Appunti di Ingegneria del Software

Indice

Introduzione

Cos'è l'ingegneria del software

Possiamo definire ingegneria del software un approccio sistematico allo sviluppo, alla mesa in opera e alla manutenzione del software, secondo criteri ingegneristici. Sono dei metodi tecnici e manageriali per prevedere e tenere sotto controllo i costi per tutto il *lifecycle* dei prodotti software. Come tutte le ingegnerie, fornisce metodi una guida per applicare la conoscenza scientifica allo sviluppo di soluzioni software *cost-effective* per risolvere problemi pratici. L'ingegneria del software studia sistemi complessi di grandi dimensioni nati dal lavoro di gruppo di diversi anni, soggetti a frequenti modifiche ed esistono in diverse versioni. In breve, si studiano i metodi da usare perché il processo porti allo sviluppo di prodotti di qualità.

Per software non intendiamo solo l'eseguibile o il codice sorgente, ma tutti gli artefatti legati ad un particolare prodotto:

- Sorgente e *build environment* e i test
- Eseguibile
- Documentazione utente e tecnica
- Documenti di progetto

Saper programmare, purtroppo, non basta per essere un buon ingegnere del software:

- Un programmatore sviluppa un programma completo partendo da specifiche fornite da altri e lavora individualmente;
- Un ingegnere del software analizza i problemi e i domini applicativi, coglie i requisiti, sviluppa specifiche progetta componenti *riutilizzabili* e lavora in (e talvolta coordina) un gruppo.

Per progettazione intendiamo la scomposizione di un problema in sotto-problemi che possono essere risolti indipendentemente con l'obiettivo di governare la complessità e rendere efficiente il processo.

Il processo di sviluppo del software

Con la nascita dell'ingegneria del software si è sviluppato un nuovo ciclo di vita del software detto a cascata che sostituiva quello non strutturato adottato in precedenza (*code and fix*). Il modello a cascata identifica fasi e attività predefinite, forza una progressione lineare tra una fase e la successiva attraverso decisioni *go/no-go* e non torna sui suoi passi. I passi del modello a cascata sono i seguenti (Royce, 1970):

1. **Studio di fattibilità:** stabilire se il progetto si debba fare e analizzare il rapporto costi-benefici. Esso produce uno "studio di fattibilità", un documento che contiene una descrizione preliminare del problema con alcuni scenari, le relative soluzioni e i costi e i tempi previsti per ogni alternativa
2. **Analisi e specifica dei requisiti:** analizzare il dominio in cui l'applicazione si trova, identificarne i requisiti e derivare una specifica per il software. Viene prodotto un altro documento: l;analisi e la specifica dei requisiti.
3. **Progettazione:** definire l'architettura del software, i vari componenti e le loro relazioni. Lo scopo e supportare lo sviluppo concorrente, definendo le diverse responsabilità per le diverse parti. Viene prodotto il documento "progetto del sistema""
4. **Implementazione e test di unità:** ogni modulo elementare viene implementato mediante il prescelto linguaggio di programmazione e viene testato *durante* lo sviluppo.
5. **Integrazione e testi di sistema:** i moduli sono collegati tra di loro e viene testata la loro interazione. Il sistema completo viene testato alla fine per verificare proprietà complessive.
6. **Messa in opera:** distribuire l'applicazione e gestire le diverse istallazioni e configurazioni presso i clienti
7. **Manutenzione:** copre i cambiamenti successivi alla fine del processo di sviluppo.

Le correzioni sono necessarie durante l'evoluzione del software, ma il loro costo aumenta in base a quante decisioni (quanto tardi nella catena) sono state eseguite. Perciò è meglio anticipare il cambiamento e non subirlo. Il ciclo di vita a cascata non offre un buon punto di partenza. In soccorso viene il modello *agile*: si procede in maniera incrementale, per successivi rilanci. Esiste in diverse forme, le più note *SCRUM* e *Extreme Programming*.

SCRUM

Si parte dal presupposto che la specifica cambi ed evolva. Si utilizzano dei team di 5/8 persone e lo sviluppo avviene a incrementi di 10/15 giorni. Durante "sprint" produttivo il software compie tutto il suo ciclo e viene prodotta una versione completa potenzialmente vendibile dell'applicativo. Successive fasi raffinano il lavoro prodotto dalle fasi precedenti. Durante uno sprint non possono avvenire cambiamenti alla specifica del progetto. Perciò la durata degli sprint è decisa sulla base di quanto a lungo si vuole evitare cambiamenti.

All'interno di ogni team vi sono diversi ruoli:

- **SCRUM Master:** il responsabile del processo;
- **Product Owner:** dialoga con l'esterno (utenti/contrattuario);
- **Team:** il gruppo di circa 7 persone responsabili delle varie fasi di sviluppo.

Extreme Programming (XP)

Si basa sull'analogia dell'extreme climbing. Ha per obiettivo la risposta ai requisiti stringenti di ridotto "time to market". Si basa sulla premessa che il software è impossibile da pre-specificare ed è un po' come guidare: anche se si ha in mente un piano, occorre un continuo aggiustamento di traiettoria. Come SCRUM, rivendica la centralità del codice rispetto ad altri

artefatti. Integra, inoltre, il testing nel processo come attività centrale (TDD). Sostiene l'utilizzo del *pair programming* (un programmatore scrive e l'altro guarda) e prevede un *refactoring* ed un evoluzione continua del codice.

DevOps

Una fase critica nello sviluppo è la messa in opera, o deployment. Solitamente il team si divide in:

1. **Devs:** si concentrano su design, implementazione, manutenzione, testing e rilascio del prodotto
2. **Ops:** sono addetti alla messa in produzione; essi installano il prodotto e offrono servizi e garanzie di funzionamento e stabilità

Per rendere più efficiente la correzione di errori, la cultura DevOps unisce le due figure in un'unica figura dove i devs devono indossare i panni degli ops e viceversa. Ciò significa una integrazione di *continuous testing and monitoring* con il processo di sviluppo che diventa collaborativo. Ciò permette la creazione di cicli di sviluppo sempre più corti con cambiamenti piccoli che viene automaticamente messo in produzione e monitorato, creando un ciclo continuo che unisce lo sviluppo del software con la monitoraggio del funzionamento.

Programmazione a oggetti (OOP)

L'idea che ha portato allo sviluppo del paradigma a oggetti è stato il riuso. Creare piccoli componenti riutilizzabili permette la riduzione dei tempi di sviluppo, minore manutenzione, maggiore robustezza, affidabilità ed efficienza. Il riuso, però può causare anche dei problemi:

- Motivi tecnici: i moduli per essere riutilizzabili devono essere adattabili, andando a creare un compromesso con il grado di ottimizzazione di ogni componente (si ottimizza per il particolare, non per il generale)
- Motivi non tecnici:
  - Paura ad affidarsi al codice scritto da altri
  - Paura di perdere efficienza (compromesso con l'adattabilità)
  - Tendenza a focalizzarsi su progetti di breve durata non ben finanziati
  - Timore di non riuscire a gestire una miriade di componenti diversi

Un sistema, per essere riutilizzabile, deve essere modulare. Un sistema è modulare se è diviso in parti che hanno una sostanziale autonomia individuale ed una ridotta interazione con le altre parti. In generale, con il termine "modulo" intenderemo un fornitore di risorse computazionali: procedure, strutture dati, tipi ecc. Per un modulo è importante distinguere cosa fa, ossia l'insieme dei servizi esportati, e come è fatto, ossia le peculiarità interne al modulo. L'interfaccia di un modulo costituisce il *contratto* tra il modulo e i suoi utilizzatori.

L'incapsulamento e ADT

L'approccio alla modularità tradizionale è quello di scomposizione funzionale *top-down*: si scompone ricorsivamente la funzionalità principale del sistema da sviluppare in funzionalità più semplici e si termina quando le funzionalità individuate sono così semplici da permettere una diretta implementazione. Un modulo secondo questo è approccio è definito come una procedura/libreria. È un modo di procedere molto ordinato e logico che però è adatto a progettare algoritmi ma non sistemi di grosse dimensioni:

1. Spesso un sistema non possiede una singola funzionalità principale
2. Le funzionalità di un programma sono soggette a frequenti cambiamenti
3. Con l'approccio *top-down* si decidono prematuramente i vincoli tra i diversi moduli.

Dove l'approccio tradizionale usava delle procedure che accedevano ad un pool di dati comune, la scomposizione in oggetti usa due astrazioni: l'incapsulamento e il tipo di dato astratto (ADT).

Per incapsulamento intendiamo la capacità di ogni dato di esporre un insieme di operazioni valide, un'interfaccia pubblica immutabile, e mantenere privata la mutevole implementazione di queste. Definiamo, allora, tipo di dato astratto (ADT) l'astrazione che classifica i dati in base al loro comportamento e non in base alla loro rappresentazione fisica. L'interfaccia pubblica sarà l'unico modo per gli utilizzatori del dato per modificarlo (*information hiding*).

Concetti di base

Oggetto

La OOP si basa, come lascia intendere il nome, sul concetto di oggetto. Per oggetto intendiamo un'astrazione o, in generale, un'entità con un significato ben preciso nel dominio applicativo. Nel nostro contesto, un oggetto è un'istanza di un ADT. Gli oggetti avranno 3 componenti:

1. L'identità: discrimina le diverse istanze dello stesso concetto: due istanze possono essere uguali (stessi dati) ma non identici (stesso oggetto);
2. Lo stato: definito da un insieme di attributi mutabili o immutabili;
3. Il comportamento: definito dall'insieme di operazioni o comandi impartibili ad un oggetto, chiamati metodi o messaggi.

Un oggetto può essere composto a sua volta da altri oggetti.

Incapsulamento

Come visto precedentemente, intendiamo la capacità di definire proprietà pubbliche (metodi) e private (attributi). Le proprietà private sono visibili e modificabili solamente attraverso apposite proprietà pubbliche. Dal punto di vista dell'utilizzatore l'oggetto deve essere una scatola nera utilizzabile solo attraverso le apposite interfacce.

Classe

Una classe è un gruppo di entità identificato da un insieme di caratteristiche comuni. Una classe è il modello per creare oggetti con stesso comportamento: ogni oggetto è infatti una istanza di una classe. Ogni classe deve catturare una e una sola astrazione. Esse consentono la definizione di tipi di dato astratto.

Ereditarietà e polimorfismo

Le classi possono essere legate tra di loro da diversi tipi di relazioni:

1. **Ereditarietà**
2. **Aggregazione:** una classe contiene un'altra classe (il volante fa parte dell'automobile);
3. **Uso:** una classe chiama metodi di un altro classe (una persona usa una macchina).

Tra questi l'ereditarietà è il più importante e caratterizza il paradigma OOP.

