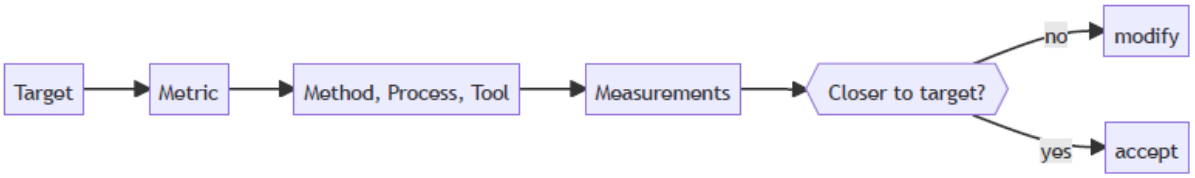
Ingegneria del Software - Riassunto - Simone Gioè

Processo di Produzione del software: Approccio ingegneristico



Ha dei problemi riguardo alle persone coinvolte (cliente != programmatore), dimensioni del software, malleabilità del software (che porta a diverse versioni e/o evoluzioni).
Qualità ricercate in un software:

- Correttezza
- Affidabilità
- Innocuità e/o Robustezza
- Usabilità
- Efficienza
- Verificabilità
- Riusabilità
- Manutenibilità (Riparabilità e Evolvibilità)

. Analogamente un processo deve godere delle qualità di:

- Robustezza (resistere a imprevisti)
- Produttività (essere veloce)
- Tempismo (cogliere l'attimo)

. Produzione del software non è solo scrivere codice, ma bisogna anche risolvere il problema della comunicazione ed essere rigorosi (formali, in quanto i metodi formali riducono gli errori di design o li eliminano presto). A seguito vi sono le fasi di produzione del software.

Studio di fattibilità

- Definizione preliminare del problema: possibile mercato e concorrenti-.
- Studio di diversi scenari di realizzazione: scelte architetturali e hardware necessario, sviluppo in proprio o subappalto.
- Stima dei costi, tempi di sviluppo, risorse necessarie e benefici delle varie soluzioni.
- Spesso difficile fare un'analisi approfondita: spesso viene commissionata all'esterno o si hanno limiti di tempo troppo stringenti.

Produce spesso un documento in linguaggio naturale.

Analisi e specifica dei requisiti

- Comprendere il dominio applicativo.
- Identificare gli stakeholders.
- Identificare le funzionalità richieste: cosa deve fare il sistema (non come).

Può produrre un output di vario genere:

- Documento di specifica: documento contrattuale approvato dal committente, base per il lavoro di design e verifica, importanza di avere una documentazione formale.
- Manuale utente o maschere di interazione: vista esterna per eccellenza.
- Piano dei test di sistema: collaudi che certificano la correttezza.

Progettazione (Design)

E' il processo di come realizzare in maniera opportuna le specifiche precedentemente trovate e come definire l'architettura del sistema (scelta di un'architettura software di riferimento, scomposizione in

moduli o oggetti, identificazione di patterns). Produce un output che può essere un documento di specifica di progetto (può essere in diversi linguaggi).

Programmazione e test di unità
Vengono realizzate le blackbox definite in fase di progettazione e vengono testati indipendentemente i singoli moduli (utilizzando un framework per il test in grado di creare moduli stub, ovvero fittizi, e moduli driver, ovvero guida).
Questa fase ha come output un insieme di moduli sviluppati separatamente, con un'interfaccia concordata e singolarmente verificati.

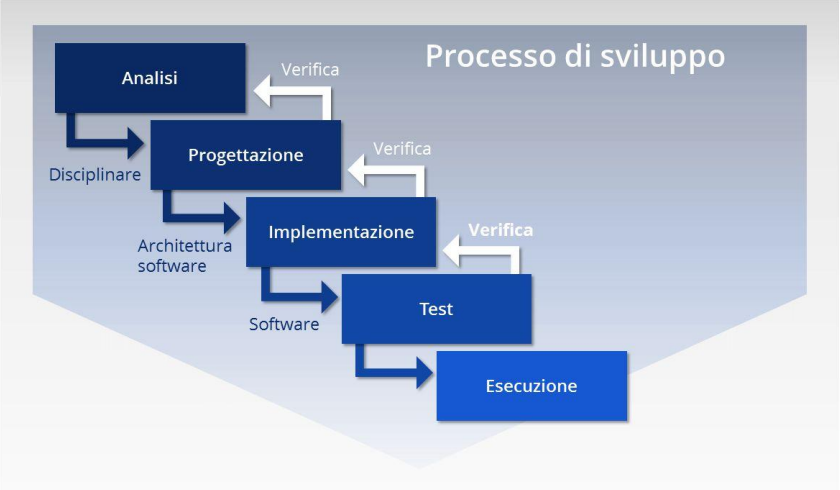
Integrazione e test di sistema
Vengono uniti i vari componenti e si effettua un test di integrazione: vengono man mano sostituiti i moduli di testing con i moduli reali (può essere bottom-up o top-down).

Manutenzione
- Correttiva
- Adattiva
- Perfettiva

Che produce come output un prodotto migliore.

Modelli di ciclo di vita del software

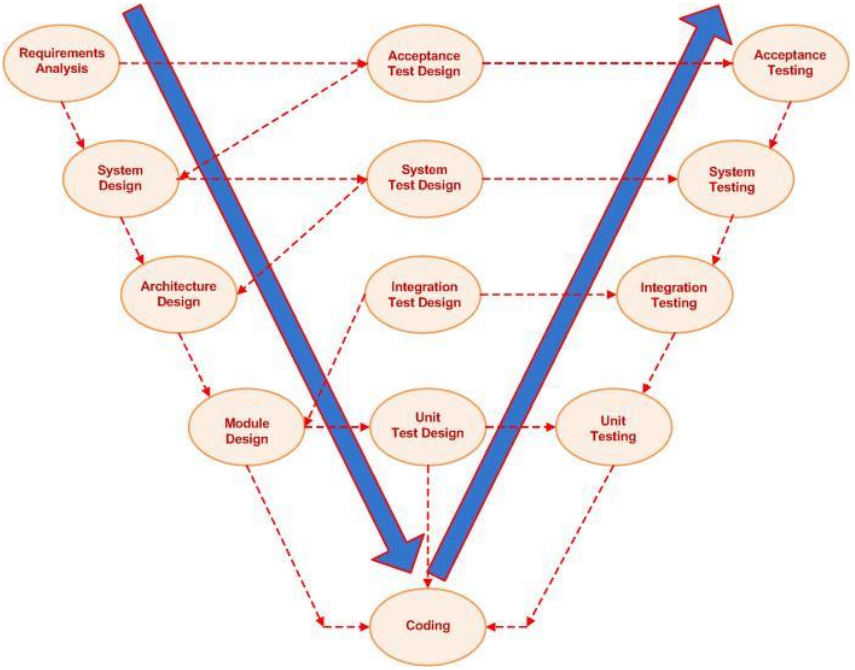
Modello a cascata
Forza una progressione lineare da una fase alla successiva. Le varie fasi comunicano attraverso semilavorati, permettendo dunque una separazione dei compiti. E' possibile fare una pianificazione dei tempi e monitoring dello stato di avanzamento, ma a senso unico.



Nel modello a cascata la manutenzione non è una fase prevista, è considerata un'eccezione. Nel modello a cascata non è possibile infatti tornare indietro e modificare specifiche, codice, documentazione.
E' dunque un modello molto rigido, nel quale le specifiche e le stime sono fatte solo nelle prime fasi. E' anche monolitico, in quanto tutta la pianificazione è orientata ad un singolo rilascio e la manutenzione è fatta solo sul codice.

Modello a V

Il modello a V espande la fase di testing e chiarisce alcune relazioni tra ogni fase del ciclo di vita dello sviluppo del software e la sua fase di testing.



Varianti del processo a cascata

- Con singola retroazione:
  Una fase può portare a modifiche nella fase precedente (iterazione)
- Modello di ciclo di vita a fontana:
  Rinizia “da capo”, incrementale.

Modelli iterativi vs incrementali

Si parla di incrementale quando nelle iterazioni viene inclusa la consegna.

- Implementazione iterativa: stressa la modularizzazione e l’identificazione di sottosistemi, si ripetono fasi di coding ed integrazione.
- Sviluppo incrementale: viene esteso a tutte le fasi (specifiche incluse), viene eseguito il modello di produzione per ogni incremento. La fase di manutenzione (in senso stretto del termine) sparisce, diventando semplicemente riciclo del prodotto.

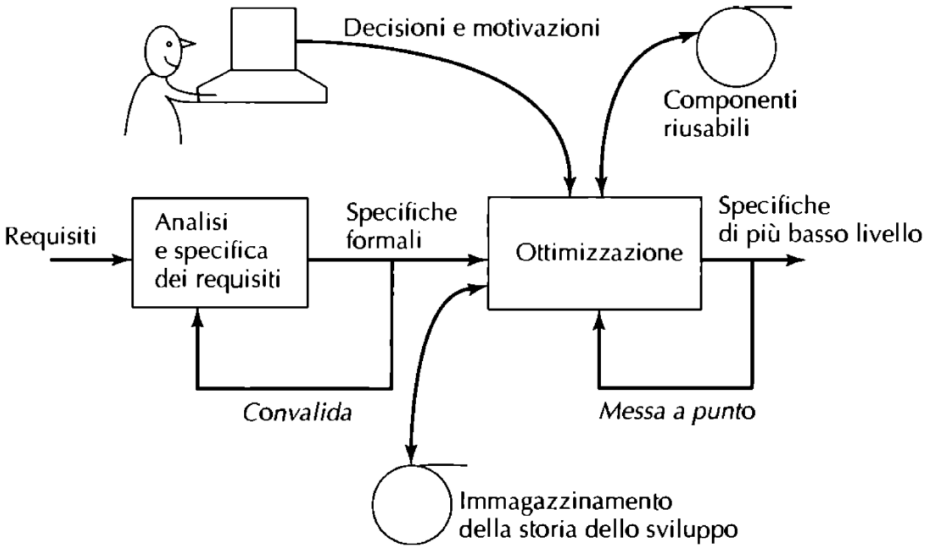
I modelli incrementali hanno però diversi problemi:

- Viene complicato il lavoro di pianificazione (stato del processo è meno visibile, necessità di pianificare tutte le iterazioni)
- Si riconosce che bisogna rimettere mano a ciò che si è fatto (il sistema può non convergere)
- Problema del definire un’iterazione e la sua durata.

//Pinball life cycle

Modelli trasformazionali

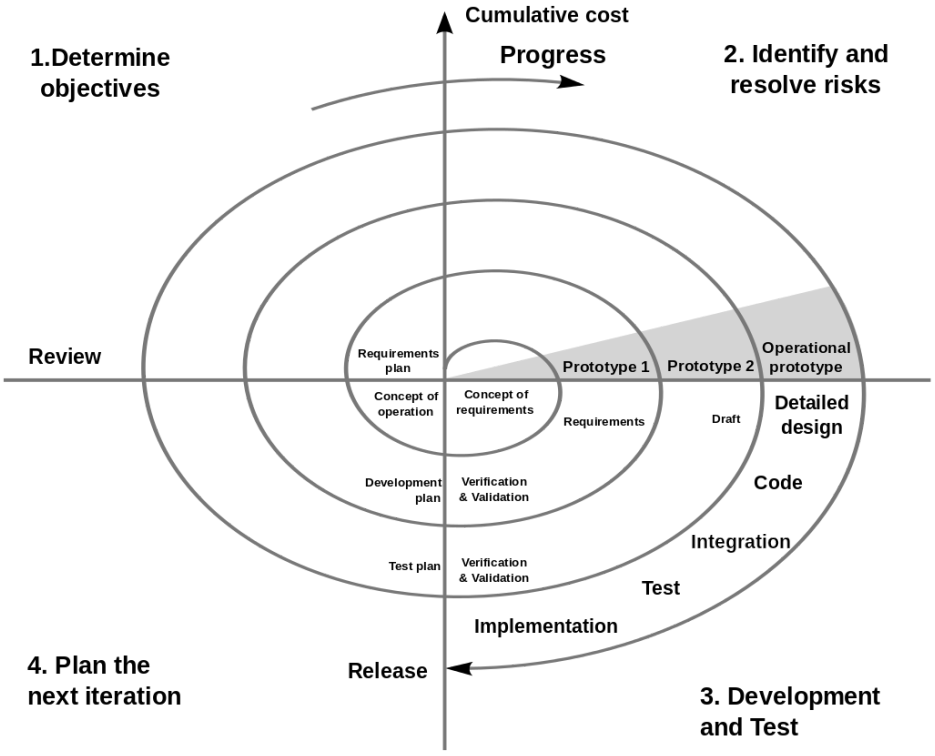
Raffinamenti di rappresentazione formali del problema, che ad ogni passo vengono specializzate, ottimizzate, rese più concrete attraverso trasformazioni automatiche o controllate.
A partire da specifiche formali si ottiene un prototipo, che differisce dal prodotto finale per efficienza e completezza. Vengono svolti passi di trasformazioni (formalmente dimostrabili come corretti) che portano verso l’ottenimento della versione finale.



Metamodello a spirale

E' un framework in cui si possono inquadrare altri modelli. E' guidato dall'analisi dei rischi e si compone delle seguenti fasi:

- Determinazione di obiettivi, alternative e vincoli.
- Valutazione delle alternative, identificazione dei rischi.
- Sviluppo e verifica.
- Pianificazione della fase successiva.



Il modello spirale win-win evidenzia le comunicazioni con i clienti che necessitano contrattazioni e negoziazione.

Il modello COTS (Component Off The Shelf), basato sul partire dalla disponibilità interna o sul mercato di moduli preesistenti e creare il sistema a partire da quelli mediante integrazione.

Metodologie Agili (eXtreme Programming)

eXtreme Programming (XP)
Basato sul concetto di “Increment then simplify”. Prevede uno sviluppo guidato dai test, ovvero il Test Driven Development (TDD). Il TDD è una tecnica di progettazione che guida verso il design più semplice, riassumibile come “test-first + baby steps”. Le fasi del TDD sono le seguenti:

- Scrittura di un test fallente (ROSSO)
- Fare passare il test scritto (VERDE)
- Rifattorizzare il codice ove necessario (REFACTORING)

, il tutto ripetuto ogni 2-10 minuti.
I principi dell’eXtreme Programming sono i seguenti:

- Feedback rapido
- Presumere la semplicità
- Accettare il cambiamento
- Modifica incrementale
- Lavoro di qualità

. Le figure in gioco nell’XP sono il Manager e/o Cliente e lo sviluppatore, ciascuno con le rispettive responsabilità.

Approcci richiesti da XP
XP richiede i seguenti approcci:

1. Planning game: una versione semplificata ed informale degli Use Cases di UML. Vengono determinate le funzionalità del prossimo rilascio, combinando priorità commerciali e valutazioni tecniche. //carte, team estimation game, anchoring effect, planning game, velocity
2. Brevi cicli di rilascio.
3. Uso di una metafora: serve come vista aggregante, fornisce nuovi elementi di discussione e un nuovo vocabolario con cui parlare con l’utente o con i nuovi arrivati. Dovrebbe inoltre permettere la nominazione di classi e metodi.
4. Semplicità di progetto: contrapposto al “Design for change”, che viene visto come un appesantimento inutile.
5. Testing: clienti scrivono test funzionali, programmatori scrivono test di unità.
6. Refactoring: modifiche al codice che non cambiano le funzionalità. Non bisogna avere paura di apportare modifiche semplificanti o che facilitano l’inserimento di funzionalità.
7. Pair programming: aiuta ad avere un controllo maggiore sul rispetto delle regole dell’XP, aiuta l’inserimento di nuovo personale e la sua formazione, aiuta ad ottenere proprietà collettiva del codice (conoscenza osmotica) e aiuta il refactoring.
8. Proprietà collettiva: il codice non è di una sola persona, ma di tutti. I cambiamenti al codice devono essere responsabilizzati sull’intero codice, anche non conoscendo tutto in egual modo.
9. Integrazione continua: integrazione svolta più volte al giorno, in modo da ottenere feedback rapidi. La coppia dopo aver risolto il problema in piccolo, deve anche risolvere i relativi problemi di integrazione.
10. Settimana di 40 ore: personale fresco, soddisfatto e meno stressato.
11. Cliente sul posto: permette una fase di specifica leggera, fonte in tempo reale di scelte e test funzionali, ma non è rappresentativo di tutti gli stakeholder. Viene visto spesso dai committenti come un problema, in quanto si toglie a loro una risorsa.
12. Standard di codifica: enfatizza la comunicazione attraverso il codice, aiutando refactoring, pair programming, proprietà collettiva.

Nel caso sia proibito anche uno solo dei precedenti approcci, non si può usare l’XP.

Fasi dell’XP

L’XP si divide nelle seguenti fasi:

1. Requirements:
    1.1. Gli utenti fanno parte del team di sviluppo.
    1.2. Consegne incrementali e pianificazioni continue.
2. Design:
    2.1. Una metafora come visione unificante di un progetto.
    2.2. Refactoring.
    2.3. Presumere la semplicità.
3. Code:
    3.1. Pair programming.
    3.2. Proprietà collettiva.
    3.3. Integrazioni continue.
    3.4. Standard di codifica.
4. Test:
    4.1. Testing di unità continuo (da scrivere prima del codice).
    4.2. Test funzionale scritto dagli utenti.

In XP la documentazione è nel codice (test di unità) e nelle persone (cliente, compagno di pair programming).

