

Contents

| | | |
|----------|---|-----------|
| 1 | Introduzione | 7 |
| 1.1 | Modelli | 7 |
| 1.2 | Definizioni | 7 |
| 2 | Modelli | 8 |
| 2.1 | Ciclo di vita del software | 8 |
| 2.2 | Waterfall | 8 |
| 2.3 | Ciclo di vita incrementale | 9 |
| 2.4 | Ciclo di vita evolutivo | 10 |
| 2.5 | Processo Unificato - UP | 11 |
| 2.6 | Nuovi approcci allo sviluppo del software | 12 |
| 2.7 | Modelli Concettuali | 13 |
| 2.8 | UML | 13 |
| 2.8.1 | Modelli di classe | 13 |
| 2.8.2 | Esempi di modelli di dominio e costrutti vari | 14 |
| 2.9 | Interpretazione del modello delle classi | 22 |
| 2.10 | Espressioni navigazionali e condizioni | 22 |
| 2.11 | Invarianti | 23 |
| 2.12 | Regole di validazione | 24 |
| 2.13 | Operatore distinct | 24 |
| 2.14 | Associazioni ordinate | 24 |
| 2.15 | Operazioni | 25 |
| 2.16 | Creare un object model | 25 |
| 2.17 | Attributi obbligatori e derivati | 26 |
| 2.18 | Relazioni derivate | 26 |
| 3 | Analisi dei requisiti | 27 |
| 3.1 | Requisiti del software | 27 |
| 3.2 | IEEE STD-830 | 28 |
| 3.3 | Model-Driven Development - con studio di caso | 28 |
| 3.3.1 | Invariante | 30 |
| 3.4 | Casi D'uso | 30 |
| 3.5 | Alcune regole e consigli | 31 |

| | |
|---|-----------|
| CONTENTS | 2 |
| 4 Meta-modelli | 32 |
| 4.1 Business Process | 34 |
| 4.1.1 Diagrammi di attività | 34 |
| 4.2 Modelli Dataflow | 36 |
| 4.2.1 Top-Down Decomposition | 42 |
| 4.3 Modelli di stato | 45 |
| 4.3.1 Transazioni Temporizzate | 47 |
| 4.3.2 Macchina a Stati Operazionali | 48 |
| 4.3.3 Macchina a stati gerarchica | 49 |
| 4.3.4 Macchine di Moore e Mealy | 51 |
| 4.4 Pattern | 53 |
| 4.4.1 Façade | 54 |
| 4.4.2 Factory Method | 54 |
| 4.4.3 Builder | 54 |
| 4.4.4 Prototype | 54 |
| 4.4.5 Singleton | 54 |
| 4.4.6 Composite | 55 |
| 4.4.7 Iterator | 56 |
| 4.4.8 Command | 56 |
| 4.4.9 Observer | 56 |
| 4.4.10 Livelli di pattern | 57 |
| 5 Reti di Petri | 59 |
| 5.1 Struttura di una rete | 59 |
| 5.2 Effetto di una transizione | 61 |
| 5.3 Marcature | 62 |
| 5.4 Control Flow | 63 |
| 5.5 Una modellazione da evitare | 63 |
| 5.5.1 Scelta libera asimmetrica | 64 |
| 5.5.2 Scelta libera estesa | 65 |
| 5.5.3 Confusione Simmetrica | 65 |
| 5.6 Producer e Consumer | 65 |
| 5.7 Costruire il grafo di raggiungibilità (copertura) | 67 |
| 5.7.1 Definizione raggiungibilità | 68 |
| 5.8 Proprietà comportamentali | 69 |
| 5.8.1 Esempi | 69 |
| 5.8.2 Archi bidirezionali | 71 |
| 5.9 Analisi di sottoclassi delle reti di Petri | 71 |
| 5.9.1 Macchine a stati | 72 |
| 5.9.2 Grafi marcati (MGs) | 72 |
| 5.9.3 Reti Free-Choice | 76 |
| 5.9.4 Reti Free Choice Estese | 77 |
| 5.9.5 Reti a scelta asimmetrica | 79 |

| | |
|--|------------|
| CONTENTS | 3 |
| 5.10 Riduzione della complessità di una rete | 80 |
| 5.10.1 Regole di riduzione in serie | 80 |
| 5.10.2 Regole di riduzione in parallelo | 81 |
| 5.10.3 Riduzioni in loop | 81 |
| 5.10.4 Esempio di riduzione | 82 |
| 6 Reti temporizzate | 84 |
| 6.1 Simulazione ad eventi discreti | 84 |
| 6.2 Calcolo del tempo ciclo | 85 |
| 7 Reti Object-Oriented | 86 |
| 7.1 Funzionamento | 88 |
| 8 Modelli di processi | 91 |
| 8.1 Scenario centralizzato | 91 |
| 8.2 Tipi di task: entry task, exit task | 94 |
| 8.3 Scelte strutturate e non strutturate | 94 |
| 8.4 Life cycle degli ordini | 95 |
| 8.5 Tipi di task | 98 |
| 8.5.1 Task Partecipativo | 98 |
| 8.5.2 Task di timeout | 99 |
| 8.5.3 Scelta semplice | 99 |
| 8.5.4 Scelta composta | 100 |
| 8.5.5 Scelta composta con restrizioni | 100 |
| 8.5.6 Test composto associativo | 100 |
| 8.5.7 Task composto generativo | 101 |
| 8.5.8 Task composto con restrizione | 101 |
| 8.5.9 Effetti dei task e dataflow | 101 |
| 8.6 B2B Systems | 105 |
| 8.6.1 Assunzioni | 106 |
| 8.6.2 Modello di collaborazione | 106 |
| 8.6.3 Blocco Opt e Loop | 109 |
| 8.6.4 Task d'interazione | 112 |
| 8.6.5 Uso dei riduttori | 114 |
| 9 SOA - Service Oriented Architecture | 116 |
| 10 Servizio sincrono | 117 |
| 10.1 Collaborazione | 117 |
| 10.2 Servizi | 118 |
| 10.2.1 Collaborazione Multipla | 118 |

| | |
|--|------------|
| 11 DDD - Domain Driven Design | 119 |
| 11.1 Bounded context | 119 |
| 11.2 Ubiquitous Language | 119 |
| 11.3 Capability | 119 |
| 12 Microservizi | 120 |
| 12.1 Evoluzione da monoliti a microservizi | 120 |
| 12.1.1 Integrazione | 120 |
| 12.1.2 Sostituzione | 120 |
| 12.1.3 Nuova applicazione basata su microservizi | 120 |
| 13 Business Process | 121 |
| 13.1 BP Models | 121 |
| 13.2 Workflow Pattern | 121 |
| 13.2.1 Basic control patterns | 121 |
| 13.2.2 Advanced control and synchronization patterns | 121 |
| 13.3 Control Flow e Data Flow | 122 |
| 13.4 Esempio | 122 |
| 13.5 Elementi grafici dei BP Models | 122 |
| 13.6 Esempio 2 | 123 |
| 13.6.1 Task | 125 |
| 13.6.2 Events | 126 |
| 13.7 Altri elementi grafici dei modelli | 128 |
| 13.7.1 Task con più input/output | 128 |
| 13.7.2 Loop | 128 |
| 13.7.3 Event-based gateway | 128 |
| 13.7.4 Oggetti Dati | 129 |
| 13.7.5 Multi-instance activities | 130 |
| 13.7.6 Sotto processi | 131 |
| 13.7.7 Boundary Events | 131 |
| 13.8 Difficoltà con BPMN | 132 |
| 14 Verifica del software | 133 |
| 14.1 Verifica e Validazione | 133 |
| 14.2 Tecniche di Verifica | 133 |
| 14.2.1 Inspection | 133 |
| 14.3 Tipi di testing | 135 |
| 14.3.1 White Box Testing | 135 |
| 14.3.2 Correlazioni | 140 |
| 14.3.3 Black box testing | 140 |
| 14.3.4 Obiettivi di testing | 142 |
| 14.3.5 Test-driven development | 143 |

| | |
|---|------------|
| CONTENTS | 5 |
| 15 Gestione della configurazione | 144 |
| 15.1 Version management | 144 |
| 15.1.1 Versionamento lineare | 144 |
| 15.1.2 Versionamento a grafo | 145 |
| 15.2 Sviluppo di un sistema | 145 |
| 15.3 Change management | 146 |
| 15.4 Issue Tracking System | 147 |
| 15.5 Git | 147 |
| 15.5.1 Contenuto del repository | 148 |
| 15.5.2 Diramazioni | 148 |
| 15.5.3 Convergenza | 148 |
| 15.5.4 Repository remoto | 148 |
| 15.5.5 Eventi | 149 |
| 15.5.6 GitHub | 149 |
| 16 Project Management | 150 |
| 16.1 Pianificazione | 150 |
| 16.2 Software process | 154 |
| 16.3 CMM | 154 |
| 16.3.1 CMMI | 154 |
| 16.3.2 Process areas | 154 |
| 17 Qualità del software | 155 |
| 17.1 functional suitability | 155 |
| 17.2 performance efficiency | 156 |
| 17.3 compatibility | 156 |
| 17.4 Usability | 156 |
| 17.5 Reliability | 156 |
| 17.6 Security | 156 |
| 17.7 Maintainability | 157 |
| 17.8 Portability | 157 |
| 17.9 Una nona caratteristica: dependability | 157 |
| 17.10 Misure | 157 |
| 17.10.1 Metriche interne | 157 |
| 18 Organizzazione dello sviluppo del software | 158 |
| 18.1 Agile Development | 158 |
| 18.2 Extreme Programming | 158 |
| 18.2.1 Planning game | 159 |
| 18.3 Scrum | 160 |
| 18.3.1 Product Backlog | 160 |
| 18.3.2 Sprint Backlog | 160 |
| 18.3.3 Elemento fatto | 160 |
| 18.3.4 Sprint | 160 |

| | |
|-----------------|---|
| <i>CONTENTS</i> | 6 |
|-----------------|---|

| | |
|--|------------|
| 18.3.5 Ruoli e competenze | 160 |
| 19 DevOps | 162 |
| 19.1 Continuous Integration e Delivery | 162 |
| 19.2 Deployment | 162 |
| 19.3 Kubernetes | 163 |
| 19.4 DevOps e Microservizi | 163 |

Chapter 1

Introduzione

1.1 Modelli

Si parla di modelli per dare la possibilità a tutti gli interessati di comprendere le soluzioni proposte. Vedremo diversi tipi di modelli per la formalizzazione di soluzioni. L'ingegneria del software è in continua evoluzione e fa uso di strumenti, processi e *best practices*. L'approccio dell'ingegneria del software è *model-driven*, cioè guidata da modelli talvolta generati automaticamente. L'ingegneria del software è una disciplina molto ampia e non comprende solo lo sviluppo del software ma può essere vista come una modalità per formalizzare un obiettivo che si vuole conseguire.

1.2 Definizioni

Software Engineering: Approccio sistematico per lo sviluppo, la messa in operazione, manutenzione e il ritiro del software. In questo corso principalmente ci basiamo e ci concentriamo sulla parte di *Development*. Si applica quello che viene definito il principio scientifico.

- Trasformare una richiesta in una soluzione funzionante
- Ritirare il software quando le richieste non sono più quelle richieste

Chapter 2

Modelli

2.1 Ciclo di vita del software

Sono delle fasi che suddividono le vari fasi che un software attraversa. Ogni fase a sua volta si suddivide in attività. Ogni fase ha una serie di input e produce una serie di output.

Lo sviluppo di un prodotto software è un'attività difficile che deve essere guidata da principi e pratiche che permettano lo sviluppo di prodotti di qualità.

'La qualità del prodotto dipende dalla qualità del processo'

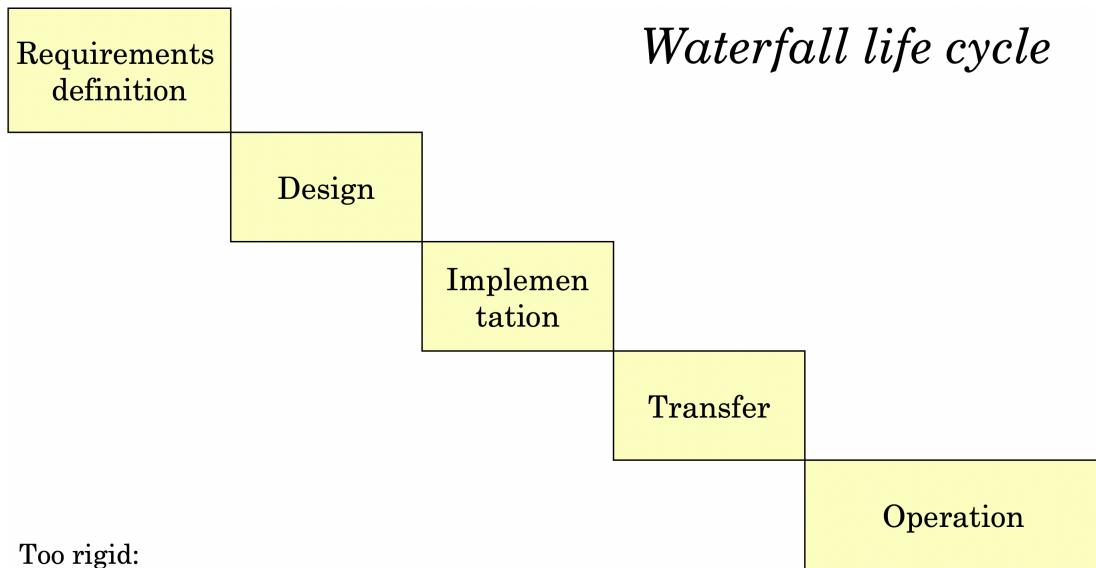
Vi sono diverse modalità per approcciare allo sviluppo software e di seguito vi è un'introduzione a tutte le tecniche.

2.2 Waterfall

La prima tecnica è Waterfall, o a cascata. Le fasi che caratterizzano un processo a cascata sono 5:

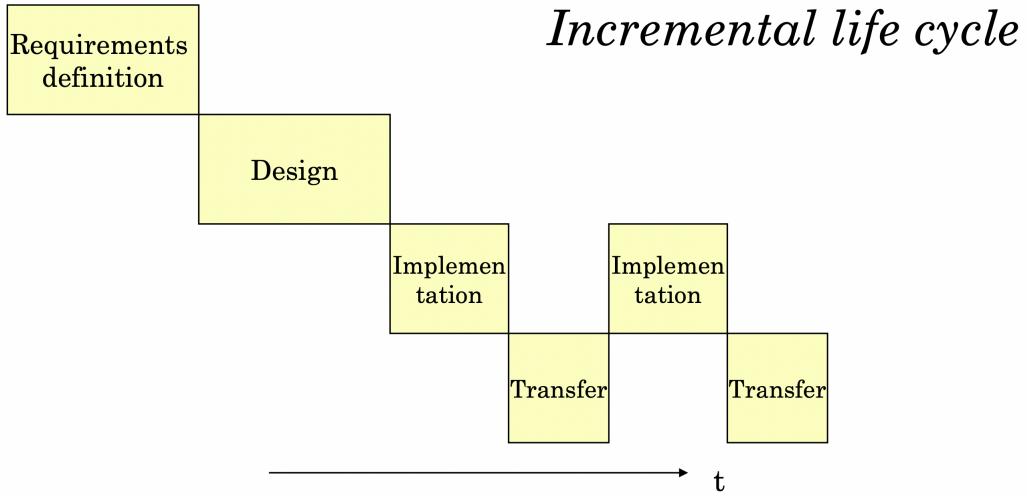
- **Definizione dei requisiti:** fase per capire i requisiti del sistema, cosa il prodotto deve soddisfare. Si cerca di definire i vincoli e le funzionalità principali
- **Design:** Definizione dei moduli che comporranno il software e delle interazioni dei moduli definiti. Si può anche decidere di importare moduli già esistenti invece di crearne una copia.
- **Implementazione:** Sviluppo del software e testing
- **Trasferimento al committente:** Il committente deve accettare lo stato dello sviluppo e dei testing ed effettuare il deployment, cioè rendere il sistema funzionante.
- **Operation**

Questa modalità è stata una delle prime ad essere utilizzata e in realtà è ancora oggi utilizzata ampiamente anche se presenta diversi problemi. Il primo problema è quello della fase di trasferimento al committente, essendo un processo che non contempla delle revisioni intermedie il tempo di rilascio è lungo quanto il tempo di sviluppo del software stesso e durante questo lasso di tempo il committente non ha nulla. L'attesa del committente potrebbe essere addirittura di anni. La seconda problematica è che i requisiti devono essere congelati dopo la definizione, non è possibile cambiarli poiché tutto quello che viene fatto dopo la fase di definizione dei requisiti è basato sui requisiti raccolti e il committente si accorgerà solo dopo il trasferimento se i requisiti sono quelli giusti o no.



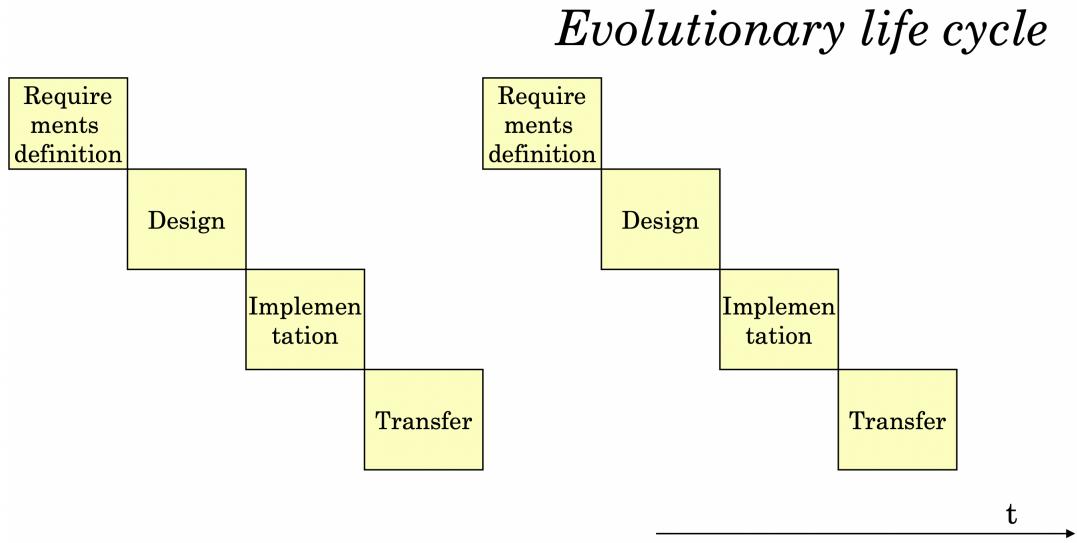
2.3 Ciclo di vita incrementale

È una prima estensione che permette di migliorare l'approccio a cascata. Le prime due fasi rimangono le stesse e sono consecutive mentre la fase di implementazione può essere eseguita più volte. In pratica si effettuano sviluppi intermedi necessari per realizzare un prodotto non finito da fornire al committente. Quindi alla fine delle diverse fasi di implementazione viene effettuato un trasferimento.



2.4 Ciclo di vita evolutivo

Molto drastico, permette di evolvere il sistema in tutto e per tutto, i requisiti sono parzialmente compresi ma si cerca comunque di rendere tutto modificabile. Ogni versione che viene rilasciata è una sorta di prototipo.



Si noti che esistono due diversi tipi di prototipi:

- **Throwaway:** Cioè usa e getta, per una soluzione che serve per rendere l'idea. Per poter creare il prototipo potrebbero essere usate delle tecnologie che non necessariamente rispecchiano quelle del prodotto definitivo.