Diciamo che una sottoclasse è figlia di una o più superclassi "padre" se è una specializzazione dell'ultima. Le sottoclassi ereditano tutte le proprietà delle superclassi e possono modificarle o aggiungerne altre. Il motivo dell'importanza dell'ereditarietà è la possibilità degli oggetti di compiere polimorfismo, ossia mutare il proprio tipo. Infatti una variabile ha un tipo statico noto a *compile time* che definisce le caratteristiche generali e un tipo dinamico noto solo a runtime che definisce il tipo specifico dell'oggetto a cui la variabile si riferisce in un certo istante. Attenzione però: tipo statico e dinamico non sono slegati! Infatti il tipo dinamico deve essere sempre una specializzazione di quello statico: una variabile polimorfica può solo specializzarsi, non generalizzarsi. Ad esempio a priori sapremo che una variabile è di tipo `Macchina` e ne avrà gli attributi, ma a runtime una variabile di tipo `Macchina` può contenere sia `MacchinaElettrica` che `MacchinaBenzina` ma non `Aereo`. Il vero tipo della variabile verrà finalizzato solo all'accesso di una delle sue proprietà, ad esempio quando viene mandato il messaggio `parti`. Solo allora si potrà decidere se usare l'implementazione fornita da `Macchina`, `MacchinaElettrica` o `MacchinaBenzina`. Il polimorfismo implica la necessità di un lookup dinamico dei metodi e degli attributi da parte del compilatore, compito che nel paradigma procedurale era delegato al programmatore.

UML

Il modello è una descrizione astratta del sistema. Modellare il sistema aiuta gli analisti a capire le funzionalità del sistema. Diversi modelli presentano il sistema da diverse prospettive. UML è un linguaggi grafico di modellazione usato per il software. Sono definiti 3 tipi di diagrammi:

1. Diagrammi di struttura: mostrano come i componenti sono strutturati internamente
2. Diagrammi di comportamento: mostrano come i componenti si comportano evolvono nel tempo
3. Diagrammi di interazione: mostrano come i vari componenti interagiscono tra di loro

Diagrammi di casi d'uso

Sono diagrammi di comportamento che descrivono i requisiti del sistema. Descrivono quali funzionalità devono essere offerte e chi le usa, senza affrontare la descrizione di come le funzionalità sono realizzate. I vari componenti possono interagire tra di loro con relazioni di "estensione" ed "inclusione".

```
@startuml
skinparam style strictuml
left to right direction

actor "Research associate" as associate
actor "Professor" as prof
actor "Assistant" as assistant

package "Student administration" {
  usecase "Query student data" as query
  usecase "Create course" as course
  usecase "Issue certificate" as cert
  usecase "Publish task" as task
}

prof -|> associate
assistant -|> associate

associate -- query
professor -- course
professor -- cert
assistant "0..1" -- cert
assistant -- task

@enduml
```

Diagrammi di classe

I diagrammi delle classi è una descrizione statica delle classi del nostro sistema. Una classe è composta da tre parti:

1. Nome
2. Attributi: lo stato; la sintassi per definire un attributo è la seguente:

   ```
   visibilita' ::= - | + | ~ | #
   attributo   ::= visibilita' nome: tipo [molteplicita'] = default {proprieta'}
   ```

   Gli attributi statici (condivisi da tutte le istanze) sono sottolineati.
3. Metodi: il comportamento; la sintassi per definire un metodo è la seguente:

   ```
   visibilita' ::= - | + | ~ | #
   direzione   ::= out | in
   parametro   ::= direzione nome: tipo = default
   metodo      ::= visibilita' nome(parametro+): tipo {proprieta'}
   ```

Attributi e metodi hanno quattro impostazioni di visibilità: + pubblica, - privata, ~ friendly (pubblico ma solo per il modulo),# protected (come friendly ma anche per i sottotipi).

La rappresentazione grafica è la seguente:

```
@startuml
class Persona {
  - nome: String
  - cognome: String
  - dataNascita: Date
```

```
  - {static} numPersone: int
  + siSposa(p: Persona) : boolean
  + compieAnni(d: Date) : boolean
}
@enduml
```

Fra le classi sono definite delle associazioni. Una associazione è una relazione tra le classi con un nome (solitamente un verbo) in cui sono specificati i ruoli svolti dalle classi nell'associazione. Gli estremi di un associazione sono *attributi impliciti*, hanno una visibilità come gli attributi normali e possono avere anche molteplicità. Le associazioni possono essere bidirezionali (entrambi i membri guadagnano l'attributo implicito) oppure avere una direzione (solamente dall'origine vi è l'attributo implicito).

```
@startuml
hide empty members
class Persona {
  - nome: String
  - cognome: String
  - anni: int
}

class Azienda {
  - nome: String
  - fatturato: int
  - numDipendenti: int
  + assume(p: Persona): int
}

class Banca

Persona "-dipendenti 0..*" -- "-azienda 1" Azienda : lavora
Persona "superiore 0..1" --- "subalterni 0..*" Persona : dirige
Banca "banche 0..*" --> "clienti 0..*" Persona
@enduml
```

Possiamo definire anche le classi associazione, ossia delle classi che definiscono delle proprietà dell'associazione tra due classi.

```
@startuml
left to right direction
hide empty members

class Studente
class Corso

class DatiCorso {
  - voto: int
  - frequenza: int
  + votoMassimo(): int
}

Studente "1..*" -- "1..*" Corso
(Studente, Corso) -- DatiCorso
@enduml
```

Esistono alcune associazioni particolari che vengono identificate in modo particolare:

1. Aggregazione: definisce una relazione *part-of*;
2. Composizione: è una aggregazione forte, ossia le parti non esistono senza il contenitore;
3. Ereditarietà: qui si intende l'ereditarietà generalizzata.

```
@startuml
hide empty members

class Automobile
class Telaio
class Motore
class Ruota

Telaio "1" --o "1" Automobile
Motore "1" --o "1" Automobile
Ruota  "4" --o "1" Automobile
@enduml
```

```
@startuml
hide empty members

class Ruota
class Automobile
class Bicicletta

Automobile *-right-> "4" Ruota
Bicicletta *-left->  "2" Ruota
@enduml
```

```
@startuml
left to right direction
hide empty members

class Child extends Mother
class Child extends Father
class Grandchild extends Child
@enduml
```

Le classi possono essere astratte (senza istanze) e vengono indicate tramite il corsivo nel nome o la proprietà `{abstract}`. Le classi astratte possono contenere metodi astratti, ossia metodi che le classi figlie devono implementare. Se una classe ha almeno un metodo astratto essa è per forza astratta. Le interfacce sono simili alle classi astratte, ma hanno la restrizione di avere solo funzioni astratte e costanti. In Java, una classe può ereditare da solo una classe astratta ma può implementare infinite interfacce.

```
@startuml
hide empty members

class Printer

interface Printable {
  + print(): void
}

class Course implements Printable {
  + name: String
  + hours: int
  + print(): void
  + getCredits(): float
}

abstract class Person implements Printable {
  + name: String
  + address: String
```

```
  + dob: Date
  + ssNo: int
  + print(): void
}

class Employee extends Person {
  + acctNo: int
}

class Student extends Person {
  + matNo: int
  + print(): void
}

Printer .left> Printable : <<uses>>
@enduml
```

In alcuni casi, possiamo usare anche una notazione alternativa (detta *lollipop*) per quanto riguarda le interfacce.

```
@startuml
class fig2 as "FiguraGeometrica" {
  - vertici: List
}

class vec2 as "Vector" {
  + add(o: Object): boolean
  + det(index: int): Object
}

together {
  class fig1 as "FiguraGeometrica" {
    - vertici: List
  }

  interface List {
    + add(o: Object): boolean
    + det(index: int): Object
  }

  class vec1 as "Vector" {
    + add(o: Object): boolean
    + det(index: int): Object
  }
}


fig1 .> List : <<uses>>
List <|. vec1
List -[hidden]--> fig2
fig2 -right(0- vec2 : List
note "Eqivalenti" as n1

List .. n1
n1 .. fig2
@enduml
```

I pacchetti sono un meccanismo di strutturazione simile ad uno spazio di nomi. Sono un'associazione grafica di più classi e godono della proprietà di ereditarietà.

Diagrammi di sequenza

È un diagramma di iterazione utilizzato per rappresentare gli scenari in termini di entità e messaggi scambiati.

Il diagramma si suddivide in sequenze che contengono attori che interagiscono con il sistema. Lungo l'asse delle ascisse vengono rappresentati gli attori, mentre lungo l'asse delle ordinate il tempo. I messaggi sono indicati da delle frecce parallele all'asse delle ascisse. È possibile anche rappresentare la durata delle chiamate e la durata di vita degli oggetti. Ogni sequenza, o frame di interazione, può essere ripetuta oppure può essere eseguita condizionatamente.

```
@startuml
skinparam style strictuml
autoactivate on

-> Autonoleggio: prenota
loop per ogni auto ordinata
  opt se il cliente è registrato
    Autonoleggio -> Cliente : preferenze
    Cliente --> Autonoleggio
  end
  Autonoleggio -> Sede : disponibilità

  alt se auto è disponibile
    Sede -> Prenotazione **
    Prenotazione -> Autovettura : associa
    Autovettura --> Sede : associamento
    Sede --> Autonoleggio
  else
    Sede --> Autonoleggio : non disponibile
  end
end
@enduml
```

Macchine a stati finiti

In questo contesto, esse rappresentano il comportamento dei singoli oggetti di una classe in termini di eventi a cui gli oggetti sono sensibili, azioni prodotte e transizioni di stato. Le condizioni di transizione l'emissione di un evento oppure delle condizioni, funzioni booleane sui valori degli oggetti.

```
@startuml
hide empty description

[*] --> Vuoto : crea
Vuoto --> Elaborazione : aggiungi destinatario

Elaborazione --> Elaborazione : aggiungi testo
Elaborazione --> Bozza : memorizza
Elaborazione --> Cancellato : cancella
Elaborazione --> Spedito : spedisci [rete.stato = disponibile]
Elaborazione --> Pronto : spedisci [rete.stato != disponibile]

Bozza --> Elaborazione : seleziona

Pronto --> Pronto : timeout [rete.stato != disponibile]
Pronto --> Spedito : timeout [rete.stato = disponibile]

Cancellato --> [*]
Spedito --> [*]
@enduml
```

Ogni stato può contenere attributi ed eseguire attività o emettere eventi in base al punto nel ciclo di vita, allo stato di entrata o a quello di uscita.

```
@startuml
state Nome : attributo: tipo = valore_iniziale
Nome : do / attività0
Nome : entry / attività1
Nome : exit / attività2
Nome : event1 / attività3

Nome --> [*] : event2 [condizione1] / attività4
Nome --> Nome : event3 [condizione2] / attività5
@enduml
```

Uno stato può anche essere composto da vari sotto-stati. In questo caso viene detto macro-stato. I sotto-stati ereditano le transizioni dei loro super-stati. I macro-stati possono rappresentare 2 modelli di esecuzione:

1. Esecuzione in sequenza (decomposizione `OR`): solo uno dei sotto-stati può essere attivi in un certo istante
2. Esecuzione in parallelo (decomposizione `AND`): sono uno stato per branch del macro-stato può essere attivo in un certo istante

```
@startuml
state Warning : entry/visualizza
state Attesa
state Memorizzazione {
   state Preparazione
   state Salvataggio : do/memorizza
   state Notifica

   Preparazione -> Salvataggio
   Salvataggio -> Notifica
}

[*] -> Attesa

Warning --> Attesa
Attesa --> Warning : scatto [non luce]

Attesa --> Preparazione : scatto [luce]
Notifica --> Attesa

Attesa -> [*] : spegni
Memorizzazione --> [*] : spegni
@enduml
```

```
@startuml
hide empty description

state t as "Sul treno"
state a as "Arrivato"

state Seduto {
   state l1 as "Legge"
   state l2 as "Legge in modo distratto"
   state o as "Origlia"
   state lc as "Libro chiuso"

   [*] -> l1
   l1 -> l2 : interesse l1 --> lc : stazione vicina
```

```
   l2 -> l1 : fine interesse
   l2 -> o  : maggior interesse
   l2 --> lc : stazione vicina

   o --> lc : stazione vicina

   lc -> [*]
   ---
   state m as "Ascolta musica"
   state v as "Volume basso"
   state ms as "Musica spenta"

   [*] -> m

   m -> v : interesse
   m --> ms : stazione vicina

   v -> m : fine interesse
   v --> ms : stazione vicina

   ms -> [*]
}

[*] -> t
t -> Seduto : trova posto

Seduto -> a
a -> [*]
@enduml
```

I macro-stati possono anche mantenere una "storia": quando l'esecuzione lascia uno stato con *history* viene salvato l'ultimo visitato. Se si ha *deep history* viene salvato l'ultimo sotto-stato foglia visitato, mentre se si ha *shallow history* viene salvato l'ultimo sotto-stato diretto. Una transizione che entra in uno stato con *history* è come se entrasse nello stato salvato nella history.

```
@startuml
hide empty description

state inactive as "Study program inactive"
state active as "Study program active" {
   [*] -> bachelor
   state bachelor as "Bachelor" {
      state b1 as "Bachelor's degree"
      state b2 as "Bachelor's degree completed"

      [*] -> b1
      b1 -> b2 : ended
      b2 --> master : enroll for master's degree
   }

   state doctorate as "Doctorate"
   doctorate -> [*] : ended

   state master as "Master" {
      state m1 as "Master's degree"
      state m2 as "Master's degree completed"
```

```
      [*] -> m1
      m1 -> m2 : ended
      m2 --> doctorate : enroll for doctorate
   }

   inactive --> [H*] : pay tuition fees
}

[*] --> inactive
inactive --> [*] : leave university

active -up-> inactive : new semester [tuition fees not paid]
active --> [*] : leave university
@enduml
```

Il linguaggio Java

Le basi

Le classi vengono definite in file con estensione `.java` e avente lo stesso nome, una classe per file. All'interno della definizione di classe verranno definiti i relativi metodi e attributi. Poiché oramai sei grande, ecco una classe esempio per spiegare la base della sintassi.

```java
// Data.java

// Commento linea
/*
 * Commento multilinea
 */

/*
 * Tutti gli oggetti ereditano implicitamente da Object
 */
public class Data {

  // private, public, protected e nulla per friendly
  private int giorno;
  private int mese;
  private int anno;

  /*
   * Le variabili statiche non sono legate ad una istanza e sono accessibile
   * tramite il tipo (Date.epochMonth). Devono essere sempre inizializzate
   */
  static public int epochMonth = 1970;

  /*
   * Se non fornisco un costruttore, java ne fornirà uno di default che
   * inizializza tutti gli attributi al valore di default
   */
  public Data(int parGiorno, int parMese, int parAnno) {
    giorno = parGiorno;
    mese = parMese;
    anno = parAnno;
  }
```

```java
  public Data() {
    this(12,12,2012);
  }

  public int getGiorno() {
    return giorno;
  }

  public int getMese() {
    return mese;
  }

  public int getAnno() {
    return anno;
  }

  public void setGiorno(int parGiorno) {
    giorno = parGiorno;
  }

  public void setMese(int mese) {
    this.mese = mese;
  }

  public void setAnno(int anno) {
    this.anno = anno;
  }
}

// Test.java
public class Test {

  /*
   * Nota bene: un programma può avere diversi `main`. Non viene a crearsi
   * ambiguità poiché al momento dell'esecuzione viene specificata la classe che
   * contiene il punto d'entrata.
   *
   * I metodi statici, come gli attributi statici, sono legati alla classe e
   * vengono invocati dal tipo (Test.main()). È ovvio che i metodi statici non
   * hanno accesso agli attributi d'istanza ma hanno accesso ad altri attributi
   * statici.
   */
  public static void main(String[] args) {
    int x = 10;
    Data d = new Data(); // creazione di un'istanza di Data
    Data d1; // auto-inizializzato a null

    int y = d.getGiorno();

    for (String s: args) { // foreach su iterabile
      ...
    }
  }
}
```

Non riporterò tutta la sintassi (eredità, interfacce, OOP base) ma solo alcune peculiarità o costrutti particolari.

Gli `enum`

Si possono dichiarare tipi enumerati per modellare insiemi con cardinalità ridotta. Gli `enum` sono vere e proprie classi con un numero limitato di istanze possibili. Per confrontare due `enum` non c'è bisogno di effettuare confronto semantico tramite `equals()`, ma basta ==. Una variabile enumerata può essere solo `null` o uno dei valori enumerati.

```
enum Size { SMALL, MEDIUM, LARGE, X_LARGE };
Size s = Size.MEDIUM;
```

A una classe enumerata si possono aggiungere costruttore, metodi e attributi che associano ulteriori informazioni alle costanti enumerate. I costruttori sono invocati solo quando vengono "costruite" le costanti. Non possono essere costruiti oggetti generici da classi enumerate.

```
public enum Size {
  SMALL("S"), MEDIUM("M"), LARGE("L"), X_LARGE("XL");

  private String abbreviation;

  private Size(String abbreviation) {
    this.abbreviation = abbreviation;
  }

  public String getAbbreviation() {
    return this.abbreviation;
  }
}
```

Tutte le classi enumerate, inoltre, offrono i seguenti metodi (per eredità):

1. `static Enum valueOf(String name)` : Restituisce la costante enumerata della classe indicata che ha quel nome.
2. `String toString()`
3. `Enum[] values()` : Restituisce un array contenente tutti i valori possibili della classe.
4. `String name()` : Restituisce il nome della costante enumerativa
5. `int ordinal()` : Restituisce la posizione (a partire da 0) della costante enumerativa.
6. `int compareTo(...)`

Il boxing

I tipi primitivi sono passati per valore. Alcune volte, però, potrebbe essere necessario ottenere un riferimento a questi. Java fornisce delle classi corrispondenti ad ogni tipo primitivo: `Integer`, `Character`, `Float`, `Long`, `Short` e `Double`. Un oggetto di tipo `Integer`, ad esempio, conterrà un solo `int` e sarà di tipo immutabile.

```
Integer i, x;
int y;

i = new Integer(5); // un modo per inizializzare
i = 5; // un altro modo. Viene effettuata la procedura di boxing
x = i; // y e x puntano allo stesso oggetto
y = i; // in y viene salvato il valore interno di i. Viene effettuato unboxing
       // automatico
```

Con il termine "boxing" e il corrispettivo "unboxing" intendiamo il processo di mettere, figuratamente, un valore (in questo caso un primitivo) in un contenitore (un oggetto) e, analogamente, il processo inverso di estrarre questo valore dal proprio contenitore.

Attenzione: i riferimenti a tipi primitivi sono oggetti veri e propri, anche se si può essere tentati di fare `x == y`, conviene lo stesso usare `equals()` poiché il compilatore potrebbe effettuare delle ottimizzazioni che rendono inconsistente il comportamento di ==.

Gestione delle eccezioni

Un metodo deve poter segnalare l'impossibilità di produrre un risultato significativo o la propria terminazione scorretta. Possono esserci 4 possibilità per segnalare ciò:

1. Termina il programma
2. Restituire un valore convenzionale che segnala l'errore
3. Portare il programma in uno stato scorretto
4. Usare un metodo speciale predefinito per la gestione degli errori

Il metodo utilizzato dai linguaggi OOP è quello delle eccezioni. Le eccezioni sono a loro volta oggetti speciali che possono essere sollevate dal chiamato e catturate dal chiamante.

```
try {
  x = x/y;
} catch (DivisionByZeroException e) {
  ...
} finally {
  ...
}
```

Le eccezioni risalgono finché non raggiungono un blocco `catch` che le cattura oppure il `main`, dove causerà la terminazione. Il blocco `finally`, invece, viene eseguito in tutti i casi, sia se viene sollevata un'eccezione, sia se non viene sollevato nulla, sia se vengono catturate eccezioni.

Il fatto che un metodo possa terminare sollevando un'eccezione va dichiarato nella sua interfaccia per mezzo della clausola `throws`:

```
public int leggiInteroDaInput() throws IOException
```

Nota bene: non vanno dichiarate le eccezioni runtime, ad esempio `NullPointerException`. Le eccezioni runtime ereditano da `RuntimeException`, invece che da `Exception`, e vengono dette *unchecked*.

