limitato da k. Questo vale per ogni posto all'interno della rete, riconducendosi di conseguenza alla definizione di limitatezza. ■

Rete strettamente conservativa

La *conservatività stretta* è un particolare caso di conservatività definibile come segue: una rete $\mathcal{P}/\mathcal{T}$ conservativa rispetto alla funzione H che assegna pesi tutti uguali a 1 si dice *strettamente conservativa*.

$$\forall M' \in R(\mathcal{P}/\mathcal{T}, M) \quad \sum_{p \in P} M'(p) = \sum_{p \in P} M(p).$$

La sommatoria del numero di gettoni per ogni posto in una *qualsiasi marcatura* è **costante**, ovvero è uguale alla sommatoria dei gettoni della marcatura iniziale per ogni posto. In altre parole, dopo lo scatto di una transazione viene forzata la **distruzione del gettone in ingresso** e la **generazione di un'altro in uscita**.

Matematicamente questo concetto si può esprimere anche tramite questa espressione:

$$\forall t \in T \quad \sum_{p \in \mathrm{Pre}(t)} W(\langle p, t \rangle) = \sum_{p \in \mathrm{Post}(t)} W(\langle t, p \rangle)$$

Per ogni transizione t la somma dei pesi degli archi che collegano ogni elemento del preset di t alla transizione t deve essere uguale alla sommatoria dei pesi degli archi che collegano la transizione t con ogni posto nel postset di t.

Le due espressioni sopra esprimono lo stesso concetto, ma la prima si riferisce alle **marcature** (stati) analizzando dinamicamente calcolando gli stati raggiungibili mentre l'altra all'**aspetto topologico** della rete (ovvero i pesi degli archi).

Si precisa che per quanto riguarda la seconda formula, le espressioni da considerare sono quelle **non morte** (di grado ≥ 1). La seconda è anche più generale rispetto alla prima, ma potrebbe erroneamente considerare **non** strettamente conservative reti che **invece lo sono**.

Stato base e rete revertibile

Una marcatura M' si dice **stato base** di una rete se per ogni marcatura M in $R(M_0)$, M' è raggiungibile da M, ovvero *qualunque* sia lo stato attuale della rete è **sempre possibile** raggiungere la marcatura M'.

Quando la marcatura iniziale M_0 è lo stato base della rete per ogni marcatura M in $R(M_0)$ allora la rete si dice **reversibile**, ovvero lo stato iniziale è uno stato base.

Analisi delle reti di Petri

Le reti di Petri sono state introdotte per poter **analizzare un sistema** ancora prima di avere il codice. Alcune domande da porsi sono:

- può essere raggiunta una determinata marcatura?
- è possibile una certa sequenza di scatti?
- esiste uno stato di deadlock all'interno della rete?
- la rete (o una certa transizione) è viva? E di che grado?

Per rispondere a queste domande esistono diverse tecniche, suddivise in:

- **tecniche dinamiche**:
  - albero (grafo) delle marcature raggiungibili (chiamato anche **grafo di raggiungibilità**);
  - albero (grafo) di copertura delle marcatura raggiungibili (chiamato anche **grafo di copertura**);
- **tecniche statiche**:
  - identificazione delle P**-invarianti** (caratteristiche invarianti riguardanti i posti);
  - identificazione delle T**-invarianti** (caratteristiche invarianti riguardanti alle transizioni).

Le tecniche dinamiche ragionano sugli **stati raggiungibili** durante l'esecuzione della rete di Petri (o di un programma), mentre le statiche sulla **topologia della rete**.

- **Albero di raggiungibilità**
- **Albero di copertura**
- **Rappresentazione matriciale**
- **Analisi statica**
- **Controllori con specifica a stati proibiti**
- **Reti con priorità**

Albero di raggiungibilità

Per generare l'*albero di raggiungibilità* di una rete di Petri si può applicare il seguente **algoritmo**.

1. **crea la radice** dell'albero corrispondente alla marcatura iniziale M_0 ed etichettala come *nuova*;
2. ***finché* esistono nodi etichettati come *"nuovi"*** esegui:
    1. **seleziona** una marcatura M etichettata come *"nuova"*;
       prendila in considerazione e **rimuovi l'etichetta** *"nuova"*.
    2. ***se*** la **marcatura** M è **identica** ad una marcatura di un altro nodo allora:
        - **etichetta** M come **"duplicata"**;
        - ***continua*** passando alla prossima iterazione.
    3. ***se*** nella **marcatura** M non è abilitata **nessuna transizione** allora:
        - **etichetta** M come **"finale"**;
        - *situazione di deadlock*.

       altrimenti esegui:
        - ***finché* esistono transizioni abilitate** in M esegui:
            - ***per ogni transizione*** t **abilitata** in M esegui:
                1. **crea** la **marcatura** M' prodotta dallo **scatto** di t;
                2. **crea** un nuovo **nodo** corrispondente alla marcatura M';
                3. **aggiungi** un **arco** nell'albero al nodo corrispondente di M al nodo di M';
                4. **etichetta** la **marcatura** M' come **"nuova"**.

Esempio

Di seguito è mostrata una consegna di un esercizio riguardo gli alberi di raggiungibilità.

> Modellare tramite una rete di Petri l'accesso ad una risorsa condivisa tra quattro lettori e due scrittori, ricordandosi che i lettori possono accedere contemporaneamente, mentre gli scrittori necessitano di un accesso esclusivo.

Come primo approccio, si possono creare due reti, una per i lettori e una per gli scrittori. È possibile successivamente procedere modellando la *Risorsa* condivisa collegando le diverse parti create.


Esempio albero di raggiungibilità

Essendo presente un solo gettone nel posto *Risorsa*, i **lettori** non sono in grado di accedervi contemporaneamente. Per risolvere questo problema, si può aumentare il numero di gettoni all'interno di *Risorsa* a 4. Per evitare che gli scrittori possano accedere alla *Risorsa* mentre viene letta, è possibile aggiungere un peso pari a 4 sugli archi da "*Risorsa*" a *"S_inizia"* e da *"S_finisce"* a *"Risorsa"*.

Così facendo, per accedere alla *Risorsa* uno **scrittore** dovrà attendere che tutti i token saranno depositati in essa, garantendo che nessun'altro sta utilizzando la risorsa.

Il **risultato finale** è il seguente.


Esempio albero di raggiungibilità completo

Costruzione dell'albero di raggiungibilità

Una volta creata la rete finale, è possibile **generare** l'albero di raggiungibilità seguendo l'**algoritmo precedente**.

Il primo passo è creare il **nodo radice** corrispondente alla marcatura iniziale e marcarlo come *nuovo*: 40420.

Successivamente, occorre procedere *per ogni nodo marcato come nuovo*. In questo caso l'unico nodo marcato come *nuovo* è 40420. Dopo aver rimosso l'etichetta *nuovo* si verifica che, partendo dalla radice dell'albero, non siano già presenti altri nodi uguali. Essendo 40420 esso stesso la radice (e unico nodo dell'albero), si procede.

A questo punto, per ogni transizione abilitata nella marcatura presa in considerazione (40420) la si fa **scattare** generando le altre marcature marcate come *nuovo* (40011 e 31320) che quindi si **collegano** con un arco alla marcatura originale (40420).

La situazione attuale è la seguente.


Esempio albero primo giro algoritmo

Si procede quindi con l'algoritmo ripetendo i passi fino ad arrivare in una situazione in cui **non esistono più nodi nuovi**, marcando nel mentre come duplicati tutti i nodi che si re-incontrano nonostante siano già presenti almeno una volta nell'albero.

La **situazione finale** sarà la seguente.


Esempio albero finale

L'albero di raggiungibilità sopra in figura è a ora **completo** e rappresenta tutti gli *stati* raggiungibili.

Grazie a questo albero, se si volesse verificare che gli scrittori sono in **mutua esclusione** con i lettori, basterà controllare se esiste una marcatura in cui il secondo e il quinto numero (rispettivamente *"LettoriAttivi"* e *"ScrittoriAttivi"*) sono entrambi contemporaneamente maggiori di zero. Si può verificare in modo esaustivo (model checking) guardando tutti i nodi dell'albero. Inoltre si può verificare se gli **scrittori** si **escludono a vicenda**, controllando se in ogni marcatura l'ultimo numero (*"ScrittoriAttivi"*) è maggiore di uno. Si infine verificare l'assenza di **deadlock**, data dalla presenza o meno di nodi terminali.

Collassando i nodi aventi la stessa marcatura, si può verificare dall'albero di raggiungibilità se la rete è **viva**.


Esempio di grafo di raggiungibilità

La rete è anche **reversibile** in quanto ogni stato è uno *stato base* ed è quindi possibile raggiungere da ogni stato tutti gli altri stati.
Avendo questo grafo è quindi facile capire che la rete è **viva**, in quanto sono rappresentate tutte le transizioni all'interno del grafo, e siccome il sistema è reversibile è sempre possibile riportarsi ad una situazione in cui si può far scattare una transizione.

Limiti

- Per poter creare un albero di raggiungibilità è necessario enumerare tutte le possibili marcature raggiungibili, di conseguenza **la rete deve essere obbligatoriamente limitata**: non sarebbe altrimenti possibile elencare tutti i nodi.
- la **crescita** (esponenziale) del numero degli stati globali può risultare velocemente **ingestibile** per una rete limitata.

Inoltre:

- Questa tecnica di analisi non è in grado di rilevare se una rete è limitata o meno;
- Nel caso in cui si sappia già che la rete è limitata:
    - l'albero di raggiungibilità non perde informazioni ed è la esplicitazione degli stati della rete (quindi ne è di fatto la FSM corrispondente).

Albero di copertura

A questo punto risulterà normale chiedersi se sia possibile creare una struttura dati (albero e grafo) anche per le **reti illimitate**, cui nodi rappresenteranno *gruppi di stati* potenzialmente

infiniti.
È bene introdurre il concetto di **copertura** prima di procedere.

*Una marcatura M **copre** una marcatura M' se e solo se:*

$$\forall p \in P \quad M(p) \geq M'(p).$$

Ovvero *se per ogni posto in P, la marcatura $M(p)$ è maggiore o uguale a $M'(p)$.*

Al contrario, M si dice **copribile** da M' se e solo se:

$$\exists M' \in R(M') \mid M'' \; copre \; M.$$

Grazie al concetto di *copertura* è possibile ridefinire quello di **transizione morte**:
una transizione t si dice **morta** se e solo se data la sua *marcatura minima* M (ovvero il minor numero di gettoni necessario in ogni posto nel suo preset per abilitarla) questa **non è copribile** a partire dalla marcatura corrente. In caso contrario la transizione t è almeno 1-viva.

Si conclude quindi che se una marcatura ne copre un'altra, tutte le azioni possibili nella prima **sono possibili** anche nella seconda. È quindi possibile modificare l'*albero di raggiungibilità* in modo tale che, quando viene creato un nodo è necessario verificare se tra i suoi predecessori ne esiste uno che lo copre, allora a questo punto nei posti dove c'è copertura propria (ovvero $M(p) \geq M'(p)$) si mette ω.
Il **simbolo** ω rappresenta un numero **grande a piacere** (e non *qualsiasi*), che può aumentare all'infinito: questo aspetto è da tenere in considerazione quando bisogna cercare quali transizioni sono abilitate da esso. Questo tipo di notazione (ω) viene introdotta per limitare l'aumento spropositato di nodi nel diagramma, comprimendo marcature uguali se non per ω.

Se una marcatura M copre una precedente M', infatti, è possibile **ripetere gli scatti** delle transizioni in M' per arrivare ad M; se al termine di questa operazione sono presenti più gettoni in un posto, allora è possibile crearne un numero grande a piacere.

È importante notare come le transizioni che erano **abilitate** in una certa marcatura M' lo saranno anche in una marcatura diversa che copre M', a meno che non ci siano **archi inibitori**.