- **Evolutivo:** Si modifica lo stesso prototipo che evolve nel tempo e diventa il prodotto definitivo.

2.5 Processo Unificato - UP

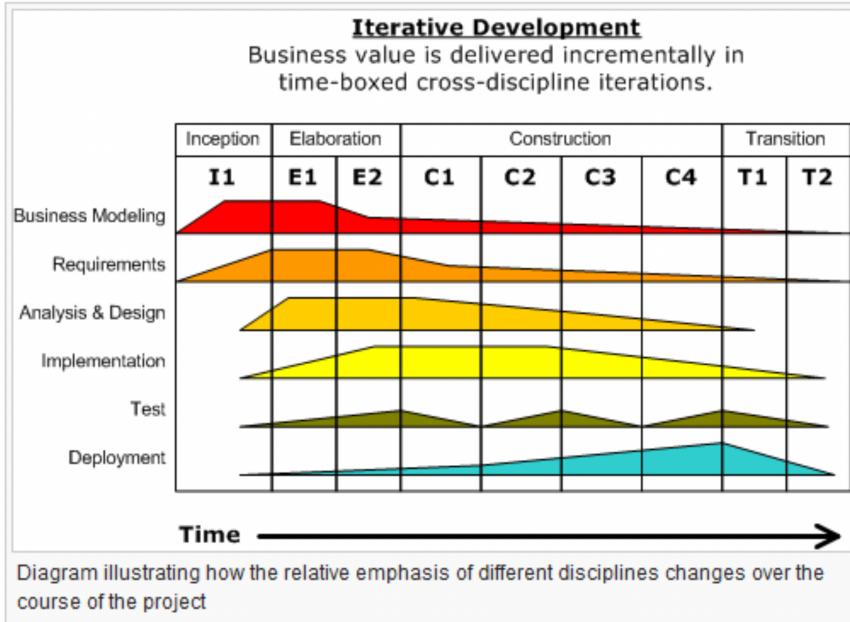
È una modalità che unisce gli approcci incrementali ed evolutivi. Anche questa modalità è divisa in diverse fasi che presentano nomi differenti dal modello a cascata:

- **Inception:** è la prima fase, in cui si devono valutare gli elementi chiave, i vincoli, i rischi e si deve iniziare a valutare lo *schedule* del progetto, cioè *project management*.
- **Elaboration:** Analisi dettagliata dei requisiti utilizzando *use cases* e *class diagrams*.
- **Construction:** Implementazione supportata da diversi UML dettagliati
- **Transition:** Deployment. Il deployment in questo caso prende il posto del trasferimento.

Vi è una grande unione tra processo unificato e UML. I diagrammi UML vengono utilizzati in diverse fasi e per diversi scopi. Nel processo unificato una fase può essere eseguita più volte e solitamente ogni fase comprende molte attività tra cui:

- **Business modelling:** Serve a definire il contesto in cui il software andrà ad operare. Questa fase in particolare serve per creare un ponte tra gli analisti del business e gli analisti del software.
- **Requirements**
- **Analysis and design**
- **Implementation**
- **Test**
- **Deployment**

Non è strano per questo modello che una parte dell'implementazione si trovi nella fase di Inception, magari per la costruzione di un prototipo, oppure che l'attività di definizione dei requisiti si trovi anche nella fase di Transition. In generale l'idea di base è che ogni fase è caratterizzata da diverse attività invece che una sola come si fa nel modello a cascata. Il grafico che segue da l'idea della presenza di ogni attività all'interno di una fase:



È visibile dal grafico che tutte le attività sono presenti in praticamente tutte le fasi, anche se delle volte in piccolissima parte. Questo da proprio il senso di iterazione, per questo il modello viene anche chiamato modello iterativo. Ogni qualvolta che viene iniziata una nuova fase si effettuano nuovamente tutte le attività. Dipendentemente dalla fase un'attività può occupare più o meno tempo. Vi sono delle attività di supporto che vengono ripetute durante tutto il ciclo di vita del software che non sono rappresentate nel grafico ma sono comunque presenti:

- **Project Management:** Planning, Staffing, Monitoring, Controlling e leading
- **Version control and configuration management**
- **Verification:** Are we building the product right? Può essere fatta in due modi, anche in parallelo, verifica statica effettuata da persone oppure tramite test
- **Validation:** Are we building the right product? Ci si accerta che il prodotto che si sta sviluppando sia quello corretto che soddisfa le richieste
- **Controllo qualità**

2.6 Nuovi approcci allo sviluppo del software

Vi sono delle modalità di sviluppo moderne che hanno principi differenti da quelle che si basano sul ciclo di vita del software:

- **Agile Development**

- **SCRUM**

- **Continuous Integration:** Alla fine di una fase di sviluppo i diversi developer che lavorano al prodotto si occupano di effettuare l'integrazione e di effettuare testing
- **Continuous Delivery:** Rilascio del prodotto, anche intermedio, al committente alla fine di una fase di sviluppo

2.7 Modelli Concettuali

Sono modelli di alto livello ma comunque rigorosi che riescono a mettere alla luce i punti chiave del sistema. Nello sviluppo software i modelli concettuali ci permettono di generare una prima fase del codice per inizializzare lo sviluppo. Un modello da cui si può generare del codice viene chiamato **modello operazionale**.

- model → code
- model = code

Le due espressioni non sono uguali, la seconda è molto più forte ed esprime la possibilità di usare i modelli per avere un codice totale, comprensivo di logica e dettagli implementativi. La prima invece è un po' più debole, indica che il modello può fornire una prima bozza del codice come ad esempio le firme dei metodi e delle classi

2.8 UML

Standardizzato dalla OMG(Object Modelling Group), è un modo per modellare attraverso l'utilizzo di notazioni, noi con notazioni intendiamo una modalità descrittiva standard grafica, quindi utilizzando disegni. Esistono due tipologie principali di modelli:

- **Strutturali:** Fa uso di modelli delle classi e modelli degli oggetti
- **Comportamentali:** Fa uso di casi d'uso, modelli delle attività, modelli di stato e modelli di sequenza

Esiste anche un linguaggio, chiamato Object Constraint Language che è un linguaggio simile a java capace di definire vincoli. Vedremo questo linguaggio per capire il perché sia molto utile.

2.8.1 Modelli di classe

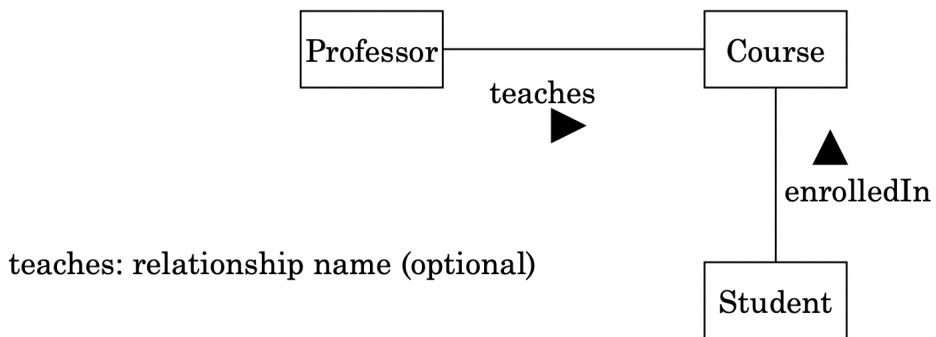
Una classe è formata da individui, chiamati anche oggetti. Gli oggetti sono legati tra loro da relazioni. Un class model rappresenta le classi, attributi e relazioni. Un modello degli oggetti rappresenta una serie di oggetti con le loro interazioni.

Vi sono una serie di aspetti dei modelli classi-relazioni che caratterizzano i diversi tipi degli elementi che usiamo per creare un modello delle classi:

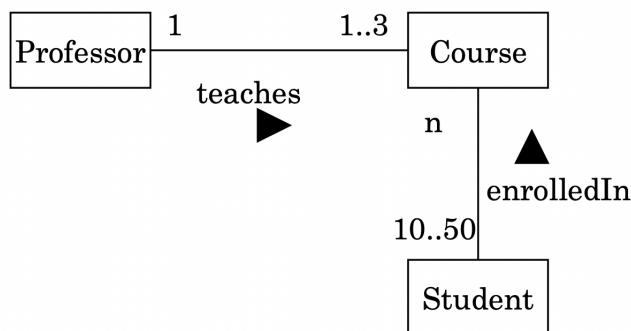
- Relazioni: Associative, composizione, ereditarietà, ricorsive
- Attributi normali e associativi
- Espressioni navigazionali
- Attributi necessari e relazioni necessarie
- Attributi derivati e relazioni derivate
- Invarianti e regole di validazione

2.8.2 Esempi di modelli di dominio e costrutti vari

Modello di dominio



Come si nota ci sono 3 classi nello schema, *Professor*, *Course* e *Student* e tra queste classi vi sono delle relazioni. Le frecce stanno ad indicare il verso di lettura della relazione in modo da dare un senso compiuto alla frase che si pronuncia. Come si vede anche dalla foto sarebbe possibile definire una relazione tra *Professor* e *Student* che però è derivata passando da *Course* quindi non è obbligatoria. Questo modello può essere arricchito con diversi dettagli, per esempio con le cardinalità:



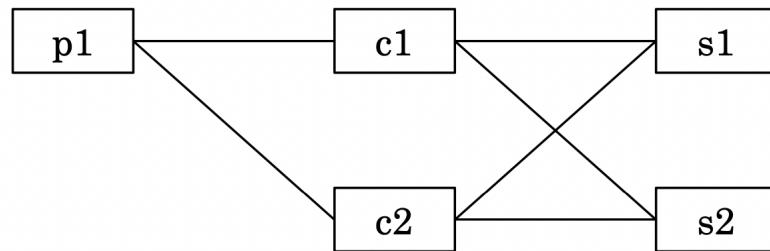
Le tipologie di cardinalità sono molteplici e sono descritti dalla tabella

| Indicator | Meaning |
|-------------|----------------------------------|
| 0..1 or 0,1 | Zero or one |
| * | Zero or more |
| "n" | One or more (not known a priori) |
| n | n (where $n > 1$) |
| 0..n | Zero to n (where $n > 1$) |
| m..n | m to n (where $n > m$) |

Inoltre se non si inserisce una cardinalità si da per scontato che sia *

Modello degli oggetti

La differenza con il modello delle classi è che il modello degli oggetti vengono rappresentati delle istanze delle classi e si usa per rappresentare le interazioni tra gli oggetti



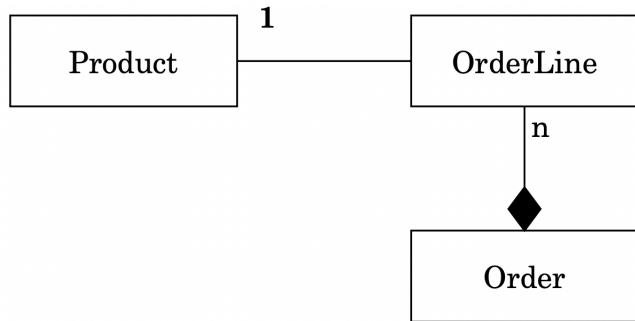
Nel modello degli oggetti i rettangolini che vediamo sono istanze di una classe; l'oggetto **p1** è istanza di *Professor*, **c1** e **c2** sono istanze di *Course* mentre **s1** e **s2** sono istanze di *Student*. Un oggetto completo, con la rappresentazione completa si presenta come segue

| |
|-------------------|
| Student |
| name = "John Doe" |
| id = "..." |

ma per semplicità si può anche scrivere semplicemente un nome identificativo dell'oggetto, come nel modello di esempio e si potrebbe anche inserire ad esempio la classe di appartenenza dell'oggetto con la dicitura **Student**: **s1** invece di solo **s1**

Composizione

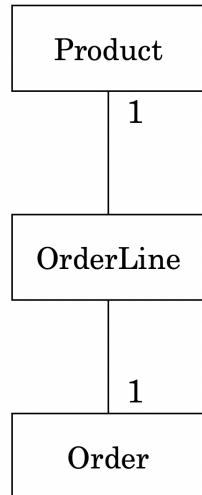
Un costrutto molto importante è la composizione che si indica con un rombo pieno e si usa quando una classe è composta da una seconda classe. UML ha anche introdotto il concetto di aggregazione che però è molto difficile da gestire e quindi non lo si usa, si preferisce una semplice associazione. Per poter capire meglio la composizione si può pensare anche a una relazione contenitore-contenuto.



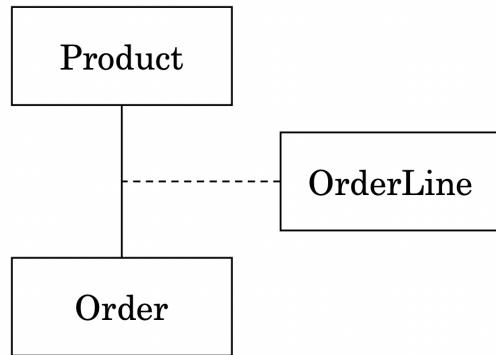
In questo esempio un oggetto *Order* è **composto** da una serie di *OrderLines*. Non può esserci un ordine senza nemmeno un istanza di *OrderLine*.

Classe associativa

Un altro importante costrutto è la classe associativa, quando si presenta una situazione del tipo



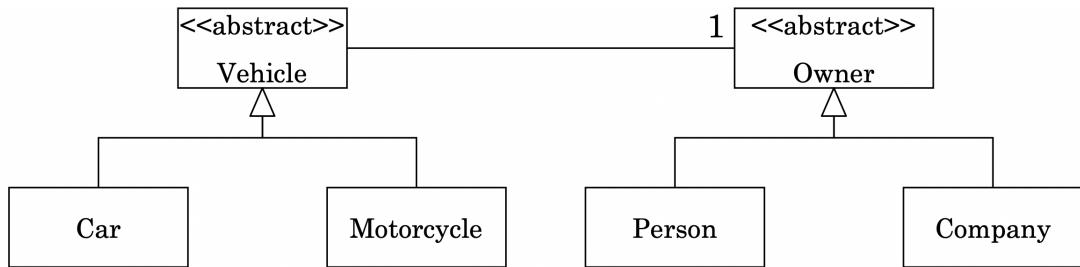
si può rappresentare in un modo differente



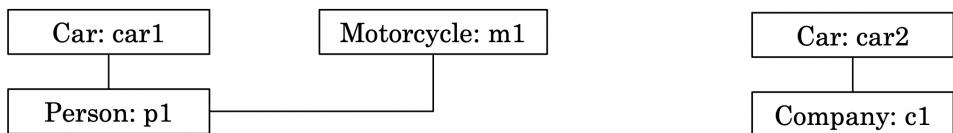
È importante notare che in questa situazione gli attributi di **OrderLine** dipendono dagli oggetti **Product** e **Order**. È questo che caratterizza una classe associativa. La si può vedere come l'entità che si inserisce per modellare una relazione molti a molti.

Ereditarietà

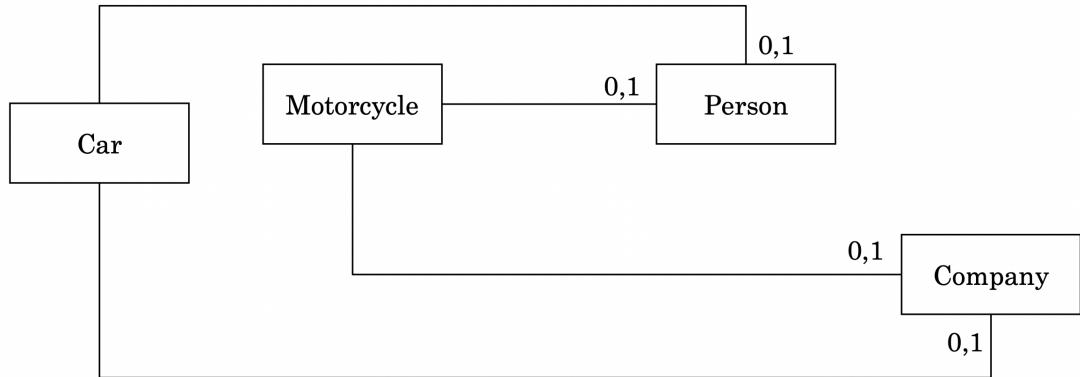
In UML esiste anche l'ereditarietà e per capirla prendiamo in esame un semplice esempio con veicoli e proprietari. Sfruttando l'ereditarietà si potrebbe fare un modello delle classi



e il corrispondente modello delle classi è

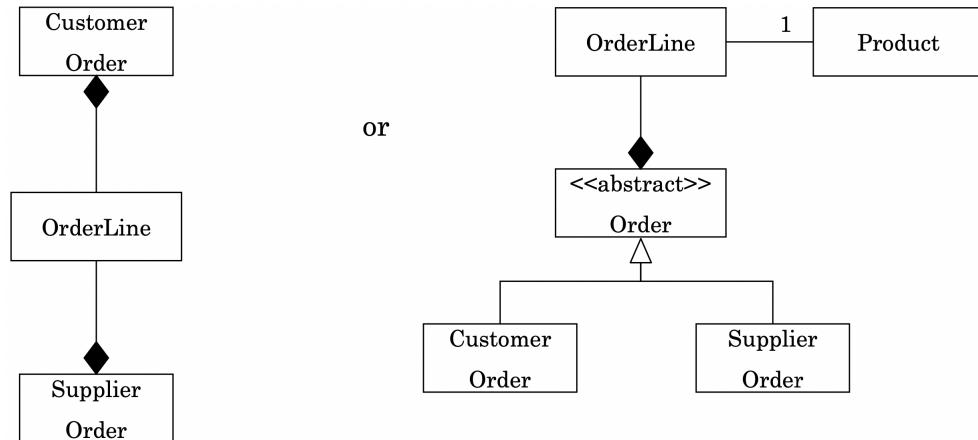


La stessa soluzione potrebbe essere ideata senza l'utilizzo dell'ereditarietà ma questo ci fa perdere di generalizzazione e rende molto più meccanica l'implementazione reale. Potrebbe anche comportare la duplicazione di codice e potrebbe far diventare più difficile la manutenzione del software.