Per sollevare un'eccezione esplicitamente si usa la parola chiave `throw`.

```
if (mandelle)
   throw new Exception();
```

Per definire nuove eccezioni basta semplicemente estendere `Exception` e ridefinire i due costruttori:

```
public class MyException extends Exception {
  public MyException() { super(); }
  public MyException(String s) { super(s); }
}
```

Iteratori

Per implementare un iteratore per una data collezione, è necessario definire una classe che implementa l'interfaccia `Iterable`. Una implementazione comune è la seguente:

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class MyCollection<T> implements Iterable {
  ...

  public Iterator<T> iterator() {
    return new MyCollectionIterator<T>();
  }

  private class MyCollectionIterator implements Iterator<T> {
    public T next() {
      ...
```

```
    }

    public bool hasNext() {
      ...
    }
  }
}
```

Annotazioni

Le annotazioni sono strumenti che aggiungono informazioni aggiuntive sul nostro codice. Una di queste l'abbiamo già incontrata: `@Override`. Altre degne di nota sono `@Dreprecated` e `@SuppressWarnings(...)`.

I design pattern

I problemi incontrati nello sviluppo di grossi progetti software sono spesso ricorrenti e prevedibili. I design pattern sono "schemi di soluzioni" riutilizzabili. Permettono quindi di non inventare da capo soluzioni ai problemi già risolti, ma di utilizzare dei mattoni di provata efficacia.

I pattern che vedremo sono classificati in 3 tipi:

1. Creazionali: Factory e Singleton
2. Strutturali: Adapter, Decorator e Proxy
3. Comportamentali: Iterator, Observer, State e Strategy

Factory

Limitare le dipendenze delle classi è desiderabile perché permette di sostituire un'implementazione con un'altra. Una eccezione è la chiamata al costruttore: il codice utente che chiama il costruttore di una classe rimane vincolate a quella classe. Per questo esistono pattern per un'operazione semplice come la creazione di un oggetto: disaccoppiano il codice che fa uso di un tipo da quello che sceglie quale implementazione del tipo utilizzare.

In Java le chiamate ai costrutti non sono personalizzabili. La soluzione è nascondere la creazione in un metodo detto *factory*: un metodo che restituisce un oggetto di una classe senza essere costruttore. Per implementare il pattern factory, bisogna rendere il costruttore `private` o `protected` e costruire un oggetto invocando metodo pubblico statico (il *factory method*).

Singleton

A volte viene usata la definizione per istanziare un solo oggetto. Usare una normale classe con soli metodi statici non assicura che esista un solo esemplare della classe, se viene reso visibile il costruttore. In una classe singleton il costruttore è protetto o privato, un metodo statico fornisce l'accesso alla sola copia dell'oggetto.

Adapter

Un software esistente usa una data interfaccia. Un altro software ne usa un'altra. Come è possibile combinarle senza cambiare il software? Basta usare un oggetto che implementa l'interfaccia del primo ed espone l'interfaccia del secondo.

Proxy

Lo scopo del proxy è di proporre o addirittura evitare l'istanziazione di oggetti pesanti se non necessaria. Interponiamo un oggetto, proxy, con la stessa interfaccia dell'oggetto "pesante" di cui fa le veci. L'oggetto può fare preprocessing o a volte rispondere direttamente alle richieste se è in grado di farlo. I clienti dell'oggetto chiamano i metodi di un *Subject*, a sua volta super-classe del proxy

Observer

Definisce due ruoli: il *Subject* (chi è osservato) e lo *Observer* (detto anche *Listener*). Più oggetti possono essere interessati ad essere informati quando un *Subject* cambia stato. Il *Subject* non dipende dal numero e tipo di osservatori, questi possono anche cambiare a runtime.

Strategy

Il pattern Strategy è utile dove è necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione. Questo pattern permette la selezione a runtime di diversi algoritmi. I nostri algoritmi devono essere intercambiabili tra loro grazie ad una interfaccia comune. Il cliente dell'algoritmo non deve fare nessuna assunzione su quale sia la strategia istanziata in un particolare istante.

Decorator

Il pattern Decorator consente di aggiungere nuove funzionalità ad oggetti già esistenti a runtime. Questo viene realizzato costruendo una nuova classe decoratore che "avvolge" l'oggetto originale passando l'oggetto originale come parametro al costruttore del decoratore. Il Decorator offre un'alternativa alle sottoclassi: permette l'aggiunta di funzionalità solo per determinati oggetti in un secondo momento, anche a runtime.

State

Nel caso di classi mutabili, si può pensare ad implementazioni multiple, i cui oggetti cambiano dinamicamente configurazione a seconda dello stato. Le implementazioni multiple corrispondono a diversi stati in cui possono trovarsi gli esemplari del tipo astratto. Nel corso della vita dell'oggetto, possono essere utilizzate diverse implementazioni senza che l'utente se ne accorga.

Model View Controller (MVC)

Il pattern è basato su tre ruoli principali:

1. Il *model* fornisce i metodi per accedere ai dati utili all'applicazione
2. Il *view* visualizza i dati contenuti nel *model* e si occupa dell'iterazione con utenti e agenti
3. Il *controller* riceve i comandi dell'utente (in genere attraverso il view) e li attua modificando lo stato degli altri due componenti

Principi di progettazione

Principio open-closed e di sostituzione di Liskov

Il principio open-closed dice che in generale un modulo deve essere aperto alle estensioni, ma chiuso alle modifiche. Le classi dovranno, quindi, essere scritte in modo che siano estendibili, senza che debbano essere modificate. Questo principio porta notevoli vantaggi in termini di manutenzione: se non modifico le classi, riduco la possibilità di introdurre errori.

Un modo per fornire estensibilità è usare l'ereditarietà. L'estensibilità tramite eredità, però, è garantita solo se tutte le sottoclassi rispettano in contratto instaurato dalla superclasse. Questo viene detto principio di sostituzione di Liskov.

Per creare oggetti che modificano il contratto di una classe ma ne riusano il codice si deve usare la composizione. In generale, si favorisce sempre la composizione rispetto all'ereditarietà poiché non abbiamo vincoli di contratto imposti dalla superclasse.

Principio di dependency inversion

Il principio di dependency inversion ci impone la dipendenza da astrazioni, non da oggetti concreti. Per inversione si intende che le classi "in alto" nella gerarchia di ereditarietà non devono dipendere dalle classi "in basso".

Una delle conseguenze della dependency inversion è il cosiddetto principio di "coding to an interface", ossia che ogni classe `C` che si ritiene possa essere estesa in futuro dovrebbe essere definita come sottotipo di un'interfaccia o di una classe astratta `A`. Tutte le volte che non è strettamente indispensabile riferirsi ad oggetti della classe concreta `C`, è meglio riferirsi

ad oggetti con tipo `A`. In questo modo sarà più facile modificare il codice cliente per utilizzare invece di `C` altre classi concrete che sono sempre sottotipi di `A`.

Valutazione della qualità

La correttezza funzionale di un progetto è fondamentale, ma possiamo adottare anche altre metriche, come la performance, la modificabilità e la stabilità. Possiamo valutare gli aspetti strutturali in un progetto studiando l'accoppiamento, la coesione e il principio open/closed.

L'accoppiamento (coupling)

L'accoppiamento cattura il grado di interconnessione tra classi. Un altro grado di accoppiamento significa alta interfipendenza tra le classi e dipendenza di modifica della classe singola. Un basso accoppiamento è requisito fondamentale per creare un sistema comprensibile e modificabile. Nel mondo OOP abbiamo 3 tipi di coupling: interaction, component e inheritance.

1. Interaction coupling: si ha quando i metodi di una classe chiamano metodi di altre classi. Le forme di interaction coupling da evitare sono: manipolare direttamente le variabili di stato di altre classi e lo scambio di informazioni attraverso variabili temporanee. Una forma che è accettabile, ma non raccomandata è la comunicazione tra metodi tramite parametri.
2. Component coupling: una class `A` è accoppiata ad un'altra classe `C` se possiede attributi di tipo `C`, parametri di tipo `C` o metodi con variabili locali di tipo `C`. Quando `A` è accoppiata a `C` essa è accoppiata anche a tutte le sue sottoclassi. Il component coupling spesso implica interaction coupling.
3. Inheritance coupling: quando una classe è sottoclasse di un'altra. Discussa già con il principio open/closed.

La coesione

La coesione è un concetto intra-modulo. Essa si concentra sul perché gli elementi stanno nello stesso modulo. Anche qui abbiamo tre tipi di coesione: method, class e inheritance.

1. Method cohesion: è la coesione all'interno di una singola funzione. La forma migliore si ha quando una funzione implementa una singola azione chiaramente definita.
2. Class cohesion: una classe dovrebbe rappresentare un solo concetto e tutte le proprietà dovrebbero contribuire alla sua rappresentazione. Più concetti sono incapsulati nella stessa classe più la coesione diminuisce.
3. Inheritance cohesion: maggiore è la coesione per ereditarietà più la gerarchia viene usate per gestire generalizzazione/specializzazione e non semplice riuso di codice.

Metriche

È possibile definire delle misure, dette metriche, delle caratteristiche affrontate:

1. Weighted Methods per Class (WMC)
2. Depth of Inheritance Tree (DIT)
3. Number of Children (NOC)
4. Coupling Between Classes (CBC)
5. Response For a Class (RFC)
6. Lack of Cohesion in Methods (LCOM)

Consigli nella realizzazione

1. Minimizzare l'interfaccia:
    - Se non si ha un motivo per rendere un metodo pubblico, deve essere privato
    - Evitare metodi di tipo setter se possibile
    - Non bisogna creare un getter per ogni singolo campo intero della classe
2. Scrivere metodi con meno parametri
3. Evitare gli anti-pattern (soluzioni comuni ma sbagliate) come ad esempio le classi Blob

Programmazione concorrente in Java

Noi vedremo programmazione concorrente solo a livello di thread, non a livello di processo.

Classe Thread

Per definire un thread dobbiamo creare una classe che estende `Thread`. Per avviare il thread dovremo istanziare la classe e chiamare il metodo `start()` che creerà e avvierà il thread.

```
class ListSorter extends Thread {
  ...
  public void run() {
    ... // codice da eseguire
  }
}

class Main {
  public static void main(String[] args) {
    ...
    ListSorter s = new ListSorter(l);
    s.start();
  }
}
```

Un altro modo per costruire un thread è implementare direttamente l'interfaccia `Runnable` e poi passare il nostro `Runnable` alla classe `Thread`:

```
class ListSorter implements Runnable {
  ...
  public void run() {
    ... // codice da eseguire
  }
}

class Main {
  public static void main(String[] args) {
    ...
    ListSorter s = new ListSorter(l);
    Thread t = new Thread(s);
    t.start();
  }
}
```

Liveness

La liveness è un proprietà di un programma concorrente. Essa implica che il programma viene eseguito entro limiti accettabili di tempo. Le principali situazioni che danneggiano la liveness da evitare attraverso attenta progettazione sono deadlock, livelock e starvation.

- **Deadlock**: due o più thread sono bloccati per sempre in mutua attesa.
- **Starvation**: situazione in cui un thread ha difficoltà a guadagnare accesso a una risorsa e ha difficoltà a procedere.
- **Livelock**: genera una sequenza ciclica di operazioni inutili ai fini dell'effettivo avanzamento della computazione.

Metodi e statement sincronizzati

I metodi sincronizzati sono definiti dalla parola chiave `synchronized` e sono quei metodi che possono essere eseguiti solo da un thread alla volta.

Java associa un *intrinsic lock* (monitor) a ciascun oggetto. Quando un metodo `synchronized` viene chiamato se nessun metodo `synchronized` è in esecuzione l'oggetto viene bloccato e il metodo eseguito, altrimenti se l'oggetto è bloccato il task chiamante viene sospeso fino a quando il task bloccante libera il lock. Il lock viene liberato alla fine del metodo ma anche al lancio di un'eccezione. Inoltre i metodi `static synchronized` usano un diverso *intrinsic lock* che fa riferimento all'oggetto classe e non all'istanza.

I costruttori non possono essere synchronized. Solo i thread che crea l'oggetto ha accesso as esso mentre viene creato. Attenzione però a non far uscire un riferimento prematuro all'oggetto.

Eventuali dati `final` possono essere letti con sicurezza anche da metodi non `synchronized`.

Possiamo anche sincronizzare sincronizzare solo un solo alcuni statement:

```
public void addName(String name) {
  synchronized(this) { // oggetto da proteggere con lock
    lastName = name;
    nameCount++;
  }
  nameList.add(name);
}
```

In questo modo si rilascia il lock all'oggetto prima di invocare un metodo che potrebbe a sua volta richiedere di attendere il rilascio di un lock. Ciò potrebbe generare potenziali problemi di liveness.

La parola chiave `synchronized` non è considerata parte dell'interfaccia di un metodo ma un dettaglio implementativo. Ciò significa che non viene ereditata! Le sotto classi devono quindi essere esplicite quando fanno override.

Precondizioni per metodi synchronized: guarded blocks

Per mettere in stato di `WAITING` un thread si può chiamare il metodo `wait()` all'interno di un blocco sincronizzato. La `wait()` oltre a cambiare stato del thread rilascia il lock all'oggetto. Attenzione: lo stato `WAITING` è diverso dallo stato di `SLEEPING`, raggiunto tramite `Thread.sleep(ms)`.

Per risvegliare uno dei thread nella coda di attesa si chiama `notify()` all'interno di un blocco sincronizzato. Nota bene: la `notify()` non implica il rilascio del lock da parte del chiamante, ma semplicemente modifica lo stato del primo thread della coda da `WAITING` a `READY`. Per risvegliare tutta la coda d'attesa si usa `notifyAll()`.

Tutti i metodi visti in questa sezione funzionano solo se chiamati sull'oggetto che detiene il lock.