Open Source Process

Free and Open Source Software (FOSS)

Un software open source vive le seguenti fasi:

1. Invenzione: qualcuno ha un’idea e la fa funzionare.
2. Espansione ed innovazione: il mondo viene a conoscenza dell’invenzione e la tecnologia inizia a espandersi rapidamente.
3. Consolidazione: pochi dei alcuni progetti si rivelano più dominanti e prendono vantaggio. I progetti minori sono incorporati o falliscono.
4. Maturità: il mercato viene ridotto a pochi prodotti. Diventa impossibile per nuovi competitor entrare nel mercato.
5. Dominio del FOSS: la comunità del FOSS inizia inesorabilmente a erodere il vantaggio tecnico delle offerte commerciali.
6. L’era del FOSS: infine la versione FOSS domina.

Il team utilizza di un FOSS utilizza strumenti di supporto per:

- Comunicazione: attraverso siti internet e/o forum in modo tale da mantenere la community unita e rispondere ai dubbi delle new entry.
- Sincronizzazione del lavoro e versioning: del codice e della documentazione.
- Automatizzazione della build.
- Bug tracking.

Software Configuration Management

Software Configuration Management

Il SCM è un 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 istante in uno stato (configurazione) ben definito. Gli oggetti di cui si controlla l’evoluzione sono detti artifact (manufatti).

Gli artifact classicamente sono file. L'SCM permette di tracciare/controllare le revisioni degli artifact e le versioni delle risultanti configurazioni, a volte fornendo supporto per la generazione del prodotto a partire da una ben determinata configurazione.

Il meccanismo di base per controllare l'evoluzione delle revisioni è che ogni cambiamento è regolato da:

- Check-out: dichiara la volontà di lavorare partendo da una particolare revisione di un artifact (o una configurazione di diversi artifacts).
- Check-in (o Commit): dichiara la volontà di registrarne una nuova (spesso chiamata change-set).

Queste operazioni vengono attivate rispetto a un repository e producono dunque spostamenti tra il repository (che contiene tutte le configurazioni) e il workspace (l'ambiente su cui si lavora nel filesystem).

Il repository mantiene informazioni comprese date, etichette (tag), versioni, diramazioni (branches), eccetera. La repository può salvare solo le differenze tra una versione e l'altra e può essere centralizzata o distribuita.

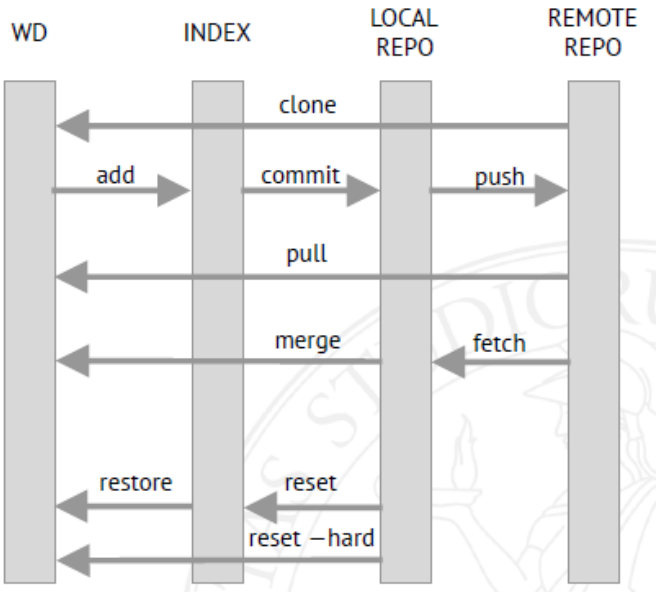
In una repository condivisa da un gruppo di lavoro è necessario gestire l'accesso concorrente. Abbiamo due modi per farlo:

- Modello "pessimistico" (RCS): il sistema gestisce l'accesso agli artifact in mutua esclusione attivando un lock al check-out.
- Modello "ottimistico" (CVS): il sistema si disinteressa del problema e fornisce supporto per le attività di merge di change-set paralleli potenzialmente conflittuali. Questo modello può essere parzialmente regolato tramite i rami paralleli di sviluppo (branch).

Git

Git è un SCM distribuito, ogni peer ha un repository e non c'è una sincronizzazione automatica. La sincronizzazione avviene attraverso dei merge. Permette di lavorare offline, velocemente, e con diverse modalità di lavoro:

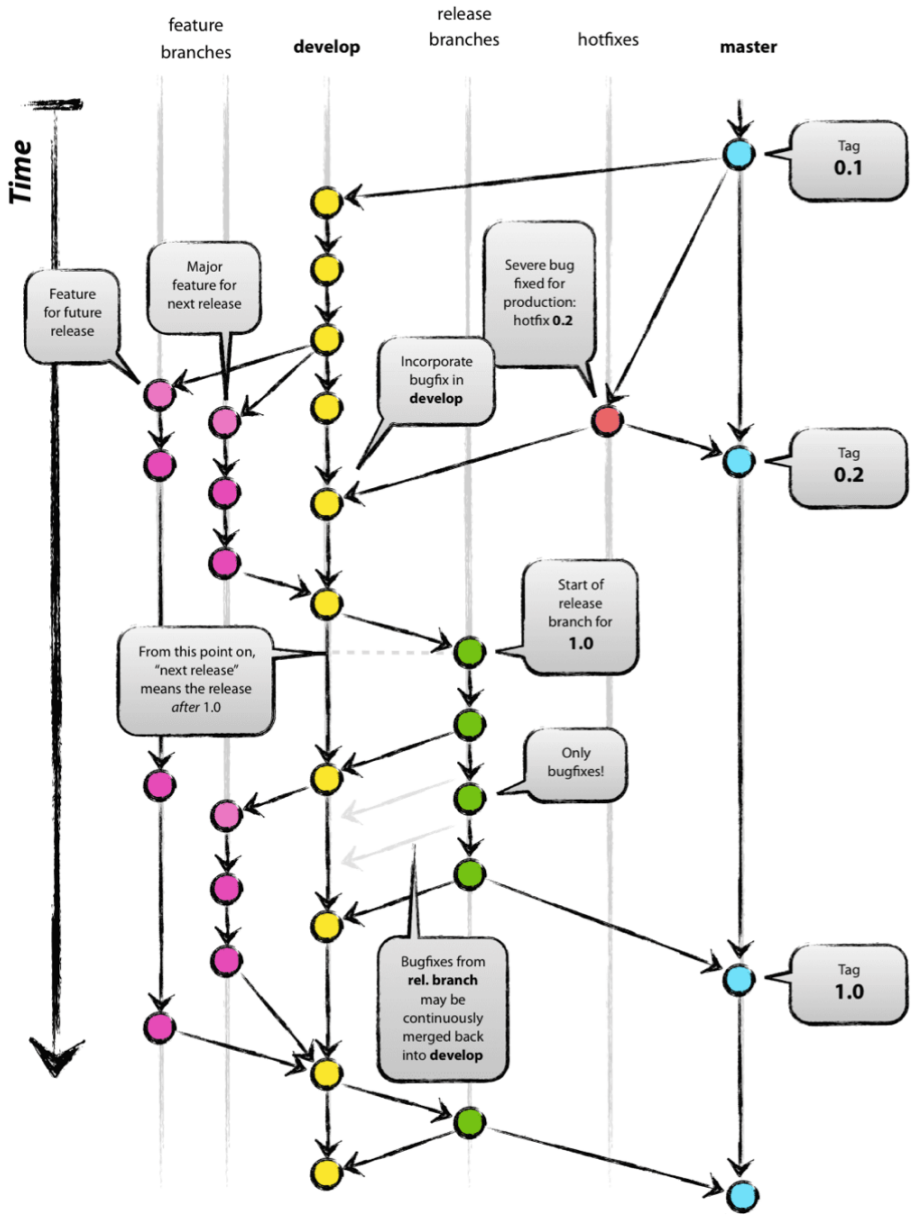
- Simil centralizzato: si ha un repository "di riferimento".
- Due peer che collaborano direttamente.
- Gerarchico a più livelli.



//Git cheat sheet: http://ndpsoftware.com/git-cheatsheet.html#loc=index

Git workflow

Per differenziare vari tipi di branch.



I rami master e develop sono due rami con “vita infinita”. Il ramo master contiene le versioni stabili e pronte alla consegna, il ramo develop è il ramo di integrazione.

Attraverso git flow possono essere iniziate e concluse feature e release con i rispettivi comandi: git flow feature start/finish <nomefeature>, git flow release start/finish <versione>.

Il ramo di hotfix è un ramo in cui vengono effettuate riparazioni veloci di difetti urgenti senza aspettare la prossima release.

Il fork risolve problemi di autorizzazioni, permettendo di mantenere legami tra repository su sito di hosting, ma con owner e autorizzazioni diversi. Inoltre ottimizza lo spazio occupato grazie alla condivisione dello spazio degli oggetti.

Tra la creazione e il deployment, vi deve essere una fase di review. A tale scopo vi sono due figure:

- Il verifier: verifica un cambiamento buildando e testandolo.
- L’approver: giudica se il cambiamento sia consono allo scopo e all’architettura del progetto, ragionando anche sul fatto che il cambiamento possa creare problemi nel futuro o meno.

Oltre ai tool di versioning si utilizzano anche tool di build automation, che si occupano di automatizzare la ricompilazione e il testing (Gradle) e tool di bug tracking, che si occupano di tracciare e gestire tutte le segnalazioni sui difetti di un software.

Refactoring e Design Knowledge

Refactoring
Il refactoring serve a migliorare il design del codice senza cambiarne le funzionalità. Viene principalmente usato per:

- Migliorare un design inizialmente tenuto “semplice” o nato male.
- Preparare il design per una funzionalità che non si integra bene in quello esistente.
- Eliminare debolezze (debito tecnico).

Design knowledge
La design knowledge può essere salvata in:

- Memoria
- Documenti di design (linguaggio naturale o diagrammi)
- All’interno di piattaforme di discussione, issue management, version control.
- Con modelli specializzati (UML). Può portare a approcci generative programming.
- Nel codice, ma spesso risulta difficile rappresentare le ragioni.

e può essere condivisa attraverso:

- Metodi
- Design pattern
- Principi

Alcuni termini e concetti utili

- Object orientation:
  - Ereditarietà
  - Polimorfismo
  - Collegamento dinamico
- Principi SOLID:
  - Single responsibility
  - Open close principle
  - Liskov substitution principle
  - Interface segregation
  - Dependency inversion
- Reference escaping: violazione dell’encapsulation.
- Encapsulation e information hiding: per facilitare la comprensione del codice e renderlo più modificabile senza effetti collaterali.
- Immutabilità: non vi è modo di cambiare lo stato dell’oggetto dopo la sua inizializzazione.
- Code smell: segnali che indicano necessità di refactoring.
- Principio Tell-Don’t-Ask: non chiedere i dati, ma dì cosa vuoi che faccia sui dati (minimizzare getter studiando cosa fare con il valore ritornato e definire funzioni opportune).

Noun extraction
Tecnica di identificazione delle classi e delle relazioni basato sulle specifiche (ad esempio i commenti esplicativi delle User Story/Use Case). Vengono estratti tutti i sostantivi o frasi sostantivizzate e si considerano tutti i candidati, dopodiché si comincia a sfoltire. Si cercano poi le relazioni, probabilmente date dai verbi.

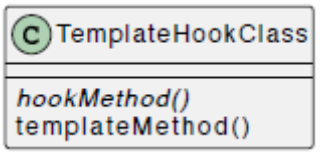
Patterns

Design patterns
I design patterns sono uno strumento concettuale che catturano la soluzione per una famiglia di problemi, esprimendo architetture vincenti.
Meta patterns identifica due elementi di base:

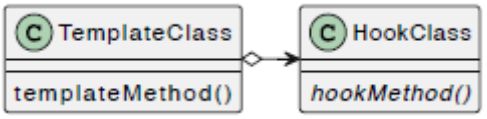
- HookMethod: metodo astratto che determina il comportamento specifico nelle sottoclassi (punto in cui si può intervenire per personalizzare, adattare lo schema).
- TemplateMethod: metodo che coordina generalmente più HookMethod (punto di invariabilità del pattern).

Hook e template possono relazionarsi in diversi modi:

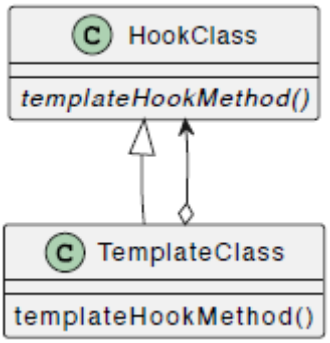
- Unification: template e hook sono nella stessa classe del framework.



- Connection: hook e template sono in classi separate indicate rispettivamente come hook class e template class, tra di loro collegate da un'associazione.



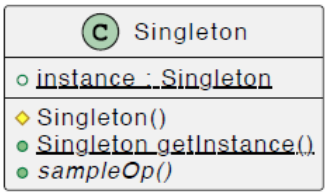
- Recursive connection: hook e template sono in classi tra di loro collegate anche tramite relazione di generalizzazione.



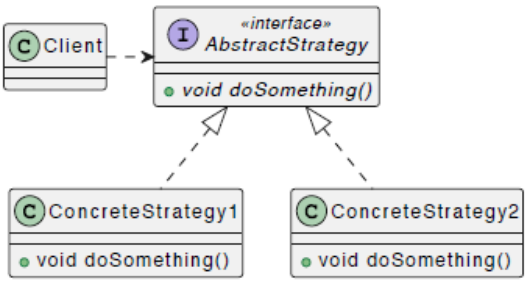
Gang of Four Patterns
Definiscono 23 pattern suddivisi in 3 categorie:

- Creazionali (creazione degli oggetti).
- Comportamentali (interazione tra gli oggetti).
- Strutturali (composizione di classi e oggetti).

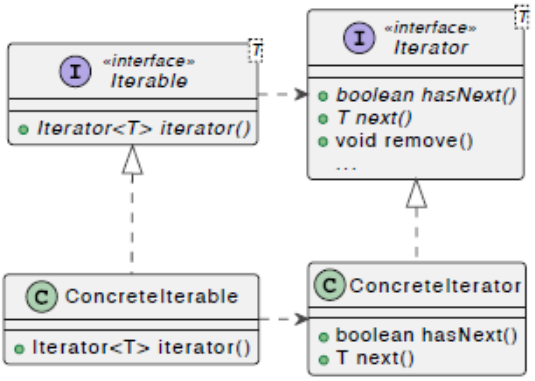
Singleton Pattern
Utilizzato quando si vuole avere un oggetto e non una classe in un linguaggio che fornisce solo classi. Si vuole rendere la classe responsabile del fatto che non può esistere più di una istanza. In java per creare singleton si utilizza enum.



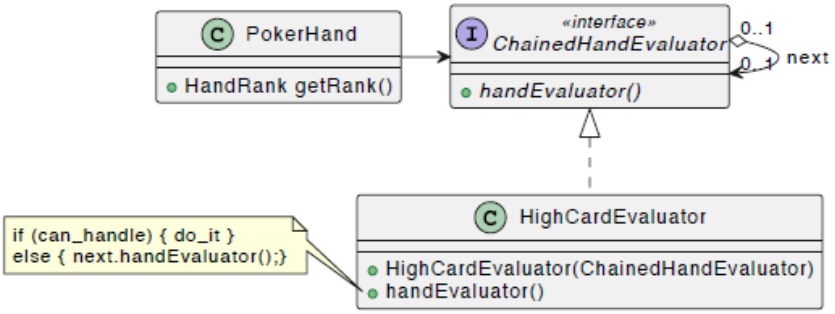
Delegation/Strategy
Definisce una famiglia di algoritmi e li rende (tramite encapsulation) tra di loro intercambiabili. Ad esempio sort di Collections che accetta sia Comparable che Comparator.



Iterator pattern
Fornisce un modo di accedere agli elementi di un oggetto aggregatore in maniera sequenziale senza esporre la rappresentazione interna.



Chain of Responsibility pattern
Permette di definire una catena di potenziali gestori di una richiesta di cui non sappiamo a priori chi sarà in grado di gestirla effettivamente.



FlyWeigth pattern
Serve a gestire una collezione di oggetti immutabili assicurandone l'unicità. Quando le istanze equivalenti sono fortemente condivise all'interno del programma diventano auspicabili sia immutabilità che unicità. Simile al singleton, ma a differenza di quello devo contenere più istanze e spesso non è definibile a priori quante sono.

```java
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()];
  }
}
```

NullObject pattern

Per evitare problemi, è meglio non usare null. Vogliamo dunque creare un oggetto che corrisponda al concetto di “nessun valore” o “valore neutro”.