È ora possibile definire l'**algoritmo** per la creazione di un albero di copertura, comunque simile in molti punti al precedente:

1. **crea la radice** dell'albero corrispondente alla marcatura iniziale M_0 ed etichettala come *nuova*;
2. ***finché*** **esistono nodi etichettati come *"nuovi"*** esegui:
    1. **seleziona** una marcatura M etichettata come *"nuova"*;
    prendila in considerazione e **rimuovi l'etichetta** *"nuova"*.
    2. ***se*** la **marcatura** M è **identica** ad una marcatura di un altro nodo allora:

        - **etichetta** M come **“duplicata”**;
        - ***continua*** passando alla prossima iterazione.
    3. ***se*** nella **marcatura** M non è abilitata **nessuna transizione** allora:
        - **etichetta** M come **“finale”**;

    altrimenti esegui:
    - ***finché*** **esistono transizioni abilitate** in M esegui:
        - ***per ogni transizione*** t **abilitata** in M esegui:
            1. **crea** la **marcatura** M' prodotta dallo **scatto** di t;
            2. ***se*** sul cammino dalla **radice** (M_0) alla **marcatura** M **esiste** una **marcatura** M'‘ **coperta** da M' allora
                - **aggiungi** ω in tutte le posizioni corrispondenti a coperture proprie;
            3. **crea** un nuovo **nodo** corrispondente alla marcatura M';
            4. **aggiungi** un **arco** nell’albero al nodo corrispondente di M al nodo di M';
            5. **etichetta** la **marcatura** M' come **“nuova”**.

Dall’albero generato da questo algoritmo è possibile arrivare al **grafo di copertura**.

Inoltre, ripetendo l’algoritmo precedente su una rete limitata viene generato un grafo di copertura senza ω e quindi equivalente a un albero di raggiungibilità.

L’algoritmo **termina** in ogni caso: è sufficiente **osservare** l’albero risultante per stabilire se la rete considerata è limitata oppure no.

Esempio

Partendo dalla rete di Petri sottostante ed applicando l’**algoritmo** appena descritto è possibile arrivare ad un **albero di copertura**.

Come visto nell’esempio della creazione di un albero di raggiungibilità, il primo passo da fare è creare il nodo radice corrispondente alla marcatura iniziale ($\boxed{100}$) e marcarlo come nuovo. Successivamente, è necessario considerare l’unico nodo esistente ($\boxed{100}$) e iterare tra le sue transizioni. In questo caso, è abilitata la transizione t_1 che porta a una marcatura $M' = \boxed{101}$.

A questo punto si può notare come la radice sia una **marcatura coperta da** M', in quanto:

- $M \mid 1 \geq 1 \mid M'$;
- $M \mid 0 \geq 0 \mid M'$;
- $M \mid 1 > 0 \mid M'$.

Nel nodo corrispondente alla marcatura M' è quindi possibile sostituire l'unica **copertura propria** (quella con il $>$ e non il $\geq$) con il simbolo ω e marcare il nodo. Questa è l'unica parte dell'algoritmo differente da quello che genera l'albero di raggiungibilità: il resto dell'esempio è quindi completato dall'immagine sottostante.

Tramite lo stesso procedimento attuato per gli alberi di raggiungibilità, è possibile trasformare il precedente albero in un **grafo di copertura**.

Considerazioni

- se ω non compare mai nell'albero di copertura la rete è **limitata**;
- una rete di Petri è **binaria** se nell'albero di copertura compaiono solo 0 e 1;
- una transizione è **morta** (0-viva) se non compare mai come etichetta di un arco dell'albero di copertura;
- condizione necessaria affinché una marcatura M sia **raggiungibile** è l'esistenza di un nodo etichettato con una marcatura che copre M (non sufficiente: *le marcature coperte non sono necessariamente raggiungibili*);
- non è possibile stabilire se una rete è **viva**.

Esempio particolare

È doveroso un ulteriore esempio particolare nel caso di reti **non vive**.
Data una *rete non viva* (come nella figura sotto) dall'albero di copertura **non è possibile** evincere se la rete è effettivamente viva o no: infatti se il nodo $\boxed{01\omega}$ è duplicato, quindi non verrà più espanso. A questo punto non è possibile aggiungere all'interno dell'albero il nodo $\boxed{010}$, in cui la rete raggiunge un deadlock. Questo però significa che questo albero di copertura è uguale a quello della stessa rete senza arco che collega p_3 a t_4, che in quel caso è una rete viva. Detto ciò si può affermare che tramite l'albero di copertura non è possibile dire se una rete è viva oppure no.

Da albero di copertura a rete

Passare dall'albero di copertura alla rete di Petri è un'operazione che rispetto all'inverso crea più **incertezza**. L'albero di copertura permette infatti di rappresentare reti potenzialmente

illimitate, è quindi normale avere come risultato reti di cui non si conosce la struttura: molte reti potrebbero essere associate **allo stesso albero**.

Nel seguente esempio si può notare come la rete ricavata presenta degli *archi tratteggiati*: **potrebbero essere presenti**, oppure no. Inoltre, sono assenti anche i pesi negli archi. Tale mancanza di informazioni è dovuta in gran parte dalla presenza di ω: un nodo con all'interno ω rappresenta **diverse marcature**.

È importante notare come le marcature **sicuramente raggiungibili** siano quelle i cui nodi nell'albero di copertura non contengono ω: delle altre non si può essere certi.

Rappresentazione matriciale

Prima di procedere con la spiegazione delle tecniche di analisi statiche, è necessario introdurre una nuovo modo per rappresentare le reti di Petri: la **rappresentazione matriciale**. Essendo tutte rappresentazioni *formali*, *non ambigue* e *complete*, data una qualsiasi rete rappresentata graficamente o in forma logica, è possibile **trasformarla automaticamente** in una rete in forma matriciale, e viceversa.

Il vantaggio principale della rappresentazione matriciale è la **maggiore semplicità** ed **efficienza** nel **trattamento matematico** delle reti.

Le matrici che verranno utilizzate sono diverse, tra cui:

- I: rappresenta gli **archi in ingresso**, ovvero le coppie di flusso che da un posto vanno nelle transizioni;
- O: rappresenta gli **archi in uscita**, ovvero le coppie di flusso che da una transizione vanno nei posti;
- vettore m: rappresenta la **marcatura** dei posti.

Definizione parte statica

Matrici I e O

Diversamente dalla rappresentazione logica in cui venivano utilizzati degli indicatori alfanumerici per riferirsi ai posti e alle transizioni, nella rappresentazione matriciale viene assegnato un **indice** ad ogni posto e ad ogni transizione. Ogni indice deve essere possibilmente **continuo** (senza salti) e **biunivoco**: ogni indice corrisponde ad un posto e ogni posto corrisponde ad un indice.

- indice dei **posti**: $p : 1..|P| \to P$
- indice delle **transizioni**: $t : 1..|T| \to T$

La **dimensione** delle due matrici è $|P| \times |T|$: la **cardinalità dei posti** corrisponde al numero di righe e il **numero delle transizioni** corrisponde al numero delle colonne.

Per ogni flusso uscente dal posto i-esimo ed entrate nella transizione j-esima, l'elemento $I[i][j]$ equivale al **peso** di tale flusso, oppure 0 se il flusso non esiste. In sintesi:

$$\forall i \in 1..|P|, \forall j \in 1..|T| \quad I[i][j] = \{ W(\langle p(i), t(j) \rangle) \quad \text{se } \langle p(i), t(j) \rangle \in F, \ 0 \quad \text{altrimen}$$

Analogamente, per la matrice degli output O:

$$\forall i \in 1..|P|, \forall j \in 1..|T| \quad O[i][j] = \{ W(\langle t(j), p(i) \rangle) \quad \text{se } \langle t(j), p(i) \rangle \in F, \ 0 \quad \text{altrime}$$

Per indicare il vettore colonna k da una matrice X spesso verrà utilizzata la notazione $X[.\,][k]$.

Marcatura m

Per ogni posto, il vettore m di dimensione $|P|$ indica la **marcatura corrente**.

$$\forall i \in 1..|P| \quad m[i] = M(p(i))$$

Che **differenza** c'è tra il vettore m e M? Entrambi logicamente indicano la **stessa cosa**, ma:

- gli indici di m sono nell'insieme $1..|P|$;
- gli indici di M sono nell'insieme P.

Definizione parte dinamica

Abilitazione di una transizione

La transizione j-esima è **abilitata in una marcatura** (espressa dal vettore m) se e solo se il *vettore colonna* della sua matrice di **input** $I[.\,][j]$ è minore o uguale alla marcatura corrente m:

$$\boxed{m \ [\ t(j) >} \iff I[.\,][j] \leq m$$

o se proprio vogliamo essere precisi...

$$\boxed{m \ [\ t(j) >} \iff \forall i \in 1..|P| \quad I[i][j] \leq m[i].$$

In sostanza, si controlla che il numero dei gettoni di ogni posto $p(i)$ del *preset* sia maggiore o uguale del peso dell'arco che collega $p(i)$ alla transizione.

Scatto di una transizione

Lo **scatto** di una transizione j in una marcatura m produce una marcatura m' che si ricava sottraendo elemento per elemento al vettore di partenza la colonna j-esima della matrice di input e quindi sommando al risultato la colonna j-esima della matrice output.

$$\boxed{m[\, t(j) >\, m'} \iff m' = m - I[.][j] + O[.][j]$$

$$o\ se\ proprio\ vogliamo\ essere\ precisi\ldots$$

$$\boxed{m[\, t(j) >\, m'} \iff \forall i \in 1..|P| \quad m'[i] = m[i] - I[i][j] + O[i][j].$$

È importante notare come nell'operazione sopra due operandi su tre sono matrici costanti (I e O): è quindi possibile **pre-calcolare** $O - I$ per efficienza.

Matrice di incidenza C

La matrice $O - I$ presentata sopra è infatti chiamata **matrice di incidenza** e si indica con la lettera C. È utile per ottimizzare l'operazione *scatto* di una rete in forma matriciale. In formule:

$$\forall i \in 1..|P|, \forall j \in 1..|T| \quad C[i][j] = O[i][j] - I[i][j].$$

C **non sostituisce** le matrici di input I e output O, in quanto I è ancora necessaria per calcolare l'abilitazione di una transizioni. Per le **reti non pure**, infatti, il valore presente in un qualsiasi posto della matrice potrebbe essere dato da una *qualsiasi combinazione* di pesi relativi ad archi in ingresso ed uscita, in quanto per la stessa posizione $\langle i,\, j \rangle$ entrambe le matrici potrebbero assumere un valore.

Sequenze di scatti

Si consideri una **sequenza** di n **scatti** che porti la rete da una marcatura iniziale M a una marcatura M^n.
Ripetendo il seguente processo per n scatti

$$\boxed{M[\, t_1 >} \boxed{M'}, \quad \boxed{M'[\, t_2 >} \boxed{M''} \rightarrow \boxed{M[\, t_1 t_2 >} \boxed{M''},$$

si rinomini la sequenza ottenuta nel seguente modo:

$$\boxed{M[\, s >} \boxed{M^{(n)}}.$$

Esiste un **legame diretto** tra la marcatura iniziale e quella finale, che non preveda eseguire i **singoli passi**? A livello di matrici, l'esecuzione in sequenza di x_1 volte di t_1, x_2 volte di t_2 fino a x_n volte di t_n è **fattorizzabile**. Definendo un vettore s tale per cui

$$\forall j \in 1..|T| \quad s[j] = \#\text{ di volte in cui } t(j) \text{ scatta}$$

è facile notare come l'**ordine di scatto non conta**. Calcolando quindi Cs è quindi possibile calcolare l'**effetto netto** dell'intera sequenza di scatti, svolgendo un'unica operazione. Sommando Cs alla marcatura iniziare M, si ottiene lo stato della marcatura finale $M^{(n)}$.

$$M^{(n)} = M + Cs.$$

È opportuno specificare che s non è in grado di determinare l'**esistenza** o l'**ordine** della sequenza presa in considerazione. Non è quindi possibile sapere se s corrisponde a una *sequenza ammissibile* di scatti, ma è facile escluderlo: se $M^{(n)}$ contiene numeri negativi, allora s corrisponde sicuramente ad una **sequenza inammissibile**. In generale, se anche in un solo passo intermedio $M^{(i)}$ è negativo, allora la sequenza considerata non è ammissibile.

In conclusione, è possibile effettuare questo calcolo solo se si è **certi** che la sequenza di scatti sia **ammissibile**.

Di seguito è presente un **esempio** che potrebbe chiarire le idee.