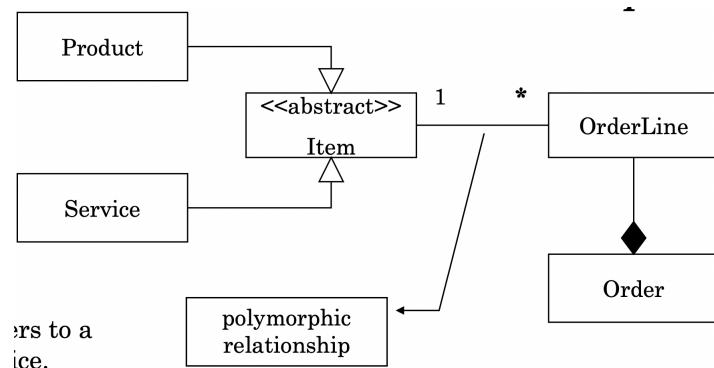


L'ereditarietà ci permette di modellare in maniera efficace diverse situazioni anche se in maniera più complicata essa risulta essere molto più efficace:

Contenitore Astratto

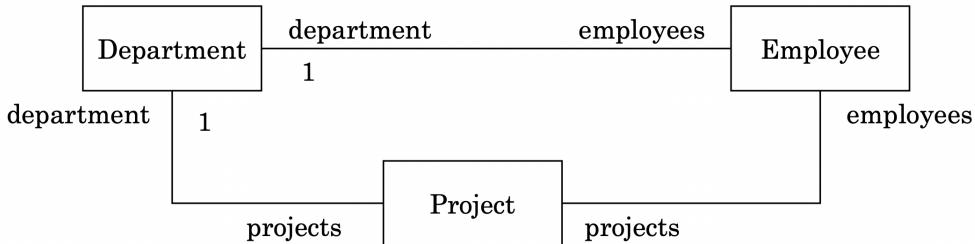


Pattern Abstract



Inoltre, per arricchire i modelli possiamo usare gli attributi associativi che sono utili per visualizzare la navigazione del modello. Per esempio il modello che segue presenta gli

attributi associativi vicino alle classi. In sostanza è utile indicare quale attributo della classe fa riferimento all'oggetto legato dall'associazione

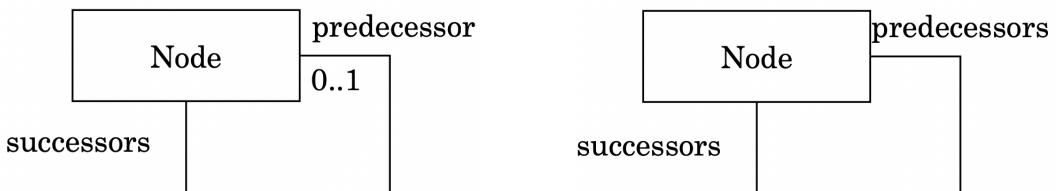


Nella figura salta subito all'occhio che *Department* ha un attributo che è una lista di oggetti *Employee* e questa lista si chiama *employees* mentre, allo stesso modo, *Employee* ha una lista di oggetti *Project* e questa lista si chiama *projects*. Chiaramente gli attributi associativi non vengono usati solo per rappresentare liste, per esempio *Project* ha un attributo di tipo *Department* e questo attributo si chiama *department*. La convenzione vuole che gli attributi singoli abbiano lo stesso nome della classe che ne rappresenta il tipo ma con la prima lettera minuscola mentre gli attributi che sono liste abbiano il nome della classe che ne rappresenta il tipo al plurale con la prima lettera minuscola.

Date le regole per ricavare gli attributi di associazione è possibile ometterli esattamente come è possibile omettere le molteplicità quando essa è *. Con queste regole è possibile ottenere uno schema semplificato con degli elementi sottintesi.

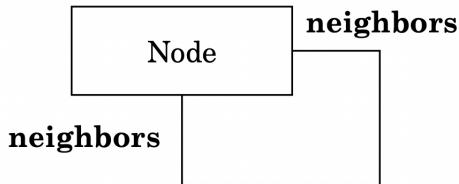
Relazione ricorsive

Le relazioni ricorsive sono delle relazioni che collegano una classe con se stessa. Solitamente questa relazione presenta degli attributi associativi che non rispettano le regole che abbiamo visto in precedenza per i nomi degli attributi associativi. Un esempio di relazione ricorsiva è quella che si può definire in un nodo di un albero o grafo.



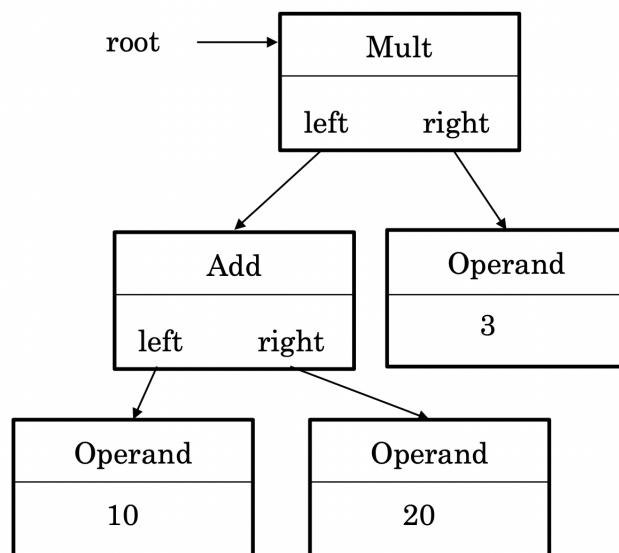
L'immagine a sinistra indica il nodo che potremmo usare per costruire un albero mentre quello a destra indica un nodo che potremmo usare per modellare un grafo. La differenza è nella cardinalità del predecessore. Nel caso di un albero ogni nodo può avere al più un predecessore(solo la radice non ha alcun predecessore) mentre nel caso di un grafo i predecessori possono essere tanti(stiamo parlando di un grafo orientato).

Vi è anche una particolare relazione ricorsiva che è quella **simmetrica** dove non vi è una gerarchia particolare tra i nodi come quella proposta in precedenza(gerarchia di precedenza). In questo caso vi è un singolo attributo associativo



Interfacce

Le interfacce vengono presentate per completezza ma non le useremo tantissimo. Comunque sono delle strutture tali per cui è possibile garantire che una classe che implementa una determinata interfaccia abbia dei comportamenti specificati dall'interfaccia. L'esempio proposto è un albero per effettuare delle operazioni aritmetiche nel giusto ordine.

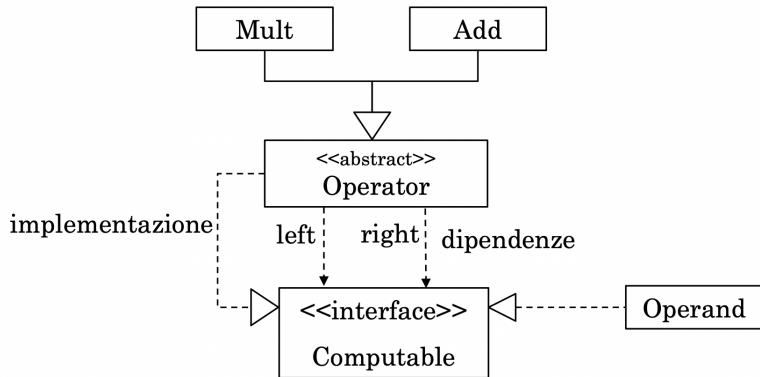


Per fare in modo che si esegua l'operazione corretta $(10 + 20) * 3$ bisogna riuscire ad accomunare le varie operazioni e i vari operandi per fare in modo che abbiano un comportamento unificato da poter utilizzare partendo dal nodo radice. Per fare questo si definisce un'interfaccia che in Java si potrebbe scrivere:

```

public interface Computable {
    int compute();
}
  
```

Facendo implementare questa interfaccia a operatori e operandi si potrebbe unificare il comportamento. Ovviamente gli operandi dovranno restituire il numero che li rappresenta mentre le operazioni dovranno effettuare l'operazione con gli operandi che hanno come attributi. Il modello delle classi completo è:



Un'implementazione completa della soluzione in java potrebbe essere la seguente:

```

public interface Computable {
    int compute();
}

public class Mult implements Computable {
    private int o1;
    private int o2;

    public Mult(Operand o1, Operand o2) {
        this.o1 = o1;
        this.o2 = o2;
    }

    @Override
    public int compute() {
        return this.o1 * this.o2;
    }
}

public class Operand implements Computable {
    private int number;

    public Operand(int number) {
        this.number = number;
    }

    @Override
    public int compute() {
        return this.number;
    }
}
  
```

È stato implementato solo l'operatore Mult perché tanto gli altri sono identici cambia solo il nome e l'operazione.

2.9 Interpretazione del modello delle classi

Vi sono due diverse interpretazioni del modello delle classi:

- Dal punto di vista *programming-oriented* il modello delle classi rappresenta un programma orientato agli oggetti. Le classi del modello vengono mappate nelle classi del linguaggio, gli attributi delle classi del modello vengono mappati negli attributi delle classi java mentre le relazioni di ereditarietà vengono mappate nell'estensione di classe
- Secondo l'approccio *object-relational* un modello delle classi indica delle informazioni persistenti salvate in un database relazionale. Secondo questo approccio due output possono essere ottenuti dal modello delle classi:
 - La definizione delle tabelle del modello relazionale
 - La collezione di classi java rappresentative dei record del database. Gli oggetti derivati dalle classi sono chiamati *entity* mentre la classe è chiamata *entity class*. In questa prospettiva il modello prende il nome di *entity model*. **Hibernate** è un framework per java che implementa il mapping tra oggetti java e record SQL

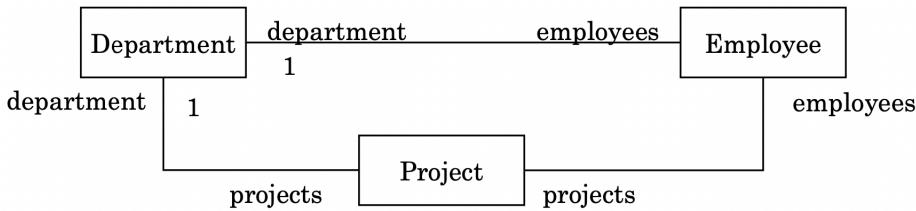
Sulle **entity** possono essere eseguite diverse operazioni tra cui le 4 operazioni CRUD e 2 operazioni aggiuntive che sono quelle riservate alla creazione e rottura dei collegamenti tra gli **entity**.

2.10 Espressioni navigazionali e condizioni

Le espressioni navigazionali e le condizioni vengono espresse in UML tramite un linguaggio chiamato OCL - Object Constraints Language, già citato in precedenza. Nel corso noi useremo una semplificazione di questo linguaggio che prende il nome di *SimpleOCL*. Le espressioni navigazionali ci permettono di ottenere collezioni di oggetti oppure oggetti singoli a partire da altri oggetti mentre le condizioni sono espressioni booleane che spesso includono espressioni navigazionali. Con le condizioni possiamo esprimere tante cose:

- Invarianti
- Regole di validazione
- pre-condizioni
- post-condizioni

Per comprendere le espressioni navigazionali diamo degli esempi:



Dato un impiegato e:

- possiamo ottenere i progetti in cui partecipa: **e.projects**
- e possiamo risalire a tutti i dipartimenti in cui lavora: **e.projects.department**
- sapere il numero di progetti in cui lavora: **[e.projects]**
- sapere il nome del suo dipartimento: **e.department.name**

Per esprimere dei filtri a delle collezioni di oggetti potremmo fare:

Vogliamo solo i progetto con budget > 1000: **e.projects (p, p.budget > 1000)**. Dentro le parentesi tonde il primo p è un cursore, indica il generico progetto corrente. Possiamo anche utilizzare una sintassi standard per prendere il Max, Min di una collezione oppure fare Sum o Avg. Le sintassi possibili sono due e sono equivalenti:
Se volessimo il budget massimo tra tutti i progetti in cui lavora l'impiegato:

- **e.projects.max(budget).budget**; Il **.budget** finale è obbligatorio perché l'espressione max ci restituisce l'oggetto che contiene il max budget e non direttamente il budget.
- **e.projects.budget.max**;

Per poter utilizzare la navigabilità e le condizioni bisogna avere un'entità contestualizzata su cui poter applicarle. Per poter esprimere l'entità del contesto usiamo la parola chiave **with**. Per esempio ipotizziamo di voler avere tutti i lavoratori di un dipartimento che lavorano ad un certo progetto:

with Department d, Project p: d.employees (e, e.projects include p)

2.11 Invarianti

Gli invarianti sono molto importante, inoltre si trovano sempre nel primo esercizio dell'esame. L'invariante altro non è che una condizione che vale sempre, ad esempio:

Ogni dipendente può avere al più 3 progetti

$$[\text{employee.projects}] \leq 3$$

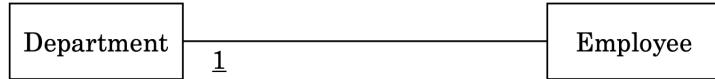
Ogni progetto di un dipendente è gestito dal dipartimento del dipendente

$$[\text{employee.projects} \text{ in } \text{employee.department.projects}] == 0$$

le parentesi quadre indicano quello che viene chiamato *quantificatore universale*. Si possono anche definire invarianti sugli attributi associativi, per esempio per verificare se sono definiti o meno, attraverso la parola chiave **def**

```
employee.department def
```

In alternativa si può definire il la relazione come necessaria per evitare di dover inserire il vincolo:



2.12 Regole di validazione

Con lo stesso meccanismo possono essere definite delle regole dette regole di validazione. Queste regole vengono scritte con la stessa sintassi e stanno ad indicare una condizione che si deve verificare sotto determinate condizioni. Ad esempio ipotizziamo che il direttore di un dipartimento debba essere necessariamente tra gli impiegati. Questa condizione, esprimibile con la regole **department.head in department.employees** si deve verificare se il dipartimento ha un direttore. Potrebbe accadere in qualche momento che il direttore non sia presente, magari in un momento di cambio del direttore.

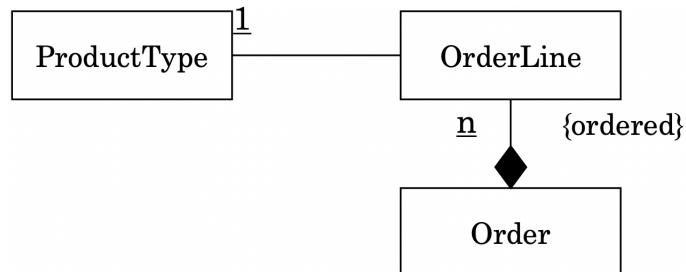
2.13 Operatore distinct

L'operatore **distinct**, come dice il nome stesso, mette un vincolo sugli elementi di una collezione imponendo che uno degli attributi dell'oggetto che compone la collezione siano tutti distinti. Per esempio in uno scontrino non è ammesso che ci siano due righe con lo stesso prodotto, ipotizzando che i prodotti multipli vengano aggregati in una sola riga.

order.orderLines.ProductType distinct

2.14 Associazioni ordinate

Se si volesse impostare l'ordinamento di un'associazione si potrebbe utilizzare il qualificatore **ordered** come nell'immagine che segue:



Per prendere un elemento specifico da una collezione possiamo usare la notazione `order.orderLines(0)`

2.15 Operazioni

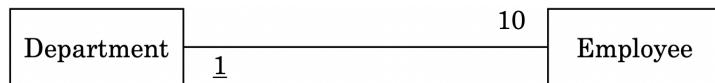
Le operazioni vengono espresse in modo dichiarativo. Per esempio se si volesse aggiungere un nuovo impiegato al dipartimento ma si vuole pure che il dipartimento non superi mai 10 impiegati bisogna solo esprimere il bisogno di questa condizione e non si deve invece implementare la logica che ne gestisce la casistica:

with Department department

pre: [department.employees] < 10

post: new Employee e, e.department == department

Si noti che lo stesso risultato si può ottenere semplicemente assegnando il valore 10 alla cardinalità dell'associazione tra *Department* ed *Employee*:



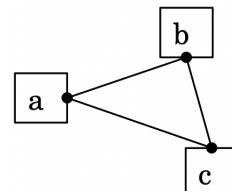
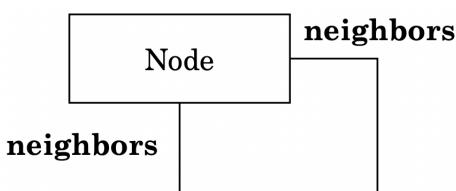
In questo modo ci è possibile eliminare le **pre**:

with Department department
post: new Employee e, e.department == department

2.16 Creare un object model

Anche per specificare un object model vi è la stessa sintassi vista in precedenza per le operazioni e si agisce anche in questo caso in maniera dichiarativa:

Dal modello delle classi a sinistra si vuole costruire il modello degli oggetti a destra:



new Node a; new Node b; new Node c;

b in a.neighbors; c in a.neighbors; c in b.neighbors

In questo esempio inserire **b** nei vicini di **a** implica che **a** sia nei vicini di **b** e questo vale anche per gli altri casi.

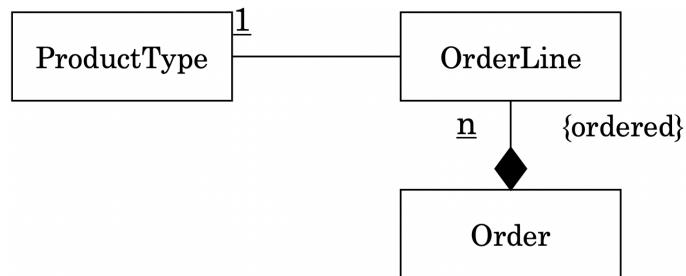
2.17 Attributi obbligatori e derivati

Gli attributi obbligatori (required attributes) devono necessariamente essere inizializzati quando si istanzia un oggetto. Gli attributi obbligatori possono essere associativi oppure ordinari. Gli attributi obbligatori sono sottolineati.

Per quanto riguarda gli attributi derivati invece sono attributi che vengono calcolati o ricavati da altri attributi. Essi sono riconoscibili dalla loro definizione che solitamente include il segno di $=$. Per esempio $\text{float } amount = n * \text{productType.price}$

2.18 Relazioni derivate

Alcune associazioni, come abbiamo anche visto in precedenza sono derivate. Certe volte è utile definire queste relazioni per evitare di dover effettuare navigazioni complicate. Prendiamo ad esempio il seguente modello delle classi:



La relazione tra *Order* e *ProductType* si può derivare passando da *OrderLine*. Tuttavia se gli accessi ai **ProductType** da **Order** sono frequenti si potrebbe pensare di utilizzare una relazione derivata per semplificare e ottimizzare l'operazione.

Order: Set<ProductType> **productTypes** = orderLines.productType;
 ProductType: Set<Order> **orders** = orderLines.order;

Chapter 3

Analisi dei requisiti

3.1 Requisiti del software

Un requisito è una richiesta da rispettare durante lo sviluppo del software. Ne abbiamo di due tipi:

- **Requisiti funzionali:** Sono le funzionalità che soddisfano una richiesta del committente. Possono essere sotto forma di attività o processo. Nella fase di definizione lo scopo non è sapere come realizzare l'implementazione di un requisito funzionale ma solo descrivere che cosa ci si aspetta da quel requisito. Spesso il *customer* e gli *end-user* sono inclusi nella fase di definizione in modo da modellare in modo completo il requisito
- **Requisiti non funzionali:** Sono requisiti che interessano il sistema come ad esempio vincoli di tempo o utilizzo. Ad esempio il tempo di computazione di una determinata operazione. Spesso sono vincoli.