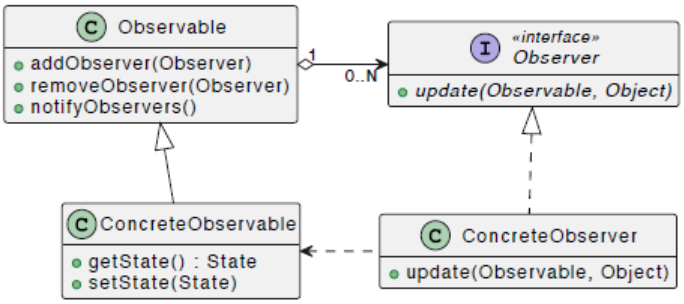
```java
public interface CardSource {
  Card draw();
  boolean isEmpty();

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

CardSource.NULL è un oggetto valido di un tipo anonimo che aderisce all’interfaccia CardSource, ma che ha particolari implementazioni per i vari metodi. Serve ad evitare di dover trattare separatamente il caso == null, ma se proprio diventasse necessario si può sempre testare == CardSource.NULL.

Observer pattern

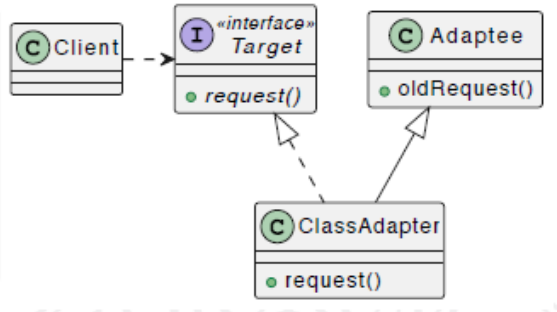
Per estrarre lo stato e metterlo in un oggetto a parte (Subject), che verrà osservato dagli altri (Observer). Lo stato modificato viene passato come argomento alla callback.



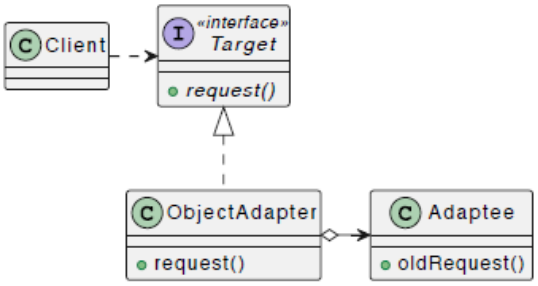
Adapter pattern

Usato per rendere compatibili più componenti. Ve ne sono di due tipi:

- Class Adapter: unico oggetto che può essere usato contemporaneamente con le due interfacce diverse (vecchia e nuova). Se un metodo non cambia, non devo fare nulla, ma potrei avere problemi con ereditarietà multipla.

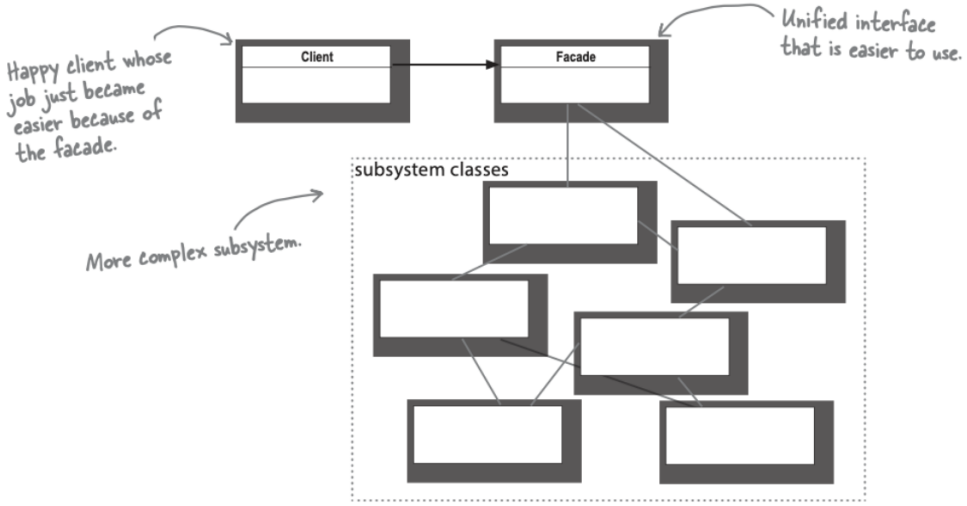


- Object Adapter: sono due oggetti distinti. L'oggetto nuovo non può più essere usato con l'interfaccia vecchia. Adatta un oggetto aderente a una interfaccia e non una classe, quindi in realtà adatta tutta una gerarchia di classi.



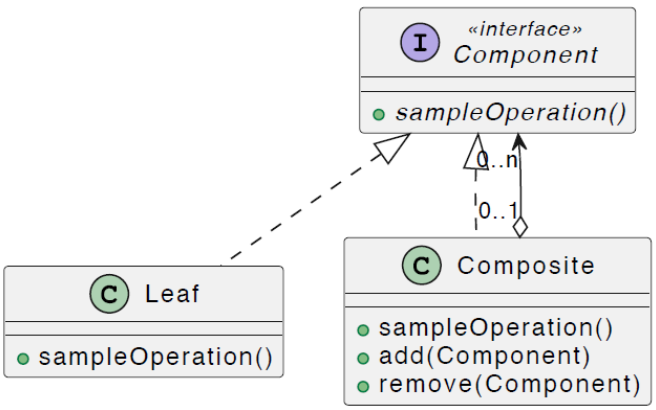
Facade pattern

Fornisce un'interfaccia unificata e semplificata a un insieme di interfacce separate.

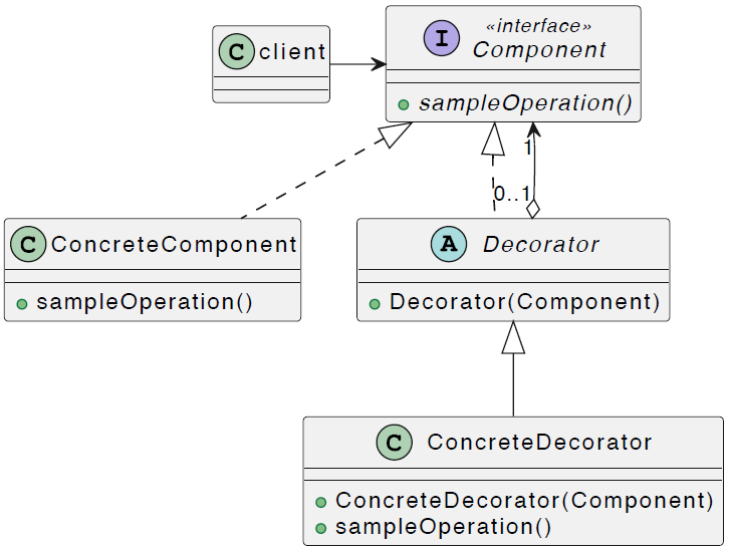


Composite pattern

Utile per gestire strutture ad albero per rappresentare gerarchie di parti e insiemi uniformemente. Il cliente interagisce esclusivamente tramite l'interfaccia Component; risulta dunque semplice perché non si deve preoccupare se sta interagendo con elemento singolo o composito.

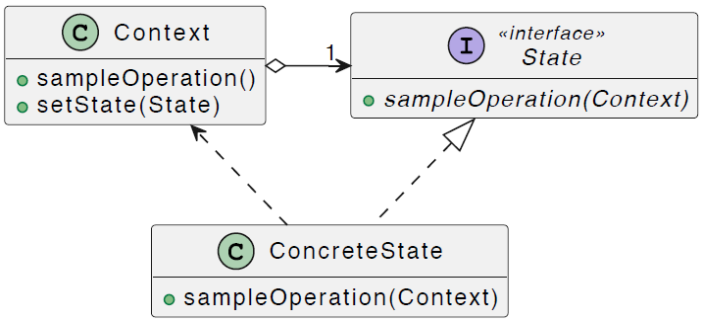


Decorator pattern
Usato per aggiungere nuove funzionalità o caratteristiche dinamicamente. Attacca le nuove responsabilità tramite l'aggiunta di nuovi oggetti.

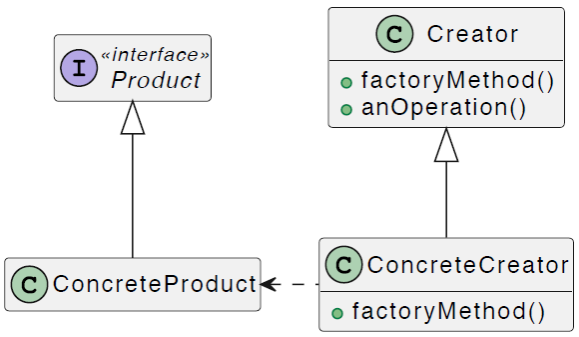


In Java viene usato nelle InputStreams.

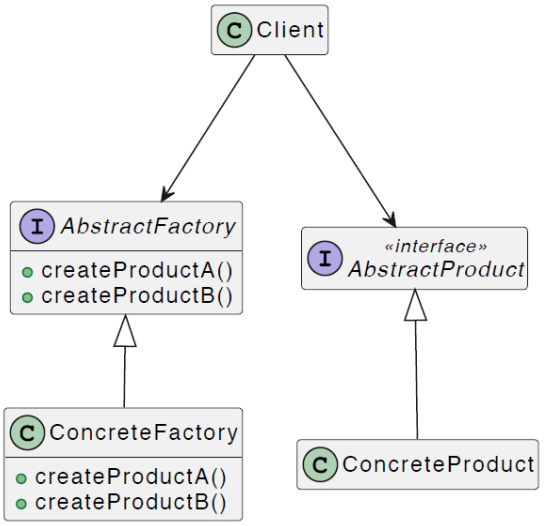
State pattern
Permette di modellare cambiamenti di comportamento al cambiare dello stato dell'oggetto. Viene partizionato il comportamento nelle varie classi stato e reso esplicito (atomico) il passaggio di stato. Gli oggetti stato possono essere in alcuni casi condivisi.



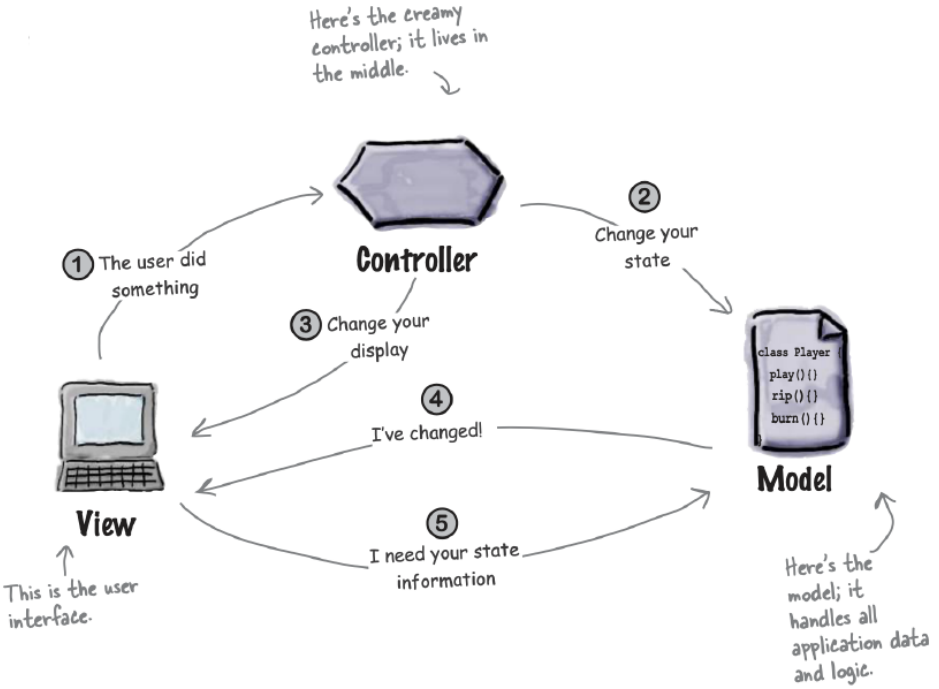
Factory Method pattern
Un pattern di tipo creazionale. Definisce un'interfaccia per creare un oggetto, ma lascia alle sottoclassi la scelta di cosa creare.



Abstract Factory pattern
Pattern di tipo creazionale. Fornisce un'interfaccia per la creazione di famiglie di oggetti relazionati o dipendenti senza specificarne la loro classe concreta.



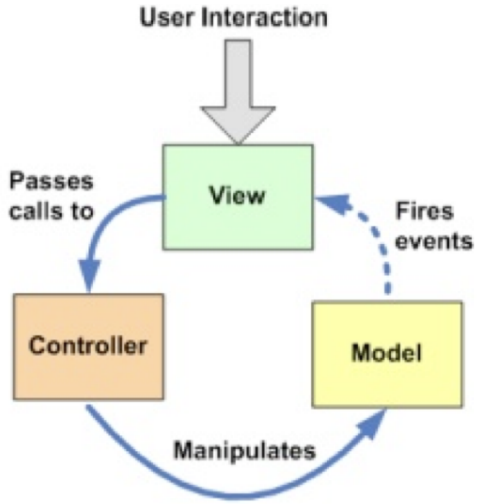
Model View Controller (MVC) pattern



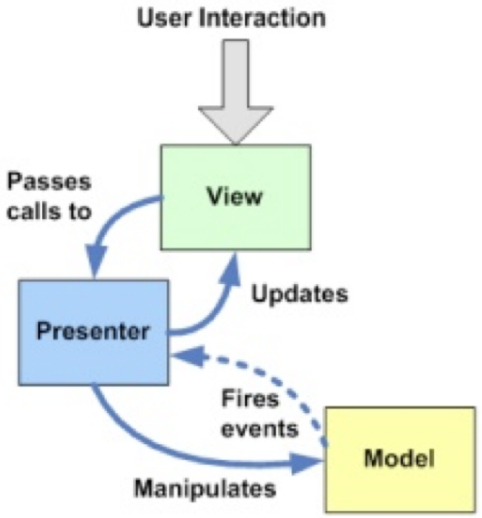
Questo pattern è composto da 3 elementi:

- View: offre una presentazione del modello. La view di solito prende lo stato e i dati che deve mostrare direttamente dal modello.
- Controller: prende l'input dell'utente e capisce cosa significa in relazione al modello.
- Model: il modello mantiene tutti i dati, lo stato e l'applicazione logica. Il modello è ignaro della view e del controller, anche se fornisce un'interfaccia per manipolare e recuperare il proprio stato e può notificare i propri cambiamenti dello stato agli observer.

Differisce dal Model View Presenter per come vengono gestiti gli eventi.

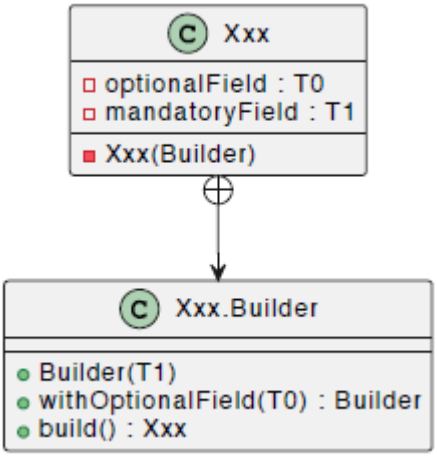


Model-View-Controller



Model-View-Presenter

Builder pattern




```java
public class Xxx {
   private final T1 mandatoryField;
   private final T0 optionalField1;
   private final T2 optionalField2;
   private Xxx(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 Xxx build() { return new Xxx(this); }
   }
}
```

Mocking

Testing

Quando si testa un oggetto (SUT, System Under Test), la sua esecuzione può dipendere da altri componenti del sistema da lui richiamati (DOC, Dependent-On Component). Test Double è un termine generico per un qualunque tipo di oggetto che viene messo al posto di DOC reale al fine di permettere il testing del SUT.

Mocking
Per ogni test di unità ci dovrebbe essere un unico componente reale (quindi un'unica new che crea il SUT) e i vari DOC implementati tramite mocking. In certi casi può essere utile o necessario mockare parte dell'oggetto sotto esame per renderlo più facilmente testabile (ad esempio creare un oggetto reale e farne uno spy per sovrascrivere alcuni metodi). Potrebbe anche essere utile iniettare un oggetto mockato in un attributo della classe in esame (@Injectock, tramite DependencyInjection).

Dummy objects
Oggetti che sono passati in giro, ma mai veramente usati. Sono usati quando:
- Non posso passare null.
- Potrei avere solo una interfaccia e non una classe.
- Potrei avere solo costruttori complessi.

```
@Test
public void testDummy() {
  MyClass dummy =  ?? ;

  List<MyClass> SUT = new ArrayList<MyClass>();

  SUT.add(dummy);

  assertThat(SUT.size()).isEqualTo(1);
}
```

Stub objects
Sono oggetti che forniscono risposte preconfezionate alle sole chiamate fatte durante il testing.

```
@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);
}
```

Mock objects
Sono oggetti che instrumentano e controllano le chiamate.

```
@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...
}
```

Spy objects

Sono oggetti che instrumentano e controllano le chiamate di oggetti reali (di cui possono usare i metodi e lo stato).