Analisi statica

È ora possibile introdurre due tecniche di **analisi statica** che si pongono come obiettivo la ricerca di **invarianti** all'interno della rete. Più nello specifico, esistono:

- P**-invarianti**: invarianti relative alla **marcatura dei posti**;
- T**-invarianti**: invarianti relative alla **sequenza di scatto**.

P-invarianti

Una P-invariante è una caratteristica relativa alla marcatura dei posti che **non cambia**; viene rappresentata da un **vettore di pesi** h di dimensione $|P|$.

Il vettore h ricorda la funzione $H : P \to \mathbb{N} \setminus 0$ dalla definizione di **rete conservativa**, con l'unica differenza che gli elementi di h possono essere nulli o negativi.
Nel caso in cui una P-invariante abbia tutti i pesi maggiori di zero allora $h \equiv H$: la rete sarebbe quindi conservativa e quindi anche **limitata**.

Tramite l'analisi delle P-invarianti è quindi possibile stabilire se una rete è conservativa e quindi limitata, fornendo un'**alternativa** al metodo dell'albero di copertura.

Per ogni marcatura m' *raggiungibile* da m, l'**invariante** è il prodotto vettoriale della marcatura con h.

$$\forall m' \text{ raggiungibile da } m \quad hm = hm'$$

Se m' è raggiungibile da m, allora esiste una **sequenza di scatti** ammissibile s tale per cui

$$m' = m + Cs,$$

è quindi possibile moltiplicare entrambi i membri per h in modo da avere

$$hm = h(m + Cs)\ hm = hm + hCs$$

quindi, semplificando i due hm,

$$\boxed{hCs = 0}.$$

Ritornando alle assunzioni iniziali, tale proprietà vale solo se esiste una **sequenza di scatti ammissibile**: le informazioni su m non sono andate perse.

Nell'ultima formula è presente la matrice C (**nota**), il vettore di pesi h (**incognita**) e il vettore s (**variabile libera**). La relazione vale infatti **per ogni** sequenza di scatti ammissibile s. La **formula precisa** è quindi:

$$\forall s \quad hCs = 0 \text{ con } s \text{ rappresentante una sequenza di scatti ammissibile.}$$

Assumendo per un momento che $hC = 0$, allora qualsiasi sia s il risultato è sempre zero, **perdendo informazione** su quest'ultima.
Analogamente, in una rete che possiede una **transizione morta** la corrispondente posizione in s sarà sempre zero causando l'azzeramento anche della relativa posizione nel risultato.

Non è quindi **necessario** che $hC = 0$ per far sì che $hCs = 0$, ma è sicuramente **sufficiente**.

In conclusione, considerando solo $hC = 0$ è possibile **escludere** la **componente dinamica** dalla proprietà ragionando solo in base alle informazioni topologiche (C) della rete. Trovare l'h che rende $hC = 0$ è quindi **condizione sufficiente** ma non necessaria per cui h è una P-invariante, tenendo a mente che esistono comunque h che non rendono $hC = 0$ ma potrebbero essere P-invarianti.

I P-invarianti determinati con l'espressione $hC = 0$ non dipendono dalla marcatura iniziale ma solo dalla **topologia** della rete: se venisse considerato anche s sarebbero P-invarianti per qualunque evoluzione della rete *a partire dalla marcatura* m.

Il sistema $hC = 0$ è un **sistema di equazioni lineare**, risolvibile con varie tecniche presentate successivamente.

Copertura di P-invarianti

Una **combinazione lineare** di P-invarianti (e quindi di soluzioni del sistema) è anch'essa una P-invariante.

Una P-invariante h avente tutti i pesi ≥ 0 è detta **semi-positiva**. Se un posto ha *peso positivo* in una P-invariante semi-positiva, allora è **limitato** nel numero di gettoni massimi che può contenere.
Se così non fosse, infatti, il contributo nella sommatoria vista precedentemente di $h[i]m[i]$ (con $h[i] \geq 0$ e $m[i] > 0$) sarebbe **potenzialmente illimitato**.
Se un posto ha peso nullo, potrebbe quindi essere **illimitato**.

Avere pesi dei posti **negativi** non fornisce nessuna informazione sulla limitatezza degli stessi nella rete.

Infine, se **per ogni posto** esiste una P-invariante semi-positiva il cui peso del posto è positivo, allora la rete è **limitata**. Matematicamente:

$$\forall i \in 1..|P|, \exists h \in \mathbb{R}^{|P|} \quad hC = 0 \land h[.] \geq 0 \land h[i] > 0 \qquad \Rightarrow C \text{ rappresenta una ret}$$

Si può anche dire che se esiste una **combinazione lineare** di P-invarianti tale per cui il P-invariante risultante è **strettamente positivo**, allora vuol dire che la funzione $H : P \to \mathbb{N}^+$ (che restituisce proprio quei pesi trovati) è una delle funzioni per cui la rete risulta **conservativa**.

Esempio

Di seguito è illustrato un esempio sulle proprietà viste delle P-invarianti.

Date le matrici I, O e $C = O - I$ sopra è necessario risolvere il sistema $hC = 0$

$$h \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ -1 & 1 & -4 & 4 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} = 0,$$

che si trasforma nel seguente sistema di equazioni lineari:

$$\begin{cases} -h_0 & +h_1 & -h_2 & & & = 0 & & & \\ +h_0 & -h_1 & +h_2 & & & = 0 & & & \\ & & -4h_2 & -h_3 & +h_4 = 0 & & +4h_2 & +h_3 & -h_4 = 0. \end{cases}$$

È facilmente notabile che la prima e la seconda equazioni sono uguali a meno di una costante moltiplicativa: sono quindi **linearmente dipendenti**, insieme alla terza e alla quarta.
Chiedendo quindi a Wolfram|Alpha di risolvere il sistema

$$\begin{cases} -h_0 & +h_1 & -h_2 & & = 0 \\ & & -4h_2 & -h_3 & +h_4 = 0, \end{cases}$$

otteniamo le seguenti **basi**:

$$\langle -1, 0, 1, 0, 4 \rangle,\ \langle 1, 0, -1, 4, 0 \rangle,\ \langle 1, 1, 0, 0, 0 \rangle,$$

che se combinate linearmente generano **infinite soluzioni**.

Tra le tre basi ottenute, l'ultima è **particolare** in quanto ha tutti gli elementi semi-positivi: di conseguenza i posti corrispondenti alle prime due posizioni (con pesi strettamente positivi) sono **limitati**.
Rimane comunque difficile stabilire se la **rete** è limitata oppure no.

Algoritmo di Farkas (1902)

E se esistesse un algoritmo che predilige la ricerca di **basi minime** per un sistema di equazioni, privilegiando quelle **semipositive**? L'algoritmo in questione è l'**algoritmo di Farkas**.

Sia n il numero di righe e m il numero di colonne della matrice di incidenza C.
L'algoritmo inizia creando una matrice D_0 ottenuta da C alla quale viene appesa una matrice di identità E_n di dimensione $n \cdot n$. Quindi, **per ogni colonna** i da 1 a m si considerano le **coppie di righe** $\langle d_1, d_2 \rangle$ aventi nella i-esima riga numeri di segno opposto (non necessariamente uguali in valore assoluto).

Per **ogni coppia di righe** $\langle d_1, d_2 \rangle$:

- si crea una riga temporanea d ottenuta **combinando linearmente** la linea d_1 moltiplicata per il valore assoluto dell'i-esimo elemento della riga d_2 e sommando il viceversa.
  Così facendo, l'i-esimo argomento della riga d è uguale a **zero**;
- per evitare instabilità numerica dovuta a numeri troppo grandi si divide d per il **massimo comun divisore** dei suoi elementi, assegnando il risultato a d';
- si estende la matrice D_{i-1} aggiungendo una nuova ultima riga d'.

Una volta terminato il ciclo sulla coppia di righe, si **scartano** tutte le righe della matrice D_{i-1} cui i-esimo elemento è diverso da 0. Infine, al termine del ciclo esterno si eliminano le prime m colonne di D_m, essendo azzerate. Nella matrice risultante (corrispondente alla matrice E_n) sono presenti i P-invarianti.

Continuazione dell'esempio con Farkas

Nell'esempio iniziato in precedenza si era arrivati ad un punto in cui si necessitava ottenere **basi semi-positive** e quindi P-invarianti semi-positivi: per fare ciò si può applicare l'algoritmo sopra descritto.

Si inizia creando la matrice $D_0 = [C \mid E_n]$:

$$D_0 = \left[\begin{array}{ccccccccc} -1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ -1 & 1 & -4 & 4 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 \end{array}\right].$$

Osservando la **prima colonna** ($i = 1$) si nota che sono presenti due coppie di righe aventi segno opposto: la prima e la seconda, la seconda e la terza.

A questo punto si possono **combinare linearmente** le coppie appendendo i risultati come **ultima riga**:

$$D_0 = \left[\begin{array}{ccccccccc} -1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ -1 & 1 & -4 & 4 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -4 & 4 & 0 & 1 & 1 & 0 & 0 \end{array}\right].$$

Le prime tre righe contengono nella colonna i-esima (la prima) elementi non nulli; si **scartano**:

$$D_1 = \begin{bmatrix} 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -4 & 4 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}.$$

Si procede iterativamente senza ulteriori azioni fino alla terza colonna, dove sono presenti **due coppie di righe** aventi segni opposti in posizione i: la prima e la seconda, la prima e la quarta. Applicando gli stessi passaggi di prima, la matrice D_3 che si ottiene è la seguente:

$$D_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

Infine, considerando la matrice D_m senza le prime colonne nulle, si ottengono le seguenti basi di h:

$$\langle 1, 1, 0, 0, 0 \rangle,\ \langle 0, 1, 1, 0, 4 \rangle,\ \langle 0, 0, 0, 1, 1 \rangle.$$

Interpretazione dei risultati ottenuti

È facile notare come la rete sia **limitata**, in quanto per ogni posizione (*posto*) esiste almeno un P-invarianti semipositivi cui valore in tale posizione è **strettamente positivo**.

Conoscendo ora possibili valori per h, nella relazione $hm = hm_0$ l'unica incognita al momento è m: la **marcatura generica** che è possibile raggiungere.

Considerando il **primo P-invariante** $h_1 = \langle 1, 1, 0, 0, 0 \rangle$ e la marcatura iniziale $m_0 = \langle 4, 0, 4, 2, 0 \rangle$ si ottiene la **relazione** $h_1 m = hm_0 = 4$ riguardante i seguenti posti:

$$\mathrm{LettoriPronti} + \mathrm{LettoriAttivi} = 4. \qquad (h_1)$$

I posti cui peso è 1 (proveniente da h_1) sono $\mathrm{LettoriPronti}$ e $\mathrm{LettoriAttivi}$, mentre il 4 dipende dal numero di gettoni in $\mathrm{LettoriPronti}$ in m_0: la somma dei due è garantita essere **costante**.
In generale, i **termini** a sinistra dipendono da h, quelli a destra da m_0.

Per $h_2 = \langle 0, 0, 0, 1, 1 \rangle$ il procedimento è lo stesso:

$$\mathrm{ScrittoriPronti} + \mathrm{ScrittoriAttivi} = 2. \qquad (h_2)$$

Il terzo risultato, per $h_3 = \langle 0, 1, 1, 0, 4 \rangle$ è il **più interessante**:

$$\mathrm{LettoriAttivi} + \mathrm{Risorsa} + 4 \cdot \mathrm{ScrittoriAttivi} = 4. \qquad (h_3)$$

In tutti i risultati è **implicito** che tutti gli **operandi** devono essere **interi maggiori o uguali a zero**.

Con dell'algebra è possibile riscrivere l'**ultimo risultato** (h_3) nel seguente modo:

$$\begin{aligned}\frac{4 \cdot \mathrm{ScrittoriAttivi}}{4} &= \frac{4 - \mathrm{LettoriAttivi} - \mathrm{Risorsa}}{4} \\ \mathrm{ScrittoriAttivi} &= 1 - \frac{\mathrm{LettoriAttivi}}{4} - \frac{\mathrm{Risorsa}}{4} \\ \mathrm{ScrittoriAttivi} &= 1 - \frac{\mathrm{LettoriAttivi} + \mathrm{Risorsa}}{4}.\end{aligned}$$

Dall'ultima espressione è possibile determinare che gli $\mathrm{ScrittoriAttivi}$ sono o **zero** o **uno**, in quanto $\frac{\mathrm{LettoriAttivi}+\mathrm{Risorsa}}{4}$ è necessariamente positivo. A questo punto "LettoriAttivi" + "Risorsa" si sa essere un valore positivo, che diviso per 4 rimane un numero positivo, a meno che non siano entrambi 0.
È quindi garantito matematicamente che gli scrittori sono in **mutua esclusione**.