I requisiti devono rispettare 4 proprietà che però non sono sempre semplici da soddisfare. Tante volte ci si limita a soddisfare solo un sottinsieme di questi:

- **Non ambigui:** Ciascun requisito deve avere una singola precisa interpretazione
- **Consistenti:** Non deve avere conflitti con gli altri requisiti
- **Verificabili:** In modo semplice e senza spendere troppe risorse si dovrebbe poter verificare il soddisfacimento di ogni requisito
- **Completi:** La definizione del requisito deve coprire tutti gli aspetti che lo interessano

Ovviamente di tutti i requisiti che sono stati definiti vogliamo che siano tracciabili e lo possiamo fare in due differenti modalità:

- **Forward:** Dai requisiti al subsystem

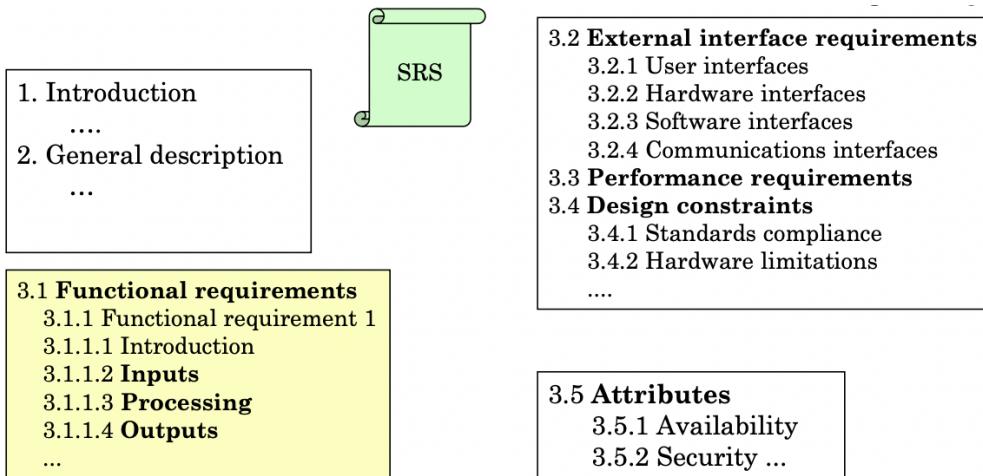
- **Backward:** Dal subsystem ai requisiti

L'analisi dei requisiti è divisa in diverse fasi:

- **Elicitation:** Raccolta dei bisogni da coloro che vengono chiamati *stakeholders* cioè tutti quelli che sono interessati al sistema
- **Analysis:** Assicurarsi che i requisiti rispettino dei criteri di qualità
- **Documentation:** Viene prodotta della documentazione e dei diagrammi basati sugli standard in vigore

3.2 IEEE STD-830

È uno standard per la scrittura dei documenti, uno dei primi. Suddivide la documentazione in parti:



3.3 Model-Driven Development - con studio di caso

Nel MDD quello che viene fatto è ricavare dei modelli dai requisiti funzionali. Tutta la fase di acquisizione dei requisiti è quindi obbligatoria per poter poi ricavare dei modelli che possano modellare il sistema. Vi sono 2 punti fondamentali che devono essere fatti in seguito all'acquisizione dei requisiti funzionali:

- Si ricava il modello delle classi di dominio con relazioni e attributi
- Si modellano le attività con i casi d'uso UML. I casi d'uso rappresentano dei *task* con pre-condizioni per i vincoli e post-condizioni per gli effetti.

Di seguito c'è il testo di un esercizio che è stato svolto in aula:

The library contains *books*: book attributes include id, title, authors, publisher. Books are added by librarians (staff).

Members are the persons entitled to get books on *loan*. Members are enrolled by librarians: the enrollment date and the librarian who made the enrollment must be recorded. Member attributes include name, address, email. A personal id is provided by the enrollment procedure.

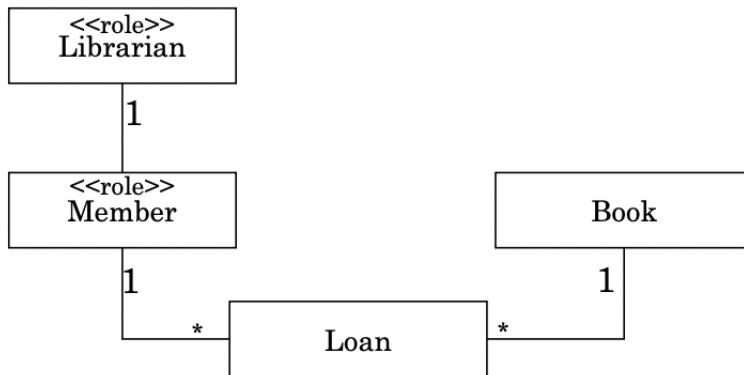
A librarian has a name and an id.

To borrow a book, members must open a loan. First, they get a list of books by specifying subject and authors, then they choose an item from the list. The loan is opened if the book is available and the number of open loans of the member does not exceed the limit (6). The duration of the loan is 14 days.

Members cannot borrow additional books if they have some loans overdue.

When a member returns a book, the system closes the loan associated with the book id.

La prima cosa che si dovrebbe fare è cercare di individuare le classi, le relazioni e gli attributi. In questo studio di caso abbiamo che le classi sono 4 di cui 2 ruoli. Di seguito il modello delle classi di dominio:



Nella classe Librarian e Member vediamo che c'è «role» che è un oggetto chiamato stereotipo. Gli stereotipi stanno ad indicare delle proprietà della classe e di tutti gli stereotipi che usiamo in un modello delle classi di dominio possiamo creare dei profili UML in cui inserirli. Gli attributi sono:

Book: String id, String title, String authors, String publisher

Member: String id, String name, String address, String email, Date enrolmentDate

Loan: Date dueDate, Date endDate

Librarian: String id, String name

Si deve fare particolare attenzione alla classe **Loan** che presenta due attributi che hanno uno scopo particolare perché vengono usati per ricavare un attributo derivato, lo stato del prestito che può assumere 3 valori:

- **Ongoing:** endDate == null
- **Past:** endDate != null

- **Overdue:** dueDate < today and endDate == null

Questo attributo sarà poi utile per effettuare delle operazioni nei casi d'uso.

3.3.1 Invariante

Si definisce il seguente invariante: $[m.loans (ongoing)] \leq 6$

3.4 Casi D'uso

Adesso, sempre facendo riferimento all'esercizio vediamo come strutturare i casi d'uso. Per prima cosa come abbiamo già detto i casi d'uso sono una descrizione con pre e post condizioni e descrivono delle interazioni tra l'utente e il server. I casi d'uso hanno delle relazioni di estensione e inclusione:

- **Extends:** Il comportamento di un caso d'uso può essere esteso da un altro
- **Includes:** Se y include x allora y è incompleto senza x

Prima di passare ai casi d'uso dell'esercizio è importante porre l'attenzione ad uno scenario, quello che descrive la presa in prestito del libro:

1. Lo user inserisce i filtri di ricerca per il libro
2. Il sistema mostra i libri che corrispondono ai filtri
3. Lo user sceglie il libro dalla lista
4. Il sistema conferma la possibilità dello user di poter prendere il libro
5. Il sistema apre un prestito

Quello che si vuole mettere in evidenza è che in realtà gli ultimi due punti non necessariamente devono essere separati ma possono scrivere anche un unico punto:

Se i vincoli sono rispettati il sistema apre un nuovo prestito altrimenti l'utente viene riportato al punto 1

La cosa che potrebbe far fallire i requisiti, la causa principale si intende, è un problema relazionata alla concorrenza. È possibile che due utenti stiano facendo la stessa procedura contemporaneamente e il secondo a terminare la stessa operazione riceverà un errore dal sistema perché non è possibile prendere il libro in prestito in due. Nella spiegazione degli scenari si prende in considerazione sempre l'happy path, cioè la situazione in cui tutto va come dovrebbe, senza errori ed è per questo che la prima soluzione è quella accettata mentre la seconda, che mostra una condizione di fallimento, non viene presa in considerazione.

enrollMember**post:** new Member

N.B.: Siccome lavoriamo in modo dichiarativo, quando scriviamo *new Member* stiamo dando per scontato che gli attributi vengano popolati. La versione estesa, troppo prolissa, sarebbe: new Member m, m.name def, m.email def, ..., m.librarian == librarian

borrowBook

L'invariante ci garantisce che l'utente non abbia più di 6 prestiti attivi

with book

pre: [member.loans(overdue)] == 0 and bool.available

post: new Loan l, l.dueDate == today + 14 giorni

Si ipotizza che le operazioni pre e post vengano fatte in una transizione. Se non si avesse l'invariante si dovrebbe specificare un'altra condizione: [member.loans(ongoing)] < 6

returnBook

with book

pre: book in member.loans(ongoing).book

post: book.currentLoan.endDate == today

Casi d'uso rimanenti

updatePersonalInformation: address modified and/or email modified

addBook: new Book

sendReminder: Da fare giornalmente. **post(informale):** un reminder deve essere mandato a loans(l, l.overdue and oggi == l.dueDate + 1 settimana).member

I casi d'uso rimanenti sono stati scritti in maniera molto informale ma è una cosa accettabile quando i requisiti sono semplici.

3.5 Alcune regole e consigli

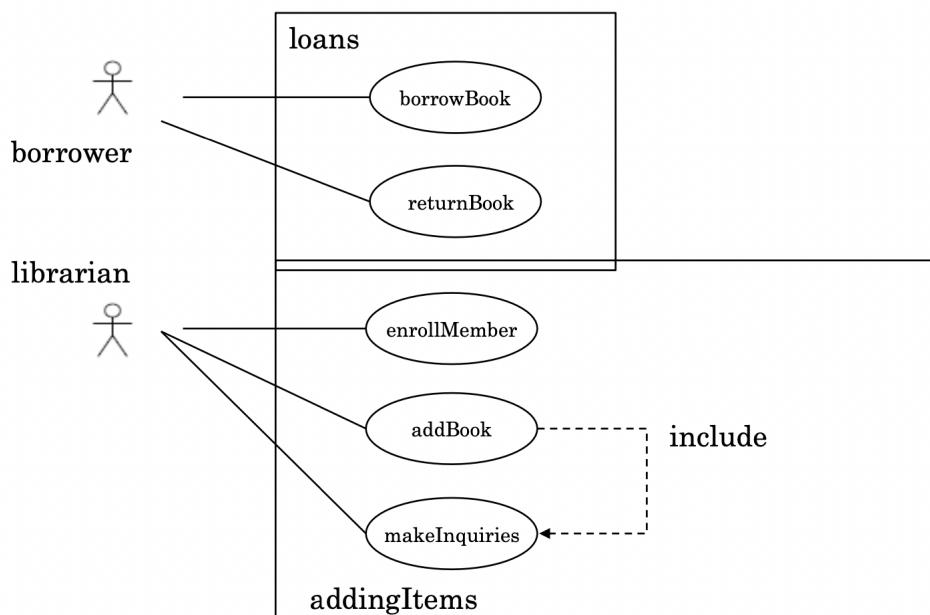
- Quando si svolgono gli esercizi c'è una convenzione per gli attributi che rappresentano uno stato ma che non è derivato. Non si esplicita il tipo dell'attributo ma si elencano tutti i possibili stati tra parentesi e si sottolinea lo stato iniziale, nel caso in cui ci fosse. Si tenga presente che se non è presente uno stato iniziale lo stato è semplicemente non definito, null.
- Quando si rappresenta un invariante, per evitare di complicare la navigazione, al 99.9% dei casi è meglio far partire la navigazione dalla classe a cui si applica l'invariante. Applicando questa regola si semplifica la navigazione e solitamente si rappresenta meglio il vincolo

Chapter 4

Meta-modelli

I meta-modelli sono dei modelli che stanno ad un livello superiore, servono per rappresentare a loro volta dei modelli. Questi modelli sono molto utili nel caso in cui bisogna strutturare un software che ha come scopo la creazione di modelli. Per esempio un software che permette di fare diagrammi delle classi, diagrammi E-R, ecc.

Per definire un meta-modello è necessario osservare il modello di cui si vuole creare il meta-modello e classificare le diverse forme che vediamo. Per esempio dato il seguente modello

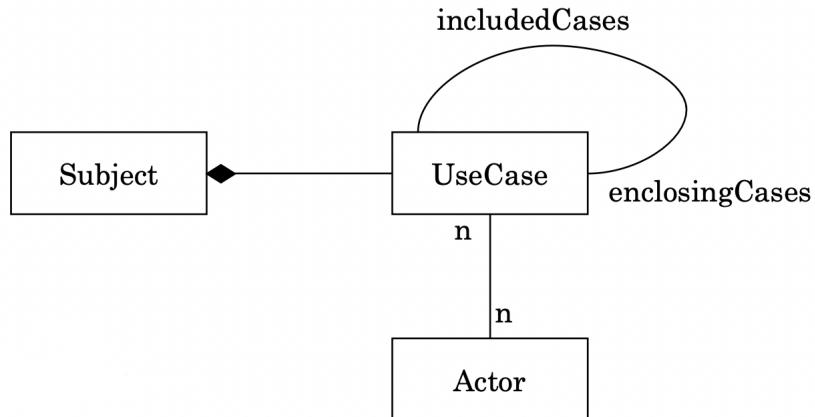


si possono individuare alcuni elementi che possiamo classificare

- Dentro gli ovali abbiamo gli **Use Case**
- Dentro i rettangoli abbiamo i **Subject**

- Ci sono gli omini che sono gli **Actor**
- Vi è una relazione di inclusione(e in realtà anche di estensione) tra i casi d'uso

Tra queste classi che abbiamo individuato vi sono delle associazioni, ad esempio vediamo che un Actor può avere a che fare con più Use Case. Con le osservazioni fatte possiamo disegnare il meta-modello:



Gli attributi per brevità non sono stati inclusi ma sono esattamente quelli che ci si aspetta guardando il modello di partenza. Invece di seguito ci sono degli invarianti e dei vincoli che devono essere definiti:

addBook in **makeInquiries.enclosingCases**

makeInquiries in **addBook.includeCases**

if([useCase.includedCases] > 0) then useCase.subject == useCase.includeCases.subject;

Questa ultima condizione, che esprime un vincolo, può essere scritta in una seconda maniera sfruttando l'implicazione logica: $[useCase.includedCases] > 0 \rightarrow useCase.subject == useCase.includeCases.subject$. L'implicazione logica ha una sua tabella della verità che è la seguente

A B Risultato

| | | |
|---|---|---|
| t | t | t |
| t | f | f |
| f | t | t |
| f | f | t |

Da quello che abbiamo potuto vedere, anche se abbiamo visto un solo meta-modello, in realtà il modo di rappresentare un meta-modello è esattamente il modo che usiamo per rappresentare un modello delle classi infatti un meta-modello è esattamente un modello delle classi dove ogni classe rappresenta un elemento grafico e ha dei legami con gli altri elementi grafici, a loro volta classi nel meta-modello.

4.1 Business Process

Vi sono due differenti definizioni di business process:

- Una serie di azioni o task fatti per ottenere qualche risultato
- Un processo di business è il coordinamento di step o task attraverso il tempo e lo spazio. Con tempo si intende quello necessario per completare gli step o task mentre con spazio si intende il posto in cui si svolgono.

Un processo Business to Business è un processo dove due aziende o grandi organizzazioni si parlano per portare a termine un obiettivo comune

4.1.1 Diagrammi di attività

Sono dei diagrammi che descrivono dei processi, anche se vengono chiamati diagrammi di attività. Un processo potrebbe essere inteso come attività ma in realtà con attività si intende un task singolo, allora per essere in linea con quanto diremo più avanti noi con "attività" intenderemo un macro task, composto da più task piccoli

Nella notazione standard non viene trattata l'assegnazione di un task mentre noi, per completezza, metteremo il ruolo che può eseguire un determinato task umano.

- L'ordine dei task è gestito dal control flow, rappresentato da rombi e si divide in più categorie: **Scelta, confluenza, fork, join**
- Il data flow è subordinato al control flow e i dati sono rappresentati da object nodes

Tutti i processi sono gestiti da un *process engine* che ha il compito di istanziare ed eseguire i processi. Con i termini *Control flow* e **Data flow** intendiamo:

- **Control flow:** Gestisce l'ordine dei task in termini di precedenza
- **Data flow:** Gli input di un task sono gli output di uno o più task precedenti. Il data flow mostra le dipendenze di dati.

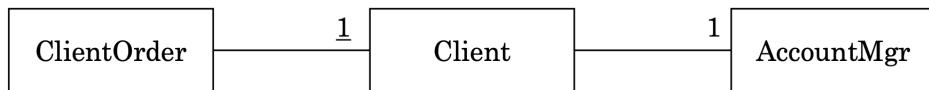
Esempio

Il processo HandleClientOrder permette ad un fornitore di trattare gli ordini provenienti dai clienti. Quando il fornitore riceve un ordine da un cliente, il process engine genera un'istanza del processo e le passa come parametro di input l'ordine del cliente. Un

ordine ha una descrizione e un importo. Se l'importo è ≤ 1000 , l'ordine è accettato automaticamente altrimenti è valutato dall'account manager associato al cliente. L'ordine ha uno stato che può essere accepted o rejected. Il processo informa il cliente dello stato dell'ordine. Nel sistema informativo del processo sono registrati i clienti e gli account manager (staff). Le relazioni con un cliente sono gestite da un accountMgr che può trattare vari clienti.

N.B.: Alcuni aspetti nel testo sono specificati perché indicano che dei dati sono già presenti nel sistema, magari acquisiti da altri processi precedenti, come ad esempio la frase "Nel sistema informativo del processo sono registrati i clienti e gli account manager (staff)". Questo dovrebbe sottolineare il fatto che il processo prende dei dati in input che sono risultati di altri processi precedenti.

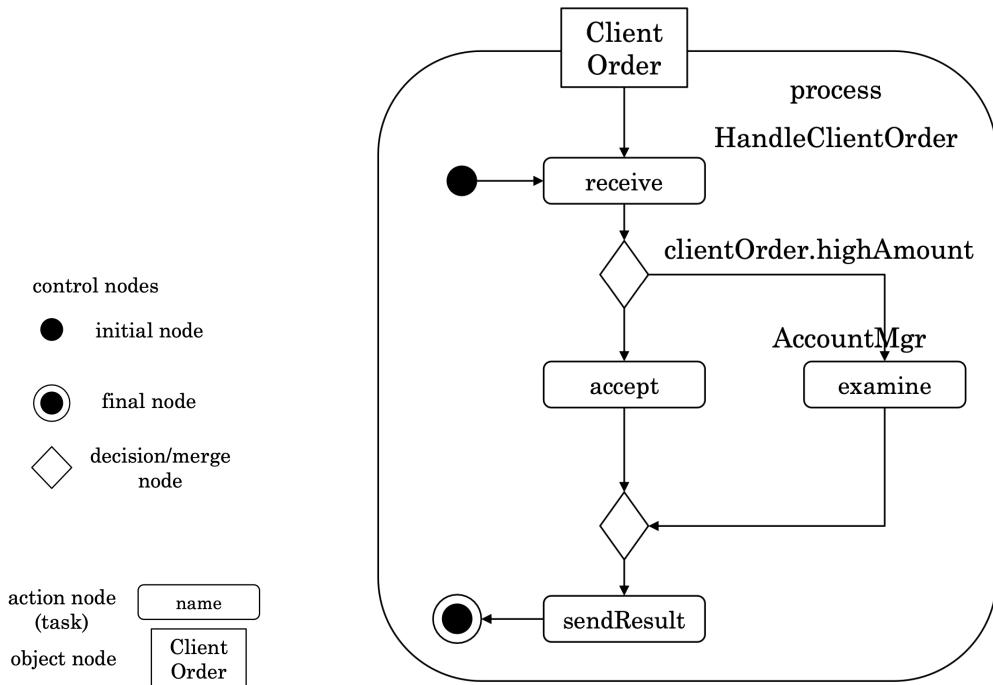
Si costruisce un piccolo diagramma delle classi che renda chiara la situazione che chiamiamo information model



Con gli attributi:

ClientOrder: String description, double amount, state (accepted, rejected), boolean highAmount = amount > 1000.