```
// SBAGLIATO!!
class ConsumerProducer {
  ...
  private Object cons = new Object();
  private Object prod = new Object();

  public synchronized void set (int d) {
    while (!empty)
      try {
        prod.wait();
      } catch (InterruptedException e) {}

    cons.notify();
    data = d;
    empty = false;
  }

  public synchronized int get () {
    while (!empty)
      try {
        cons.wait();
      } catch (InterruptedException e) {}
```

```
    prod.notify();
    empty = true;
    return data;
  }
}
```

Oggetti Lock

Il codice `synchronized` definisce un caso elementare di lock. Il pacchetto `java.util.concurrent.lock` ci permette delle funzionalità più avanzate tramite l'oggetto `Lock`. La funzionalità più importante è la possibilità di ritirarsi dalla richiesta di lock tramite due metodi:

1. `tryLock()` esce immediatamente dall'attesa se il lock non è disponibile
2. `lockInterruptibly()` esce se viene interrotto uno dei thread in attesa allo stesso lock.

Esecutori

La programmazione concorrente implementata con la classe `Thread` impone una stretta relazione tra il dover essere eseguito (interfaccia `Runnable`) e l'oggetto stesso. I due concetti possono, però, essere separati mediante l'interfaccia `Executor` che eseguono thread tramite thread pools e `fork()/join()`.

Gli esecutori sono predefiniti e consentono una gestione efficiente che riduce l'overhead dovuto alla gestione dei thread. Sono definite 3 interfacce: `Executor`, `ExecutorService` e `ScheduledExecutorService`. Per far partire un esecutore ci basterà solamente chiamare `e.execute(r)`, evitando la creazione di un oggetto `Thread`. Le implementazioni dell'interfaccia `Executor` usano un pool di `Thread`, come ad esempio una a lunghezza fissa come quella di `FixedThreadPool`. I task saranno inviati al pool attraverso una coda.

Programmazione funzionale in Java

Classi anonime e funzioni lambda

La programmazione funzionale è un altro paradigma di programmazione che favorisce l'espressione della computazione come applicazione di una o più funzioni sui dati. In questo caso le funzioni sono definite in senso matematico: niente side effects e deterministiche senza stato.

Il principale metodo per implementare la programmazione funzionale è usare delle classi anonime, ossia delle classi senza nomi, che possono essere solo istanziate una volta e mai più riusate.

```
Collections.sort(l, new Comparator<Persona>() {
  @Override
  public int compare(Persona p1, Persona p2) {
    ...
  }
});
```

Java 8 offre della sintassi per supportare facilmente il pattern delle classi anonime. Le interfacce con un solo metodo sono definite *functional interfaces*. La nuova sintassi per definire il metodo di una *functional interface* è la lambda expression.

```
Comparator<Persona> c = (Persona p1, Persona p2) -> {
  ...
};

Collections.sort(l, (Persona p1, Persona p2) -> {
  ...
});
```

L'input delle funzioni lambda è sempre immutabile (`final`) e il tipo può essere omesso. Le lambda con un solo statement hanno un return implicito e possono omettere le graffe, mentre se si ha più di un statement diventano obbligatorie graffe e return.

Il pacchetto `java.utils.function` offre alcuni tipi e funzioni utili per la programmazione funzionale:

- `Function<T, R>`: funzione con un argomento `T` e risultato `R`
- `BiFunction<T, U, R>`: funzione con due argomenti di tipo `T` e `U` e risultato `R`
- `Consumer<T>`: funzione con un argomento di tipo `T` che non ritorna nessun risultato
- `Predicate<T>`: funzione con un argomento `T` che ritorna un booleano

I tipi sopra sono ridefinite per i corrispettivi tipi primitivi, ad esempio `IntFunction<R>`.

Optional

Il tipo `Optional<T>` nasconde il fatto che un elemento possa essere `null`. Possiamo usare i seguenti metodi:

- `Optional.of(T val)`: crea un `Optional` con valore `val`;
- `Optional.empty()`: crea un `Optional` vuoto;
- `Optional.ifPresent(Consumer<T>)`: chiama la funzione se e solo se l'`Optional` non è vuoto;
- `Optional.flatMap(Function<T, Optional<U>>)`: se `Optional` è `empty`, ritorna `empty`, altrimenti esegue la funzione;
- `Optional.orElse(T val)`: se `Optional` non è vuoto ne ritorna il valore, altrimenti ritorna `val`.

```
public class Indirizzo {
  final String via;
  final Optional<String> commenti;

  public Indirizzo (String via, String commenti) {
    this.via = via;
    this.commenti = Optional.of(commenti);
  }

  public Indirizzo(String via) {
    this.via = via;
    this.commenti = Optional.empty();
  }

  ...

  public Optional<String> getCommenti() {
    return commenti;
  }
}

public class Persona {
  final String nome;
  final Optional<Indirizzo> indirizzo;

  public Indirizzo (String nome, Indirizzo indirizzo) {
    this.via = via;
    this.indirizzo = Optional.of(indirizzo);
  }

  public Indirizzo(String nome) {
    this.via = via;
    this.indirizzo = Optional.empty();
  }

  ...

  public Optional<Indirizzo> getIndirizzo() {
    return indirizzo;
```

```
  }
}

...
  Optional<Persona> o = ...
  String commenti = o
    .flatMap(p -> p.getIndirizzo)
    .flatMap(i -> p.getCommenti)
    .orElse("no comment");
...
```

Il pattern applicato con `Optional` è un pattern ricorrente nella programmazione funzionale chiamato composizione monadica. L'idea è di inglobare un valore in una struttura che permette di concatenare funzioni, mascherando alcuni aspetti.

Stream

Gli stream permettono di concatenare funzioni che agiscono su una collection. Partendo da un `Collection<T>` il metodo `stream()` ritorna l'oggetto `Stream<T>`. L'interfaccia stream ci permette di modificare in modo interessante (e in modo funzionale) gli elementi di una collection con i seguenti metodi:

- `forEach(Consumer<T>)`: prende in ingresso un `Consumer<T>` e lo esegue su ogni elemento della collection;
- `filter(Predicate<T>)`: prende in ingresso un `Predicate<T>` e ritorna uno stream che contiene solo gli elementi per cui il predicato è vero;
- `map(Function<T, U>)`: prende in ingresso una `Function<T, U>` e la applica ad ogni elemento della stream, ottenendo una `Stream<U>`;
- `distinct()`: rimuove i duplicati; `sort()`: ordina gli elementi tramite
- ordine naturale (interi e stringhe) o un `Comparator<T>`;
- `flatMap(Function<T, Stream<U>>)`: applica `Function<T, Stream<U>>` a ogni elemento dello stream, infine unisce tutti gli stream in unico stream;
- `reduce`: la riduzione è un'operazione che permette di raccogliere un solo valore a partire da uno stream (come `sum` o `average`). La riduzione può anche essere implementata manualmente fornendo un valore iniziale e una funzione che prende un risultato parziale e aggiunge il nuovo elemento al risultato parziale (esempio `reduce(0, (a,b) -> a+b)`);
- `collect(Supplier<R>, Accumulator<T, R>, Combiner<T, T, R>)`: analogo alla reduce, non crea un nuovo valore per ogni elemento, ma modifica sempre il medesimo valore. Nel caso delle strutture dati standard, esistono dei *collectors* predefiniti;

Uno stream non memorizza i suoi elementi; essi possono essere generati on-demand o salvati in una collection. Le operazioni su stream non possono modificare lo stream a cu si applicano ma possono solo restituire nuovi stream che contengono i risultati. Inoltre le operazioni sono *lazy* quando possibili, ossia sono eseguite solo quando il loro risultato è necessario.

Java permette di analizzare gli elementi di uno stream in parallelo semplicemente invocando la funzione `parallel()`. Attenzione ai side effects nell'uso di parallel!

```
// NO
final List<Integer> result = new ArrayList<>();
list.stream().parallel()
  .map(x -> x.size())
  .forEach(x -> result.add(x));

// OK
final List<Integer> result = list.stream().parallel()
  .map(x -> x.size())
  .collect(Collectors.toList())
```

Closures

Supponiamo di avere una funzione lambda che vogliamo usare molte volte. Come abbiamo visto, possiamo definirla un volta e riusarla successivamente:

```
Predicate<String> pred = s -> s.startsWith("L");
...
list.stram().filter(pred)
```

E se volessimo generalizzare il nostro predicato in modo da usare qualsiasi carattere? Possiamo usare una chiusura, ossia uno snapshot dell'ambiente circostante alla lambda:

```
public static Predicate<String> pred(String prefix) {
  return s -> s.startsWith(prefix);
}
...
list.stream().filter.filter(pred("X"));
```

L'unica limitazione che la closure ci impone è che le variabili dello scope esterno usate all'interno della lambda devono essere `final` o *effectively* `final`, ossia mai variate dopo la loro inizializzazione.

Astrazioni nella progettazione del software

Le astrazioni usate nella progettazione software che vedremo saranno:

1. Astrazioni sulle operazioni: *Procedural abstractions*
2. Astrazioni sui dati: *Abstract data type*
3. Astrazioni sul controllo: iteratori
4. Astrazioni da tipi individuali a famiglie di tipi

Per astrarre nella progettazione possiamo usare diversi meccanismi:

1. **Abstraction by parameterization**: generalizza moduli in modo che siano utilizzabili su dati diversi passati tramite parametri
2. **Abstraction by specification**: non importa come una funzione è implementata, ma conta solo il comportamento. I vantaggi offerti sono la località, la specifica può essere letta o scritta senza la necessità di esaminare l'implementazione, e modificabilità, le funzioni possono essere re-implementate senza danneggiare i clienti

Astrazioni procedurali

Le astrazioni procedurali definiscono tramite una specifica un'operazione complessa su dati generici (o parametri). La specifica può essere data usando il linguaggio naturale oppure una semplice notazione matematica. Noi useremo la seconda tramite JML.

Il contratto è il termine usato spesso per descrivere la specifica di astrazioni procedurali o interfacce di moduli.

Un contratto può essere soddisfatto in molti modi. Ciò che cambia tra le varie implementazioni sono le proprietà non funzionali come efficienza e consumo di memoria.

Ogni funzione deve avere definito una precondizione (`requires`) e una postcondizione (`ensures`). La precondizione di un metodo ci dice cosa deve essere vero per poterlo chiamare. La postcondizione normale ci dice che cosa deve essere vero quando il metodo ritorna normalmente; la postcondizione eccezionale ci dicono cosa è vero quando il metodo ritorna con un eccezione.

JML

Le specifiche JML sono contenute in annotazioni rappresentate con la seguente sintassi nei commenti:

```
//@ ...