Procedendo similmente per i lettore, si ottiene che:

$$\mathrm{LettoriAttivi} = 4 \cdot (1 - \mathrm{ScrittoriAttivi}) - \mathrm{Risorsa}.$$

Si può quindi concludere che:

- $\mathrm{ScrittoriAttivi} \leq 1$;
- $\mathrm{LettoriAttivi} \leq 4$;
- $\mathrm{LettoriAttivi} > 0 \implies \mathrm{ScrittoriAttivi} = 0$;
- $\mathrm{ScrittoriAttivi} > 0 \implies \mathrm{LettoriAttivi} = 0$.

T-invarianti

I T-invarianti sono concettualmente **molto simili** ai P-invarianti, ma pongono alcuni vincoli di **invariabilità** sulle **sequenze di scatti**, ovvero:

- si **possono ripetere** ciclicamente;
- portano alla situazione iniziale (**stato base**).

Partendo dall'equazione $m' = m + Cs$, poniamo il **vincolo** $m' = m$ in quanto la sequenza deve tornare alla marcatura iniziale. Le **soluzioni** del sistema sono quindi:

$$Cs = 0,$$

con C costante e s un **vettore di incognite**, rappresentante una sequenza ammissibile.

Se si risolve il sistema e si trova un vettore s rappresentante una sequenza di scatti ammissibile, allora tale sequenza è **ciclica** per cui s è un T-invariante.

A differenza dei P-invarianti (trovarne uno è *condizione sufficiente* purché sia valido), per un T-invariante soddisfare l'equazione è **condizione necessaria** ma non sufficiente per la **validità** della sequenza.

Se una rete è **limitata** e copribile da T-invarianti, allora è dimostrabile che è anche **viva**.

Controllori con specifica a stati proibiti

Tramite le reti di Petri si possono **modellare dei controllori** che forzano o limitano certi comportamenti del sistema: se si desidera cioè che la rete rispetti una certa **invariante** si introduce un controllare che la forzi. *Controllare* significa **assicurarsi** che vengano rispettate certe proprietà.

È possibile definire gli **stati** come situazioni che *si possono verificare*, e le **transizioni** come *eventi che si verificano*. Lo scopo è **controllare** che le transizioni possano svolgere certe operazioni, oppure no.

Esistono due **classi di problemi** che limitano la capacità espressiva dei controllori:

- **non tutte le transizioni sono osservabili**: il controllore non ne ha le capacità, oppure è un'attività troppo onerosa;
- l'osservazione di alcune situazioni ne **comporta il cambiamento**.

Inoltre, **non tutto è controllabile**: non si può chiedere ad una centrale nucleare in surriscaldamento di non esplodere, ma si possono attivare i sistemi di sicurezza.

Nel modello del **controllore a stati proibiti**, l'attività di *controllo* si traduce formalmente in una **combinazione lineare** delle marcature che deve rimanere sotto una certa soglia.
Si vincola quindi per un **sottoinsieme di posti** che la combinazione lineare di una marcatura M con un **vettore dei pesi** L sia minore o uguale (e non solo *uguale* come nei P-invarianti) di una soglia data:

$$LM \leq b.$$

Come abbiamo visto nel corso di *Ricerca Operativa*, è **sempre** possibile riportare un **sistema di disequazioni** ad un sistema di equazioni **inserendo variabili aggiuntive** (***slack***) semipositive:

$$LM + x = b \,|\, x \geq 0.$$

Mutua esclusione

Il problema della mutua esclusione è l'**accesso esclusivo** a zona critica da parte di più soggetti. Nel seguente esempio si vuole imporre che non sia possibile avere gettoni contemporaneamente in P_1 e P_3.

Matematicamente il vincolo si può esprimere con la seguente **disequazione**.

$$P_1 + P_3 \leq 1$$

La tecnica del *controllore a stati proibiti* aggiunge tanti **posti di controllo** quanti sono il **numero di disequazioni** (e quindi il numero di variabili di *slack*) per modificare il comportamento delle transizioni.

In questo caso, per trasformare la disequazione in un'equazione si aggiunge una variabile di slack, rappresentante il nuovo **posto controllore** P_c.

$$P_1 + P_3 + P_c = 1$$

Per collegare P_c alle diverse transizioni occorre aggiungere una riga C_c nella **matrice di incidenza** C_s:

$$C_{\text{nuova}} = \begin{bmatrix} C_s \\ C_c \end{bmatrix}.$$

Inoltre, bisogna aggiungere la marcatura iniziale M_{0c} del posto P_c alla **marcatura iniziale** del sistema M_{0s}:

$$M_0 = \begin{bmatrix} M_{0s} \\ M_{0c} \end{bmatrix}.$$

Riscrivendo quindi il vincolo tramite le matrici otteniamo:

$$LM_s + M_c = b.$$

Sia $\begin{bmatrix} LI \end{bmatrix}$ la giustapposizione tra L e la **matrice identità** I e M la giustapposizione di M_s e M_c, allora:

$$\begin{bmatrix} LI \end{bmatrix} M = b.$$

L'espressione sopra ricorda la definizione di P-invariante ($hm = 0$). Volendo forzare che $[LI]$ sia un'invariante, riprendiamo quindi la relativa definizione:

$$\begin{bmatrix} LI \end{bmatrix} C = 0,$$

che, rifacendosi al vincolo originale, si può a sua volta riscrivere come

$$LC_s + IC_c = 0$$
$$\boxed{C_c = -LC_s}.$$

Le righe da aggiungere al sistema C_c sono quindi **uguali** a $-LC_s$, dove:

- C_s è la **matrice di incidenza** del **sistema** originale;
- L è il **vincolo desiderato**, fissato;
- C_c la si trova con un **semplice calcolo matriciale**.

Sintesi del controllore

Continuando l'esempio precedente, l'obiettivo è trovare

$$C_s = \begin{bmatrix} 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix}$$
$$-LC_s = \begin{bmatrix} -1 & -1 & 1 & 1 \end{bmatrix}.$$

Il vettore $-LC_s$ definisce gli **archi in ingresso** e **in uscita** dalle transizioni per il **posto controllore** P_c:
il posto ha in ingresso T_0 e T_1 (gli elementi con -1) mentre in uscita T_2 e T_3 (gli elementi con 1).

Da questi risultati è possibile ottenere anche la **marcatura iniziale** del posto controllore (M_{0_c}):

$$LM_{0s} + M_{0c} = b$$
$$\boxed{M_{0c} = b - LM_{0s}}.$$

Essendo tutti termini noti, è facile rispondere che la **marcatura iniziale** di P_c è uguale a 1.

In **conclusione**, le due formule principali da conoscere sono le seguenti:

- $\boxed{C_c = -LC_s}$ per calcolare le **righe** da aggiungere alla **matrice di incidenza** C_s;
- $\boxed{M_{0c} = b - LM_{0s}}$ per calcolare la **marcatura iniziale** del posto controllore P_c.

Esempio

Riprendendo il classico esempio dei **lettori** e **scrittori**, lo scopo di questo esercizio è collegare le due parti assicurando l'accesso esclusivo alla risorsa.

Dovendo imporre la **mutua esclusione** tra lettori e scrittori, poniamo i seguenti vincoli:

$$\begin{cases} \text{LettoriAttivi} + \text{ScrittoriAttivi} \leq 1 \\ \text{LettoriAttivi} + 4 \cdot \text{ScrittoriAttivi} \leq 4. \end{cases}$$

Il primo vincolo è incluso nel secondo, quindi possiamo ignorarlo.
Date le **seguenti informazioni**, possiamo realizzare nella rete i vincoli sopra.

$$M_0 = \begin{bmatrix} 4 & 0 & 2 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \quad (\text{Dati della r}$$

$$LM \leq b \qquad (\text{Vincolo})$$

$$L = \begin{bmatrix} 0 & 1 & 0 & 4 \end{bmatrix} \quad b = 4 \qquad (\text{Parametri del vincolo})$$

È sufficiente quindi sfruttare le formule viste prima per trovare la **nuova riga della matrice di incidenza** e la **marcatura iniziale** di P_0.

$$\begin{aligned} C_c &= -LC_s = \begin{bmatrix} -1 & 1 & 4 & -4 \end{bmatrix} \\ M_{0_c} &= b - LM_{0_s} = 4. \end{aligned}$$

Reti con priorità

Ad ogni transizione è associata una **priorità**: quando in una marcatura n transizioni sono abilitate, la scelta della prossima da far scattare è **determinata** dalla sua priorità.

Date le opportune priorità, è quindi possibile **guidare** la progressione della rete verso la soluzione richiesta.

Ci sono due svantaggi principali a questo approccio:

- rischio di creare di **cicli infiniti**;
- si perde la *località di decisione* della abilitazione di una transizione: non è quindi più possibile fare analisi locale.

Reti temporizzate

Abbiamo visto come le reti di Petri siano un modello estremamente potente per modellare un'infinita varietà di situazioni anche molto diverse tra loro. Tuttavia, alcune categorie di problemi richiedono un approccio più mirato, ovvero un'**estensione delle reti di Petri** specifica per il loro studio: è questo il caso dei **sistemi Hard Real-time**, di cui ora tratteremo approfonditamente.

In molte applicazioni il **tempo** è un fattore essenziale: si pensi per esempio ad un termostato intelligente che deve accendere e spegnere i termosifoni di una casa in base ad un programma giornaliero oppure ad un autovelox, che in base al tempo di andata e ritorno di un'onda elettromagnetica dev'essere in grado di calcolare la velocità di un veicolo.
Ma non tutti i sistemi basati sul tempo sono uguali: alcuni di essi richiedono infatti il rispetto assoluto di una serie di **vincoli temporali stretti**, ovvero requisiti sul tempo di esecuzione di certe operazioni che devono essere rispettati per evitare gravi conseguenze. Considerando per esempio il sistema di controllo di una centrale nucleare, qualora si inizi a rilevare un'aumento eccessivo delle temperature nel reattore tale software dev'essere in grado di reagire entro un certo tempo strettissimo, pena l'esplosione dell'apparato. Sistemi di questo tipo prendono il nome, come già detto, di **sistemi Hard Real-time**, dove l'aggettivo "Hard" indica proprio la durezza richiesta nel rispetto dei vincoli temporali.

Visto il loro tipico impiego in situazioni di rischio o di pericolo, i committenti di sistemi di questo tipo potrebbero voler avere **prova del loro corretto funzionamento** prima ancora che questi vengano installati. I modelli finora descritti potrebbero però non essere sufficienti: non è per esempio abbastanza un'*analisi stocastica* della rete, in quanto in virtù dei rischi a cui un malfunzionamento del sistema esporrebbe bisogna essere assolutamente **certi** del suo corretto funzionamento, certezza che può essere ottenuta solo con un **modello deterministico**.
Ecco quindi che come strumento di specifica e comunicazione col cliente vengono sviluppate una serie di estensioni alle reti di Petri progettate specificamente per trattare il concetto di *tempo* e *ritardo*: tra queste distingueremo in particolare le **reti Time Basic** (*Ghezzi et al., 1989*), oggi le più usate.

Modelli temporali

Esistono una serie di proposte per modellare il concetto di **tempo** (*deterministico*) all'interno delle reti di Petri. Esse si dividono sostanzialmente in due grandi categorie:

- quelle che introducono **ritardi sui posti**;
- quelle che introducono **ritardi sulle transizioni**.

Tempo sui posti

Il tempo associato a ciascun posto rappresenta il **tempo che un gettone deve rimanere in tale posto prima di essere considerato per l'abilitazione** di transizioni che hanno tale posto nel proprio preset.