E dopo aver fatto l'information model passiamo a fare l'activity diagram:



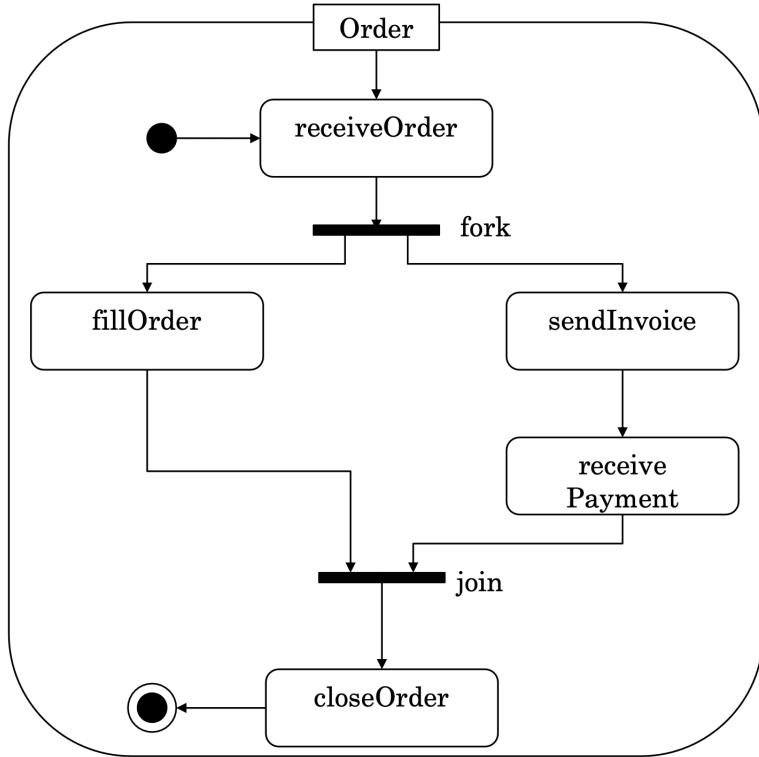
A cui dobbiamo abbinare anche delle post condizioni per accertarci che il processo sia stato eseguito come si deve:

```

receive: this.clientOrder == clientOrder;
accept: clientOrder.state == accepted; examine: clientOrder.state def (performer ==
clientOrder.client.accountMgr)
  
```

Nelle post condizioni di examine si vede che c'è una sorta di vincolo sul performer. Quel vincolo sta ad indicare che che fa il task non è un Account Manager qualunque ma bensì è colui che è stato assegnato al cliente che ha effettuato la richiesta, è uno specifico.

A scopo dimostrativo di seguito c'è un Activity Diagram che fa uso delle fork e delle join:



Si noti che se una fork prende più input e ha più output essa viene interpretata come **Join Fork** cioè un elemento che prima fa il join degli input e successivamente il fork del risultato del join.

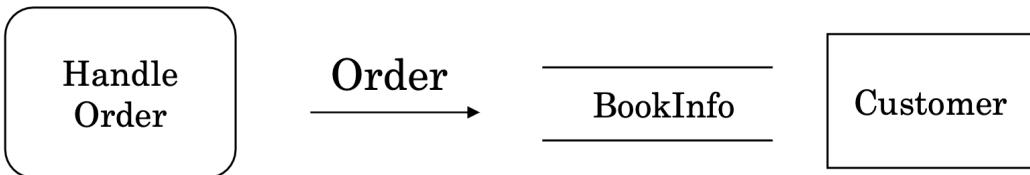
Nelle slide sono disponibili diversi esempi con soluzioni dei modelli di attività.

4.2 Modelli Dataflow

I modelli *Dataflow* sono tali per cui l'attivazione dei task successivi è subordinata all'output dei task precedenti ed è per questo che è un flusso basato sui dati. Un determinato Task riceve degli input e produce degli output, che hanno dei nomi, che serviranno ai task successivi per poter attivarsi. È un flusso prettamente top down in cui il primo livello rappresenta gli input dall'esterno.

Vi è un data dictionary che sostanzialmente è un dizionario dei termini dove viene spiegato che cosa rappresentano i dati.

Per comporre un *Dataflow* abbiamo a disposizione dei simboli, principalmente 4, di cui una rappresentazione di seguito:



I dettagli di ogni simbolo sono (in ordine da sinistra verso destra):

1. **Attività:** Unito elemento attivo del diagramma, attivo vuol dire che è capace di consumare dati.
2. **Dataflow:** Indica la direzione del dato. Da dove parte la freccia è stato prodotto il dato e dove arriva la freccia è dove verrà consumato
3. **Datastore:** Immagazzina dei dati che possono essere consultati in ordine diverso da quello di produzione. Per esempio una serie di numero possono essere prodotto in ordine da 1 a 10 ma mettendo un Datastore tra due Attività si da la possibilità all'attività che consuma i dati di poterli leggere in qualsiasi ordine
4. **Attore esterno**

Come abbiamo detto l'attività è l'unico elemento del diagramma che è in grado di consumare dati, questo vuol dire che le seguenti casistiche non sono ammesse:

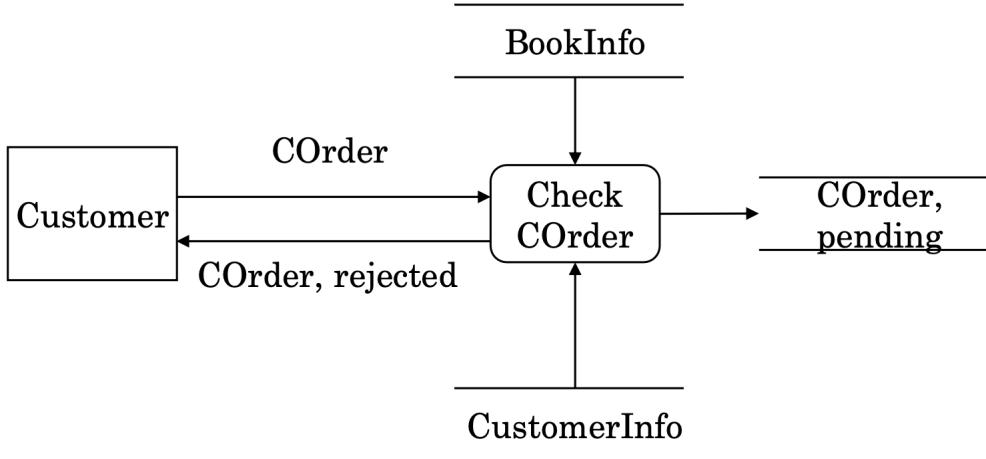
- Dataflow tra due Datastore
- Dataflow tra due attori

Un modello *Dataflow* quindi sta a rappresentare come i dati si spostano tra le varie attività e come facciamo per gli altri diagrammi lo ricaviamo da un testo dove sono descritti i requisiti. Di seguito un esempio di requisiti dove viene modellato un sistema di ordini di libri. L'esempio è molto vecchio, del 1979, per cui gli ordini venivano fatti via posta (non elettronica).

Requisito 1

Incoming orders are checked against a list of available books and against a customer file to see if the customer is in good credit standing (TBD, to be defined). Valid orders are put in a container of pending orders. If the checks fail, the order is rejected. External actors: customers and publishers. External data flows: customer orders, rejection notifications. Activities: check customer order (resulting in valid order or rejection notification). Remark: customer in good credit standing is a condition to be defined. Exceptions are not considered in this preliminary analysis.

Per i requisiti di cui sopra si può ricavare il modello Dataflow che segue



Si noti che alcuni elementi hanno due nomi come ad esempio la freccia da *Check COrder* e *Customer* oppure il datastore *COrder, pending*. Il secondo nome da una spiegazione di massa dello stato dei dati quando si trovano a passare in quegli elementi. Questi nomi devono poi avere un ulteriore approfondimento in linguaggio naturale per approfondirne il significato.

Un'attività, che come sappiamo è l'unico elemento attivo del diagramma, può inserire nuovi dati in un datastore oppure modificarne uno già esistente e allo stesso modo, in lettura, è possibile che il dato venga prelevato senza rimuoverlo oppure prelevarlo rimuovendolo dal datastore.

Che cosa è un Ordine?

Quello che bisogna fare è dare una descrizione da inserire nel data dictionary:

Un Ordine è formato da 3 parti

- Header: Ha un order-date, [orderNum]
- customerDetail: nomeOrganizzazione, indirizzo
- booksDetails: {bookDetail}
- bookDetail: {authorName}, title, publisherName, nOfCopies

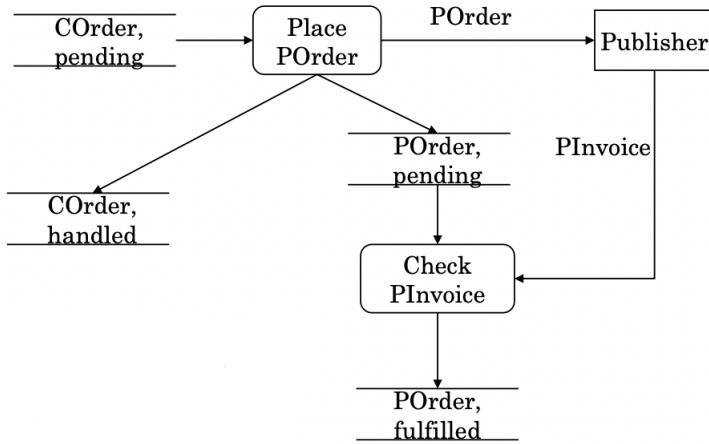
Della descrizione precedente le parentesi quadre stanno ad indicare che l'attributo è opzionale mentre le parentesi graffe indicano che vi è una ripetizione di una determinata informazione.

Requisito 2

Valid customer orders are put in a store of pending orders until a batch of orders can be assembled into a bulk order for a publisher. Publishers send an invoice with each

shipment, detailing its contents; it must be compared with the corresponding publisher order.

Con questi requisiti si vuole modellare la generazione dell'ordine e di emissione delle ricevute. Partendo dai COrder, pending il sistema si occupa di effettuare un ordine, chiamato POrder, verso il fornitore di libri ed effettua delle operazioni per la generazione e il controllo della fattura.



Si nota che da Place POrder escono 3 rami di cui i due più a destra sono **paralleli** mentre il terzo, quello che va verso il datastore COrder, handled potrebbe non essere parallelo agli altri perché è possibile che di un ordine non tutti i libri vengono gestiti allo stesso momento. Per poter distinguere i libri ordinati di un determinato ordine si può pensare semplicemente di utilizzare un flag di stato.

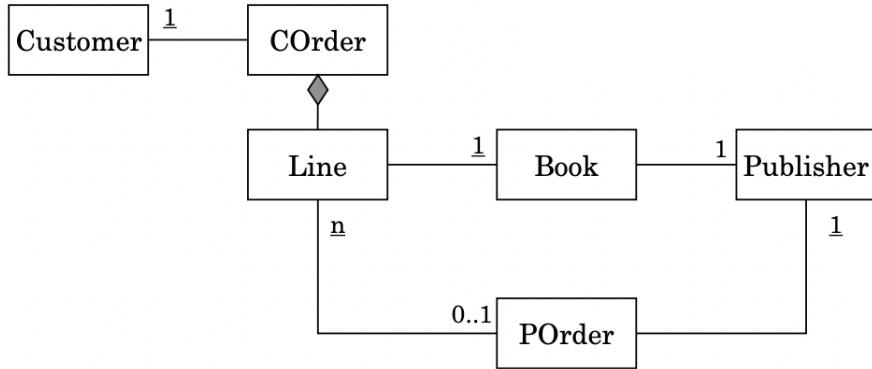
Un COrder deve specificare i libri che il customer vuole ricevere e quante copie di ogni libro. Un esempio di COrder:

```

cOrder 100 from customer 1 includes
  2 copies of book 1
  1 copy of book 2.
cOrder 101 from customer 2 includes
  3 copies of book 1
  1 copy of book 3.
  
```

Facendo riferimento al flag di stato che avevamo citato in precedenza, nel momento in cui si elabora un determinato ordine si mette un flag ad esempio in corrispondenza di *e copies of book 1* per indicare che sono stati ordinati mentre si lasciano senza spunta le altre righe in modo da capire che devono ancora essere ordinati. Un COrder viene considerato completato quando tutte le righe hanno una spunta. Quando parliamo di spunte stiamo parlando di qualcosa di scritto, si ricorda che questo esempio è del 1979 per cui queste operazioni venivano fatte su carta. La spunta viene applicata durante il piazzamento di un POrder.

Di questo esempio, anche se è molto vecchio, potremmo generare il modello informativo per poterlo rappresentare in maniera moderna. La prima cosa che facciamo è fare il modello delle classi che rappresenta la realtà descritta:



Nel modello delle classi viene esplicitata la composizione tra COrder e Line perché in COrder è effettivamente composto da una serie di linee che hanno a loro volta il numero di copie, è analogo ad uno scontrino. La relazione tra Line e POrder è opzionale perché gli ordini potrebbero essere respinti per cui nessuna linea di ordine viene associata a un POrder se il COrder viene respinto. Si può definire un invarianto:

$$\text{pOrder.publisher} = \text{pOrder.lines.book.publisher}$$

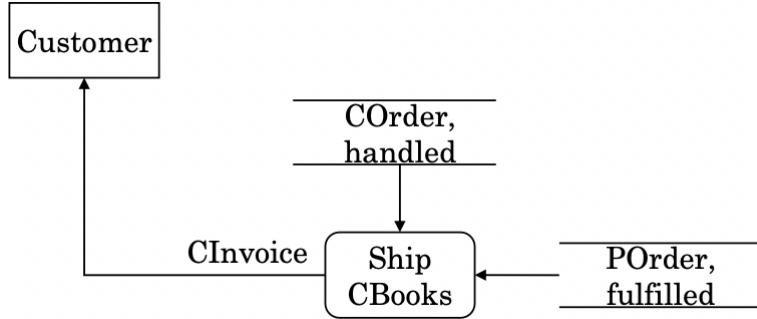
Questo ci garantisce che il publisher richiesto nell'ordine sia lo stesso dei libri che sono stati richiesti. Si potrebbe introdurre la relazione derivata tra COrder e POrder, molti a molti:

- COrder: Set<POrder> pOrders = lines.pOrder
- POrder: Set<COrder> cOrders = lines.cOrder

Requisito 3

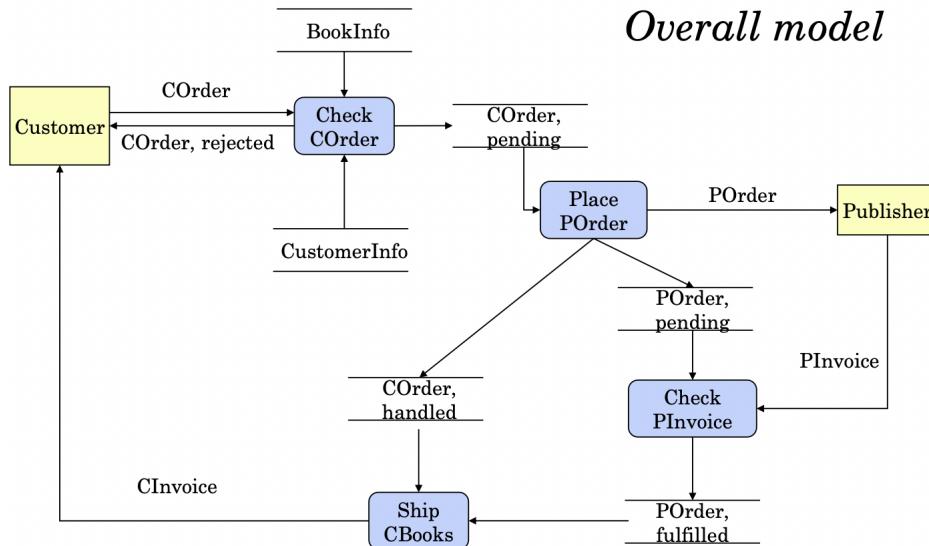
When all the books included in a customer order have been received, they are sent to the customer along with an invoice.

Questo requisito copre l'ultima parte del processo, quella di spedizione dei libri al cliente che li ha ordinati.



Come si evince da quanto detto in precedenza nel datastore COrder Handled si trovano solo gli ordini per cui sono stati ordinati, e in questo stadio anche ricevuti dall'editore, tutti i libri. Quindi per ogni dato all'interno del datastore viene effettuata la spedizione e per ogni ordine per cui si effettua la spedizione si modifica lo stato dell'ordine in *fulfilled* che rappresenta il completamento.

Mettendo insieme tutti e 3 i requisiti è possibile creare il modello generale del processo comprensivo di tutti i passaggi:



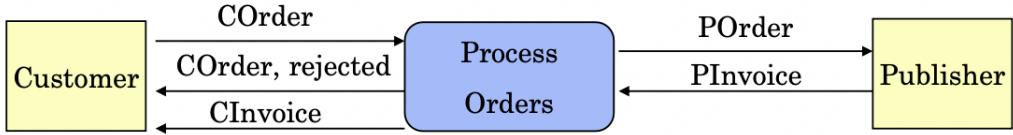
Il processo illustrato è un processo che viene chiamato ***processo singleton*** cioè che ha una singola istanza. In questo caso è necessario che il processo sia unico perché i libri che possono comporre un POrder possono essere accumulati da diversi ordini per cui è necessario che sia un singolo elemento ad effettuare questa operazione. Se ci fossero più processi istanziati non sarebbe possibile aggregare i libri dai diversi COrder prima di effettuare un POrder cumulativo o comunque sarebbe possibile ma non in maniera semplice ed efficiente.

Il processo che abbiamo definito comprende comunque un numero discreto di classi per cui inizia ad essere molto denso e difficile da rappresentare per cui è stata introdotta

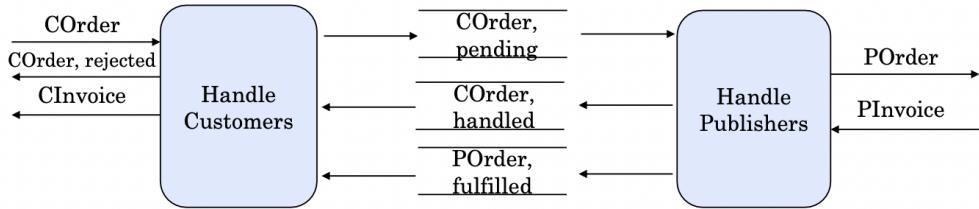
una metodologia di rappresentazione dei processi chiamata *Top-down decomposition*.