/*@ ...
  @ ...
  @*/
```

Le asserzioni sono espressi booleane Java che non devono avere side-effect. Esse sono precedute da opportune keyword (`requires`, `ensures`...) e possono contenere alcuni operatori non Java: `a ==> b`, `a <== b`, `a <==> b`, `a <=!=> b`,

`\old(E)`, `\result`, `\forall` ecc… Si possono scrivere asserzioni in linguaggio naturale usando `(*...*)`; hanno valore sempre `true`.

```
//@ requires in >= 0;
//@ ensures Math.abs(\result * \result - in) < 0.0001;
public static float sqrt(float in);
```

Se l'oggetto è mutabile, ci si può riferire allo stato iniziale dell'oggetto usando `\old(expr)`.

Per segnalare che un parametro può essere modificato si usa `assignable expr`. Se un metodo non ha side-effects si può usare `assignable \nothing`. Se `assignable` è omesso non si promette la mancanza di side-effects.

```
//@ assignable a[*];
//@ ensures (* a è una permutazione di \old(a) *) &&
//@   (* a è ordinato per valori crescenti *)
public static void sort(int[] a);
```

Il lancio di una eccezione viene descritto tramite l'asserzione `signals(exception) expr`. Cosa cambia tra una precondizione e una postcondizione eccezionale? Procedure con precondizioni non vuote vengono dette parziali, poiché hanno comportamento specificato solo per un subset di valori in ingresso. La presenza di molte procedure parziali può danneggiare la robustezza del programma poiché può mandarlo in uno stato inconsistente. Le eccezioni, invece, mandano in crash il programma se non gestite evitando stati inconsistenti. Perciò per metodi pubblici di classi pubbliche è buona norma gestire le precondizioni tramite eccezioni. In generale, però, non si lanciano eccezioni quando il chiamante può fare il controllo della precondizione multo meglio del chiamato o quando il controllo è molto difficile/inefficiente (esempio una binary search).

Per parlare degli elementi di una collection si possono usare i metodi pubblici della collezione che non hanno side-effect, come ad esempio `equals`, `contains`, `containsAll`, `get` e `sublist`. Possiamo anche usare i soliti quantificatori matematici tramite `\forall` e `\exists`, funzioni quantificatrici come `\sum`, `\product`, `\min` e `\max`, e il quantificatore numerico `\num_of`:

```
//@ (\quant var; range; cond|operation)
```

Abstract data types

Creare un nuovo tipo di dato per cui siano stati specificati valori e operazioni possibili è anche detto data abstraction. Astraendo dai dettagli di rappresentazione dei valori e d'implementazione delle operazioni il resto del programma dipende solo dalla specifica del tipo e non dalla sua implementazione.

La specifica di un ADT è l'unione di sintassi e semantica. La signature dei metodi, o l'interfaccia, definiscono solo la sintassi. Per la semantica, non basta specificare i metodi come se fossero procedural abstractions, perché questi agiscono sulle variabili di stato. Possiamo usare Java e JML per definire ADT.

Nella specifica JML di un metodo pubblico (non statico) possono comparire solo gli elementi pubblici del metodo e della classe, in particolare i parametri forma il, `\result`, ma anche metodi pubblici `pure` o attributi pubblici. Vengono detti metodi puri i metodi (non statici) dichiarati con la keyword JML `pure`. Essi non hanno effetti collaterali e possono chiamare solo altri metodi puri. Anche i costruttori possono essere dichiarati puri e possono inizializzare gli attributi dichiarati nella classe. I metodi puri sono anche detti *observers* poiché possono solo osservare lo stato. I metodi che invece modificano lo stato sono detti *mutators*.

```
//@ ensures (* \result è cardinalità di this *)
public int /*@ pure @*/ size();
```

La specifica di un'astrazione descrive un *oggetto astratto*. L'implementazione di una astrazione definisce *l'oggetto concreto*. I due oggetti sono differenti e non vanno confusi.

Proprietà delle data abstractions

Abbiamo velocemente introdotto due categorie di metodi: *observers* e *mutators*. Approfondiamole e aggiungiamone delle altre:

1. *creators*: producono nuovi oggetti da zero
   - Sono costruttori, di solito puri. NB: non tutti i costruttori però sono creatori

2. *producers*: dati in input oggetti del proprio tipo creano oggetti del proprio tipo
    - Solitamente sono puri
3. *mutators*: modificano oggetti del proprio tipo
    - Non sono mai puri e non hanno dichiarazione `assignable` in JML
4. *observers*: dati in input oggetti del proprio tipo restituiscono risultati di altri tipi
    - Tipicamente dichiarati come puri
    - A volte sono combinati con i modificatori (ad esempio chooseAndRemove)

Un tipo è adeguato se fornisce operazioni sufficienti perché il tipo possa essere utilizzato semplicemente ed efficientemente:

- Se il tipo è mutabile deve definire almeno *creators*, *observers* e *mutators*
- Se il tipo è immutabile deve avere almeno *creators*, *observers* e *producers*

Il tipo deve essere totalmente popolato, ossia deve essere possibile ottenere convenientemente ogni possibile stato astratto usando *creators*, *producers* e *mutators*. Non bisogna, però, includere troppe operazioni, sennò si rende troppo pesante l'astrazione. L'adeguatezza è, perciò, un concetto informale e dipendente dal contesto d'uso. Se il contesto è limitato basteranno, quindi, poche operazioni.

Le proprietà astratte sono proprietà osservabili, con i metodi pubblici *observer*. Le proprietà astratte possono essere di due tipi:

1. Evolutive: esprimono una relazione tra uno stato astratto osservabile e uno futuro
2. Invarianti: proprietà dei singoli stati astratti osservabili. In JML possono essere specificati tramite `public invariant` e si possono usare solo attributi e metodi pubblici della classe.

Gli invarianti vanno rispettati da ogni metodo della specifica che modifica lo stato. È il progettista della specifica che deve assicurare che questa non violi gli invarianti prefissati.

Implementazione di un ADT

L'implementazione di un tipo deve fornire una rappresentazione per gli oggetti del tipo, cioè una struttura dati per rappresentarne (*rep*) i valori, e l'implementazione di tutte le operazioni.

La scelta del *rep* dipende da efficienza, semplicità e riuso di strutture dati esistenti.

Definiamo funzione di astrazione una funzione che assegna ad ogni stato concreto al più uno stato astratto. Solitamente è una funzioni non iniettiva in quanto ogni stato concreto può essere associato allo stesso stato astratto. Per implementare una funzione d'astrazione si può usare un invariante dichiarato `private`, ossia che ha accesso anche agli attributi privati della classe, definendo una relazione tra le parti private e i metodi *observer* della classe.

La funzione di astrazione è tipicamente implementata in java tramite la funzione `toString()`: il metodo tipicamente restituisce una rappresentazione testuale del valore dell'oggetto.

Non tutti gli oggetti concreti, però, sono rappresentazione "legali" degli oggetti astratti. Le rappresentazioni legali sono quelle per cui valgono tutte le ipotesi sottese all'implementazione. È detto *invariante di rappresentazione* un predicato booleano che è valido solo per oggetti legali. Di norma l'invariante di rappresentazione contiene solo riferimenti agli attributi privati della classe in quanto le relazioni fra un oggetto astratto (`public`) e il concreto (`private`) sono definite dalla funzione d'astrazione.

Ragionare sulle data abstractions

Scrivendo un programma, spiegandolo o leggendolo, si cerca di convincersi che sia corretto, ragionandovi. Fare ragionamenti sulle data abstractions è complicato: occorre considerare l'intera classe e non solo una singola procedura; il codice manipola il *rep*, ma ci si deve convincere che soddisfa la specifica degli oggetti astratti. Possiamo usare un approccio a due passaggi:

1. Mostriamo che l'implementazione conserva l'invariante di rappresentazione
2. Mostriamo che le operazioni sono corrette rispetto alla specifica dell'astrazione, eventualmente utiulizzando l'invariante di rappresentazione e le proprietà dell'astrazione.

Estensione e principio di sostituzione di Liskov

Armati dei nostri nuovi strumenti, come possiamo garantire che il contratto dell'estensione sia compatibile con quello della sopraclasse? Basta garantire le seguenti tre proprietà:

1. Regola della segnatura: un sottotipo deve avere tutti i metodi del sopratipo e le segnature dei metodi del sottotipo devono essere compatibili
2. Regola dei metodi: le chiamate ai metodi del sottotipo devono comportarsi come le chiamate ai metodi corrispondenti del sopratipo
3. Regola delle proprietà: il sottotipo deve preservare tutti i `public invariant` degli oggetti del sopratipo

Regola della segnatura

La regola della segnatura ci garantisce type-safety, ossia che ogni chiamata corretta per il sopratipo sia corretta anche per il sottotipo. Questa correttezza è verificabile staticamente.

In Java la regola della segnatura è ristretta ulteriormente imponendo l'identità tra le segnatura invece della compatibilità. Una sottoclasse può, però, rimuovere delle eccezioni dalla segnatura.

La regola della segnatura ci permette di definire dei tipi di trasformazioni tra tipi:

1. Una trasformazione P è covariante se per tutti i tipi T e S, con $S \leq T$, allora $P(S) \leq T(S)$.
2. Una trasformazione P è controvariante se per tutti i tipi T e S, con $S \leq T$, allora $T(S) \leq P(S)$.
3. Negli altri casi P è detta invariante.

Per mantenere la type-safety garantitaci dalla regola della segnatura possiamo usare covarianza e controvarianza in questo modo: controvarianza per i parametri (impossibile da realizzare in Java a causa della restrizione della regola) e covarianza per i risultati.

Regola dei metodi

La regola dei metodi esprime una proprietà relativa al comportamento di un metodo. Ciò rende impossibile la sua verifica da parte del compilatore.

Un metodo per garantire che la chiamata a un metodo del sottotipo abbia lo stesso effetto, basta che la specifica sia la stessa. Spesso, però, è necessario modificare la specifica nel sottotipo, perciò sono definite delle regole per garantire la compatibilità delle specifiche:

1. Precondizione più debole: se la precondizione del metodo ridefinito è più debole di quella del metodo originale, allora tutti i casi in cui si chiamava il metodo originale si può chiamare anche il metodo ridefinito:

$$pre_{super} \implies pre_{sub}$$

2. Postcondizione più forte: se rafforziamo la postcondizione, allora la postcondizione originale sarà comunque verificata:

$$post_{sub} \implies post_{super}$$

In JML, una classe eredità le pre e postcondizioni dei metodi pubblici e protetti della superclasse e gli invarianti pubblici. Per ridefinire delle asserzioni bisogna prefissare il blocco con `also`:

```
//@ requires var >= 10;
//@ ensures \result >= 2;
public int method(int param);

...