Dopo lo scatto di una transizione i gettoni generati in un posto non fanno cioè funzionalmente parte della sua marcatura prima che sia passato un dato intervallo di tempo Δ. Tale Δ può quindi essere considerato la **durata minima di permanenza** del gettone in tale posto, bloccando così quella porzione di stato del sistema per un certo periodo.

Tempo sulle transizioni

Quando si associa un tempo ad una transizione è bene indicare che cosa esso rappresenti. Il tempo di una transizione può infatti rappresentare due concetti molto differenti:

- la **durata della transizione**, ovvero il tempo richiesto dopo lo scatto della transizione perché vengano generati i gettoni nel suo postset (una sorta di *ritardo di scatto*);
- il **momento dello scatto** della transizione, che può essere espresso in modo diverso a seconda del modello.

Esistono a dire il vero anche modelli misti che permettono di specificare sia la durata di una transizione che il suo tempo di scatto.

Equivalenza tra tempi sui posti e sulle transizioni

È facile dimostrare che sia le reti che definiscono tempi sui posti che quelle che definiscono tempi sulle transizioni, sia come durata che come momento dello scatto, sono **funzionalmente equivalenti**, ovvero permettono di rappresentare lo stesso insieme di sistemi.

Ciò è testimoniato dal fatto che, come mostra la figura sottostante, ogni rete avente tempo sui posti può essere trasformata in una rete con durata delle transizioni semplicemente aggiungendo una transizione di "ritardo" e separando il posto in due. Ovviamente vale anche il viceversa.

Similmente, reti con durata delle transizioni possono essere trasformate in reti con tempi di scatto per le transizioni modellando esplicitamente con un posto il ritardo con cui vengono generati gettoni nel postset della transizione originale.

Tempi di scatto

Ritornando un attimo sui nostri passi, diamo ora un'occhiata migliore a come si possono definire i tempi di scatto di una transizione.

Nella definizione dei tempi di scatto delle transizioni esistono infatti una serie di alternative molto differenti. Innanzitutto, i tempi possono essere:

- **unici**, ovvero ogni transizione scatta (o può scattare) in *uno e un solo* specifico momento;
- **multipli**, ovvero ogni transizione scatta (o può scattare) in *uno in un insieme* di momenti. A seconda del modello considerato tali insiemi possono essere veri e propri **insiemi matematici** (*es. reti TB*) oppure **intervalli**.

Si noti come i primi possono essere visti come casi particolari dei secondi.
Considerando ciò, gli insiemi (anche unitari) di tempi di scatto si distinguono poi in due categorie:

- **insiemi costanti**, ovvero tali per cui l'insieme dei tempi di scatto è **definito staticamente** ed è sempre uguale indipendentemente dall'evoluzione della rete;
- **insiemi variabili**, ovvero tali per cui l'insieme dei tempi di scatto può **variare dinamicamente** in base allo stato della rete o a porzioni di esso (*es. reti TB e HLTPN*).

Anche in questo caso i primi possono essere visti come un caso particolare dei secondi, in cui cioè l'insieme *potrebbe* variare ma non varia mai.
Infine, i tempi di scatto stessi possono essere divisi in base a come vengono definiti:

- **tempi relativi**, ovvero espressi *solo* in termini relativi al tempo di abilitazione della transizione (*es. "2 ms dopo l'abilitazione"*);
- **tempi assoluti**, ovvero espressi in termini relativi a tempi assoluti e ai tempi associati ai gettoni che compongono l'abilitazione (*es. "dopo 3 minuti dall'avvio del sistema" o "dopo 4 ms dal tempo associato all'ultimo gettone nell'abilitazione"*) (*es. reti TBe TCP*).

Nuovamente, i primi possono essere visti come un sottoinsieme dei secondi.

Tutto questo insieme di variabili permette di definire reti temporizzate basate su tempi di scatto delle transizioni anche molto diverse tra di loro. Avremo per esempio le *reti Time Petri*, che utilizzano tempi di scatto relativi, multipli e a intervalli costanti; le *reti Time Petri colorate*, simili alle precedenti ma che usano tempi assoluti; le *reti Time Petri ad alto livello*, che usano insiemi variabili, e molte altre.

Tra tutte queste tipologie, tuttavia, ci concentreremo sulle **reti Time Basic**. In virtù delle inclusioni di cui abbiamo già detto tali reti saranno quindi le più generali possibile e, dunque, anche le più interessanti.

Reti Time Basic

Prima di darne una vera e propria definizione matematica iniziamo a introdurre le **reti Time Basic** (**TB**) in modo informale.

Introdotte per la prima volta da Ghezzi e dai suoi collaboratori nel 1989, le reti TB associano **insiemi variabili** di tempi di scatto **assoluti** alle transizioni: ciascuna transizione possiede cioè un insieme di tempi in cui *potrebbe* scattare, definito in maniera dinamica a seconda dello stato. Tali tempi di scatto potrebbero poi essere definiti sia in termini assoluti che in termini dei **tempi associati ai gettoni**.

Nelle reti TB infatti i gettoni non sono più anonimi, ma caratterizzati ciascuno da un **timestamp** che indica il **momento in cui sono stati creati** ($\mathrm{t}(posto)$). A differenza delle normali reti di Petri i gettoni sono quindi **distinguibili**: questo non significa che due gettoni non possano avere lo stesso timestamp, ma solo che non tutti i gettoni sono uguali (*mentre gettoni generati dalla stessa transizione o da transizioni diverse scattate in parallelo avranno invece lo stesso timestamp*).

Per ogni transizione viene poi introdotto il concetto di **tempo di abilitazione** ($\mathbf{enab}$), ovvero il **momento in cui la transizione viene abilitata**: poiché una transizione è abilitata quando tutti i posti nel suo preset contengono tanti gettoni quanto il peso dell'arco entrante in essa, il tempo di abilitazione di una transizione sarà pari al **massimo tra i timestamp** dei gettoni che compongono la **tupla abilitante**.
Poiché i posti nel preset della transizione potrebbero contenere più gettoni di quelli necessari per farla scattare, una transizione potrebbe avere **più tempi di abilitazione diversi** in base ai gettoni considerati per la tupla abilitante.

Ovviamente i **tempi di scatto** delle transizioni **non potranno essere minori** del tempo di abilitazione, in quanto una transizione non può scattare prima di essere abilitata. Gli insiemi dei tempi di scatto potranno invece *dipendere* dal tempo di abilitazione: così, per esempio, una transizione potrebbe scattare 2 secondi dopo essere stata abilitata, oppure tra 3 e 5 minuti dall'abilitazione.

A tal proposito, molto spesso i tempi di scatto saranno rappresentati come **intervalli** $[min, max]$ piuttosto che come insiemi: nei nostri esempi adotteremo questa convenzione, ma è bene tenere in mente che tali insiemi potrebbero avere qualunque possibile forma.

Definizioni matematiche

Facciamo un po' di chiarezza introducendo delle definizioni rigorose per tutto quanto citato nell'introduzione.
Una rete Time Basic è una **6-tupla** del tipo $\langle P, T, \Theta, F, tf, m_0 \rangle$, dove:

- P, T, F sono identici all'insieme dei posti, delle transizioni e al flusso delle normali reti di Petri;
- Θ (*theta*) è il **dominio temporale**, ovvero l'insieme numerico che contiene le rappresentazioni degli istanti di tempo;
- tf è una funzione che associa ad ogni transizione $t \in T$ una **funzione temporale** $\mathrm{tf_t}$ che data in input la **tupla abilitante** $\mathbf{en}$, ovvero l'**insieme dei timestamp** dei gettoni scelti per l'abilitazione nel preset, restituisce un **insieme di tempi di scatto possibili**:

  $$\mathrm{tf_t}(en) \subseteq \Theta$$

  Per esempio, se per una transizione t i tempi di scatto sono nell'intervallo $[min, max]$, allora $\mathrm{tf_t}(en) = r \,|\, min \leq r \leq max$.
- m_0 è un multiset che esprime la **marcatura iniziale**: si tratta cioè di una funzione che ad ogni **posto** associa un insieme di coppie **timestamp-molteplicità** che indicano il numero di gettoni con tale timestamp all'interno del posto:

  $$m_0 : P \to (\theta, \mathrm{mul}(\theta)) \,|\, \theta \in \Theta$$

  Tutte le **marcature** esprimibili per le reti Time Basic assumeranno la forma di simili funzioni.

Con questi costrutti matematici siamo in grado di descrivere completamente lo stato di una rete Time Basic. Tuttavia sorge ora spontanea una domanda: dovendo modellare il concetto di tempo, come **evolve** una rete TB?

Evoluzione

Dovendo modellare lo **scorrere del tempo**, le reti Time Basic dovranno operare una serie di accortezze per quanto riguarda la loro evoluzione.

Abbiamo per esempio già detto che il tempo di scatto di una transizione dovrà necessariamente essere maggiore del suo tempo di abilitazione, e che tale tempo di scatto sarà pari al timestamp dei gettoni generati dalla transizione. Tuttavia, questo non è abbastanza: il concetto di tempo è particolarmente sfuggente e, soprattutto, **difficile da definire in maniera univoca**.
Al contrario, per le reti Time Basic vengono definite diverse **semantiche temporali**, ovvero diverse interpretazioni del concetto di "tempo" che richiederanno il rispetto di una serie di assiomi durante l'evoluzione della rete. Tali interpretazioni, ciascuna utile per modellare diversi sistemi e requisiti di tempo, variano anche in complessità; in questo corso partiremo dunque dalla semantica più semplice per poi costruire su di essa quelle più complesse.

- **Semantica temporale debole** (WTS)
- **Semantica temporale monotonica debole** (MWTS)
- **Semantica temporale forte** (STS)
- **Semantica temporale mista**

Semantica temporale debole (WTS)

Informalmente, la **semantica temporale debole** (*Weak Time Semantic*, **WTS**) impone che una transizione possa scattare *solo* in uno degli **istanti identificati dalla sua funzione temporale** e **non possa scattare *prima* di essere stata abilitata**.

Tuttavia, ***una transizione non è costretta a scattare*** anche se abilitata: essa *potrebbe* scattare, ma non è forzata a farlo. Questo permette di modellare eventi solo **parzialmente definiti**, ovvero che potrebbero accadere sotto determinate condizioni ma non è possibile dire se lo faranno o no: esempi notevoli sono guasti o decisioni umane, eventi cioè non completamente prevedibili. Si noti che a differenza dei modelli stocastici delle reti di Petri in questo caso non ci interessa la *probabilità* con cui gli eventi potrebbero accadere, ma solo che potrebbero accadere.

Per imporre questa interpretazione del concetto di tempo l'evoluzione di una rete Time Basic deve seguire i seguenti **assiomi temporali**:

- (**A1**) **Monotonicità rispetto alla marcatura iniziale**: tutti i tempi di scatto di una sequenza di scatto devono essere **non minori** ($\geq$) di uno qualunque dei timestamp dei gettoni della marcatura iniziale.
  Ogni marcatura deve cioè essere **consistente**, ovvero non contenere gettoni prodotti "nel futuro".

- (**A3**) **Divergenza del tempo** (*non-zenonicità*): **non** è possibile avere un **numero infinito** di scatti in un intervallo di **tempo finito**.
  Questo assioma serve ad assicurarsi che il tempo **avanzi**! Esso assicura cioè che il tempo

non si possa fermare e soprattutto che esso non possa essere suddivisibile in infinitesimi: il sistema evolve soltanto quando il tempo va avanti.

Le sequenze di scatti che soddisfano questi due assiomi vengono dette **sequenze ammissibili in semantica debole**.

Semantica temporale monotonica debole (MWTS)

Come i più attenti avranno notato, nell'elencare gli assiomi necessari per la semantica temporale debole abbiamo saltato un ipotetico assioma A2. Ebbene, ciò non è un caso: esiste infatti un'estensione della semantica WTS che aggiunge tra i propri requisiti il rispetto di tale assioma.