4.2.1 Top-Down Decomposition

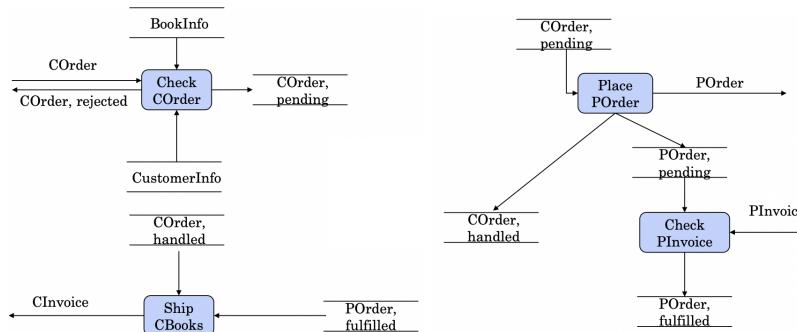
In sostanza quello che viene fatto è creare diversi modelli a livelli di astrazioni differenti partendo da quello con il livello di astrazione più alto e poi specificare gli elementi diminuendo di astrazione man mano:



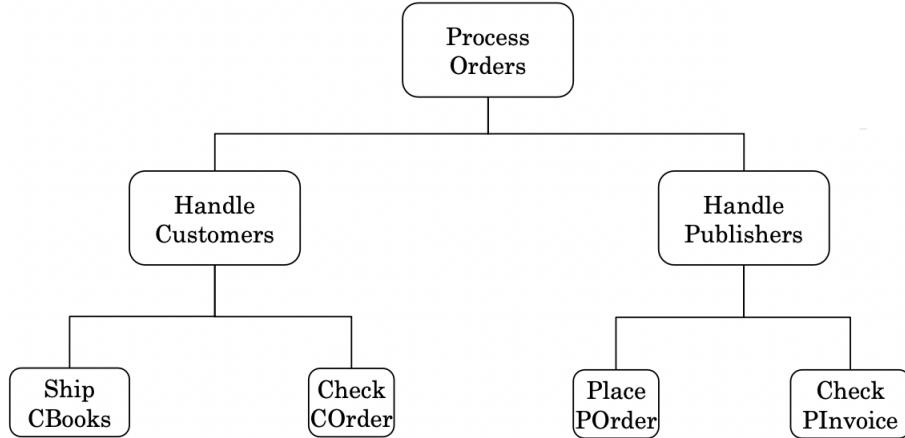
Vengono rappresentati gli attori esterni che comunicano con il processo che in questo caso viene rappresentato come se fosse una macro attività che riceve degli input e degli output. In questo modello l’astrazione è massima. A questo punto si potrebbe pensare di rappresentare il *Process Orders* con una rappresentazione simile a quella iniziale ma sarebbe comunque troppo confusionaria quindi si potrebbe aggiungere un livello di astrazione intermedio:



E a questo punto, avendo semplificato notevolmente la rappresentazione si possono rappresentare i due sotto-processi *Handle Customers* e *Handle Publisher* come diagrammi piatti:



A questo punto si può rappresentare il processo top down attraverso un diagramma che ne metta in evidenza le differenze di astrazione:



Si noti che anche se nel nostro caso abbiamo prima modellato il diagramma piatto e solo successivamente abbiamo costruito il modello Top-Down questo non vuol dire che bisogna fare così. Quello che abbiamo fatto non è solo una semplificazione di un modello ma rappresenta le diverse modalità di rappresentazione dei processi. Può essere che in un primo momento si abbia una visione generale del processo per cui è opportuno approcciare Top-Down proseguendo con approfondimenti successivi delle macro attività individuate.

Il problema della complessità del modello potrebbe essere portata anche nel modello delle classi e si potrebbe risolvere effettuando una scomposizione orizzontale, cioè raggruppando delle classi in una singola classe mantenendo un punto di contatto.

meta-modello di un Dataflow

Come abbiamo sempre fatto per definire un meta-modello si deve guardare ai diversi oggetti presenti e ai loro collegamenti. Oltre ai vari elementi presenti che sono

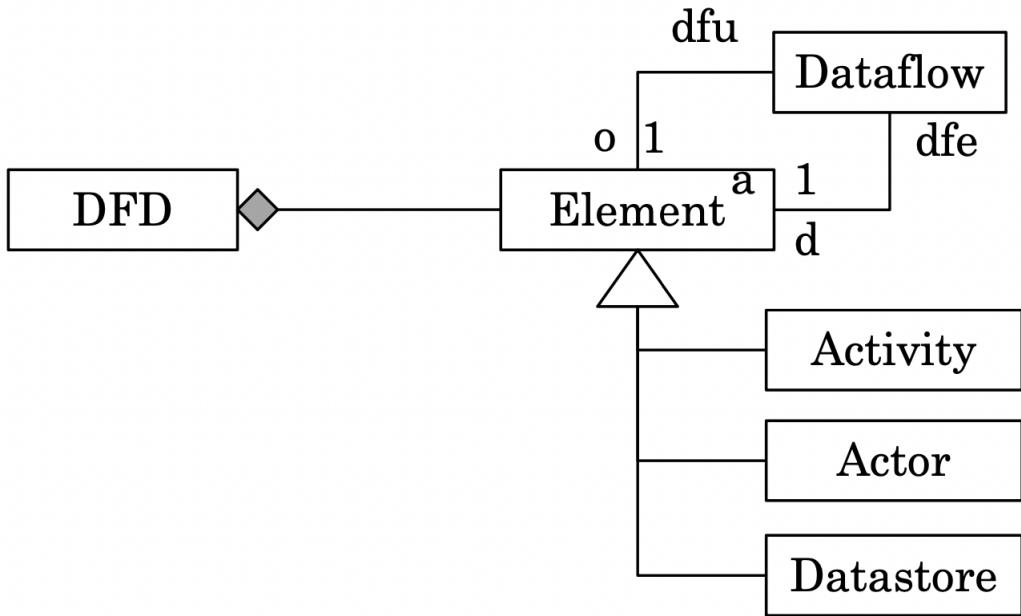
- Attori Esterni
- Attività
- Datastore
- Dataflow

vi sono anche dei collegamenti che non sono permessi che sono sostanzialmente tutti quelli tra gli oggetti non attivi:

- Attore → Attore
- Attore → Datastore
- Datastore → Datastore
- Datastore → Attore

Un attore ha almeno 1 collegamento, un datastore ha almeno 1 collegamento, una activity ha almeno 1 collegamento di input e 1 di output. Attori e attività hanno un nome, i datastore hanno un nome e uno stato (opzionale), i dataflow tra attività e quelli tra attori e attività hanno due etichette (tipo e stato) di cui la seconda può essere nulla.

Per la soluzione di questo meta-modello si può utilizzare lo stesso schema usato per i precedenti:



I cui attributi possono essere:

- DFD: String name;
- Element: String name
- Datastore: String state;
- Dataflow: String type, String state;

Di cui derivati: `List<Element> s = dfu.d; List<Element> p = dfe.o;`

Bisogna inoltre definire dei vincoli per evitare che ci siano i collegamenti non ammessi definiti in precedenza. I vincoli possono essere descritti con un po' di libertà purché siano chiari:

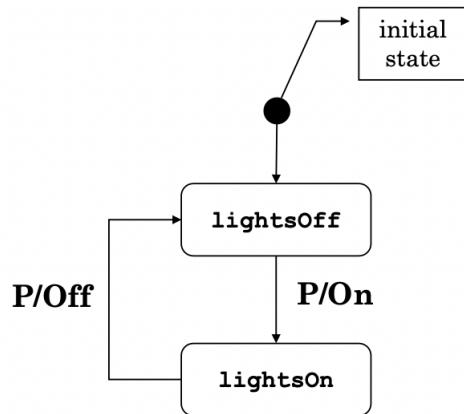
- Si nota che attori e datastore sono collegati solo ad attività: `actor.s instanceof Activity; actor.p instanceof Activity`
- Un attore ha almeno un collegamento, un datastore ha almeno 1 collegamento e un'activity ha almeno un collegamento in input e uno in output: $[actor.p] + [actor.s] > 0$; idem per datastore; $[activity.p] > 0$; $[activity.s] > 0$

Altri vincoli da dover definire sono quelli sulle etichette:

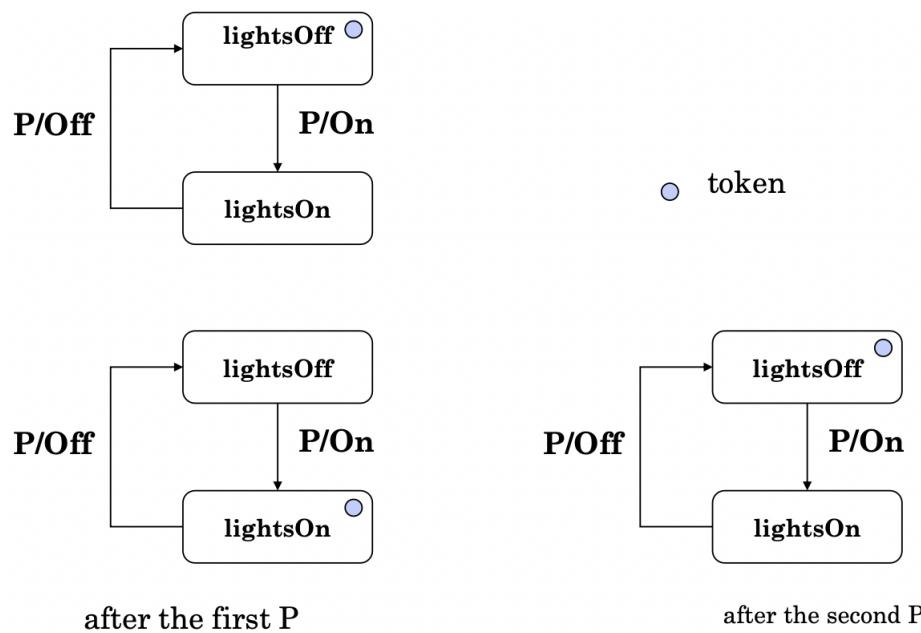
- Un dataflow ha archi entranti e uscenti privi di etichette: `datastore.dfe.type == null; datastore.dfe.state == null;` lo stesso va fatto per `datastore.dfu`
- I dataflow di un attore hanno almeno un etichetta di tipo: `actor.dfe.type def; actor.dfu.type def.`
- Un dataflow che collega due attività ha almeno l’etichetta di tipo: `dataflow.type def`

4.3 Modelli di stato

Un modello di stato è un altro modo per rappresentare processi. In un modello di stato un oggetto attraversa una serie di stati. Il ciclo di vita è gestito da eventi esterni. Normalmente un oggetto aspetta che succeda qualcosa. Per comprendere un modello di stato possiamo usare un semplice esempio con due soli stati:

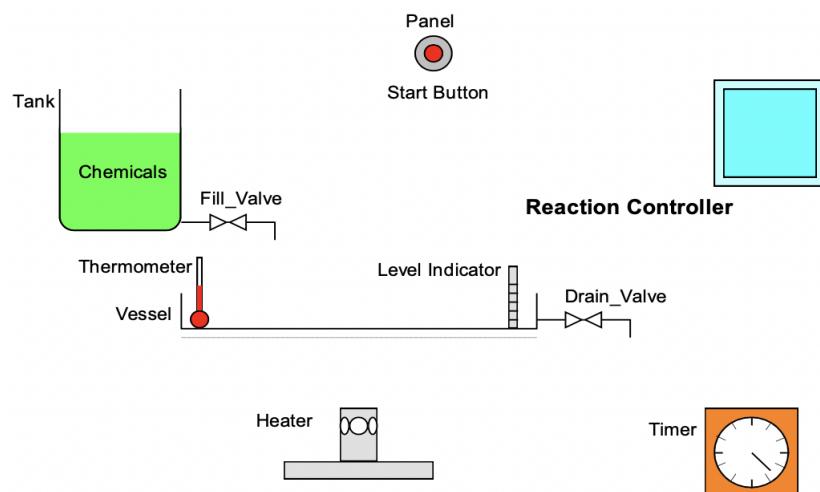


In questo modello gira un token che in base a dove si trova rappresenta lo stato attuale:



Il token è stato inserito solo per dare l'idea del passaggio di stato ma nello diagramma standard non esiste. Nel modello originale lo stato iniziale viene rappresentato dal pallino pieno nero. Quello che vediamo nell'esempio è un diagramma di stato ciclico ma esistono anche diagrammi di stato con un inizio e una fine. Inoltre quello che dobbiamo assicurarci è che una macchina a stati deve essere deterministica, cioè non ci deve essere ambiguità tra gli eventi che comportano un passaggio di stato.

L'esempio che segue è più complesso e rappresenta un controllo a reazione:



Il funzionamento è semplice. Dal **Tank** viene scaricato del liquido nel **Vessel** in cui è presente un controllo di livello che si occupa di chiudere la **Fill Valve** quando si arriva

al livello massimo che il Vessel può contenere. Il **Thermometer** si occupa di accendere o spegnere **Heater** in base alla temperatura attuale. Il **Timer** è preimpostato.

Bisogna individuare gli eventi e i comandi del processo:

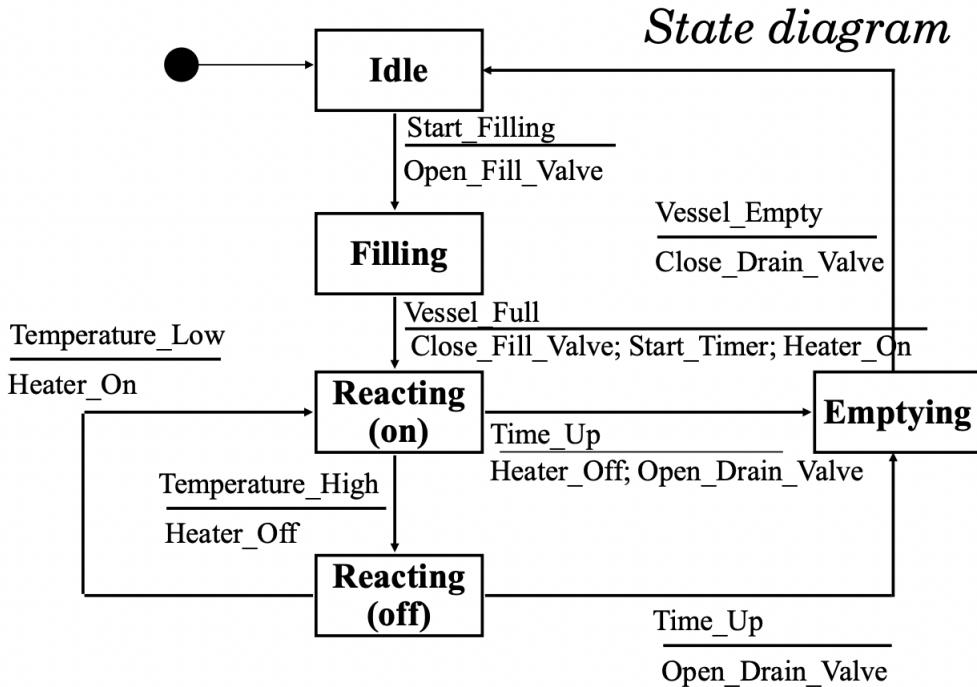
Gli eventi sono

- **Panel**: Start_Filling
- **Vessel**: Vessel_Full, Vessel_Empty, Temperature_High, Temperature_Low
- **Timer**: Time_Up

Mentre i comandi sono:

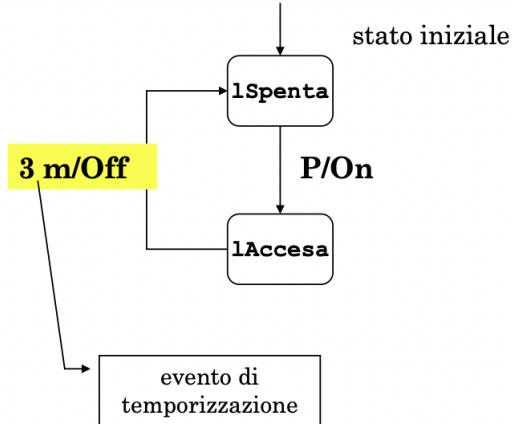
- **Fill_Valve**: Open_Fill_Valve, Close_Fill_Valve
- **Drain_Valve**: Open_Drain_Valve, Close_Drain_Valve
- **Heater**: Heater_On, Heater_Off

Lo stato iniziale del sistema è un uno stato di attesa perché il sistema aspetta che viene avviato il processo.



4.3.1 Transazioni Temporizzate

È possibile specificare un tempo per fare scattare automaticamente una determinata transizione. Un esempio, come quello fatto in precedenza, è una lampadina che si spegne in automatico:



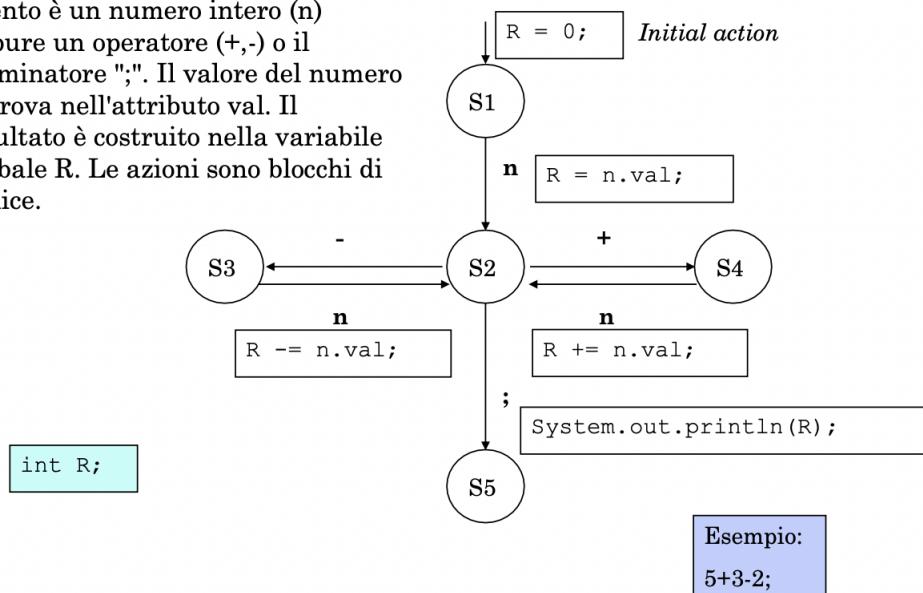
Una volta accesa la lampadina lo stato **lAccesa** rimane attivo per 3 minuti dopo di che si attiva la transizione che fa cambiare lo stato da **lAccesa** a **lSpenta**. Se durante i 3 minuti scattasse una transizione che fa cambiare stato la temporizzazione viene cancellata.

4.3.2 Macchina a Stati Operazionali

Si porta come esempio una calcolatrice che esegue solo somme e sottrazioni. Un evento è un numero intero oppure un operatore (+,-) o il terminatore di operazione ; La chiamiamo Operazionale perché le azioni sono blocchi di codice. Vi sono variabili globali e variabili locali.

Si tratta di una calcolatrice che esegue somme e sottrazioni. Un evento è un numero intero (n) oppure un operatore (+,-) o il terminatore ;. Il valore del numero si trova nell'attributo val. Il risultato è costruito nella variabile globale R. Le azioni sono blocchi di codice.