//@ also
//@ requires var >= 5;
//@ ensures \result >= 1;
public int method(int param);
```

Le precondizioni della classe figlia sono messe in disgiunzione con quelle della classe antenato: le precondizioni sono indebolite. Le postcondizioni della classe figlia, invece, sono messe in congiunzione: le postcondizioni vengono rafforzate.

Con la regola della postcondizione enunciata, dove non vale pre_{super}, può valere una $post_{sub}$ diversa o incompatibile con $post_{super}$. Rafforziamo perciò la regola della postcondizione in questo modo:

$$(pre_{super} \land post_{sub}) \implies post_{super} \equiv pre_{super} \implies (post_{sub} \implies post_{super})$$

La regola completa della postcondizione è sempre verificata in JML grazie al modo in cui sono unite le specifiche delle classi.

Java permette di rimuovere delle eccezioni quando si ridefinisce un metodo, ciò però potrebbe violare la regola dei metodi! Possiamo eliminare un'eccezione solo se non è effettivamente utilizzata o il suo lancio è opzionale. Un esempio di un caso del genere è quando l'eccezione era usata per segnalare una condizione che non si può più verificare più.

```
class Socket {
  public int connect() throws CantConnectException;
}

class RetrySocket extends Socket {
  public int connect();
}
```

Addendum: forza e debolezza delle condizioni Si definisce una condizione più forte si un'altra se la prima è più restrittiva della seconda. Viceversa è definita la debolezza. In logica matematica la relazione di "forza" tra condizione è formalizzata dall'implicazione: α è più forte di β se $\alpha \implies \beta$. La condizione più forte sarà `false` mentre la più debole `true`. Un modo intuitivo per pensare alla forza/debolezza di due condizioni è confrontare la cardinalità degli insiemi dei modelli: la condizione più forte è quella resa vera da meno valori. Le operazioni logiche modificano la forza della formula risultato in questo modo:

1. Disgiunzione: $\alpha \implies (\alpha \lor \beta)$ (indebolisce)
2. Congiunzione: $(\alpha \land \beta) \implies \alpha$ (rafforza)
3. Implicazione: $\beta \implies (\alpha \implies \beta)$ (indebolisce)

Regola delle proprietà

Per dimostrare che si rispetta la regola delle proprietà occorre mostrare che tutti i metodi nuovi o ridefiniti, inclusi i costruttori, del sottotipo conservano le proprietà invarianti e le proprietà evolutive del sopratipo osservabili con i metodi pubblici *observer* della sopraclasse.

Programmazione distribuita

Socket TCP

La programmazione socket su TCP viene eseguita tramite l'uso delle classi `Socket` e `ServerSocket`. La connessione è rappresentata da un'istanza della classe `Socket`: il client la crea tramite `new`, il server la riceve da `ServerSocket.accept()` che la ritorna appena un client esegue una richiesta di connessione.

La comunicazione avviene tramite due stream di array di byte: un input stream di ottenuto tramite `Socket.getInputStream()` e un output stream ottenuto tramite `Socket.getOutputStream()`. Questi stream possono essere convertiti in stream testuali tramite `Scanner` e `PrintWriter` per rispettivamente input e output.

La connessione viene chiusa tramite il metodo `Socket.close()`. Per `ServerSocket` la `close()` termina la `accept()` e lancia una `IOException`.

RMI

Introduciamo della terminologia:

1. **Oggetto remoto**: un oggetto i cui metodi possono essere invocati da una JVM diversa da quella dove l'oggetto risiede.
2. **Interfaccia remota**: interfaccia che dichiara quali sono i metodi che possono essere invocati da una diversa JVM.
3. **Server**: insieme di uno o più oggetti remoti che, implementando una o più interfacce remote, offrono risorse a macchine esterne distribuite sulla rete.

Il client colloquia con un proxy locale del server detto *stub* che ne implementa l'interfaccia ed è capace di fare forward delle chiamate di metodo. Esiste il corrispettivo anche a lato server detto *skeleton*. Gli argomenti dei metodi subiranno un processo di serializzazione (marshalling) per poter essere trasmessi sulla rete.

Il registro RMI (*registry*), si occupa di fornire al client lo stub richiesto: in fase di registrazione il server potrà fornire un nome canonico per il proprio oggetto remoto e il client potrà reperirlo tramite suddetto nome. Se client e server risiedono sulla stessa macchina, è possibile indicare al client il path locale per lo stub, s invece risiedono su macchine differenti è necessario utilizzare un server HTTP per permette al registro di spedire lo stub. Il registry girerà su un processo a parte e dovrà essere lanciato prima dell'esecuzione del nostro server.

Testing

Verifica e validazione

Definiamo rigorosamente i termini che useremo:

1. **Verifica**: assicurarsi che il software sia costruito correttamente (stiamo rispettando la specifica?)
2. **Validazione**: assicurarsi che il software faccia ciò che l'utente ha relamente chiesto (stiamo costrurendo il prodotto giusto?)

Al processo di verifica e validazione dovrà essere sottoposto ogni artefatto prodotto dallo sviluppo e anche l'attività di verifica stessa!

Possiamo effettuare 2 tipi di verifiche: verifiche statiche e dinamiche:

1. **Verifiche dinamiche** - Le verifiche dinamiche consistono principalmente nel collaudo del software. Il sistema viene eseguito con dati di test e si osserva il suo comportamento confrontandolo con i risultati attesi.
2. **Verifiche statiche** - Le verifiche statiche sono test eseguiti a mano o automaticamente su semilavorati non eseguibili.

Le verifiche dinamiche non garantiscono l'assenza di errori (vedi API). Alcuni requisiti/semilavorati, inoltre, possono essere verificate solo con una delle due metodologie (ad esempio la performance e altre proprietà di runtime possono essere verificate solo tramite verifiche dinamiche e semilavorati non eseguibili possono essere verificati solo staticamente) perciò essere vanno usate congiuntamente.

Analisi statica

Uno dei metodi per eseguire verifiche statiche è l'analisi statica, ossia un insieme di verifiche automatiche su documenti non eseguibili. L'analisi statica può generare informazione utilizzabile durante una eventuale ispezione umana del documento. Le sacre scritture di API ci rivelano che l'analisi automatica è molto limitata poiché molte proprietà dei programmi che vorremmo analizzare sono indecidibili. Perciò possiamo adottare 2 approcci per raggirare la limitazione dell'indecidibilità:

1. Usare un **approccio rigoroso**, ossia segnaliamo solo gli errori certi, tralasciando quelli possibili.
2. Usare un **approccio pessimistico**, ossia segnalare tutti gli errori, anche quelli possibili.

Alcuni tipi di analisi statica sono il linting (ricerca di errori nella correttezza del programma come utilizzo di variabili non inizializzate o pezzi di codice irraggiungibile o inutilizzato) e model checking.

Analisi dinamica

L'analisi dinamica, o testing, si applica al risultato finale dello sviluppo a programmi eseguibili e ha lo scopo di far emergere quanti più difetti possibile. Esistono diversi tipi di test, detti livelli di test, eseguiti in momenti diversi dello sviluppo e con obiettivi diversi:

1. **Test di unità** - Si occupa di testare un singolo modulo, ossia la più piccola unità del software. Generalmente è scritto da un solo programmatore o un piccolo team che lo ha anche ideato poiché la qualità del test dipende dalla

sua esperienza e dalla conoscenza interna del modulo. Tipicamente viene eseguito dal programmatore prima del rilascio.

2. **Test di integrazione di sottosistema** - Corrispondente alla fase in cui è disponibile il progetto dettagliato, si occupa di testare l'integrazione tra i moduli di uno stesso sottosistema. In alcuni casi questa fase può essere unita a quella del testing di unità.
3. **Test di integrazione** - Corrispondente alla fase in cui è disponibile il progetto di alto livello, si occupa di testare l'integrazione tra i vari sottosistemi che formano il sistema. Viene effettuato da un gruppo di testing apposito di cui tipicamente non fanno parte i programmatori. Le interfacce dei moduli vengono testate una alla volta tramite *driver* e *stub* che sono rispettivamente codice per chiamare le interfacce e codice per simulare interfacce su cui dipende il sottosistema.
4. **Test di sistema e di accettazione** - Corrispondente alla fase di specifica del sistema software, si occupa di verificare che l'intero sistema funzioni come specificato e come desidera l'utente. Viene detto test di sistema se condotto dal produttore per verificare la qualità, test di accettazione se condotto dal committente.

   A volte non esiste un committente, perciò non si può parlare di test di accettazione. In questo caso si usa il cosiddetto beta test: il sistema viene rilasciato in prova ad un certo numero di utenti selezionati che lo provano e segnalano eventuali errori. In contrasto si chiama alpha test il testing effettuato dal produttore.
5. **Test di non regressione** - Condotto post-release, server a verificare che le caratteristiche positive consolidate da una versione precedente del software persistano anche nelle versioni nuove.

Il testing ha come compito quello di trovare eventuali difetti, non identificare il vero e proprio difetto che sta all'origine dei malfunzionamenti. Il testing deve essere ripetibile in modo da poter permettere un debugging.