Si tratta della **semantica temporale monotonica debole** (*Monotonic WTS*, **MWTS**), e differisce dalla semantica WTS perché impone necessariamente che i tempi di scatto delle transizioni all'interno di una sequenza siano monotoni non decrescenti, forzando così il fatto che nell'intera rete **il tempo non possa tornare indietro**.
Più formalmente, la semantica introduce il seguente assioma:

- (**A2**) **Monotonicità dei tempi di scatto di una sequenza**: tutti i tempi di scatto di una sequenza di scatti devono essere ordinati nella sequenza in maniera **monotonicamente non decrescente** ($\geq$).
  Anche questo serve a garantire la proprietà intuitiva di **consistenza**, evitando cioè che il tempo torni indietro. Non richiedendo però che i tempi siano disposti in modo strettamente crescente ma ammettendo che nella sequenza lo **stesso tempo sia ripetuto** si lascia aperta la possibilità che nella rete più transizioni scattino in contemporanea, oppure che due transizioni scattino in tempi talmente ravvicinati che la granularità temporale del modello non è in grado di rilevare la differenza.

Le sequenze di scatti che soddisfano gli assiomi A1, A2 e A3 vengono dette **sequenze ammissibili in semantica monotonica debole**.
Sebbene sembri una differenza da nulla, imporre la monotonicità dei tempi di scatto ha in realtà ripercussioni piuttosto grandi: in una rete che segue la MWTS quando si analizzano gli scatti è necessario non solo fare un'analisi locale del preset e del tempo di abilitazione e scatto della transizione, ma anche assicurarsi che non ci sia nessuna transizione nella rete in grado di scattare prima. Si **perde cioè la caratteristica di località**, introducendo la necessità di mantenere un'informazione comune sull'**ultimo scatto** nella rete.

WTS $\equiv$ MWTS

Fortunatamente per noi esiste un teorema che afferma che *per ogni sequenza di scatti ammissibile in semantica debole \(S*{WTS}\) **esiste** una sequenza di scatti ammissibile in semantica monotonica debole S_{MWTS} **equivalente** ottenibile per semplice **permutazione** delle occorrenze degli scatti._

Non si tratterà di sequenze uguali, ma entrambe le sequenze produrranno la **stessa marcatura finale**. Questo è un enorme vantaggio, in quanto ciò ci permette di infischiarcene della monotonicità degli scatti durante l'analisi della rete, potendo così sfruttare la **località** e conseguentemente le **tecniche di analisi per le reti di Petri** (ad alto livello) già viste in precedenza.

Esempio di traduzione

Si prenda in esame la rete in figura:

Assumendo i timestamp iniziali di tutti i gettoni uguali a zero, si consideri la seguente sequenza ammissibile WTS di scatti:

$$\mathrm{T1\ scatta\ al\ tempo\ 12} \rightarrow \mathrm{T3\ scatta\ al\ tempo\ 14} \rightarrow \mathrm{T2\ scatta\ al\ tempo\ 4}$$

Tale sequenza non rispetta la monotonicità, in quanto T2 scatta "nel passato" dopo lo scatto di T3, e produce la marcatura $\langle 0, 0, 1, 0, 1 \rangle$. Tuttavia, riordinando la sequenza come:

$$\mathrm{T2\ scatta\ al\ tempo\ 4} \rightarrow \mathrm{T1\ scatta\ al\ tempo\ 12} \rightarrow \mathrm{T3\ scatta\ al\ tempo\ 14}$$

è possibile ottenere una marcatura identica ma con una sequenza che rispetta ora la monotonicità, essendo cioè ammissibile in semantica temporale monotonica debole.

Semantica temporale forte (STS)

Finora abbiamo lasciato aperta la possibilità che una transizione pur *potendo* scattare non lo facesse. Questa alternativa non è però contemplata in molti modelli temporizzati, in cui il **determinismo** gioca un forte ruolo: spesso si vuole che se una transizione può scattare, allora **deve** scattare entro il suo massimo tempo di scatto ammissibile.

Per forzare questo comportamento viene creata una semantica temporale apposita, che prende il nome di **semantica temporale forte** (*Strong Time Semantic*, **STS**): essa impone che *una transizione **deve scattare** ad un suo possibile tempo di scatto **a meno che non venga disabilitata** prima del proprio massimo tempo di scatto ammissibile*. Aggiungere quest'ultima

clausola permette alle transizioni di non dover prevedere il futuro: se esse fossero programmate per scattare in un certo istante ma prima di esso lo scatto di un'altra transizione le disabilitasse non si richiederebbe che esse tornino indietro nel tempo per scattare all'ultimo istante di tempo *utile*.

Essendo un ulteriore irrigidimento rispetto alla semantica temporale monotonica debole, la STS dovrà sia rispettare gli assiomi A1, A2 e A3, sia la seguente nuova coppia di assiomi, che porta il totale a cinque:

- (**A4**) **Marcatura forte iniziale**: il **massimo tempo di scatto** di tutte le transizioni abilitate nella **marcatura iniziale** dev'essere **maggiore o uguale del massimo timestamp** associato ad un gettone in tale marcatura.
  Questo assicura cioè che la marcatura iniziale sia **consistente con la nuova semantica** temporale: un gettone dotato di timestamp superiore al tempo di scatto massimo di una transizione abilitata non sarebbe potuto essere generato *prima* che la transizione scattasse (cosa che deve fare!), rendendo quindi la marcatura in questione non più quella iniziale.

- (**A5**) **Sequenza di scatti forte**: una sequenza di scatti ammissibile in semantica **MWTS** che parta da una **marcatura forte iniziale** è una **sequenza di scatti *forte*** se per ogni scatto il tempo di scatto della transizione **non è maggiore** del massimo tempo di scatto di un'altra transizione abilitata.
  Si sta cioè accertando che ogni transizione scatti entro il suo tempo massimo se non viene disabilitata prima da un altro scatto: per fare ciò, si permette alle transizioni di scattare *solo* se non ci sono altre transizioni abilitate che sarebbero già dovute scattare, costringendo quindi queste ultime a farlo per far continuare a evolvere la rete.

Ecco dunque che sequenze di scatto che soddisfano gli assiomi A1, A2, A3, A4 e A5 vengono dette **sequenze ammissibili in semantica forte**.

STS $\neq$ MWTS

In virtù dell'ultimo assioma si potrebbe pensare che esista un modo per trasformare ogni sequenza di scatti MWTS in una sequenza STS, realizzando così un'equivalenza. Purtroppo, però, non è così: una sequenza STS è sempre anche MWTS, ma **non è sempre vero il contrario**.

Poiché infatti non è più possibile a causa dell'assioma A2 riordinare le sequenze per ottenerne altre di equivalenti, è possibile trovare numerose sequenze che sono MWTS ma non STS. Riprendendo la rete già vista in precedenza e assumendo anche in questo caso dei timestamp iniziali nulli per i gettoni:

è facile vedere che la sequenza ammissibile in semantica monotonica debole:

$$\text{T2 scatta al tempo } 6 \rightarrow \text{T1 scatta al tempo } 12 \rightarrow \text{T3 scatta al tempo } 14$$

non è invece una sequenza ammissibile in semantica forte, in quanto lo scatto di T2 abilita la transizione T3, che dovrebbe quindi scattare entro il tempo 9 ($enab = 6$) ma non lo fa.

Semantica temporale mista

Può però capitare che dover imporre una semantica temporale fissa per l'intera rete si riveli limitante nella modellazione di sistemi reali: questi potrebbero infatti includere sia agenti deterministici (*es. computer*) che agenti stocastici (*es. esseri umani*).

Si introduce quindi una nuova **semantica temporale mista** (*Mixed Time Semantic*), in cui la semantica temporale debole (monotonica) o forte viene associata alle **singole transizioni** piuttosto che all'intera rete. In questo modo:

- le **transizioni forti** *dovranno* scattare entro il loro tempo massimo a meno che non vengano disabilitate prima;
- le **transizioni deboli** *potranno* scattare in uno qualunque dei loro possibili tempi di scatto.

Essendo meno comuni, solitamente sono le transizioni deboli ad essere esplicitamente indicate graficamente nelle reti con una W all'interno del rettangolo che le rappresenta: tutte le altre transizioni sono invece di default considerate forti.

Analisi di abilitazione in presenza di transizioni fortis

Introdotta quindi la possibilità che esistano all'interno delle reti delle transizioni forti che devono necessariamente scattare entro il loro tempo massimo di scatto non è ora più tanto semplice fare **analisi di abilitazione**, vale a dire quel tipo di analisi che cerca di tracciare su una linea temporale gli intervalli durante i quali certe transizioni sono abilitate.

Per capire perché, osserviamo la seguente rete Time Basic, che segue una semantica temporale mista:

Analizzando localmente le singole transizioni, come se avessero tutte semantica temporale debole, si può ottenere il seguente diagramma temporale di abilitazione:

Tuttavia, questo diagramma è **scorretto**. La presenza di una transizione forte che deve scattare entro il tempo 10, ovvero T2, non ci permette di dire nulla oltre tale tempo, in quanto *il suo scatto potrebbe disabilitare altre transizioni*. Questo era vero anche per la transizione debole T1, ma il suo essere debole permetteva comunque di ignorare tale eventualità nella prospettiva che la transizione, pur potendo, non scattasse: questo tipo di ragionamento non è però purtroppo più possibile in semantica temporale forte.

In sostanza, **le transizioni forti bloccano il nostro orizzonte temporale**.
Ecco dunque che il vero diagramma temporale di abilitazione della rete è il seguente:

Analisi delle reti Time Basic

Definite le reti Time Basic in ogni loro aspetto è dunque arrivato il momento di **analizzarle**.

Esattamente come le reti di Petri, infatti, le reti TB fanno parte di quei **linguaggi operazionali** utilizzati per illustrare il funzionamento di un sistema senza entrare nei dettagli della sua effettiva implementazione ($\neq$ *linguaggi dichiarativi/logici, che si usano invece per costruire il sistema a partire dalle proprietà richieste*). Visto questo ruolo, è necessario possedere una serie di **strumenti di analisi** specifici che permettano di "simulare" il funzionamento della rete per comprenderne l'evoluzione e le proprietà: ciò appare particolarmente evidente se si ricorda che le reti TB vengono spesso utilizzate per modellare sistemi *Hard Real-Time* in cui si deve avere la **certezza** del fatto che il sistema rispetterà tutta una serie di caratteristiche prima ancora di iniziare i lavori.

Reti TB come reti ad alto livello?

Ma è davvero necessario sviluppare un'intera serie di nuove tecniche di analisi specifiche per le reti Time Basic, o è possibile riutilizzare almeno in parte metodologie già discusse?

Si potrebbe infatti immaginare di considerare le reti TB come un tipo particolare di **reti ad Alto Livello** (ER). Come abbiamo già accennato, questo tipo di reti permettono infatti ai gettoni di avere un **qualunque contenuto informativo** e di definire le transizioni come una coppia **predicato-azione**: il predicato descrive la condizione di abilitazione della transizione in funzione dei valori dei gettoni nel preset, mentre l'azione determina che valore avranno i gettoni creati nel postset.
Volendo modellare il **tempo come concetto derivato**, ovvero non delineato esplicitamente ma che emerga comunque dal *funzionamento* della rete, si potrebbero quindi creare delle reti ad Alto Livello con le seguenti caratteristiche:

- **contenuto informativo dei gettoni**: un'unica variabile temporale **chronos** che contiene il timestamp della loro creazione;
- **predicati delle transizioni**: funzioni che controllano i timestamp dei gettoni nel preset e i propri tempi di scatto per determinare l'abilitazione o meno;
- **azioni delle transizioni**: generazione di gettoni dotati tutti dello *stesso dato temporale*, il quale è *non minore dei timestamp di tutti i gettoni nella tupla abilitante*.

Con queste accortezze è possibile riprodurre i timestamp e le regole di scatto di una rete TB. Come sappiamo bene, tuttavia, questo non basta per modellare il *concetto* di tempo: per avere un'espressività simile a quella delle reti Time Basic è infatti necessario anche il rispetto di una **semantica temporale** e, in particolare, di quella più stringente, ovvero la semantica temporale forte (STS).