```
@Test
public void testConSpy()  {
  MyClass spy =  ?? ; // esiste classe reale MyClass

  MyList<int> SUT = new MyList<int>();

  res = SUT.somma(spy);

  assertThat(res).isEqualTo(14);
    // assert che getValue è stata chiamata 3 volte
    // prima una volta con parametro 0 e poi...
}
```

Fake objects
Sono oggetti che implementano il DOC, ma usando qualche scorciatoia, in maniera non realistica o non installabile; ad esempio mantenere un database in memoria invece che un database reale. Inefficiente per casi di dimensione significativa.

Stati e UML

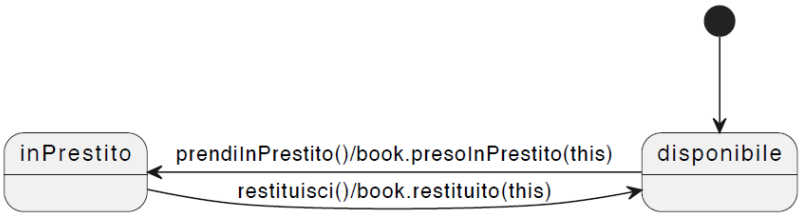
Stato di un oggetto
Lo stato concreto di un oggetto dipende dalla sua implementazione. Lo stato astratto di un oggetto è invece un sottoinsieme arbitrario degli stati concreti.
Gli oggetti possono anche non avere stato (ad esempio gli oggetti funzione) o avere un solo stato, ad esempio gli oggetti immutabili.
Va evidenziato però che una classe immutabile non ha un solo stato (può teoricamente assumere infiniti valori), mentre sono le sue istanze che una volta create non possono più cambiare stato.

Diagramma degli stati
Un automa è una n-upla <S, I, U;, t, s0>, dove:

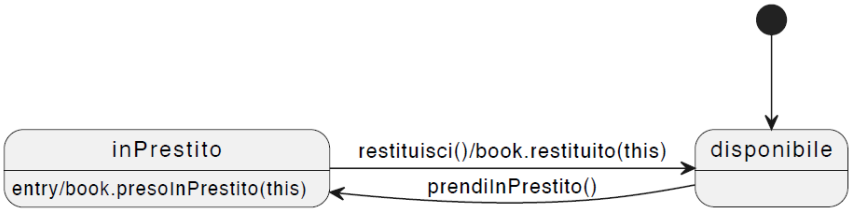
- S - l'insieme finito e non vuoto degli stati.
- I - l'insieme finito dei possibili ingressi.
- U - l'insieme finito delle possibili uscite.
- δ - la funzione di transizione. Stabilisce i possibili passaggi da uno stato ad un altro δ: S x I -> S.
- t - la funzione di uscita.
- s0 - lo stato iniziale.

Negli automi di Mealy, a differenza di quelli di Moore, le uscite dipendono non solo dallo stato raggiunto, ma anche da come viene raggiunto.

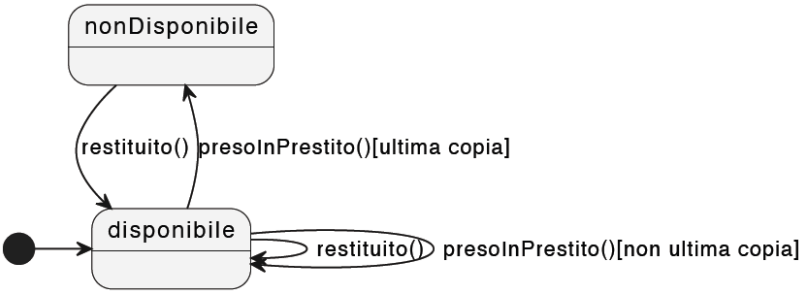
UML
In UML il diagramma degli stati è derivato da StateCharts. UML viene usato per definire il comportamento di un oggetto (o meglio di una classe di oggetti).
In UML se non vi è una freccia tra due stati, allora quell'azione non deve essere possibile (viene sollevata un'eccezione). Le azioni sono le uscite del diagramma, indicanti che l'oggetto che fa qualcosa verso l'esterno.



Le azioni possono anche essere interne allo stato.



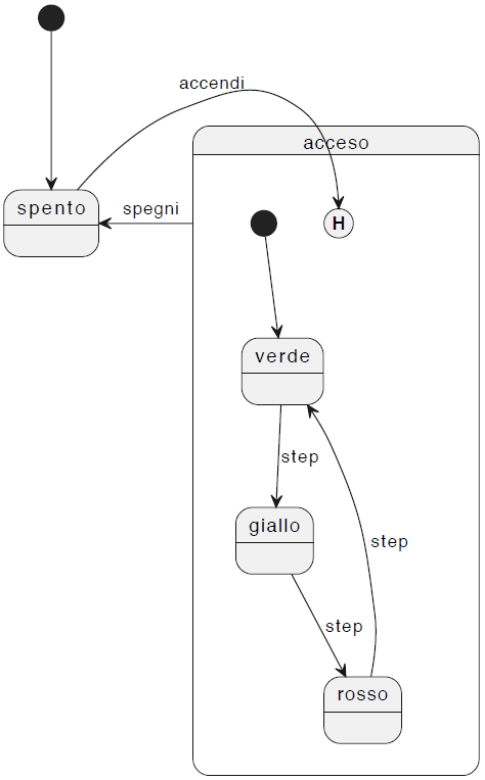
Vengono anche usate le guardie per disambiguare transizioni causate da uno stesso evento e uscenti da stesso stato.



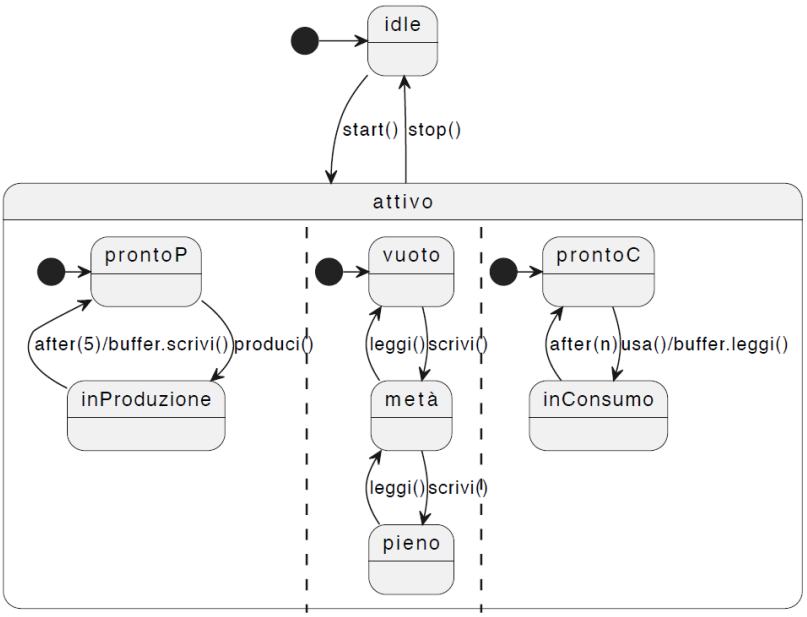
Vi sono anche altri tipi di eventi come:

- Time event: After(duration) - indica una durata massima di permanenza nello stato.
- Change event: When(condition) - la condizione è espressa in termini di valori degli attributi.

Un superstato è uno stato composto da più sottostati, permettendo dunque la creazione di una gerarchia di stati.



Vi può anche essere concorrenza all'interno di uno stato (vengono eseguiti diversi cammini contemporaneamente).



UML: Use case

Sono una classe di funzionalità fornite dal sistema, cioè un'astrazione di un insieme di scenari relazionati tra di loro. Al suo interno vengono date, in maniera testuale non formalizzata, informazioni circa:

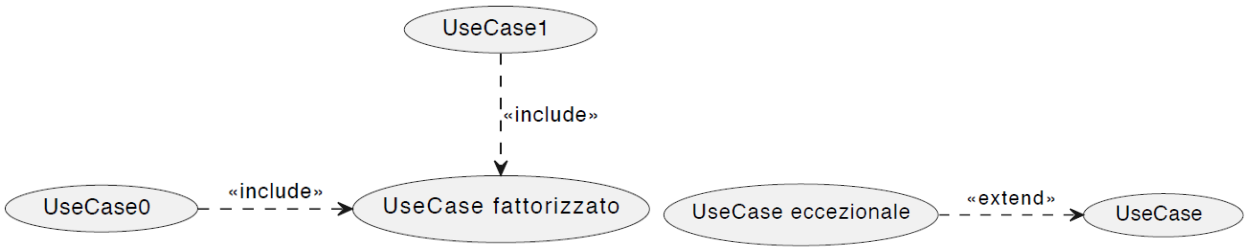
- Pre e post condizioni
- Flusso normale di esecuzione
- Eccezioni e il loro trattamento

Vengono inoltre spesso collegati ad altri diagrammi (come diagramma di Sequence e di Activity) che ne illustrano il flusso.

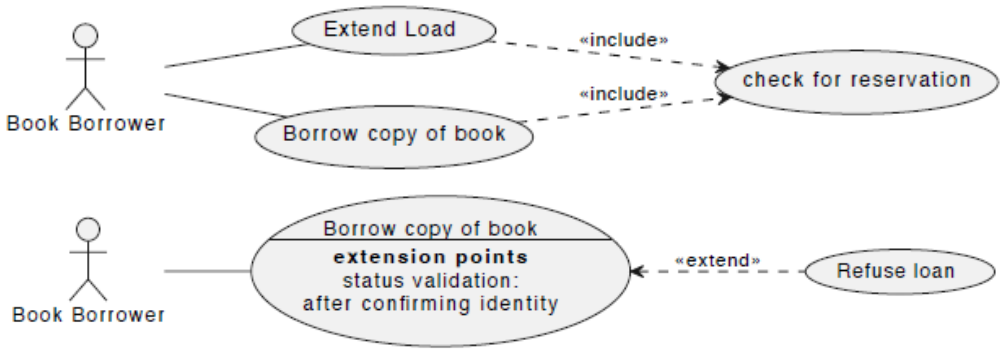
Un attore è un'entità esterna al sistema, che interagisce con il sistema. L'identificazione degli use cases può partire dalle funzionalità del sistema oppure dagli attori.

Negli use cases vanno identificati gli attori e le associazioni tra attori e casi d'uso e tra casi d'uso.
Associazioni:

- Use cases / Attori
  - Uno use case deve essere associato ad almeno un attore.
  - Un attore deve essere associato ad almeno uno use case .
  - Esiste un attore detto primario che ha il compito di far partire lo use case.
- Tra use case
  - include
  - extend



Ad esempio:



Gli scenari di UML sono descrizioni di come il sistema è usato in pratica. Sono utili nella raccolta dei requisiti perchè più semplici da scrivere di affermazioni astratte di ciò che il sistema deve fare. Possono essere inoltre usati anche complementarmente a schede di descrizione dei requisiti (come esempi).

Activity Diagram

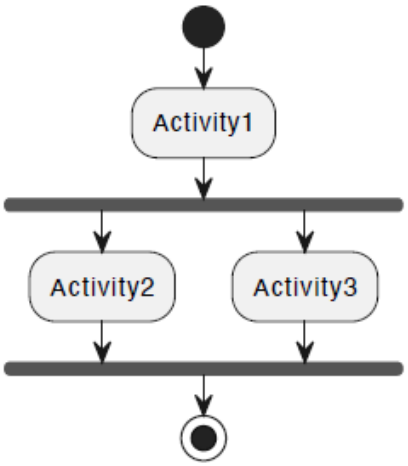
In UML gli activity diagram hanno molti punti in comune con il diagramma degli stati, ma hanno anche le seguenti differenze:

- Gli stati vengono chiamati attività
- Le transizioni di solito non sono etichettate con eventi (ma sono tutte del tipo "quando è terminata l'attività")
- Le azioni di solito sono inserite dentro le attività
- Le attività possono essere interne o esterne al sistema
- I blocchi di sincronizzazione non sono "eccezione"

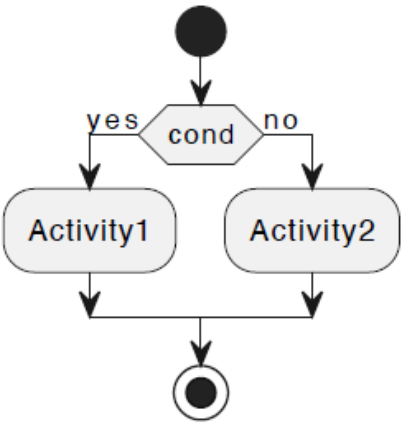
Gli activity diagram possono essere usati per rappresentare:

- Il flusso all'interno di un metodo
- Il flusso di uno use-case
- La logica all'interno di un business process

Attraverso l'uso di barre orizzontali si possono stabilire dei punti di sincronizzazione, tipicamente detti Fork/Join di attività. Se non specificato diversamente i join sono in and.



Posso anche evidenziare un momento di decisione, rappresentato da un piccolo rombo.

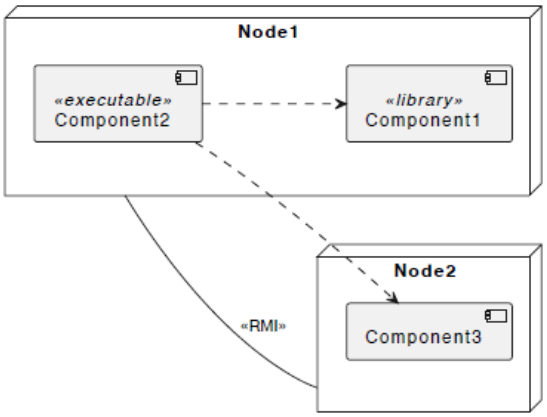


Le swim lane sono peculiari degli activity diagram. SI possono partizionare le attività al fine di rappresentare le responsabilità sulle singole attività. Vengono identificate delle “corsie” verticali, da cui il nome.

Deployment diagram
Un componente definisce una parte rimpiazzabile del sistema, svolgendo una funzione ben determinata. Può essere annidato in altri componenti e di esso vengono indicate chiaramente quale interfacce realizza (supporta) e le relazioni di dipendenza e composizione.

Un deployment diagram permette di specificare la dislocazione fisica delle istanze di componenti. E’ una vista statica della configurazione a run-time. Permette di specificare:

- Nodi del sistema (le macchine fisiche)
- Collegamenti tra nodi
- Dislocazione delle istanze dei componenti all’interno dei nodi e le loro relazioni



Verifica e Convalida

Terminologia

- Convalida: confronto del software con i requisiti (informali) posti dal committente.
- Verifica: confronto del software con le specifiche (formali) prodotte dall'analista.
- Malfunzionamento o Guasto (Failure): funzionamento non corretto di un programma. E' legato al funzionamento del programma e non al suo codice.
    - Esempio: la funzione raddoppia invocata con parametro 3 ritorna 9.
- Difetto o Anomalia (Fault): è legato al codice ed è condizione necessaria (ma non sufficiente) per il verificarsi di un malfunzionamento.
    - Esempio: il difetto che causa il malfunzionamento è ad una riga x del codice.
- Sbaglio (Mistake): è la causa di una anomalia. Generalmente si tratta di un errore umano (concettuale, di battitura, di scarsa conoscenza del linguaggio).
    - Esempio: scrivere "*" invece di "+".

Statico vs Dinamico

- Tecniche statiche: sono basate sull'analisi del codice
    - Metodi formali
    - Analisi data flow
    - Modelli statistici
- Tecniche dinamiche: sono basate sull'esecuzione del programma eseguibile
    - Testing
    - Debugging

Metodi formali

Sono tecniche che si prefiggono di provare l'assenza di anomalie nel prodotto finale, come:

- Analisi data flow.
- Dimostrazione di correttezza delle specifiche logiche.

Queste tecniche si basano sul principio di inaccuratezza pessimistica: se non riesce a dimostrare assenza del problema, dice che non va bene.

Testing

Sono tecniche che si prefiggono di rilevare malfunzionamenti o fornire fiducia nel prodotto (test di accettazione). Ne esistono di diverse tipi, come: white box, black box, gray box.
Queste tecniche si basano sul principio di inaccuratezza ottimistica: se non riesce a dimostrare la presenza di problemi, dice che va bene.

Debugging

Sono tecniche che si prefiggono di localizzare le anomalie che causano malfunzionamenti rilevati in precedenza.

Testing

Correttezza di un programma
Consideriamo un generico programma P come una funzione da un insieme di dati D (dominio) a un insieme di dati R (codominio).
P(d) indica l'esecuzione di P sul dato in ingresso $d \in D$.
Il risultato P(d) è corretto se soddisfa le specifiche, altrimenti scorretto.
ok(P, d) indicherà la correttezza di P per il dato d.

P è corretto se e solo se $\forall d \in D$ ok(P, d)

Test e caso di test
Un test T per un programma P è un sottoinsieme di D.
Un elemento t di un test T è detto caso di test.
L'esecuzione di un test consiste nell'esecuzione del programma $\forall t \in T$.
Un programma passa o super un test se:
ok(P, T) $\Leftrightarrow \forall t \in T$ ok(P, t)

Un test T ha successo se rileva uno o più malfunzionamenti presenti nel programma P:
successo(T, P) $\Leftrightarrow \exists t \in T \neg$ok(P, t)

T è ideale per P se e solo se ok(P, T) $\Rightarrow$ ok(P, D), cioè se il superamento del test implica la correttezza del programma. In generale è impossibile trovare un test ideale. Non esiste un algoritmo che dato un programma arbitrario P, generi un test ideale finito (il caso T = D non va considerato).

Tesi di Dijkstra: il test di un programma può rilevare la presenza di malfunzionamenti, ma non dimostrarne l'assenza.

Criterio di selezione
Il criterio di selezione è il ragionamento che seguiamo nel selezionare un sottoinsieme di D, sperando che approssimi il test ideale.
Un criterio di selezione può essere:

- Affidabile: un criterio C si dice affidabile se presi T1 e T2 in base al criterio C, allora o entrambi hanno successo o nessuno dei due ha successo. In simboli:
  - affidabile(C, P) $\Leftrightarrow (\forall T1 \in C, \forall T2 \in C$ successo(T1, P) $\Leftrightarrow$ successo(T2, P))