Come sappiamo da API, il testing non è capace di trovare tutti i difetti ma può solo limitarsi a trovarne il più possibile. Dobbiamo, perciò, adottare una strategia di testing che ci permetta di sfruttare intelligentemente le risorse per massimizzare la probabilità che un errore si verifichi se esso esiste.

Testing black-box

Esistono diverse modalità di testing possibili: si può decidere di collaudare un oggetto ignorando la sua organizzazione e il suo funzionamento interno, oppure possiamo cercare di sfruttare queste conoscenze. Il primo viene detto *black box testing* mentre il secondo *white box testing* o *testing strutturale*.

Nel black box testing il programma è considerato una scatola nera, ossia osserveremo solo input e output. I casi di test sono scelti a partire da un'analisi delle specifiche dei requisiti: sceglieremo i dati in modo da sollecitare tutte le funzioni previste in tutte le condizioni di applicazione previste. Poiché esso dipende solo dalle specifiche, può essere programmato presto nella pipeline di sviluppo. Inoltre, affinché sia efficace, il testing black box richiede che le specifiche siano il più precise e formali possibili er rimuovere ogni ambiguità sul funzionamento.

Esistono diverse tecniche per identificare i casi di test nella metodologia black box:

1. **Testing guidato dalla sintassi** - È più utile per programmi che intetpretano stringhe di testo secondo una grammatica formale. L'idea è usare le stesse regole che il programma da testare usa per riconoscere le stringhe per generare casi di test. Ovviamente la grammatica farà parte della specifica.
2. **Testing guidato dagli scenari d'uso** - È possibile derivare i casi di test dalla specifica degli scenari d'uso (ad esempio UML Use Case diagram). Per ogni caso d'uso si possono elencare tutti i possibili ingressi che lo caratterizzano e provare il comportamento del programma a fronte di tali casi.
3. **Derivazione semiautomatica dalle specifiche** - Se vengono usati linguaggi formali di specifica basati su logica temporale o FSA, è spesso possible ricavare automaticamente casi di test con dati di input e dati di output previsti.

Il **testing guidato da boundary conditions** si basa sul concetto di classi di equivalenza, ossia degli insiemi di valori per il quale un programma si comporta in modo simile, e sull'assunzione che errori tipici si concentrino in prossimità del confine tra queste. È quindi opportuno aggiungere tra i vari casi di test alcuni casi che vanno a verificare una corretta implementazione per i casi limite.

Un modo piuttosto comune per determinare un insieme minimo di casi di test si basa sull'utilizzo dei cosiddetti grafi causa-effetto. Questi grafi mettono in relazione dei fatti elementari, espressi come espressioni booleane, con i risultati attesi. Essi sono, inoltre, ricavabili dalle specifiche. Verrà assegnato ai nodi alle estremità (a sinistra gli input e a destra gli output) un significato preciso dipendente dalle specifiche. Conoscendo La dipendenza di un risultato dalla combinazione

degli ingressi potremo stimolare il sistema esattamente con tale combinazione e verificare se il risultato reale coincide con quello atteso.

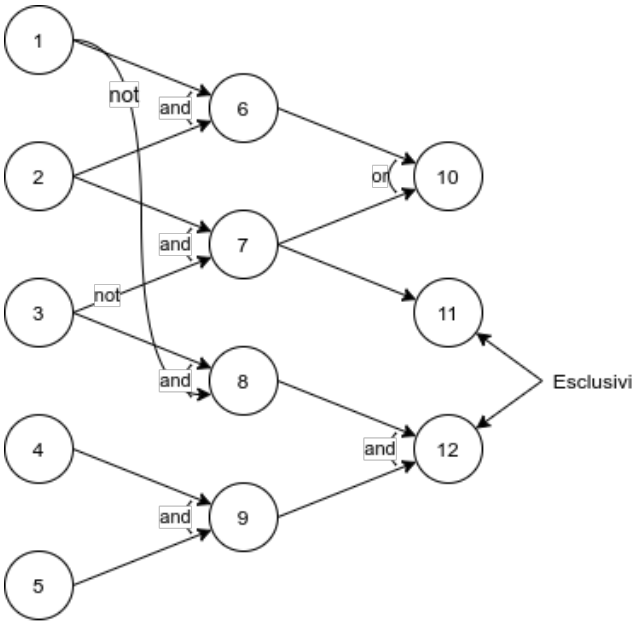

Figure 1: Grafo di causa-effetto

Talvolta risulta conveniente usare tabelle così organizzate:

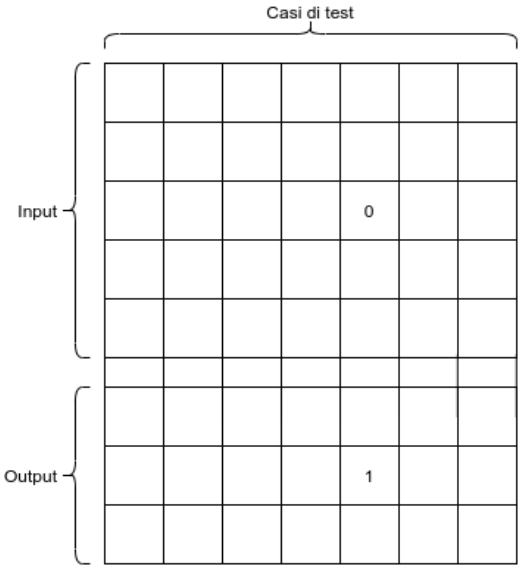

Figure 2: Tabella equivalente ad un grafo causa-effetto

Il testing black box si può effettuare sempre purché siano note le specifiche. Tuttavia è particolarmente utile per test di modulo, d'integrazione e di sistema/accettazione.

Testing strutturale

Il testing strutturale (o *white box*), a differenza del testing black box, tiene conto della struttura interna del programma. I casi di test vengono infatti derivati da un'analisi diretta dell'implementazione. L'obiettivo è sollecitare tutte le parti del programma. L'applicazione di questa metodologia in modo esaustivo è un procedimento molto costoso, perciò di solito viene effettuato solo su regioni critiche del codice.

Per il testing strutturale esistono diverse strategie, una più approfondita dell'altra. Più approfondita è la strategia, minore è la probabilità che non vengano rilevati errori.

Il il primo criterio è quello di testare tutte le istruzioni di cui è composto il codice, detto **statement coverage**. L'insieme dei casi di test verrà quindi selezionato in modo che ogni istruzione venga eseguita almeno una volta. Lo statement coverage purtroppo non è abbastanza rigoroso da assicurare la mancanza di errori. Questa limitazione è dovuta alla possibilità di non affrontare alcuni dei rami del grafo di decisione.

Per superare le limitazione dello statement coverage è stato introdotto il criterio di **edge coverage**. Questo criterio prevede che l'insieme di test debba essere definito in modo che ogni ramo del *control flow graph*, una semplificazione del diagramma di flusso del programma, venga attraversato almeno una volta. Per applicare il criterio dobbiamo prima costruire il control flow graph, abbreviato in CFG:

1. Una sequenza di istruzioni semplici diventa semplicemente un arco del CFG. Una sequenza di blocchi diventa un arco che collega i CFG corrispondenti ai blocchi.
2. Le istruzioni condizionali vengono espanse in due rami contenenti i CFG corripondenti ai due rami `then` e `else` (se è presente). Alla fine i due rami si ricongiungono in un nodo.
3. Per rappresentare i cicli usiamo due rami, uno contiene il CFG del corpo del ciclo e ricongiunge al nodo di partenza e l'altro ramo rappresenta l'uscita dal ciclo.

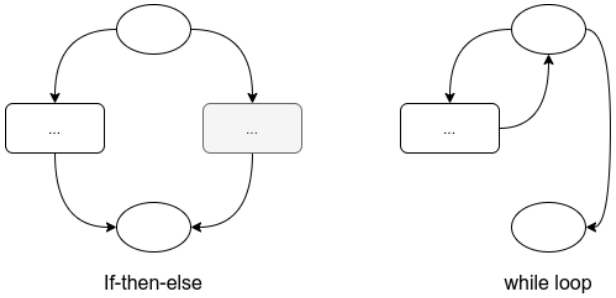

Figure 3: CFG dei 3 blocchi condizionali fondamentali

Purtroppo neanche la edge coverage basta per garantirci l'assenza di errori. Il problema in questo caso è dovuto al fatto che non distinguiamo come le condizioni vengano soddisfatte: `x < 0 || y > 0` può essere soddisfatta sia se `x < 0` oppure `y > 0`. Dobbiamo perciò espandere le condizioni composte e fornire a ciascuno degli atomi che la compongono un ramo. Questa espansione del criterio di edge coverage è detto di **edge and condition coverage**. Purtroppo non abbiamo ancora raggiunto la massima sicurezza contro gli errori: non stiamo assicurando di coprire ogni singolo cammino possibile. L'ultimo criterio, e anche il più costoso, è quello di **path coverage** e richiede, appunto, che l'insieme dei casi di test percorra ogni possibile cammino che porti dal nodo iniziale a quello finale una o più volte. Il numero di casi di test applicati con il path coverage esplode (servono 2^n casi di test per gestire n condizionali in serie) e quindi non è applicabile nella pratica. Nonostante ciò è l'unico dei criteri che può garantire l'assenza di errori.

Abbiamo visto vari criteri di definizione dei casi di test, ognuno che assicura una sicurezza e una copertura maggiore a discapito del numero dei casi di test. In generale occorre individuare accuratamente le porzioni di codice critiche per la correttezza del sistema, solo per queste useremo testing strutturale con un livello di copertura desiderato. Il livello di copertura verrà stabilito tramite metriche adeguate non trattate.

Test d'integrazione

Nel test di integrazione vengono osservati i sistemi e i sottosistemi in composizione. Conviene che sia un testing di tipo black-box visto che si sta studiando solo l'integrazione dei componenti e non la loro struttura.

Il testing strutturale fa uso di moduli placeholder: gli stub e i driver. Gli stub simulano l'interfaccia di un componente che non è ancora disponibile. I driver sono il complementare: simulano le chiamate di un utilizzatore non ancora implementato.

Per eseguire testing d'integrazione abbiamo due strategie possibili:

1. "Big bang" - Consiste nell'integrare tutti i componenti in una volta. Ciò rende più difficoltoso il testing, soprattutto in progretti complessi dove il numero di moduli e interfacce e numeroso.

La strategia “big bang” richiede numerosi stub e driver poiché i moduli devono essere anche testati in isolamento prima dell’integrazione. Le anomalie emergono solo dopo la pubblicazione di tutti i moduli. Inoltre viene resa difficoltosa la localizzazione degli errori.

2. Incrementale - Per ovviare ai problemi della strategia “big bang” si ricorre ad un approccio incrementale: i moduli vengono testati man mano che vengono scritti, integrandone un numero progressivamente maggiore.

   Ci sono due possibili modi di procede:

   1. Approccio bottom-up - Si sviluppano prima i moduli di più basso livello e si testano con i driver necessari. Si procede poi verso l’alto, introducendo ad ogni livello i nuovi moduli e i driver necessari.
   2. Approccio top-down - Si sviluppano prima i moduli più di alto livello e si testano con gli stub necessari. Si procede poi verso il basso, introducendo ad ogni livello i nuovi moduli e gli stub necessari.

   L’approccio incrementale richiede meno driver/stub rispetto all’approccio “big bang” e grazie all’integrazione immediata dei moduli le anomalie possono essere scoperte immediatamente. Il passo incrementale inoltre aiuta con la localizzazione degli errori e permette ad alcuni moduli core di essere testati il più a lungo possibile.

Test delle interfacce

L’ultima tipologia di test che vedremo è il test delle interfacce. Esso avviene quando si integrano più moduli per produrre sistemi più ampi. L’oggetto dell’analisi sarà il corretto utilizzo delle interfacce dei moduli, quindi il corretto passaggio dei parametri, l’utilizzo dell’interfaccia per lo scopo corretto, errori di sincronizzazione e violazione di vincoli real-time.