Nelle reti ad Alto Livello bisognerà dunque far rispettare i cinque assiomi temporali perché la traduzione da reti TB a reti ER sia completa. Vediamo dunque come ciò potrebbe essere fatto:

- gli assiomi **A1** e **A3** sono sufficientemente semplici da modellare all'interno dei predicati delle transizioni, rendendo così la **semantica temporale debole** rappresentabile nelle reti ad Alto Livello;

- l'assioma **A2** è già più complesso da realizzare, in quanto richiede che lo scatto di una transizione generi dei gettoni con un timestamp maggiore di quello di tutti gli altri gettoni nella rete (*imponendo così che il tempo avanzi*). Tuttavia tale limite può essere aggirato con l'aggiunta di un **posto globale** contenente il **gettone dell'ultimo scatto** e aggiunto al preset di ogni transizione: in questo modo il gettone nel posto globale rappresenterà il *tempo corrente della rete* e imporrà che i nuovi gettoni generati abbiano timestamp maggiore di esso. In questo modo una rete ER può realizzare anche la **semantica temporale monotonica debole**;

- gli assiomi **A4** e **A5**, invece, si rivelano **estremamente problematici**: essi richiedono infatti che ciascuna transizione conosca il massimo tempo di scatto di tutte le altre transizioni per "decidere" se poter scattare oppure no. I predicati delle transizioni dovrebbero cioè avere in input l'intero stato della rete: sebbene questa cosa sia *teoricamente* realizzabile con l'aggiunta di un posto globale in ingresso e uscita da ogni transizione in cui un gettone contenga come *contenuto informativo l'intero stato della rete*, nella pratica cio è **irrealizzabile**.

Come si vede, dunque, la necessità di modellare la **semantica temporale forte** fa sì che **non si possa ridurre le reti TB a un caso particolare delle reti ER**, perdendo così anche la possibilità di utilizzare le tecniche di analisi già sviluppate per esse.

Reti Time Petri ad Alto Livello (HLTPN)

Appurato che non è possibile tracciare un'equivalenza tra le reti TB e le reti ER viene dunque introdotto un modello ancora più completo, che racchiuda al suo interno sia gli **aspetti funzionali** delle reti ad Alto Livello sia gli **aspetti temporali** delle reti Time Basic: si tratta delle **reti Time Petri ad Alto Livello** (*High-Level Time Petri Nets*, **HLTPN**).

Come appare chiaro dalla figura, all'interno delle reti HLTPN **gli aspetti funzionali possono dipendere da quelli temporali e viceversa**, espandendo così incredibilmente le capacità rappresentative del modello. Si tratta di reti ovviamente molto complesse, anche se a dire il vero gran parte della complessità giunge dall'analisi della componente temporale: se riusciremo ad analizzare le reti temporizzate potremo riuscire ad analizzare anche le reti HLTPN.

Analisi di raggiungibilità temporale

Rassegnatici dunque alla necessità di creare nuove tecniche di analisi specifiche per le reti temporizzate iniziamo a parlare di **analisi dinamica** a partire dall'**analisi di raggiungibilità**, ovvero la tecnica con cui nelle reti di Petri classiche eravamo in grado di **enumerare gli stati finiti raggiungibili**.

Provando ad adottare lo stesso approccio nei confronti delle reti TB ci si rende però subito conto di un enorme **problema**: le reti temporizzate hanno sempre **infiniti stati**, in quanto lo scatto di una singola transizione può produrre un'infinità di stati di arrivo che si differenziano unicamente per il timestamp dei gettoni generati. Sebbene la marcatura sia identica, le informazioni temporali legate ai gettoni sono differenti, distinguendo così ciascuno di tali stati della rete.
Bisogna inoltre considerare che per sua stessa natura il tempo avanza, rendendo così le reti temporizzate in grado di **evolvere all'infinito**: anche raggiungendo una marcatura che non abilita alcuna transizione, la rete continua ad evolvere in quanto il *tempo corrente* continua ad avanzare.

Dovendo costruire un **albero di raggiungibilità** questo sarebbe quindi sicuramente **infinito**, anche se in un modo diverso rispetto a quanto già visto per le reti non limitate: in quel caso infatti i gettoni non erano distinguibili, cosa che ci aveva permesso di raggrupparne un numero qualsiasi sotto il simbolo ω, mentre in questo caso le differenze nei timestamp impediscono un simile approccio.

Al contrario, per ottenere per le reti TB un'analisi simile all'analisi di raggiungibilità delle reti classiche è necessario **ridefinire** completamente il concetto di **stato raggiungibile**.

Stati simbolici

Per riformulare il concetto stesso di raggiungibilità partiamo da innanzitutto da quello di **marcatura**: nelle reti temporizzate queste associavano infatti a ciascun posto un *multiset* in cui ad ogni timestamp era associato il numero di gettoni con tale timestamp presente nel posto.

Per evitare la difficoltà di distinguere tra gettoni con timestamp diversi viene introdotto nelle reti TB il concetto di **stato simbolico**, un oggetto matematico che sostituendo ai timestamp specifici degli identificatori simbolici dei gettoni permette di *rappresentare un **insieme di possibili stati** con in comune lo stesso numero di gettoni in ciascun posto* (esattamente come la marcatura delle reti classiche).

Più formalmente, uno stato simbolico è una **tupla** $[\mu, C]$, dove:

- μ è la **marcatura simbolica**, che associa a ciascun posto un *multiset* di **identificatori simbolici** che rappresentano i timestamp dei gettoni in tale posto. Timestamp uguali saranno rappresentati dallo stesso simbolo, anche se si trovano in posti diversi: questo permette di mantenere l'**identità** tra timestamp;
- C è un sistema di **vincoli** (*constraint*), ovvero equazioni e disequazioni che rappresentano le relazioni tra gli identificatori simbolici dei gettoni. In questo modo è possibile mantenere le **relazioni** tra i timestamp dei gettoni pur non rappresentando esplicitamente il loro valore.

Un'esempio aiuterà a chiarire ogni dubbio. Immaginiamo di avere una rete TB con 3 posti $(P1, P2, P3)$, ciascuno con un solo gettone al loro interno, e la seguente marcatura iniziale: $\langle 0, 1, 0 \rangle$. Volendoci disfare dei timestamp espliciti dei gettoni, che tutto sommato ci interessano relativamente, dobbiamo **mantenere due informazioni**:

- che i gettoni in $P1$ e $P3$ hanno lo *stesso timestamp*;
- che il gettone in $P2$ ha *timestamp maggiore* di 1 del timestamp dei gettoni negli altri due posti.

Per fare ciò lo stato simbolico generato assegnerà lo stesso identificatore ai gettoni in $P1$ e $P3$ e espliciterà nei vincoli la relazione tra tempi. Otterremo dunque lo stato simbolico iniziale:

$$\mu(P1) = \tau_0, \quad \mu(P2) = \tau_1, \quad \mu(P3) = \tau_0$$

$$C_0 : \quad \tau_1 = \tau_0 + 1$$

Infine, ci si potrebbe accorgere che in realtà non ci interessa che il timestamp del gettone in $P2$ sia esattamente $\tau_0 + 1$, ma solamente che esso sia maggiore di τ_0. Ecco dunque che spesso si mutano i vincoli in **disequazioni**:

$$C_0 : \quad \tau_0 < \tau_1$$

Albero di raggiungibilità temporale

Utilizzando la definizione di stato simbolico appena vista è possibile costruire tramite un algoritmo un **albero di raggiungibilità temporale**, in cui gli stati distinguibili solo dai timestamp vengono condensati in stati simbolici che conservano però le *molteplicità* dei gettoni nei posti e le *relazioni* tra i timestamp.

Prima di fare ciò, però, è necessario rinnovare un'assunzione già fatta in precedenza: anche in questa analisi assumeremo che le funzioni temporali tf_t associate alle transizioni siano esprimibili come **intervalli con estremi inclusi** $[\mathbf{tmin_t}, \mathbf{tmax_t}]$, dove questi ultimi possono dipendere ovviamente dai timestamp dei token in ingresso nonché da tempi assoluti.

Fatte queste premesse, possiamo partire a costruire effettivamente l'albero di raggiungibilità temporale di una rete TB secondo il seguente algoritmo:

1. **Inizializzazione**: si trasforma la marcatura iniziale della rete in uno stato simbolico, introducendo una serie di vincoli che descrivano le (pre-)condizioni iniziali della rete. Tale stato viene poi trasformato in un nodo, diventando la radice dell'albero, e aggiunto alla lista dei nodi da esaminare;

2. **Scelta del prossimo nodo**: tra i nodi dell'albero non ancora esaminati si seleziona il prossimo nodo da ispezionare;

3. **Identificazione delle abilitazioni**: in base allo stato simbolico rappresentato dal nodo si individuano le transizioni abilitate al suo interno;

4. **Aggiornamento di marcatura e vincoli**: ciascuna transizione abilitata trovata viene fatta scattare generando un nuovo stato simbolico, che viene rappresentato nell'albero come nodo figlio del nodo considerato e aggiunto alla lista dei nodi da esaminare;

5. **Iterazione**: si ritorna al punto 2.

Di questo semplice algoritmo approfondiamo dunque le due fasi più interessanti: l'aggiornamento di marcatura e vincoli e l'identificazione delle abilitazioni.

Aggiornamento di marcatura e vincoli

Come si fa a **generare un nuovo stato simbolico** a partire dallo stato simbolico corrente quando scatta una transizione abilitata? Sostanzialmente il processo si divide in due fasi: la **creazione e distruzione di gettoni** e l'**espansione dei vincoli**.

Il primo step è abbastanza semplice: è sufficiente distruggere i gettoni e i relativi simboli nel preset della transizione e generare nuovi gettoni nel suo postset, questi ultimi identificati tutti

dallo **stesso nuovo identificatore simbolico**.

La generazione di nuovi vincoli è invece più complessa. Essa deve infatti tenere in considerazione quattro diversi aspetti:

- I **vecchi vincoli** devono continuare a valere: essi esprimono infatti relazioni tra gli identificatori temporali che non possono essere alterate dallo scatto di una transizione;
- Il **nuovo timestamp** deve avere il **valore massimo** nella rete: esso rappresenta infatti il tempo di scatto dell'ultima transizione scattata, e per monotonicità del tempo esso dovrà essere maggiore di tutti gli altri;
- Il **nuovo timestamp** dev'essere **compreso nell'intervallo** dei possibili tempi di scatto della transizione scattata;
- Il **nuovo timestamp** dev'essere **minore del massimo tempo di scatto** di tutte le **transizioni forti abilitate** il cui intervallo di scatto non sia nullo: per la semantica temporale forte, infatti, se così non fosse la transizione non potrebbe scattare prima che tale transizione forte scatti (cambiando anche potenzialmente l'insieme delle transizioni abilitate).

Tutte queste considerazioni devono essere condensate in un'**unica espressione logica**. Si dimostra quindi che *dato uno stato simbolico precedente avente vincoli* C_n*, detto* $maxT$ *il timestamp massimo all'interno della rete e* \(\tau{n+1}\) l'identificatore simbolico dei gettoni generati dalla transizione t è possibile definire i vincoli dello stato simbolico prodotto con la seguente **formula_**:

$$C_{n+1} = C_n \wedge \tau_{n+1} \geq maxT \wedge tmin_t \leq \tau_{n+1} \leq tmax_t \bigcap_{t_{STS}} (tmax_{t_{STS}} < tmin_{t_{STS}} \vee tmax_{t_{STS}} < maxT \vee tmax_{t_{STS}} \geq \tau_{n+1})$$

dove per t_{STS} si intende una qualunque **transizione forte** diversa da t; per **ciascuna** di esse bisognerà infatti aggiungere la condizione tra parentesi, relativa appunto alla semantica STS.

Tale catena di condizioni può ben presto diventare soverchiante, ma fortunatamente essa può essere semplificata sfruttando le **implicazioni** e le proprietà degli operatori logici. In particolare:

- se una condizione A implica un'altra condizione B con cui è in **AND** ($\wedge$), allora la condizione **implicata** B può essere cancellata;
- se una condizione A implica un'altra condizione B con cui è in **OR** ($\vee$), allora la condizione **implicante** A può essere cancellata.

Identificazione delle abilitazioni

Al contrario di quanto ci si potrebbe aspettare, però, la creazione di questo nuova catena di vincoli non è relegata alla sola creazione di un nuovo stato simbolico, ma è invece necessaria anche per **identificare le transizioni abilitate**. Avendo infatti introdotto degli identificatori simbolici che mascherano i timestamp dei gettoni, capire se una transizione sia abilitata o meno non è più così facile.