- Valido: un criterio C si dice valido se, qualore P non sia corretto, allora esiste almeno un T selezionato in base a C, che ha successo per il programma P. In simboli:
  - valido(C, P) $\Leftrightarrow (\neg$ok(P, D) $\Rightarrow \exists T \in C$ successo(T, P))

Da notare che:
affidabile(C, P) ^ valido(C, P) ^ $T \in C$ ^ $\neg$successo(T, P) $\Rightarrow$ ok(P, D)
ovvero che se prendiamo un test selezionato in base a un criterio sia affidabile che valido e il test non trova errori, allora il programma è corretto. Cioè un criterio affidabile e valido selezionerebbe test ideali, che però sappiamo non esistere.

Per scegliere un criterio di selezione è necessario identificare le caratteristiche (delle specifiche o del codice) che rendono utile un caso di test. Ad ogni criterio è possibile associare una metrica che ne misuri la copertura e che ci permetta di:

- Decidere quando smettere
- Decidere quale altro caso di test è opportuno aggiungere
- Confrontare bontà di test diversi

Un caso di test per poter portare a evidenziare un malfunzionamento causato da una anomalia, deve soddisfare tre requisiti:
- Eseguire il comando che contiene l'anomalia.
- L'esecuzione del comando contenente l'anomalia deve portare il sistema in uno stato scorretto.
- Lo stato scorretto deve propagarsi fino all'uscita del codice in esame, in modo da produrre un output diverso da quello atteso.

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 contenuto in T.

La metrica è la frazione dei comandi eseguiti su quelli eseguibili.

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 contenuto in T.

La metrica è la frazione delle decisioni che sono state rese sia vere che false su quelle per cui è possibile farlo.

La copertura delle decisioni implica la copertura dei comandi, ma non è garantito l'inverso.

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 contenuto in T.

La metrica è la frazione delle condizioni che sono state rese sia vere che false su quelle per cui è possibile farlo.

Non implica nessuno dei criteri precedenti.

Copertura 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 singola condizione che compare nelle decisioni del programma vale sia vero che falso per diversi casi di test in T.

Questo criterio contiene i due precedenti.

Copertura 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 in T.

Questo criterio contiene il precedente.

Il numero di casi cresce molto velocemente rispetto al numero di condizioni base (esponenziale).

Copertura decisioni/condizioni modificate

Si dà importanza, nella selezione delle combinazioni, al fatto che la modifica di una singola condizione base porti a modificare la decisione.
Cioè devono esistere per ogni condizione base due casi di test che modificano il valore di una sola condizione base e che modificano il valore della decisione.
Si può dimostrare che se ho N condizioni base, necessito “solo” di N+1 casi di test (lineare).

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 test in T.

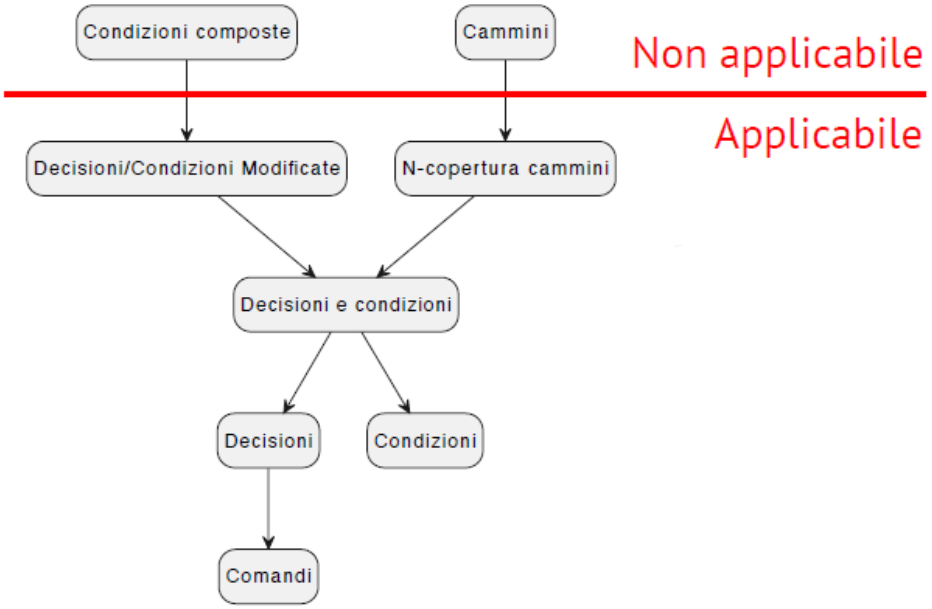
La metrica è la frazione dei cammini percorsi su quelli effettivamente percorribili.

Spesso impraticabile, anche per programmi semplici.

N-Copertura dei cicli
Un test soddisfa il criterio di n-copertura se e solo se ogni cammino del grafo contenente al massimo un numero di iterazioni di ogni ciclo non superiore a n viene percorso per almeno un caso di test.

Si limita il numero massimo di percorrenze dei cicli, ma trovare il valore di n ottimale è complesso. Il numero dei casi di test necessari cresce molto rapidamente al crescere di n.

Implicazioni tra criteri di copertura



Analisi statica
Si basa sull’esame di un insieme finito di elementi (le istruzioni del programma) contrariamente all’analisi dinamica (insieme degli stati delle esecuzioni). E’ meno “costosa” (onerosa) del testing e non soffre del problema dell’esplosione dello spazio degli stati. Non può però rilevare anomalie dipendenti da uno specifico valore assunto a run-time delle variabili.

I compilatori eseguono diversi tipi di analisi statiche:
- Analisi lessicale: identificazione dei token del linguaggio, permette di identificare la presenza di simboli non appartenenti al linguaggio.
- Analisi sintattica: identifica la relazione tra token; controlla la conformità del codice alla grammatica del linguaggio.
- Controllo dei tipi: controlla violazioni sulle regole d’uso dei tipi.

- Analisi del flusso dei dati: rileva problemi relativi a evoluzione del valore associato alle variabili.

L'analisi del data flow (utilizzata anche nell'ottimizzazione dei compilatori) identifica staticamente il tipo di operazione che un comando esegue su una variabile:
- Definizione (d): se il comando assegna un valore alla variabile.
- Uso (u): se il comando richiede il valore di una variabile.
- Annullamento (a): se al termine dell'esecuzione dell'istruzione il valore della variabile non è significativo/affidabile.

In questo modo ciascuna variabile può essere ridotta da una sequenza di istruzioni a una sequenza di d, u, a.

Bisogna porre particolare attenzione all'ordine delle operazioni:
- L'uso di una variabile deve sempre essere preceduto in ogni sequenza da una definizione senza annullamenti intermedi.
- La definizione di una variabile deve sempre essere seguita da un uso prima di un suo annullamento o definizione.
- L'annullamento di una variabile deve essere sempre seguito da una definizione prima di un uso o di un altro annullamento.

//non sono regole assolute, la classificazione dipende dal linguaggio.

Siano:

def(i) è l'insieme delle variabili che sono definite in i.

du(x, i) è l'insieme dei nodi j tali che:
- $x \in def(i)$.
- x usato in j.
- esiste un cammino da i a j, libero da definizioni di x.

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 appartenente a def(i), T include un caso di test che esegue un cammino libero da definizioni da i ad almeno uno degli elementi di du(i, x).

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 def(i), per ogni elemento j di du(i, x) T include un caso di test che esegue un cammino libero da definizioni da i a j. (deve coprire tutti gli usi di una definizione).

Analisi mutazionale

Viene generato un insieme di programmi Π simili al programma P in esame. Su di essi viene eseguito lo stesso test T previsto per il programma P. Se P è corretto, allora i programmi in Π devono essere sbagliati e per almeno un caso di test devono quindi produrre un risultato diverso.

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 in T la cui esecuzione produca per π un risultato diverso da quello prodotto da P.

La metrica è la frazione di mutanti riconosciuta come diversa da P sul totale di mutanti generati.

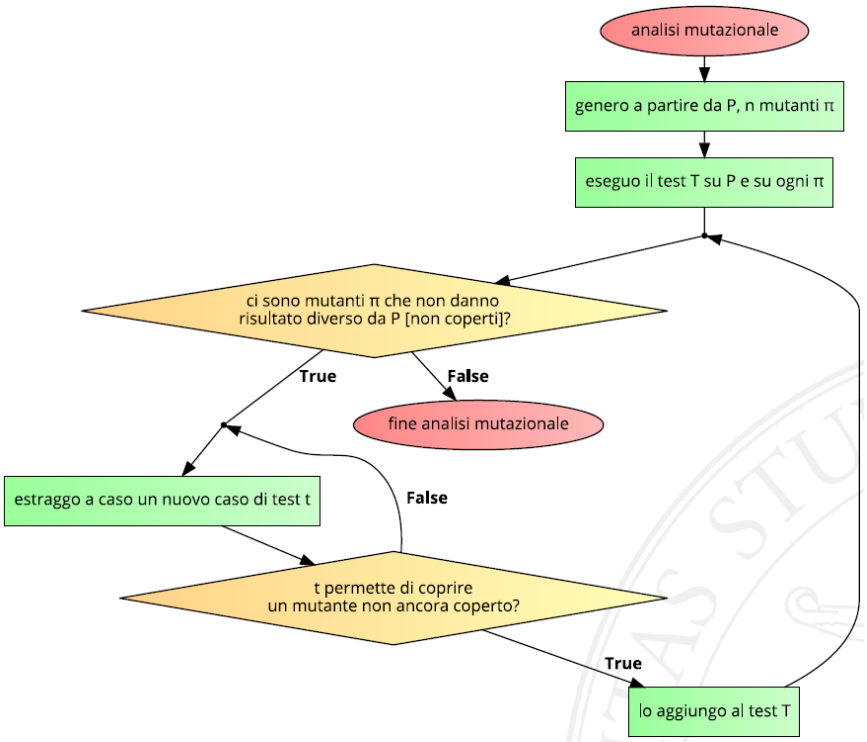
Idealmente i mutanti dovrebbero avere differenze minime da P e bisognerebbe avere un mutante per ogni possibile difetto (virtualmente infiniti). La generazione dei mutanti è pertanto facilmente automatizzabile.

Gli operatori mutanti sono funzioni che dato P generano uno o più mutanti. I più semplici svolgono semplicemente modifiche sintattiche che comportano modifiche semantiche (ma non errori di compilazione), i più sofisticati vengono invece detti HOM (High Order Mutation) e svolgono non solo una modifica; inoltre a volte sono più difficili da identificare che le due modifiche prese singolarmente.

Gli operatori mutanti si distinguono in base all'oggetto su cui operano, ad esempio:
- Costanti, variabili
- Operatori ed espressioni del programma
- Sui comandi del programma

e possono essere specifici di alcuni tipi di applicazioni, come sistemi concorrenti o sistemi object oriented.

Per automatizzare la generazione si può utilizzare un algoritmo di questo tipo:



Object Orientation testing
Con l'Object Orientation si complica il testing e pertanto è conveniente isolare la classe, costruendo classi stub per renderla eseguibile indipendentemente dal contesto. Si bypassa inoltre l'incapsulamento e si costruisce una classe che permetta di istanziare oggetti e chiamare i metodi secondo il criterio di copertura scelto.

Lo stato dell'oggetto potrebbe essere rappresentato staticamente come una macchina a stati finiti composta dagli stati (gli stati dell'oggetto) e dalle transizioni (chiamate di metodi che alterano lo stato). In questo modo abbiamo un diagramma e possiamo scegliere di coprire:
- Tutti i nodi (ovvero tutti gli stati dell'oggetto)
- Tutti gli archi (ovvero tutti i metodi per ogni stato)
- Tutte le coppie di archi in/out (ovvero considero anche come sono arrivato in uno stato)
- Tutti i cammini identificabili nel grafo

Test funzionale
Un test funzionale è black box, ovvero non sfrutta la conoscenza del codice. I dati di test possono essere derivati dalle specifiche (ovvero dai requisiti funzionali), concentrandosi dunque sul dominio delle informazioni invece che sulla struttura di controllo. Permette di identificare errori non sintetizzabili con

criteri strutturali, come funzionalità non implementate e cammini di flussi dimenticati. Trova anche errori di interfaccia e di prestazioni.

Si utilizzano diverse tecniche, come metodi basati su grafi, suddivisione del dominio in classi di equivalenza, analisi dei valori limite (test di frontiera), collaudo per confronto.

Classi di equivalenza
La suddivisione del dominio dei dati in ingresso in classi di dati è utile per derivarne casi di test. Una classe di dati è infatti un insieme i cui componenti dovrebbero essere trattati in maniera analoga dal programma.
Si cerca dunque di individuare casi di test che rivelino eventuali classi di errori (ad esempio un'elaborazione scorretta per numeri negativi).

Un 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. Un dato può essere ad esempio:
- Un valore
- Un intervallo
- Un insieme di valori correlati