Operational State Machine



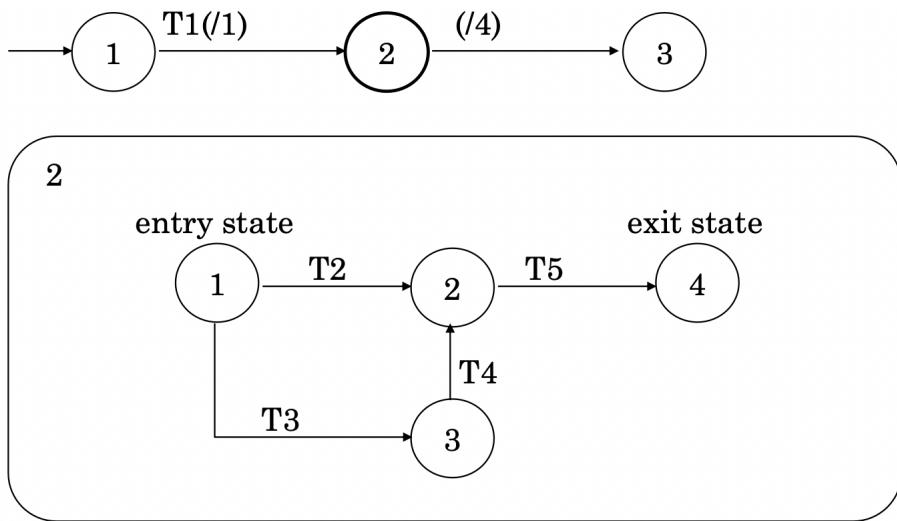
Con questo esempio si vuole introdurre la possibilità di inserire delle azioni al posto dei comandi cioè pezzi di codice che vengono eseguite durante il passaggio tra uno stato e l'altro

4.3.3 Macchina a stati gerarchica

Introduciamo 3 concetti che fanno parte delle macchine a stati gerarchiche:

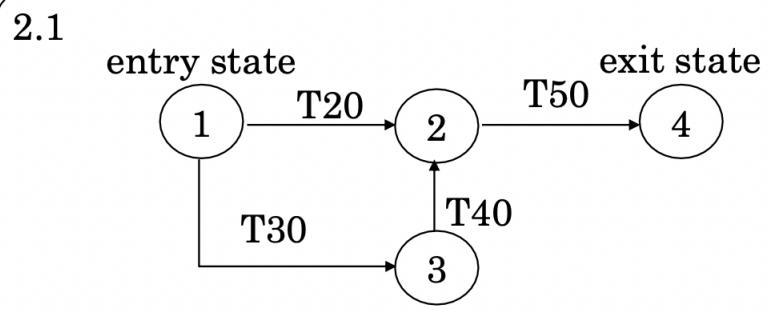
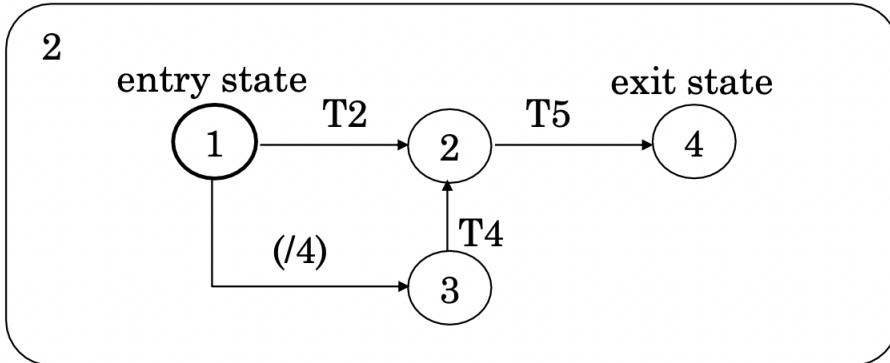
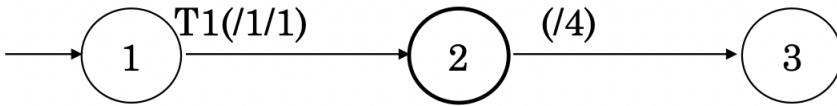
Stati Composti

Uno stato composto è uno stato che rispetta la decomposizione top down. Questo vuol dire che si rappresenta con un nodo a bordo spesso, nell'immagine che segue 2, una cosa più complessa.



con questa rappresentazione nasce l'esigenza di indicare in modo esplicito quale sia il nodo che rappresenta l'entry state del nodo composto 2 e allo stesso modo quale sia l'exit state. Per farlo usiamo la notazione che vediamo nell'immagine precedente cioè con la notazione (/1) e (/4). Questa notazione diventa particolarmente significativa nel momento in cui abbiamo più punti di ingresso o più punti di uscita per cui bisogna specificare quale si intende utilizzare mentre nell'esempio precedente potrebbero essere omessi perché vi è un solo punto di uscita e un solo punto di ingresso. Inoltre si noti che lo stato **exit state** è uno stato virtuale perché il token non si ferma mai in questo stato ma viene solo attraversato ma comunque rimane importante assegnare un nome a questi stati di uscita perché risultano utili quando vi sono più exit state.

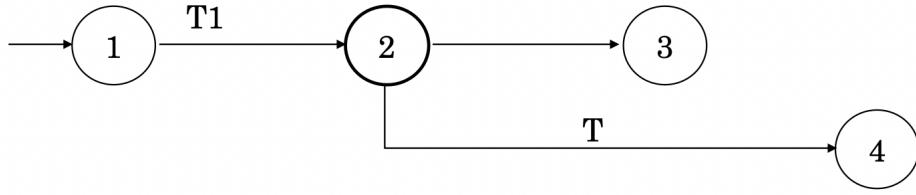
La notazione ci permette anche di annidare a più livello con la stessa modalità:



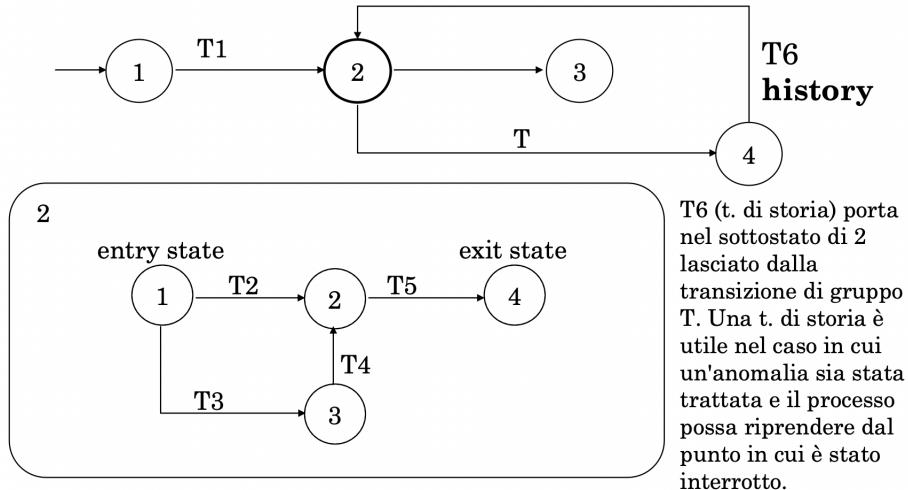
La differenza fondamentale è che nella freccia entrante allo stato composto bisogna specificare gli entry state di tutti i sottolivelli, se necessario. Inoltre notiamo una cosa riguardo la transizione 2: potrebbe sembrare che la transizione T2 non venga mai utilizzata ma in realtà quella transizione viene chiamata **transizione di gruppo** il cui scopo è quello di estrarre tutti i token da uno stato composto, indipendentemente dal suo posizionamento all'interno dei sottostati. Questa tipologia di transizione è utile per la gestione delle anomalie. Si noti che se ci fossero 100 altri livelli verrebbe estrapolato il token indipendentemente dal livello in cui si trova.

transizione di gruppo

Si potrebbe fare qualcosa di più con questa transizione di gruppo, per esempio si potrebbe avere uno stato per gestire le anomalie:



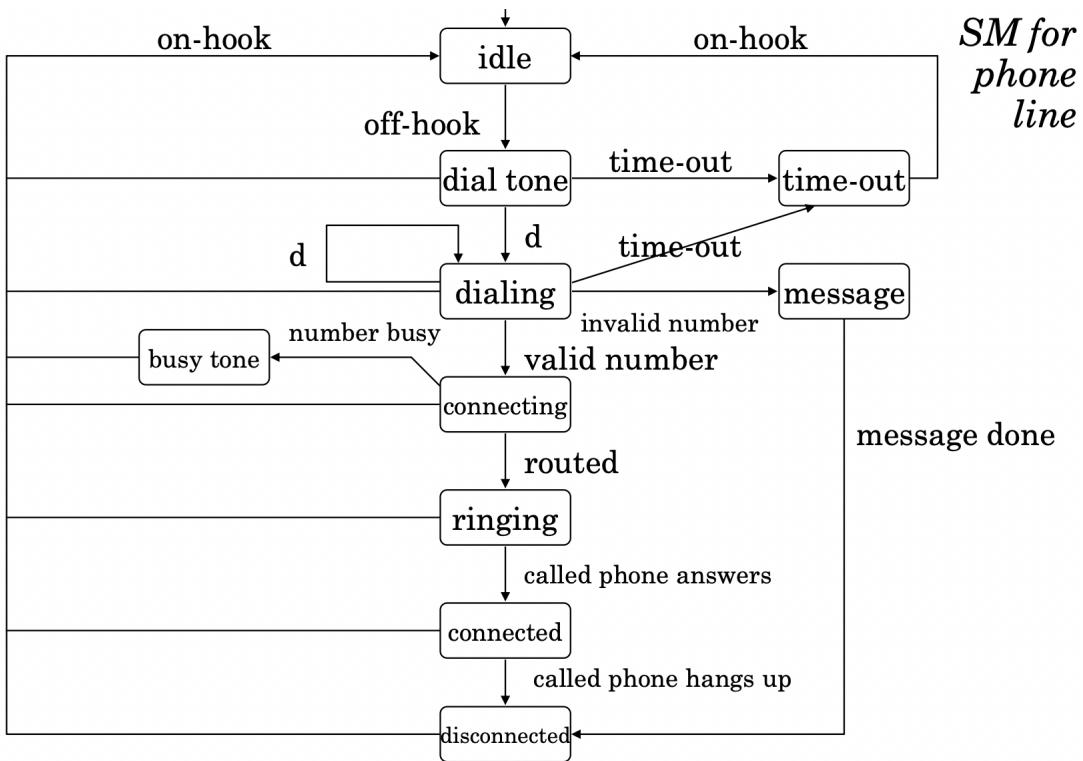
E ancora di più si potrebbe voler gestire l'anomalia è dopo tornare al punto in cui si era rimasti. Per arrivare a questa soluzione si può mischiare la transizione di gruppo con una transizione di storia, come segue:



4.3.4 Macchine di Moore e Mealy

In una Mealy machine le azioni sono associate alle transizioni, mentre in una Moore machine sono associate agli stati. I due modelli sono equivalenti. Nelle implementazioni le azioni si possono trovare associate sia alle transizioni sia agli stati.

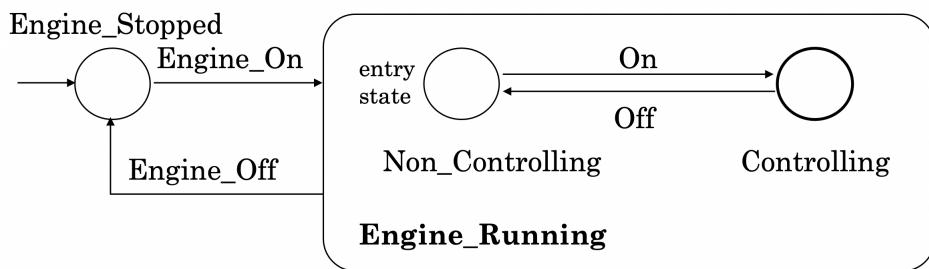
Iniziamo da un esempio di macchina di Mealy che modella una chiamata telefonica tradizionale:



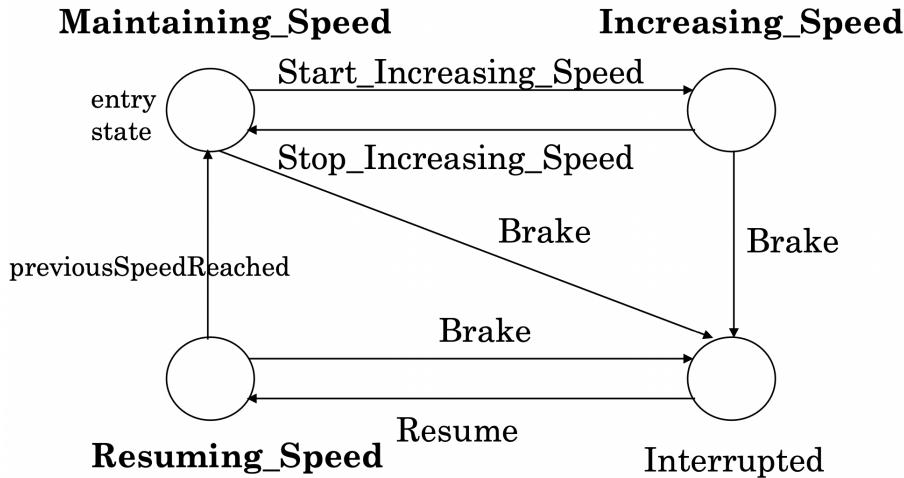
Gli stati rappresentano le azioni in questo caso infatti si nota come, ad esempio, lo stato **dialing** è l'azione di digitazione del numero che come si vede ha un loop che rappresenta la raccolta dei numeri che rappresentano il numero telefonico.

Il nome dello stato danno un significato all'azione come per esempio **dial tone** fa subito capire che il telefono sta squillando. Un altro esempio è quello di modellazione del sistema di Cruise Control di un'automobile:

Il dispositivo è attivo quando il motore è acceso. Quando il guidatore preme il pulsante On, il dispositivo entra in funzione: registra la velocità corrente e la mantiene. In caso di frenata (Brake) il controllo è sospeso; il pulsante Resume consente di tornare alla velocità impostata in precedenza. Due ulteriori pulsanti (Start_Increasing_Speed e Stop_Increasing_Speed) permettono l'accelerazione automatica.



Dove lo stato composto è come segue:



Si noti che uno stato può avere due particolarità:

- **entry action**: azione che viene eseguita dopo una transazione entrante
- **exit action**: azione che viene eseguita prima di una transazione uscente

queste due particolarità non sono da confondere con **entry state** ed **exit state** che sono invece due stati.

Per quanto riguarda il cruise control possiamo individuare 3 diverse azioni:

- **Maintaining_Speed**: entry action: copia la velocità attuale nella variabile targetSpeed. Azione che esegue periodicamente l'algoritmo 1
- **Increasing_Speed**: L'azione esegue periodicamente l'algoritmo 2
- **Resuming_Speed**: L'azione esegue periodicamente l'algoritmo 3

Se la velocità corrente è uguale alla velocità target (o la supera), l'azione interna termina dopo aver emesso l'evento endogeno **previousSpeedReached**.

4.4 Pattern

Soluzione riutilizzabile per problemi ricorrenti

Inizialmente introdotti nel campo dell'architettura da Christopher Alexander che scrisse libri se centinaia di pattern il cui scopo era quello di fornire soluzioni a problemi ricorrenti senza specificarne i dettagli che cambiano da implementazione a implementazione.

Nel campo dell'informatica i pattern sono stati poi introdotti inizialmente da 4 informatici che tutti insieme vengono chiamati la Gang of Four nel 1994 con la pubblicazione del libro *Design Patterns: Elements of Reusable Object-Oriented Software*. Nel libro ogni pattern ha un nome, una spiegazione e una soluzione. Vediamo qualche pattern con i relativi dettagli.

4.4.1 Façade

L'obiettivo di questo pattern è quello di separare un package di libreria da un package applicativo. Questa separazione implica due code

- La libreria deve esporre una classe di facciata, da questo il nome del pattern, con cui è possibile interagire dall'esterno oltre che agli oggetti coinvolti nei vari metodi di libreria di cui il package applicativo deve necessariamente conoscere i dettagli.
- Il package applicativo può interagire con la libreria solo attraverso la classe facciata per cui si riesce a mantenere limitato l'utilizzo che l'applicativo fa della libreria

4.4.2 Factory Method

L'obiettivo è quello di poter generare un nuovo oggetto senza richiamarne il costruttore. Un esempio potrebbe essere un metodo per l'inserimento di un libro in una libreria

```
addLibro(titolo, nVolumi, autori)
```

Come si vede dall'esempio al metodo addLibro non viene passato un libro ma bensì i campi che lo compongono. L'oggetto libro sarà successivamente creato all'interno del metodo in maniera protetta.

4.4.3 Builder

L'intenzione è quella di dividere la costruzione di oggetti complessi dalla sua finale rappresentazione. Quello che viene fatto è costruire una catena di metodi che accodati tra loro possano arricchire la costruzione dell'oggetto fino ad arrivare al risultato finale.

```
Car car = new CarBuilder()
    .setNofSeats(5)
    .setTripComputer()
    .getResults();
```

Questo pattern è spesso utilizzato per la costruzione di oggetti che sono composti da prodotti.

4.4.4 Prototype

Lo scopo è quello di creare un nuovo oggetto a partire da un prototipo già esistente il cui implementa il metodo **clone**. La cosa importante da sottolineare è che si tratta di una copia di valori e non copia di riferimento per cui a seguito di una copia le modifiche che vengono apportate ad un clone non si riflettono a tutti gli altri cloni.

4.4.5 Singleton

Ci garantisce che di un oggetto vi sia una singola istanza in tutto l'applicativo. Di seguito l'implementazione in Java, che sarebbe la stessa in qualsiasi linguaggio di programmazione ad oggetti:

```

public class Dispositivo {
    private static Dispositivo dispositivo =
        new Dispositivo();
    private boolean attivo = false;

    private Dispositivo() {

    }

    public static Dispositivo getDispositivo() {
        return this.dispositivo;
    }

    public void start() {
        if(!attivo) {
            System.out.println("start");
            attivo = true;
        }
    }

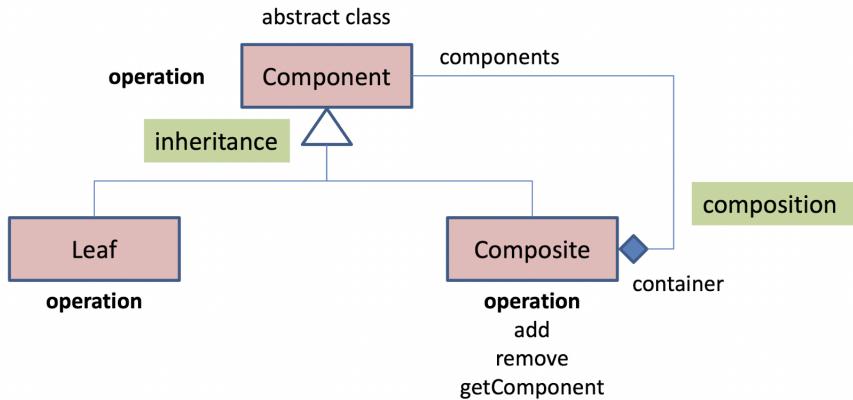
    public void stop() {
        if(attivo) {
            System.out.println("stop");
            attivo = false;
        }
    }
}

```

4.4.6 Composite

Lo scopo di questo pattern è quello di gestire una gerarchia intero-parte trattando oggetti semplici e composti in maniera uniforme. In sostanza si definisce una gerarchia in cui un oggetto composto viene generato a partire da oggetti semplici aggregati tra loro che offrono tutti le stesse operazioni. In sostanza quello che si fa è creare un'interfaccia che si occupa di definire il comportamento e che dia la possibilità di generare oggetti più complessi.

Per capire meglio il pattern basta pensare a una interfaccia FormaGeometrica e due classi che la implementano Rettangolo e Triangolo e adesso immaginiamo di voler creare un Trapezio. Per creare un trapezio sarebbe facile creare una classe che implementa figura geometrica ma per esercizio immaginiamo di voler definire il Trapezio con due triangoli rettangoli e un rettangolo. Questa composizione avrebbe gli stessi comportamenti della figura singola come se fosse una singola figura.



4.4.7 Iterator

Lo scopo è quello di astrarre l'accesso alle collection in maniera sequenziale senza esporre la rappresentazione sottostante del dato. In realtà questo è nato come un pattern ma siccome è molto dipendente dai dettagli del linguaggio di programmazione ogni linguaggio ha provveduto a implementarlo in modo tale da renderlo disponibile al programmatore.

4.4.8 Command

L'obiettivo è di incapsulare la richiesta di una operazione in un oggetto.