Tuttavia, è possibile dimostrare che *la **soddisfacibilità** del vincolo generato da un eventuale scatto della transizione **implica** la sua **abilitazione***: se esiste cioè un assegnamento di timestamp agli identificatori simbolici che **rende vero** il vincolo allora la transizione è abilitata e può scattare. Il motivo di ciò appare evidente quando ci si accorge che nella generazione del vincolo abbiamo già tenuto in conto di tutti gli aspetti che avremmo osservato per stabilire se la transizione fosse abilitata o meno.

Proprio riguardo la soddisfacibilità viene poi fatta una distinzione a livello grafico nell'albero: essendo gli stati simbolici *insiemi* di marcature è possibile che una transizione sia abilitata in alcune di esse mentre è disabilitata in altre.
Quando questo succede, lo stato generato dalla transizione **potrebbe essere uno stato finale**, in quanto potrebbe aver disabilitato tutte le transizioni: ciò si comunica graficamente con un **nodo circolare**, mentre i nodi (e quindi gli stati) i cui vincoli sono **sempre soddisfacibili** si indicano con dei **nodo rettangolari**.

Alcuni operano poi una distinzione sulle frecce che collegano i nodi dell'albero: una freccia con punta nera indica che la transizione è sempre possibile, mentre una freccia con alla base un pallino bianco indica che per rendere possibile la transizione è stato violato qualche parte del vincolo, per cui non è detto che la transizione sia possibile in nessuna delle marcature rappresentate dallo stato simbolico.

Proprietà *bounded*

Eseguendo l'algoritmo a mano per un paio di iterazioni ci si rende ben presto conto di una cosa: il processo **non termina**!

Questo è dovuto al fatto che **non avendo una forma normale** per i vincoli che permetta di confrontare tra di loro gli stati simbolici non è possibile stabilire se si sia già visitato o meno uno stato: i vincoli si allungano così sempre di più, creando sempre stati simbolici nuovi (almeno sulla carta).

Si ottiene cioè un **albero infinito**. Nonostante ciò, tale albero è comunque particolarmente utile perché permette di **verificare proprietà entro un certo limite di tempo**: si parla per

esempio di **bounded liveness** e **bounded invariance**, delle caratteristiche molto preziose sopratutto per lo studio dei sistemi Hard Real-Time.

Dall'albero al grafo aciclico

Esattamente come per le reti di Petri classiche, costruito l'albero di raggiungibilità ci piacerebbe ristrutturarlo per trasformarlo in un **grafo** che illustri più concisamente l'evoluzione del sistema rappresentato dalla rete. Di certo non possiamo sperare di ottenere un *grafo ciclico* in quanto per sua stessa natura *il tempo non può tornare indietro*, ma c'è qualche chance di ottenere un **grafo aciclico**?

Abbiamo già detto che a causa di come sono costruiti i nuovi stati simbolici è impossibile riottenere più volte lo stesso esatto stato. Ammettendo tuttavia di **dimenticare la storia** di come si è giunti in un certo nodo (ovvero l'insieme di transizioni che hanno portato ad esso) si potrebbe sperare di **ritrovare alcuni stati** che pur caratterizzati da storie diverse possiedono la *stessa marcatura simbolica* e lo *stesso insieme di vincoli sugli identificatori simbolici* presenti al suo interno.

Vediamo dunque una serie di tecniche che permettono, al costo di **perdere una serie di informazioni**, di individuare le *somiglianze* tra diversi stati simbolici in modo da raggrupparli in una serie di "super-stati" che fungano da nodi per il grafo che intendiamo costruire.

Semplificare i vincoli: l'algoritmo di Floyd

Per dimenticare la storia di come si è giunti in un certo stato simbolico è innanzitutto necessario **semplificare i vincoli**: come abbiamo visto nella formula precedente, ogni nuovo stato simbolico ereditava infatti i vincoli del precedente, cosa che permette di distinguere marcature identiche a cui si è giunti in modo diverso.

È dunque necessario **esprimere i vincoli solo in termini della marcatura corrente**: possiamo infatti considerare i vincoli sugli identificatori simbolici non più presenti nella rete come sostanzialmente inutili. Tuttavia, non basta cancellarli per risolvere la questione: sebbene il simbolo a cui fanno riferimento sia scomparso, essi potrebbero ancora esprimere **vincoli indiretti** sugli identificatori ancora presenti nella marcatura, che vanno ovviamente mantenuti. \ Si immagini per esempio di avere i vincoli $B - A \leq 5$ e $C - B \leq 6$ e che l'identificatore B sia ormai scomparso dalla rete. Sebbene si riferiscano a un simbolo ormai non più presente, tali vincoli contengono ancora delle informazioni: sommando le due disequazioni membro a membro si ottiene infatti che $C - A \leq 11$, un vincolo su variabili presenti che era espresso *indirettamente*.

Per rimappare in modo semplice gli effetti dei vincoli sulle variabili non più presenti nella marcatura su quelle presenti si utilizza spesso l'**algoritmo di Floyd**, che funziona come segue:

1. Si riconducono tutti i vincoli alla **forma $\mathbf{A - B \leq k}$**, dove A e B sono identificatori simbolici e k è una costante numerica; per esempio:

$$A + 2 \leq B \leq A + 5 \quad \longrightarrow \quad A - B \leq -2 \text{ e } B - A \leq 5$$

$$B \leq C \leq B + 6 \quad \longrightarrow \quad B - C \leq 0 \text{ e } C - B \leq 6$$

2. Si costruisce una **matrice** in cui ad ogni riga e colonna corrisponde un identificatore simbolico e l'intersezione tra la riga A e la colonna B corrisponde al **valore $\mathbf{k}$ tale per cui $\mathbf{A - B \leq k}$** in base ai vincoli ricavati al punto precedente, mentre per i valori non noti si scrive semplicemente un **punto di domanda**;

3. Si **riempiono tutti i posti contrassegnati da punti di domanda** utilizzando la seguente formula:

$$\boxed{m[i, j] = m[i, k] + m[k, j]}$$

   che mima la somma membro a membro delle disequazioni che rappresentano i vincoli. In questo modo è possibile scoprire i **vincoli indiretti** tra variabili;

4. Si **esplicitano i vincoli indiretti** relativi agli identificatori simbolici presenti nella marcatura corrente e **si eliminano** i vincoli che contengono gli identificatori non inclusi.

Applicato l'algoritmo di Floyd, ottenuti i vincoli impliciti ed eliminati i vincoli contenenti identificatori simbolici è possibile semplificare di molto l'insieme dei vincoli di nodi del grafo, identificando anche le prime *somiglianze* tra nodi.

Inclusione tra stati

Il raggruppamento offerto dalla semplificazione dei vincoli tramite l'algoritmo di Floyd non è però sufficiente a ottenere grafi aciclici soddisfacenti. Si consideri per esempio la rete in figura:

Come si vede, essa genera una serie di nodi nel grafo di raggiungibilità tutti diversi nonostante essi abbiano la **stessa marcatura** e l'unica differenza sia data dalla costante nel vincolo. Per semplificare situazioni come queste viene introdotto il concetto di **inclusione** (o *contenimento*) **tra stati**: sapendo infatti che gli stati simbolici rappresentano *insiemi* di marcature è opportuno chiedersi se alcuni di essi possano essere *sottoinsiemi* di altri.

Ecco dunque che si dice che *uno stato A è **contenuto** in un altro stato B se e solo se **tutte** le marcature rappresentate da A sono rappresentate anche da B*. Ciò avviene quando:

- A e B hanno lo **stesso assegnamento di timestamp**;
- **i vincoli di A implicano i vincoli di B**, ovvero $C_A \rightarrow C_B$.

Decidiamo quindi di rappresentare nel grafo **solo gli stati contenuti**, introducendo però una distinzione grafica: la punta bianca della freccia indica che spostandosi lungo di essa si arriva ad un **sottoinsieme** del "super-stato" di arrivo, ovvero uno stato avente *vincoli più stringenti* di quelli mostrati.

Tempi relativi

Osservando l'evoluzione di una rete Time Basic ci si può poi accorgere di un ulteriore fatto: se le funzioni temporali delle transizioni **non fanno riferimento a tempi assoluti**, ovvero a specifici istanti di esecuzione della rete, per comprendere come la rete può evolvere a partire da una certa marcatura è **sufficiente osservare i vincoli relativi tra i timestamp**.

Si prenda per esempio in considerazione la rete in figura:

ci si accorge che mantenere il riferimento ai tempi assoluti 0 e 10 introdotti dai vincoli iniziali farebbe sì che vengano generati **infiniti stati** anche considerando la possibilità di inclusione. Poiché *l'unica transizione presente nella rete non fa alcun riferimento a tempi assoluti*, si può quindi **eliminare i vincoli legati a tempi assoluti** ottenendo il secondo grafo in figura: esso rappresenta alla perfezione l'evoluzione della rete (che può far scattare la transizione $T1$ un numero infinito di volte) pur ignorando i vincoli sul valore iniziale del timestamp dell'unico gettone presente.

Tempi anonimi

Si può infine introdurre un'ulteriore astrazione: ci si rende infatti conto che *se il timestamp associato ad un gettone in una marcatura M non verrà **mai più utilizzato** per stabilire come evolverà la rete a partire da quella marcatura*, allora è possibile **anonimizzare** il tempo di tale gettone. L'identificatore simbolico del gettone viene cioè sostituito da un **identificatore anonimo** τ_A e i vincoli che lo coinvolgono cancellati: questo permette di riconoscere la somiglianza tra stati simbolici che, pur diversi a livello di timestamp dei gettoni, **evolvono nello stesso modo**.

Si consideri per esempio la rete in figura:

All'interno di questa rete, il timestamp del gettone in $P2$ non ha alcuna influenza sull'evoluzione della rete: esso funge infatti unicamente da *zero relativo* per determinare il timestamp del gettone in $P1$, che sarà maggiore di esso di tante unità quanto il numero di volte che è scattata la transizione $T1$. Il gettone può dunque essere reso anonimo, eliminando l'unico vincolo che, a conti fatti, non aggiunge nulla alla nostra conoscenza della rete.

Non esiste una vera e propria regola formale per capire quali gettoni siano anonimizzabili, ma esistono una serie di **euristiche** che possono suggerire tale eventualità: così, per esempio, è molto probabile che i **gettoni morti** (gettoni in posti che non appartengono al preset di alcuna transizione) possano essere resi anonimi.

Conclusioni

L'utilizzo delle tecniche di **raggruppamento degli stati simbolici** appena viste permette di costruire dei grafi di raggiungibilità più coincisi per le reti Time Basic, ma non senza **sacrificare** una serie di **informazioni**. Infatti:

- la tecnica di **inclusione** introduce la possibilità che nel grafo esistano dei **cammini non percorribili** dovuti al fatto che muovendosi tra *sottoinsiemi* degli stati rappresentati è possibile che lo stato simbolico reale in cui ci si trova non permetta una certa transizione che è invece possibile a parte degli stati rappresentati dal nodo;
- con l'abolizione dei **vincoli assoluti** si perdono informazioni sulle **relazioni precise** tra gli stati (anche se è possibile arricchire le informazioni sugli archi per non perderne troppe);
- **anonimizzando** alcuni gettoni potrebbe non essere sempre possibile verificare la **raggiungibilità** di una marcatura definita da vincoli sui timestamp.

Si tratta di un equo prezzo da pagare per una rappresentazione semplice ed efficace dell'evoluzione della rete.

Albero di copertura temporale?

Avevamo detto che sulle reti TB non era possibile utilizzare la tecnica di analisi di copertura vista per le normali reti di Petri a causa della **distinguibilità** tra gettoni dovuta ai rispettivi timestamp.

Tuttavia, l'introduzione della possibilità di **anonimizzare i gettoni** è in grado di far riconsiderare tale conclusione: i gettoni anonimi sono infatti tutti **equivalenti** e indistinguibili, motivo per cui potrebbero essere rappresentati globalmente solo dal loro **numero** $\omega_{\tau A}$.

Non approfondiamo la questione, ma esiste un'ipotesi non dimostrata che suppone che *le reti TB non limitate siano non limitate **solo** sul numero di gettoni anonimi* in quanto in caso contrario

bisognerebbe avere una rete che si interessi del passato all’infinito.