Se ci si aspetta un valore specifico, vengono definite una classe valida ed una non valida. Ad esempio, volendo un codice PIN:
1. PIN corretto
2. Numero di 4 cifre qualsiasi diverso da PIN

Se ci si aspetta un valore in un intervallo, vengono definite una classe di equivalenza valida e due non valide (maggiore e minore degli estremi dell'intervallo).
Ad esempio: [100, 700]:
1. Numero tra 100 e 700
2. Numero minore di 100
3. Numero maggiore di 700

Category partition
E' un particolare metodo di suddivisione in classi di equivalenza usabile a diversi livelli di granularità, come test di unità, di integrazione, di sistema. E' composto dai seguenti passi:
1. Analizzare le specifiche: identificare le unità funzionali individuali che possono essere verificate singolarmente e per ogni unità identificare le caratteristiche (categorie) dei parametri dell'ambiente.
2. Scegliere dei valori per le categorie.
3. Determinare eventuali vincoli tra le scelte.
4. Scrivere test e documentazione .

Test di frontiera
Complementare a classi di equivalenza, ma seleziono un elemento ai confini del dominio, in quanto gli errori tendono ad accumularsi ai limiti del dominio. Cerco di selezionare casi di test che esercitano i valori limite.

Software inspection
E' una tecnica manuale per individuare (e correggere?) gli errori basata su un'attività di gruppo. La tecnica di ispezione più diffusa è la Fagan Code Inspections, estesa anche a fase di progetto e raccolta dei requisiti.

Vi sono diversi ruoli in questa tecnica:

- Moderatore: coordina i meeting, sceglie i partecipanti, controlla il processo.
- Readers e Testers: leggono il codice al gruppo e cercano i difetti.
- Autore: è un partecipante passivo, che risponde ad eventuali domande.

Il processo di ispezione del software è composto dalle seguenti fasi:
- Planning: il moderatore sceglie i partecipanti e fissa gli incontri.
- Overview: per fornire il background e assegnare i ruoli.
- Preparation: attività svolte offline per la comprensione del codice o della struttura del sistema.
- Inspection
- Rework: l'autore si occupa dei difetti individuati.
- Follow-up: possibile re-ispezione.

L'ispezione ha lo scopo di registrare il maggior numero di difetti, ma non correggerli. Viene parafrasato il codice riga per riga, in modo tale da risalire allo scopo del codice partendo dal sorgente. Si utilizza una "checklist" di domande, per vedere se siano soddisfatte o meno alcuni requisiti del codice.

Una variante è la Active Design Reviews, nella quale l'autore fa domande ai revisori, ciascuno specifico per diversi aspetti e con adeguata esperienza in quell'aspetto.

La software inspection è cost-effective e incrementa la produttività, qualità e stabilità del progetto. Infatti è un processo rigoroso e dettagliato basato sull'accumulo di esperienza.

Mettere insieme le tecniche di verifica e convalida rende più completa l'individuazione degli errori.

Debugging
Mira a localizzare e rimuovere le anomalie che sono le cause di malfunzionamenti riscontrati nel programma. Non deve essere usato per rilevare malfunzionamenti. L'attività è definita per un programma e un insieme di dati che causano malfunzionamenti nel programma; è dunque basato sulla riproducibilità del funzionamento.

Il debugging può essere svolto attraverso diverse tecniche.

La tecnica naive è quella di stampare i valori intermedi delle variabili. E' facile da applicare, ma richiede la modifica del codice ed è poco flessibile.

Una tecnica naive "avanzata" invece può essere un miglioramento della tecnica precedente sfruttando le funzionalità del linguaggio, come #ifdef in C.

Un'altra tecnica è quella del dump di memoria, ovvero la produzione di un'immagine esatta della memoria dopo il passo di esecuzione, ma è spesso difficile per la differenza tra la rappresentazione astratta dello stato (legata alle strutture dati del linguaggio utilizzato) e la rappresentazione fornita dallo strumento. Produce però anche molti dati che sono per la maggior parte inutili.

Debugging simbolico: tecnica in cui gli stadi intermedi sono prodotti usando una rappresentazione compatibile con quella del linguaggio usato.

Debugging per prova: per visualizzare ed esaminare automaticamente gli stati ottenuti.

Delta (differential) debugging: per automatizzare il debugging.

Reti di Petri

Definizione di Rete di Petri
Sono in parte simili a macchine a stati finiti, ma nascono specificatamente per descrivere sistemi concorrenti, cambiando sia il concetto di stato che di transizione.
Lo stato non è più visto a livello di sistema, ma come composizione di tanti stati parziali.
Le transizioni non operano più quindi su uno stato globale, ma si limitano a variarne una parte.

Una rete di Petri è una quintupla [P, T; F, W, M0], dove:

- P insieme dei posti
- T insieme delle transizioni
- F relazione di flusso

  $F \subseteq (P \times T) \cup (T \times P)$
- W funzione che associa un peso ad ogni flusso

  $W: F \Rightarrow N - \{0\}$
- M0 la marcatura iniziale

  $M0: P \Rightarrow N$

Siano inoltre:

- Pre(a) = {d ∈ (P ∪ T) | <d, a> ∈ F} l'insieme dei preset, ovvero l'insieme di posti e transizioni tali per cui esiste una relazione di flusso che va da d ad a.
- Post(a) = {d ∈ (P ∪ T) | <a, d> ∈ F} l'insieme dei postset, ovvero l'insieme di posti e transizioni tali per cui esiste una relazione di flusso che va da a a d. .

Una transizione t ∈ T è abilitata in M se e solo se:

$\forall p \in Pre(t)\ M(P) >= W(<p,t>)$

e si scrive

$M [t >$

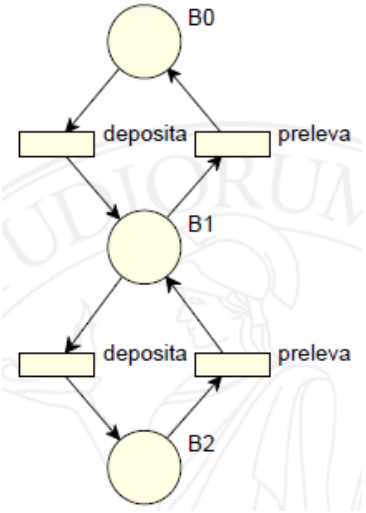
.

Lo scatto di una transizione t in una marcatura M produce una nuova marcatura M', che si scrive:

$M [t > M'$

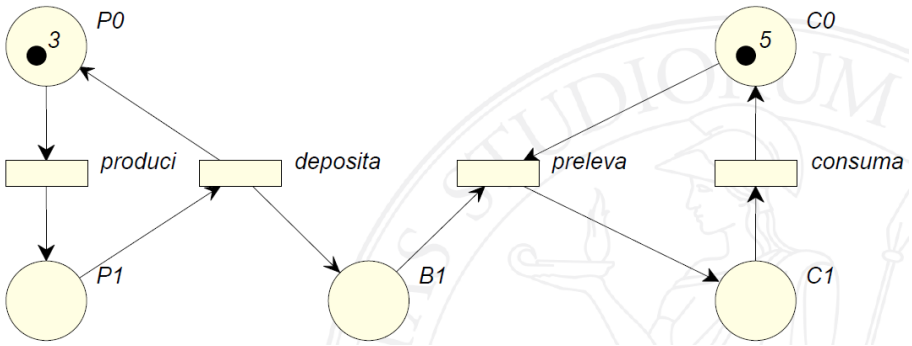
.

Ad esempio: il consumatore e il buffer



Nelle reti di Petri, se vi sono più transizioni abilitate, non posso dire quale deve scattare.

Un esempio di rete di Petri è la seguente, che implementa il problema del produttore e consumatore:



Modificando il numero di token e/o modificando il peso degli archi, è possibile modellare situazioni diverse.

L'insieme di raggiungibilità di una rete a partire da una marcatura M è il più piccolo insieme di marcature tale che:

- $M \in R(P/T, M)$
- $(M' \in R(P/T, M) \wedge \exists t \in T\ M' [t > M'') \Rightarrow M'' \in R(P/T, M)$

Una rete P/T con marcatura M si dice limitata se e solo se:

$$\exists k \in N\ \forall M' \in R(P/T, M)\ \forall p \in P\ M'(p) <= k$$

cioè se è possibile fissare un limite al numero di gettoni della rete.

Se una rete è limitata, allora l'insieme di raggiungibilità è finito. E' possibile allora definire un automa a stati finiti corrispondente, i cui stati sono le possibili marcatura dell'insieme di raggiungibilità.

Relazione di sequenza tra transizioni

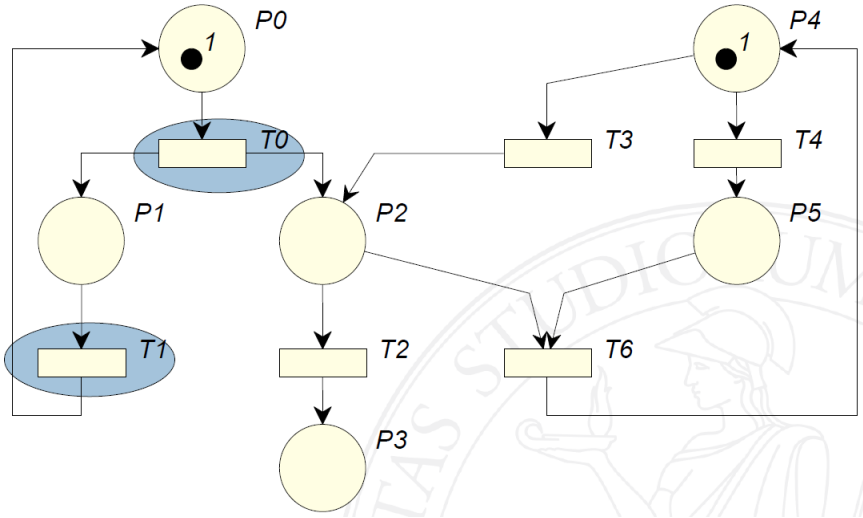
Una transizione t1 è in sequenza con una transizione t2 in una marcatura M se e solo se:

$$M [t1 > \wedge \neg M [t2 > \wedge M [t1t2 >$$

ovvero se:

- t1 è abilitata in M
- t2 NON è abilitata in M
- t2 viene abilitata dallo scatto di t1 in M

Esempio:

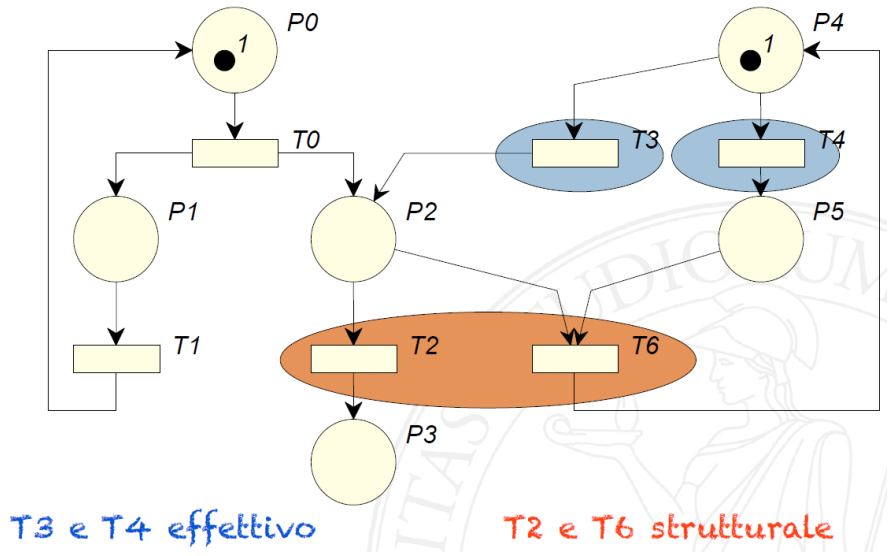


Relazione di conflitto tra transizioni

Due transizioni (t1, t2) sono in conflitto:

- Strutturale se e solo se
  Pre(t1) ∩ Pre(t2) ≠ Ø
- Effettivo in una marcatura M se e solo se:
  M [t1 > ^ M [t2 > ^ ∃ p ∈ Pre(t1) ∩ Pre(t2) | (M(p) < W(<p, t1>) + W(<p, t2>)) ,
  ovvero se t1 e t2 sono abilitate in M ed esiste un posto in ingresso ad entrambe che non ha abbastanza token per far scattare entrambe.

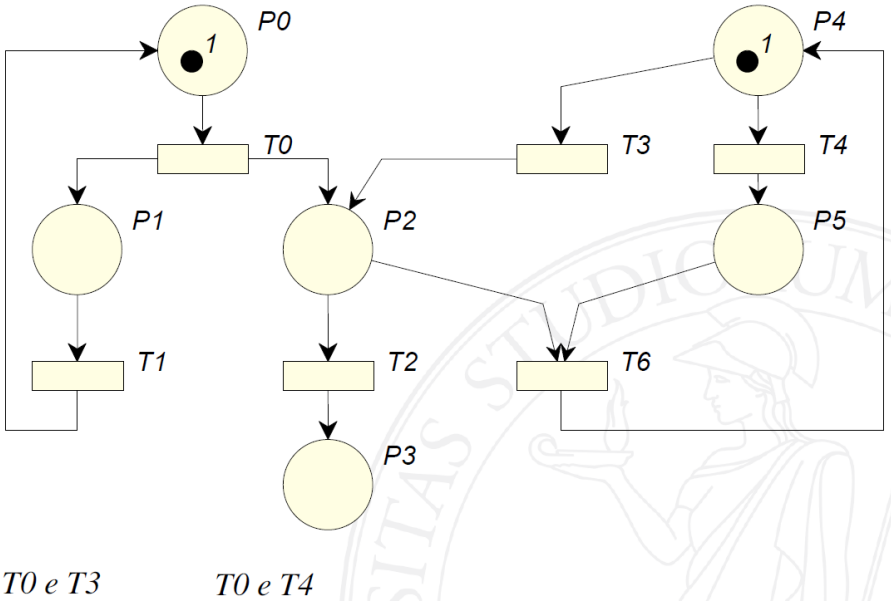
Esempio:



Relazione di concorrenza tra transizioni
Due transizioni (t1, t2) sono in concorrenza:

- Strutturale se e solo se
  Pre(t1) ∩ Pre(t2) = Ø
- Effettivo in una marcatura M se e solo se:
  M [t1 > ^ M [t2 > ^ ∀ p ∈ Pre(t1) ∩ Pre(t2) | (M(p) >= W(<p, t1>) + W(<p, t2>)) ,
  ovvero se t1 e t2 sono abilitate in M e tutti i posti in ingresso ad entrambe hanno abbastanza token per far scattare entrambe.

Esempio:



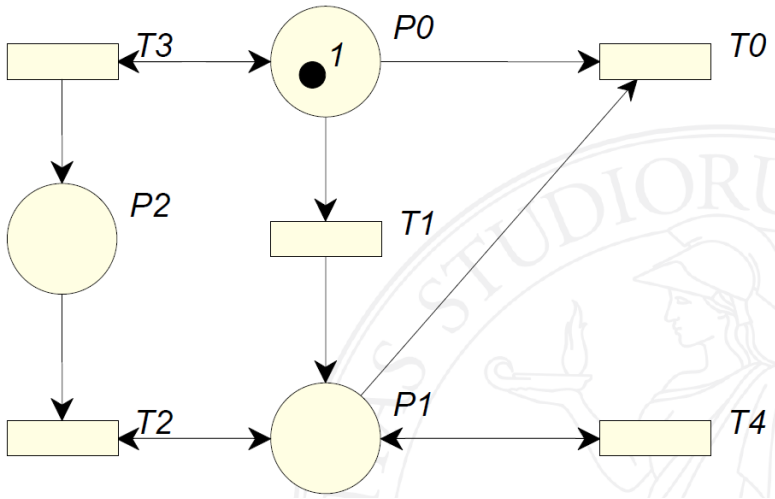
Vitalità di una transizione

Una transizione t in una marcatura m è viva a:

- Grado 0: non è abilitata in nessuna marcatura appartenente all'insieme di raggiungibilità (è morta).
- Grado 1: esiste almeno una marcatura raggiungibile in cui è abilitata.
- Grado 2: per ogni numero n esiste almeno una sequenza ammissibile in cui la transizione scatta n volte.
- Grado 3: esiste una sequenza di scatti ammissibile in cui scatta infinite volte.
- Grado 4: in qualunque marcatura raggiungibile, esiste una sequenza ammissibile in cui scatta (è viva).

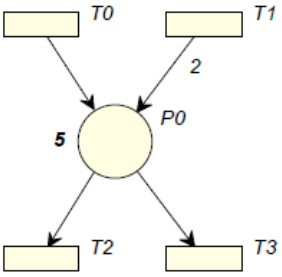
Una rete è viva se tutte le sue transizioni sono vive.

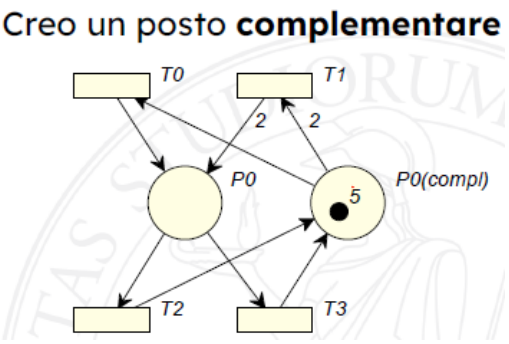
Esempio (transizione Tn è viva a n in questo esempio):



Capacità dei posti

Una possibile estensione delle reti di Petri consiste nel fissare un massimo numero di token ammissibili in un posto, in modo da forzare la limitatezza.





Un posto pc è complementare di p se e solo se:

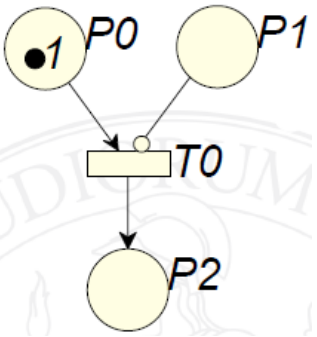
$\forall t \in T\ (\exists\ <p,t> \in F \Leftrightarrow \exists\ <t, pc> \in F \quad W(<p,t>) = W(<t,pc>)$

$\forall t \in T\ (\exists\ <t,p> \in F \Leftrightarrow \exists\ <pc, t> \in F \quad W(<pc,t>) = W(<t,p>)$

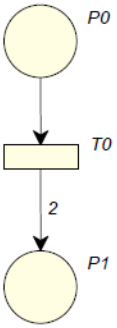
ovvero solo per reti pure.

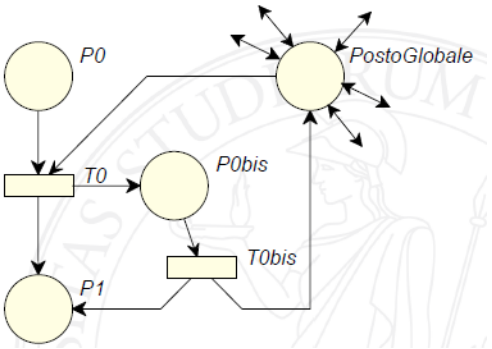
Archi inibitori

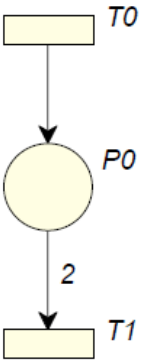
Permettono di dire che non deve essere presente un token perché la transizione sia abilitata.

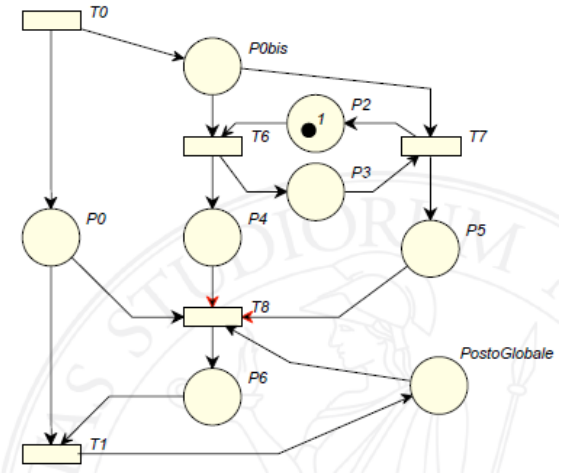


Eliminazione dei pesi degli archi









Reti Condizioni Eventi (C/E)

Una rete viene detta C/E se:

- Tutti gli archi hanno peso 1.
- Tutti i posti hanno capacità 1.

Se i posti (Condizioni) in ingresso a t contengono un token, la transizione t (Evento) può scattare. Una rete P/T limitata ha una corrispondente nella classe C/E.

Rete conservativa

Data una funzione di pesi H:

$H: P \Rightarrow N - \{0\}$

una rete P/T con marcatura M si dice conservativa rispetto a tale funzione se e solo se:

$\forall M' \in R(P/T, M) \ \ \Sigma_{[p \in P]} H(p) M'(p) = \Sigma_{[p \in P]} H(p) M(p)$

Rete strettamente conservativa
Una rete P/T conservativa rispetto alla funzione che assegna pesi tutti uguali a 1 si dice strettamente conservativa se:

- Il numero di token nella rete non cambia mai, ovvero:
  $\forall M' \in R(P/T, M)\ \ \Sigma_{p \in P} M'(p) = \Sigma_{p \in P} M(p)$
- Il numero di token consumati dallo scatto di una transizione è uguale al numero di gettoni generati dallo stesso, ovvero:
  $\forall t \text{ NON MORTA} \in T\ \Sigma_{p \in Pre(t)} W(<p, t>) = \Sigma_{p \in Post(t)} W(<t, p>)$

Stato base e Rete reversibile
Una marcatura M' è detta stato base (home state) se per ogni marcatura M in R(M0), M' è raggiungibile da M.
Una rete di Petri è detta reversibile se per ogni marcatura M in R(M0), M0 è raggiungibile da M (lo stato iniziale è uno stato base).

Tecniche di analisi

- Dinamiche
  - Albero (grafo) delle marcature raggiungibili
  - Albero (grafo) delle copertura delle marcature raggiungibili
- Statiche (Strutturali)
  - Identificazione P-invarianti
  - Identificazione T-invarianti

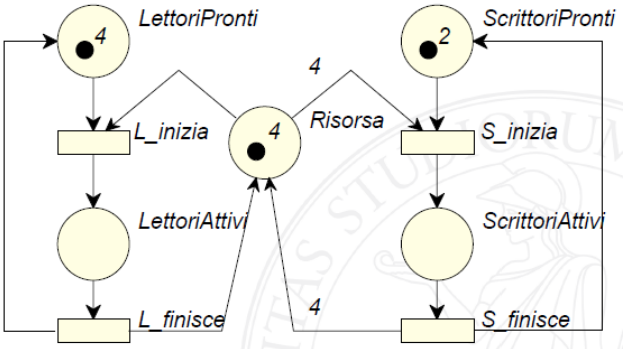
Albero di raggiungibilità

1. Crea la radice corrispondente alla marcatura iniziale, etichetta il nodo come “nuovo”.
2. Finchè esistono nodi etichettati “nuovo” esegui i seguenti passi:
   2.1. Seleziona una marcatura M con etichetta “nuovo” e togli l’etichetta.
   2.2. Se M è identica ad una marcatura sul cammino della radice ad M, etichetta M come “duplicata” e passa ad un’altra marcatura.
   2.3. Se nessuna transizione è abilitata in M, etichetta la marcatura come “finale”.
   2.4. Finchè esistono transizioni abilitate in M esegui i seguenti passi per ogni transizione t abilitata in M:
      2.4.1. Crea la marcatura M’ prodotta dallo scatto di t.
      2.4.2. Crea un nodo corrispondente a M’, aggiungi un arco da M a M’ ed etichetta M’ come “nuovo”.

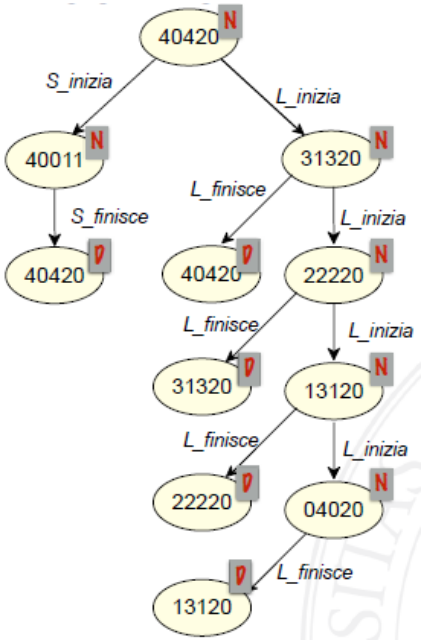
Questa tecnica ha il limite di dover enumerare tutte le possibili marcature raggiungibili. Se la rete non è limitata sono infinite e dunque non può essere completato.

Questa tecnica non sa dire se una rete è limitata, ma se è limitata ci sa dire quasi tutto su di essa (è la esplicitazione degli stati della rete, l’automa a stati finiti corrispondente). La crescita degli stati della rete è però esponenziale.

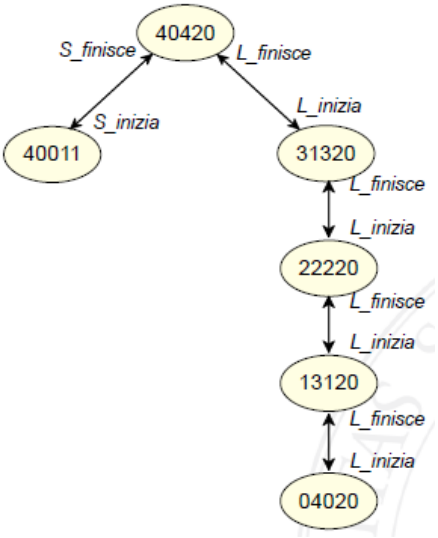
Esempio: a partire da questa rete di Petri



Otteniamo il seguente albero di raggiungibilità



Corrispondente al seguente grafo di raggiungibilità



Copribilità (coverability)

Diciamo che una marcatura M copre una marcatura M' (M' è coperta da M) se e solo se:

$\forall p \in P\ M(p) >= M'(p)$

Una marcatura è detta copribile a partire da una marcatura M' se esiste una marcatura M'' in R(M') che copre M.

Se M è la marcatura minima per abilitare t (esattamente il numero di token necessari in ogni posto del preset):

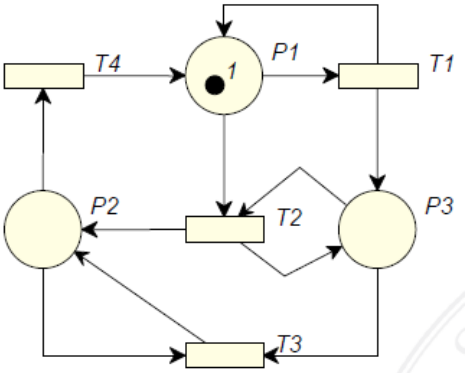
- La transizione t è morta se e solo se M non è copribile a partire dalla marcatura corrente.
- Altrimenti la transizione t è almeno 1-viva.

Albero di copertura

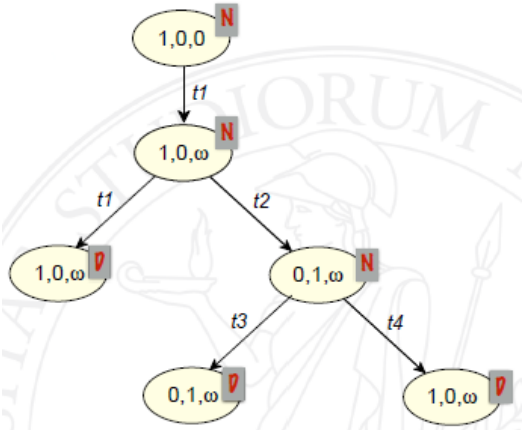
Differisce dall'albero di raggiungibilità solo per 2.4.2:

1. Crea la radice corrispondente alla marcatura iniziale, etichetta il nodo come "nuovo".
2. Finchè esistono nodi etichettati "nuovo" esegui i seguenti passi:
   - 2.1. Seleziona una marcatura M con etichetta "nuovo" e togli l'etichetta.
   - 2.2. Se M è identica ad una marcatura sul cammino della radice ad M, etichetta M come "duplicata" e passa ad un'altra marcatura.
   - 2.3. Se nessuna transizione è abilitata in M, etichetta la marcatura come "finale".
   - 2.4. Finchè esistono transizioni abilitate in M esegui i seguenti passi per ogni transizione t abilitata in M:
     - 2.4.1. Crea la marcatura M' prodotta dallo scatto di t.
     - 2.4.2. Se sul cammino della radice a M esiste una marcatura M'' coperta da M', aggiungi ω in tutte le posizioni corrispondenti a coperture proprie.
     - 2.4.3. Crea un nodo corrispondente a M', aggiungi un arco da M a M' ed etichetta M' come "nuovo".

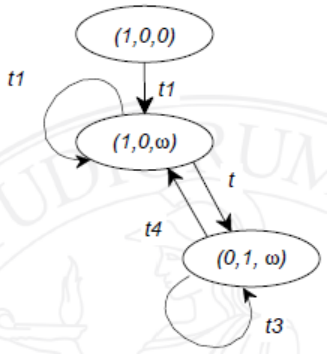
Esempio: dalla seguente rete di Petri



Otteniamo il seguente albero di copertura



Corrispondente al seguente grafo di copertura



Proprietà dell'albero di copertura

- Una rete di Petri è limitata se ω non compare in nessun nodo dell'albero di copertura.
- Una rete di Petri è binaria se nell'albero di copertura compaiono solo 0 e 1.
- Una transizione è morta (0-viva) se non appare come etichetta di un arco nell'albero di copertura.
- Condizione necessaria affinchè una marcatura M sia raggiungibile è l'esistenza di un nodo etichettato con una marcatura che copre M, ma non è una condizione sufficiente.
- Non è possibile decidere se una rete è viva.

Rappresentazione matriciale di una rete di Petri

E' possibile rappresentare (definire) una rete di Petri mediante delle matrici con una trasformazione automatica e facilmente trattabile matematicamente. E' necessario usare diverse matrici:

- I archi in input alle transizioni
- O archi in output alle transizioni
- m marcatura dei posti

Per le matrici I e O devo assegnare un indice ad ogni posto:

$p : 1\ldots|P| \rightarrow P$

ed assegnare un indice ad ogni transizione:

$t : 1\ldots|T| \rightarrow T$

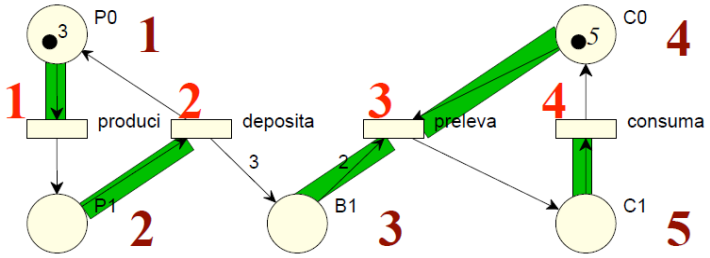
Le due matrici I e O sono $|P| \times |T|$, riempite attraverso i seguenti criteri:

- $\forall <p(i), t(j)> \in F\ I[i][j] = W(<p(i),t(j)>)$
- $\forall <p(i), t(j)> \notin F\ I[i][j] = 0$
- $\forall <t(j), p(i)> \in F\ O[i][j] = W(<t(j),p(i)>)$
- $\forall <t(j), p(i)> \notin F\ O[i][j] = 0$

ovvero nel caso di I una matrice composta da colonne ciascuna delle quali rappresenta la quantità di token necessari in ingresso alla transizione da quale stato, mentre nel caso di O una matrice composta da colonne ciascuna delle quali rappresenta la quantità di token uscenti dalla transizione, verso quale stato.

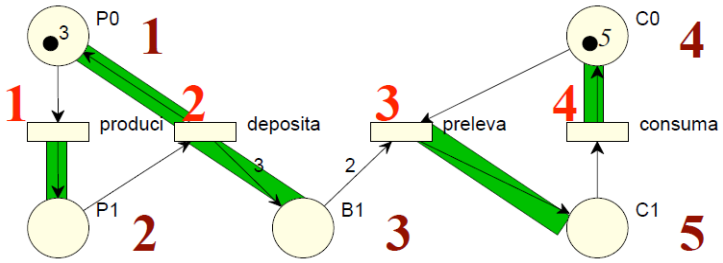
Indicheremo il vettore colonna k di una matrice X con la notazione X[.][k].

Esempio matrice I:



	1	2	3	4
1	*1*	0	0	0
2	0	*1*	0	0
3	0	0	*2*	0
4	0	0	*1*	0
5	0	0	0	*1*

Esempio matrice O:



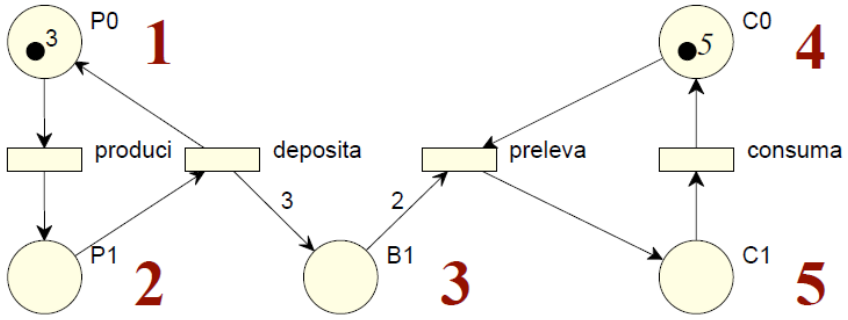
	1	2	3	4
1	0	1	0	0
2	1	0	0	0
3	0	3	0	0
4	0	0	0	1
5	0	0	1	0

Marcatura m è invece un vettore colonna di dimensione |P| e si calcola a partire dalla funzione marcatura:

m[i] = M(p(i))

ovvero cosa c'è in ingresso ad ogni transizione.

Esempio vettore m:



1	3
2	0
3	0
4	5
5	0

La transizione j-esima è abilitata in una marcatura espressa dal vettore m, ovvero:

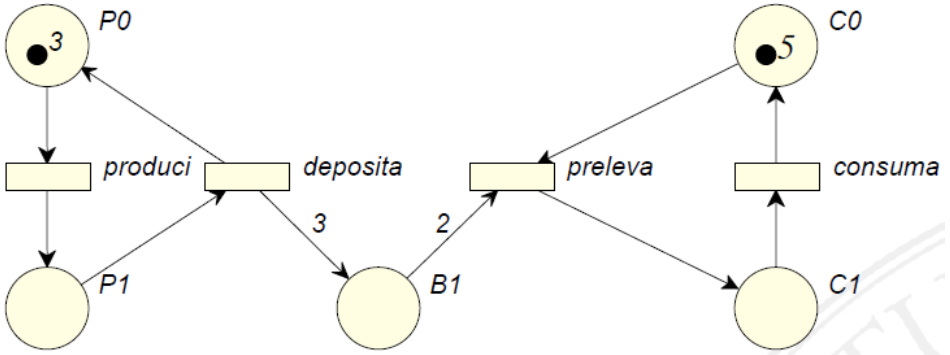
m [tj >

se e solo se:

I[.][j] <= m

elemento per elemento.

Ad esempio, usando questa rete:



La transizione 1 (produci) è abilitata se tutti gli I[.][j] <= m e dunque la transizione 1 è abilitata. La transizione 2 (deposita) invece non è abilitata, come si può evincere dalle seguenti immagini:

I[.][1]		m
1	≤	3
0	≤	0
0	≤	0
0	≤	5
0	≤	0

I[.][2]		m
0	≤	3
1	≤	0
0	≤	0
0	≤	5
0	≤	0

Quando la transizione j-esima scatta in una marcatura m, produce una nuova marcatura m' (m [tj > m'), equivalente a:

$$m' = m - I[.][j] + O[.][j]$$

La matrice di incidenza C è data da:

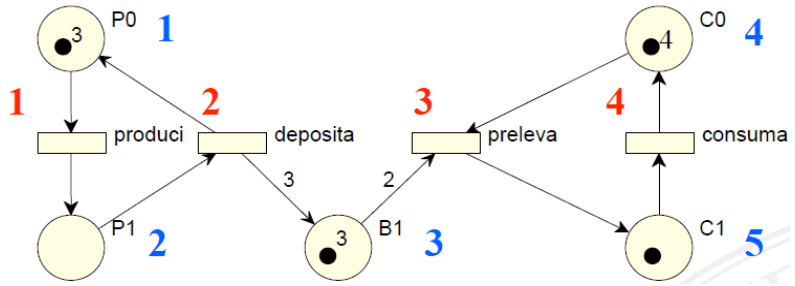
$$C = O - I$$

e risulta utile per ottimizzare lo scatto, ma non è sufficiente per l'abilitazione, in quanto non garantisce la condizione

$$Pre(t) \cap Post(t) = \varnothing$$

ovvero la condizione che stabilisce se una rete è una rete pura.

Esempio di matrice C:



	1	2	3	4
1	-1	1	0	0
2	1	-1	0	0
3	0	3	-2	0
4	0	0	-1	1
5	0	0	1	-1

Una sequenza di scatti:

$$M [t1 > M' \wedge M' [t2 > M'' \rightarrow M [t1t2 > M''$$

ovvero da un posto M, eseguendo una sequenza di scatti Sn, arrivo in un posto Mn:

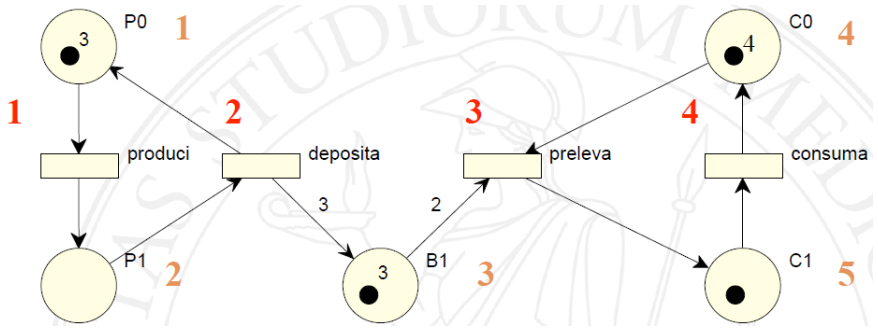
$$M [Sn > Mn$$

che può essere calcolato come segue:

$$Mn = M + C\ s$$

, dove s è il vettore di dimensione |T| contenente il numero di scatti per ogni transizione.

Ad esempio, data la seguente rete di Petri:



e data una sequenza di scatto ammissibile:

$$s = t1, t1, t4, t2, t1, t3, t2, t4, t3, t2, t4, t3, t1, t3$$

, ma in cui non importa l'ordine, quindi riscrivibile come segue:

s = 4 t1, 3 t2, 4 t3, 3 t4

e dunque

s = [4 3 4 3]

per calcolare una nuova marcatura:

|P|×|T| * |T| → |P|

	1	2	3	4
1	-1	1	0	0
2	1	-1	0	0
3	0	3	-2	0
4	0	0	-1	1
5	0	0	1	-1

1	4
2	3
3	4
4	3

1	3
2	0
3	3
4	4
5	1

+

1	-1
2	1
3	1
4	-1
5	1

=

1	2
2	1
3	4
4	3
5	2

Tecnica di analisi basata su invarianti

E' una tecnica basata sulla ricerca di invarianti all'interno della rete:

- P-invarianti, ovvero invarianti sui posti, relativi alla marcatura.
- T-invarianti, ovvero invarianti su sequenze di scatto.

I P-invarianti sono rappresentati per mezzo di un vettore di pesi h di dimensione |P|. Il prodotto vettoriale h m deve essere costante:

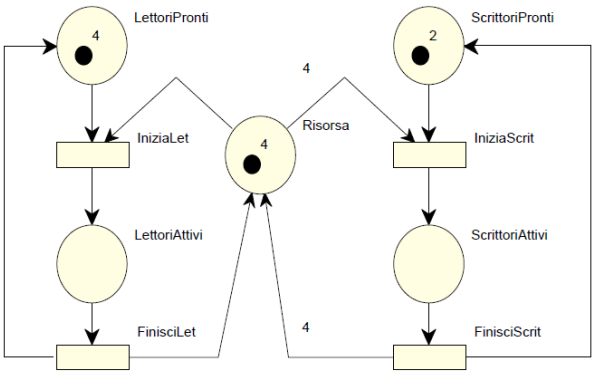
- hm = hm' per ogni m' raggiungibile da m
- m' = m + C s
- hm = hm + hCs
- hCs = 0 per ogni s che rappresenti una sequenza ammissibile

Basta trovare la soluzione del sistema lineare

hC = 0

ovvero un sistema lineare dato dalla funzione h associato ad ogni colonna di C.

Ad esempio:



I	IniziaLet	FinisciLet	IniziaScrit	FinisciScrit
LettoriPronti	1	0	0	0
LettoriAttivi	0	1	0	0
Risorsa	1	0	*4*	0
ScrittoriPronti	0	0	1	0
ScrittoriAttivi	0	0	0	1
O	IniziaLet	FinisciLet	IniziaScrit	FinisciScrit
LettoriPronti	0	1	0	0
LettoriAttivi	1	0	0	0
Risorsa	0	1	0	*4*
ScrittoriPronti	0	0	0	1
ScrittoriAttivi	0	0	1	0
C	IniziaLet	FinisciLet	IniziaScrit	FinisciScrit
LettoriPronti	-1	1	0	0
LettoriAttivi	1	-1	0	0
Risorsa	-1	1	*-4*	*4*
ScrittoriPronti	0	0	-1	1
ScrittoriAttivi	0	0	1	-1

Risolviamo il sistema hC = 0, otteniamo:

-h0 +h1 -h2 = 0
+h0 -h1 +h2 = 0
-4h2 -h3 +h4 = 0
+4h2 +h3 -h4 = 0

Per trovarne le soluzioni, utilizziamo l'algoritmo di Farkas, che trova le basi minime semipositive:

—---------- Algoritmo di Farkas

D0 := (C | En);
for i := 1 to m do
    for d1, d2 rows in Di−1 such that d1(i) and d2(i) have opposite signs do
        d := |d2(i)| · d1 +|d1(i)| · d2; (* d(i) = 0 *)
        d⇥ := d/gcd(d(1), d(2), . . . , d(m +n));
        augment Di−1 with d⇥ as last row;
    endfor;
    delete all rows of the (augmented) matrix Di−1 whose i-th component
    is different from 0, the result is Di ;
endfor;
delete the first m columns of Dm

—----------

Una combinazione lineare di P-invarianti è anch'essa un P-invariante.

Un P-invariante che ha tutti pesi >= 0 è detto semipositivo.

Se un posto ha peso positivo in un P-invariante semipositivo, allora il posto è limitato.

Una rete P/T si dice ricoperta da P-invarianti se per ogni posto esiste almeno un P-invariante semipositivo il cui peso di tale posto sia positivo, cioè se è una rete limitata.

I T-invarianti fanno riferimento a sequenze di scatti:
- cicliche (cioè che possono essere ripetute)
- che riportano nella condizione iniziale

in cui valgono:
- m’ = m + Cs
- m’ = m

Perciò si cercano soluzioni del sistema

$$Cs = 0$$

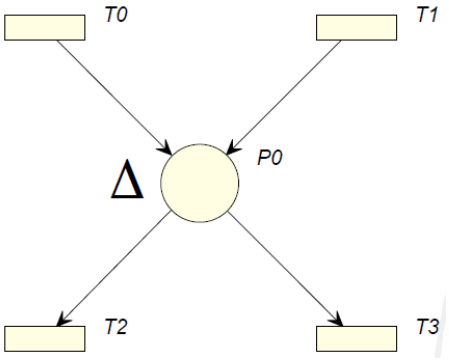
, ma non è detto che siano tutte valide.

Reti Temporizzate

Sono utilizzate anche per modellare sistemi Hard Real-time, in cui bisogna soddisfare dei vincoli temporali senza errori.

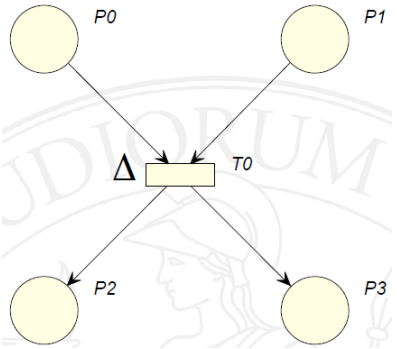
Vi sono diversi modi per aggiungere il tempo (deterministico) alle reti di Petri, ad esempio:
- Ritardi sui posti
- Ritardi sulle transizioni
- Tempi di scatto sulle transizioni
  - Unici o multipli
  - Fissi o variabili
  - Assoluti o relativi

Tempo sui posti
Il tempo associato ai posti indica il tempo che un gettone deve rimanere nel posto stesso prima di potere essere considerato come parte di una abilitazione, ovvero la durata minima di permanenza del gettone nel posto (quanto quella parte di sistema rimane in quello stato).

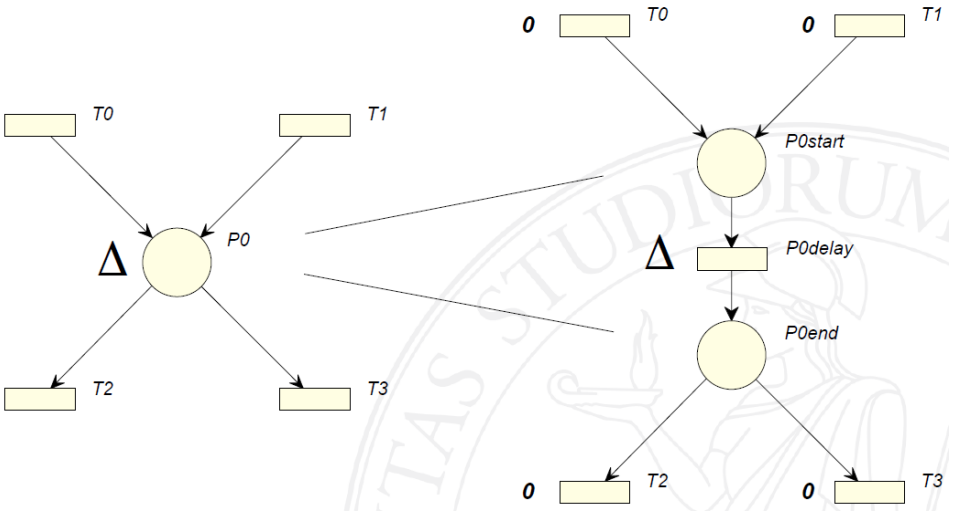


Tempo sulle transizioni
Il tempo associato alle transizioni può essere usato per indicare due cose diverse:
- Un ritardo di scatto (cioè la durata di una azione): le transizioni scattano non appena possibile, mentre gli scatti hanno una durata fissa.
- Il momento in cui lo scatto avviene: le transizioni scattano in un momento fissato (in diverse maniere, dipendentemente dai diversi modelli), mentre lo scatto è istantaneo.

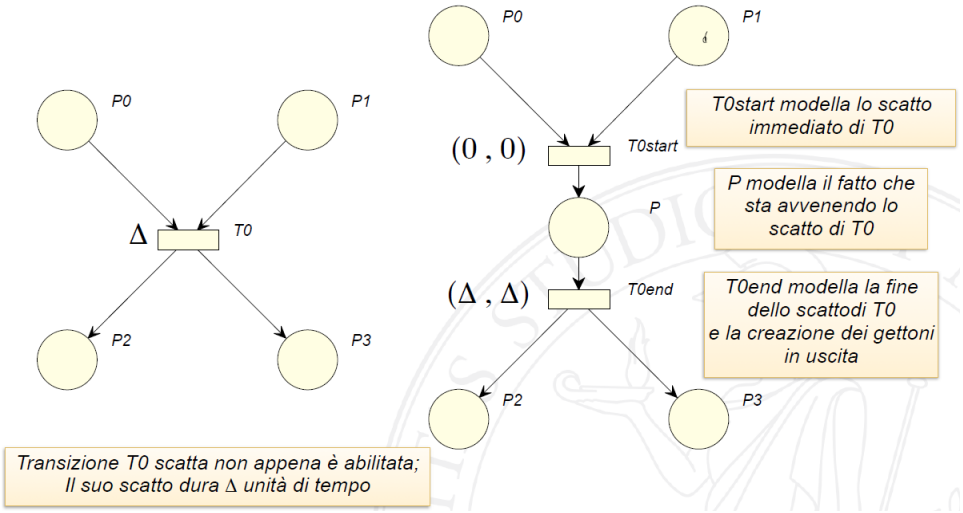
Esistono anche modelli misti, in cui si può specificare sia l’istante che la durata dello scatto.



Si può spostare il tempo associato ai posti nelle transizioni nel seguente modo:



E viceversa:



Momenti di scatto unici o multipli

Vi sono due tipi di momento di scatto:

- Momento di scatto unico: alla transizione viene associato un valore singolo.
- Momenti di scatto multipli: alla transizione vengono associati più possibili valori; tra questi si sceglierà poi il tempo effettivo di scatto della transizione.

Il primo può essere visto come un caso particolare del secondo.

Insiemi costanti o variabili

Gli insiemi possono essere di due tipi:

- Constanti: l'insieme dei tempi di scatto è definito staticamente (ad esempio TPNs: gli estremi dell'intervallo dei possibili tempi di scatto sono costanti).
- Variabili: l'insieme dei tempi di scatto può variare dinamicamente (ad esempio: TB nets: gli insiemi dei tempi di scatto sono definiti come funzioni dei timestamps gettoni che abilitano la transizione, oppure HLTPNs (Ghezz): gli insiemi dei tempi di scatto sono definiti come funzioni dei timestamps e dei valori gettoni che abilitano la transizione).

Il primo può essere visto come un caso particolare del secondo.

Tempi di scatto assoluti o relativi

I tempi di scatto possono essere:

- Relativi: i tempi di scatto possono essere espressi solo in termini relativi al tempo di abilitazione (TPNs)
- Assoluti: i tempi di scatto possono essere espressi in termini relativi a tempi assoluti e/o al tempo dei singoli gettoni che compongono l'abilitazione (TB nets, TCPNs)

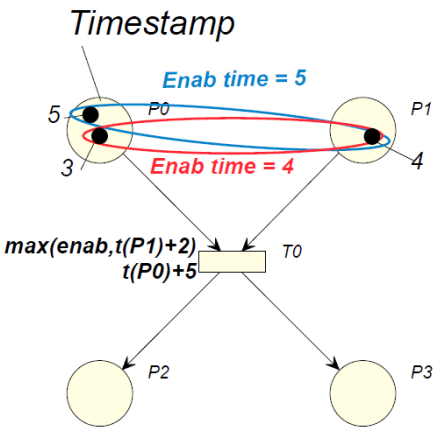
Il primo può essere visto come un caso particolare del secondo.

Time Basic nets (Ghezzi, 1989)

In questo tipo di reti temporizzate il tempo è associato alle transizioni.
Vengono associati degli insiemi di tempi di scatto possibili (definiti in maniera dinamica) come funzioni che possono fare riferimento a tempi assoluti e ai tempi dei singoli gettoni.

In esse i gettoni non sono anonimi, bensì sono dei timestamp.

Il tempo di abilitazione "enab" è uguale al massimo tra i timestamp dei gettoni che compongono la tupla abilitante (enabling tuple).



L'insieme dei tempi di scatto non possono essere minori del tempo di abilitazione (ovviamente una transizione non può scattare prima di essere abilitata).

Il tempo di scatto è scelto all'interno del set dei possibili timestamp di tutti i gettoni prodotti.

Formalmente una rete TB è una sestupla <P, T, Θ; F, tf, m0>, dove:

- P, T, F sono come nelle reti di Petri normali.
- Θ è un insieme numerico (il dominio temporale).
- tf associa ad ogni transizione una funzione temporale tf_t, dove data una tupla abilitante en, tf_t(en) ⊆ θ.
- m0: P → { (θ, mul(θ)) | θ ∈ Θ }, ovvero un multiset che esprime la marcatura iniziale.

Semantica temporale debole (Weak Time Semantic, WTS)

Informalmente in una rete TB con semantica temporale debole (WTS):

- Una transizione può scattare solo in uno degli istanti identificati dalla sua funzione temporale.
- Una transizione non può scattare prima di essere stata abilitata.
- Una transizione anche se abilitata non è forzata a scattare

Una rete di questo tipo è dunque utile per modellare eventi solo parzialmente definiti, ad esempio eventi che dipendono da componenti non modellati o modellizzabili, oppure una decisione umana o un guasto. Al contrario dei modelli stocastici, non ci interessa modellare la probabilità con cui possa accadere qualcosa, ma solo che può accadere.

Definiamo ora diversi assiomi:

Assioma 1: Monotonicità rispetto alla marcatura iniziale
Tutti i tempi di scatto di una sequenza di scatto devono essere non minori di uno qualunque dei timestamp dei gettoni della marcatura iniziale.

Ovvero la marcatura deve essere consistente: cioè non deve contenere gettoni prodotti nel futuro.

Assioma 2: 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.

E' consistente con la proprietà intuitiva del tempo, poiché il tempo non torna indietro e inoltre due o più transizioni possono scattare nello stesso istante.

Assioma 3: Divergenza del tempo (non-zenonicità)
Non è possibile avere un numero infinito di scatti in un intervallo di tempo finito.

E' consistente rispetto alla proprietà intuitiva del tempo, in quanto:

- Il tempo avanza
- Non si può fermare
- Non è suddivisibile in infinitesimi

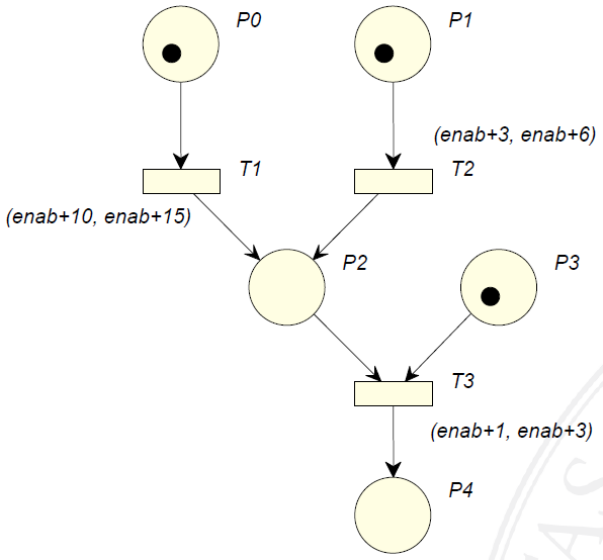
Le sequenze di scatti che soddisfano gli assiomi 1 e 3 sono chiamate sequenze ammissibili in semantica debole (WTS).

Le sequenze di scatti che soddisfano gli assiomi 1, 2 e 3 sono chiamate sequenze ammissibili in semantica monotonica debole (MWTS).

Teorema WTS ≡ MWTS (equivalenza tra WTS e MWTS)
Per ogni sequenza di scatti debole s esiste una sequenza di scatti monotonica debole ottenibile per semplice permutazione delle occorrenze degli scatti.

Possono essere usate delle tecniche di analisi per reti di Petri di alto livello anche con reti con semantica debole.

Un esempio di equivalenza:



Una possibile sequenza WTS (assumendo timestamp iniziali tutti uguali a zero):

- T1 scatta al tempo 12
- T3 scatta al tempo 14
- T2 scatta al tempo 4

La equivalente MWTS:

- T2 scatta al tempo 4
- T1 scatta al tempo 12
- T3 scatta al tempo 14

Semantica temporale forte (strong)

Informalmente in una rete TB con semantica temporale forte:

- Una transizione può scattare solo in uno degli istanti identificati dalla sua funzione temporale.
- Una transizione non può scattare prima di essere stata abilitata.
- 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.

E' la semantica più diffusa (utile per i sistemi deterministici). E' la semantica di default in molti modelli temporizzati (TPNs).

Assioma 4: Marcatura forte iniziale

Il massimo tempo di scatto di tutte le abilitazioni nella marcatura iniziale deve essere maggiore o uguale del massimo timestamp associato ad un gettone della marcatura.

La marcatura iniziale deve essere consistente con la nuova semantica. Ovvero, il gettone non avrebbe potuto essere creato a un istante successivo a quel timestamp senza che prima fosse scattata la transizione (entro il suo tempo massimo).

Assioma 5: Sequenza di scatti forte (STS)

Una sequenza di scatti 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.

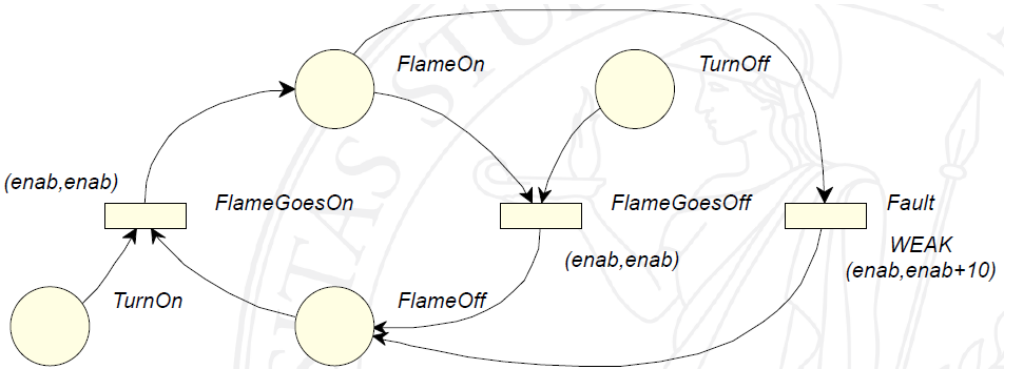
Ovvero una transizione abilitata DEVE scattare entro il suo massimo tempo di scatto, se non viene disabilita prima da un altro scatto.

Sequenze di scatto che soddisfano gli assiomi 1,2,3,4 e 5 sono dette sequenze ammissibili in semantica forte (STS).

Mixed time semantics

In questa modalità la semantica forte o debole viene associata alle singole transizioni invece che all'intera rete:

- Transizioni forti devono scattare entro il loro tempo massimo a meno che non vengano disabilitate prima.
- Transizioni deboli possono scattare entro il loro insieme di tempi di scatto.



//manca lezione 23

//simboli: ∃ ∀ ∈ ∉ ⇔ ¬ ⇒ ⊂ ⊆ ⊃ ⊇ ∪ ∅ × ≠ ∩ Σ ω θ Θ ≡ → TODO DELETE THIS LINE

//simboli: ∃ ∀ ∈ ∉ ⇔ ¬ ⇒ ⊂ ⊆ ⊃ ⊇ ∪ ∅ × ≠ ∩ Σ ω θ → TODO DELETE THIS LINE