4.4.9 Observer

Anche conosciuto come pattern publish/subscribe, il suo scopo è quello di avere un numero di observer che vengono notificati nel momento in cui avviene un cambio di stato. Solitamente l'implementazione prevede che

- Sia implementato un metodo per l'aggiunta e la rimozione degli observer
- Gli observer implementino una interfaccia che ha un metodo *notify*

```

public interface ObserverI {
    void notify(String news);
}

public class Observer implements ObserverI {
    private String name;
    public Observer(String name) {
        this.name = name;
    }

    public void notify(String news) {
  
```

```

        System.out.println(name + ": " +
                           news);
    }
}

public class Subject {
    ArrayList<ObserverI> observerList = new
        ArrayList<>();

    public void addObserver(ObserverI observer) {
        observerList.add(observer);
    }

    public void publish(String news) {
        observerList.forEach(o -> {
            o.notify(news); });
    }

    public static void main(String[] args) {
        Observer john = new Observer("John");
        Observer mary = new Observer("Mary");
        Subject agency = new Subject();

        agency.addObserver(john);
        agency.addObserver(mary);
        agency.publish("Breaking news");
    }
}

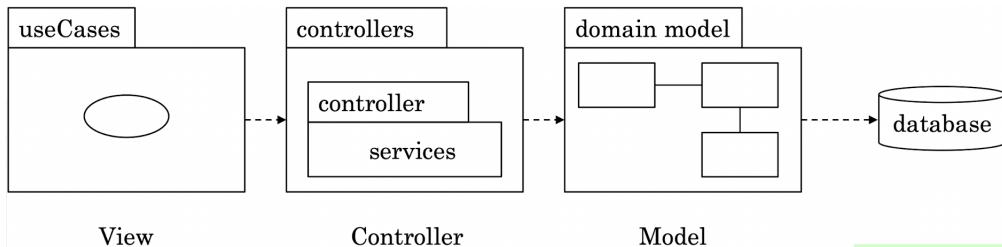
```

Si noti che è conveniente utilizzare un'interfaccia `ObserverI` in modo che si possano avere più oggetti diversi che però funzionano sotto lo stesso `Subject`.

4.4.10 Livelli di pattern

- **Programmazione:** Un pattern a livello di programmazione specifica come risolvere un determinato problema, ad esempio come leggere un file linea dopo linea
- **Progetto:** un pattern a livello di progetto consiste nella generazione di classi e interfacce per ottenere un determinato comportamento, ad esempio mettere in relazione fonti di notizie con entità interessate a riceverle
- **Architettura:** Un pattern a livello di architettura specifica come sottosistemi devono comunicare tra loro per ottenere un comportamento. Un esempio molto famoso di pattern architettonico è MVC.

Nella figura che segue viene illustrato come funziona il pattern MVC attraverso dei simboli che sono quelli dei package in UML



- **Model:** Gestisce la persistenza dei dati e effettua operazioni su DB
- **Controller:** Verificano la correttezza delle richieste e smistano ai sotto-servizi responsabili di gestirla
- **View:** Interfaccia utente

Le frecce tratteggiate che si vedono tra i componenti indicano delle dipendenze per cui un la View dipende dal Controller, il Controller dipende dal Model

Chapter 5

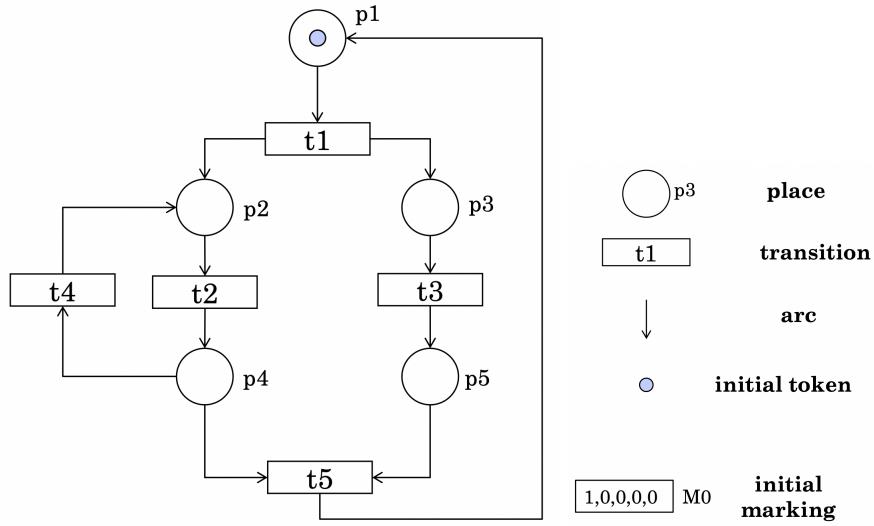
Reti di Petri

La reale motivazione per cui sono nate le reti di Petri, che prendono il nome dal suo ideatore che le portò come tesi di laurea, è che sono ottime per mostrare la concorrenza. Vi sono diverse reti di petri con diversi formalismi:

- Reti ordinarie che si concentrano sul mostrare la concorrenza
- Reti colorate che permettono di rappresentare aspetti funzionali in maniera compatta
- Reti Time-dependent che permettono di effettuare un'analisi dei vincoli di tempo
- Reti operazionali che offrono un linguaggio di altissimo livello per i sistemi di simulazione e sviluppo del software

5.1 Struttura di una rete

Gli elementi che compaiono in una rete di petri sono 3: Posto, Transizione e Arco e li vediamo nella rete che segue:



Inoltre come si vede dalla figura vi sono altri due elementi che però sono solamente rappresentativi che sono il token e la marcatura iniziale di cui parleremo più avanti. I tre componenti Posto, Transizione e Arco sono gli elementi statici del modello, questi non cambiano mai, mentre marcatura e token sono la parte dinamica del modello perché evolvono nel tempo. C'è anche da notare che la rete in questione non prende in considerazione il tempo, non ancora almeno.

Una marcatura rappresenta la disposizione dei token in un certo momento e la marcatura iniziale è fondamentale perché indica come la rete è disposta all'inizio. Vi sono una serie di definizioni formali che descrivono la rete di Petri:

Una rete di petri è definita dalla sua struttura N e dalla sua marcatura iniziale M_0 . La struttura N è una 4-tupla $N = (P, T, F, W)$ in cui ogni elemento è definito come segue

- $P = p_1, p_2, \dots, p_m$ set di posti della rete
- $T = t_1, t_2, \dots, t_n$ set di transizioni della rete
- F è un sotto set di $(P \times T) \cup (T \times P)$ set non vuoto di archi
- $W : F \rightarrow N^+$ funzione dei pesi

Nella definizione si trovano i pesi anche se non ne abbiamo parlato. In generale un arco può essere pesato e questo può indicare quanti token servono per attraversare quell'arco oppure quanti token escono da quell'arco.

Inoltre altre definizioni utili sono

- Il preSet di un posto (transizione) è l'insieme di transizioni (posti) in input al posto (transizione). Il preSet di e : $\bullet e$
- Il postSet di un posto (transizione) è l'insieme di transizioni (posti) in output al posto (transizione). Il postSet di e : e^\bullet

Per fare un esempio di pre e post set, facendo riferimento alla rete vista in precedenza:

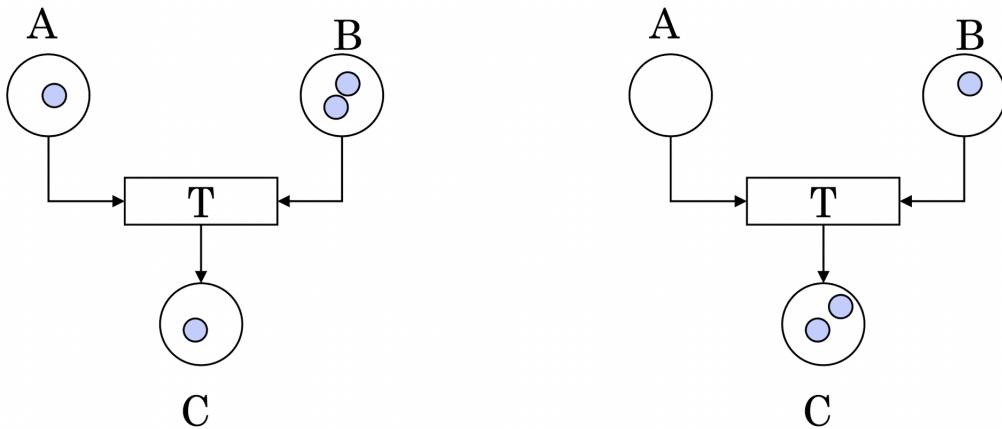
- $\bullet t_5 = (p_4, p_5)$
- $p_4^\bullet = (t_4, t_5)$

In realtà questa notazione potrebbe essere utilizzata anche per dei set, quindi per esempio posso scrivere qualcosa tipo $(t_4, t_5)^\bullet = (p_2, p_1)$

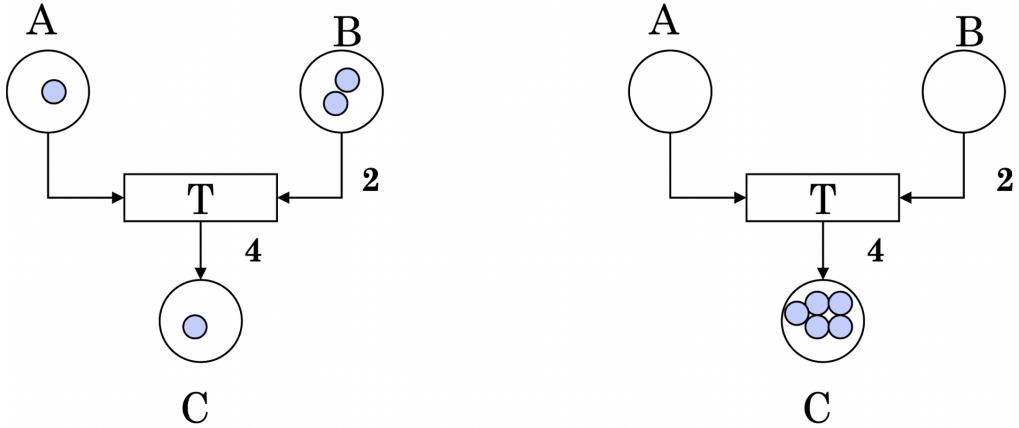
5.2 Effetto di una transizione

Quando una transizione si attiva viene effettuato uno spostamento di token da uno o più posti a uno o più posti. Per capirlo bene vediamolo prima con tutti i pesi pari a 1:

Prima di tutti una transizione si attiva solo se è abilitata e lo è quando gli input sono soddisfatti. Quando una transizione si attiva viene tolto 1 token da ogni posto in input e viene aggiunto 1 token per in ogni posto in output. L'attivazione di una transizione è un'operazione non deterministica perché quando due o più transizioni si attivano contemporaneamente non si sa quale si è attivata prima. Solitamente le transizioni sono operazioni atomiche e istantanee



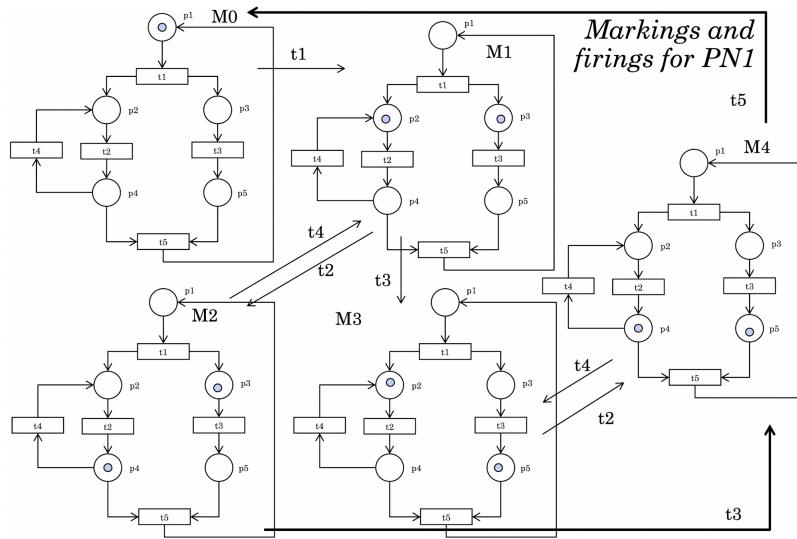
Quando in gioco ci sono i pesi quello che succede non è molto diverso solo che il numero di token necessari per attivare una transizione è subordinato al peso dell'arco come lo è anche il numero di token che esce da una transizione.



5.3 Marcature

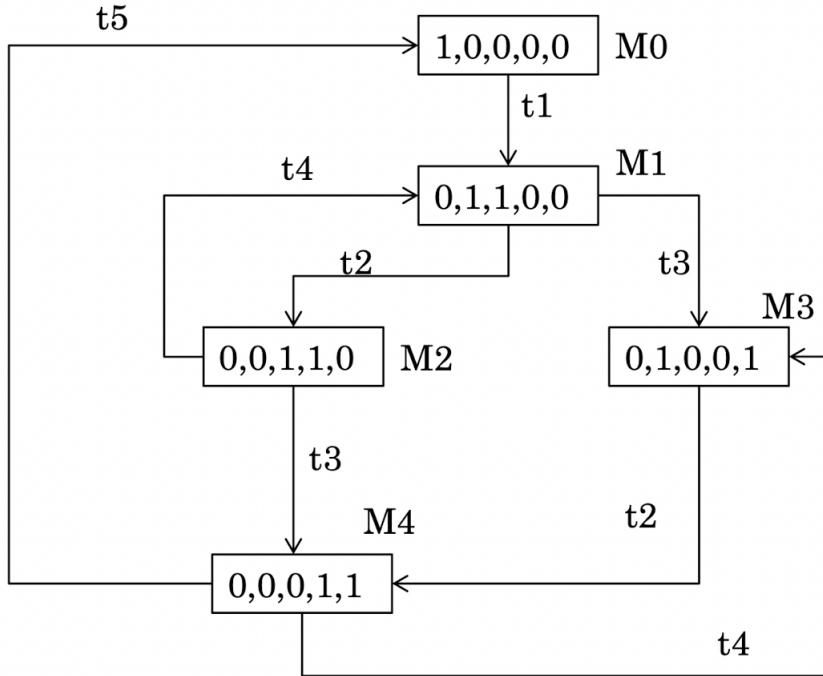
Come abbiamo detto in precedenza in ogni momento si può definire la marcatura corrente della rete che rappresenta il modo in cui sono disposti i token nella rete. In modo formale una marcatura è un vettore $M[i]$ che contiene il numero di token posizionati in p_i posti.

Un **grafo di raggiungibilità** mostra tutte le possibili marcature di una rete e le sequenze di attivazione. Inoltre ci potrebbero essere situazioni in cui far scattare determinate portino a generare infiniti token in un posto. In questo posto, invece di inserire un numero visto che sarebbe troppo alto o addirittura infinito, si inserisce il simbolo ω . Si noti che il grafo di raggiungibilità può essere fatto solo per reti che presentano posti con un numero limitato di token. Per quanto riguarda tutti gli scatti possibili della rete vista in precedenza:



da cui è possibile ricavare il grafo di raggiungibilità che altro non è che un grafo che presenta tutte le marcature possibili con degli archi che rappresentano quale transazione

deve scattare per passare da una marcatura e l'altra:



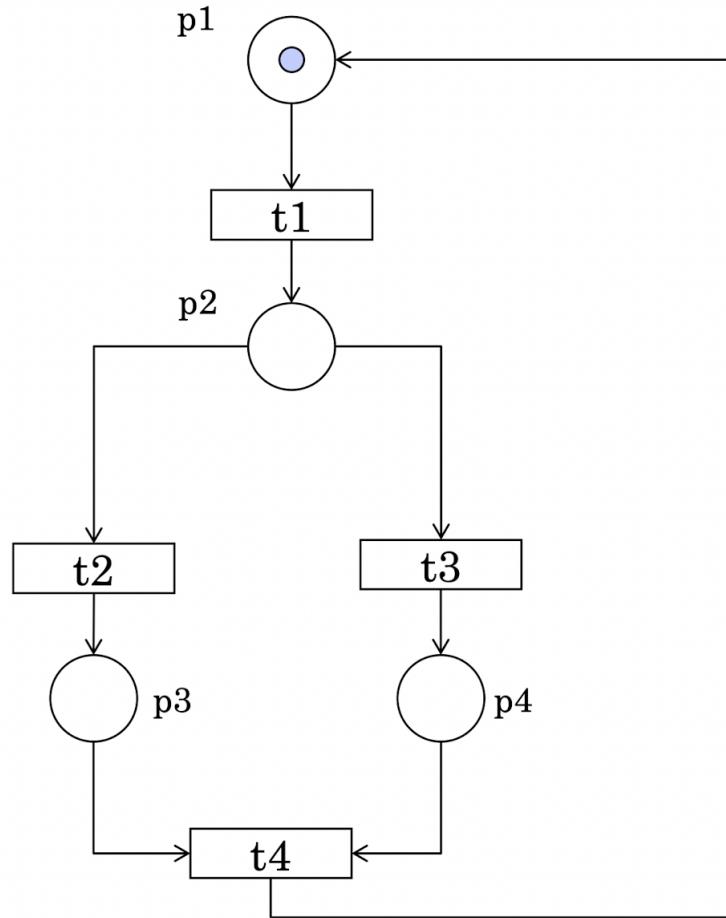
5.4 Control Flow

Il control flow di una rete di Petri è molto simile a quelli visti in precedenza:



5.5 Una modellazione da evitare

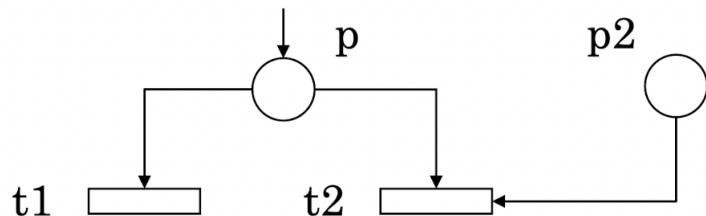
Quando si modella una rete di Petri si potrebbero fare tanti errori che potrebbero far bloccare la rete. Uno da evitare assolutamente è quello di mettere una scelta libera prima di un Join. Una scelta libera è in pratica un posto da cui escono due archi e quindi bisogna scegliere in quale arco mandare il token.



Un posto è una scelta libera se e solo se $[p^\bullet] > 1$ and $\bullet(p^\bullet) == p$ cioè se il postSet di p ha cardinalità > 1 e il preSet del postSet == p . Ci sono diverse tipologie di scelte libere a cui stare attenti quando si modella che però possono anche essere utili in certe circostanze per cui è importante conoscerle, di seguito la modellazione di tutte le scelte libere possibili:

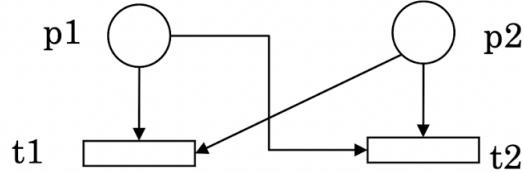
5.5.1 Scelta libera asimmetrica

È una scelta libera che però da uno dei rami dipende anche da un altro posto:



5.5.2 Scelta libera estesa

È una scelta libera che però richiede un token in due posti per poter scegliere quale transizione fare scattare:

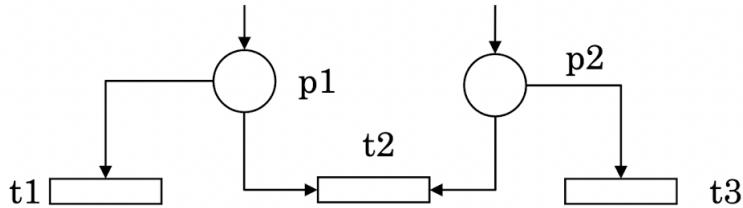


Una scelta libera estesa si individua se:

$$\text{foreach } p \text{ in } P \{p^\bullet == P^\bullet\} \text{ and foreach } t \text{ in } P^\bullet \{\bullet t == P\}$$

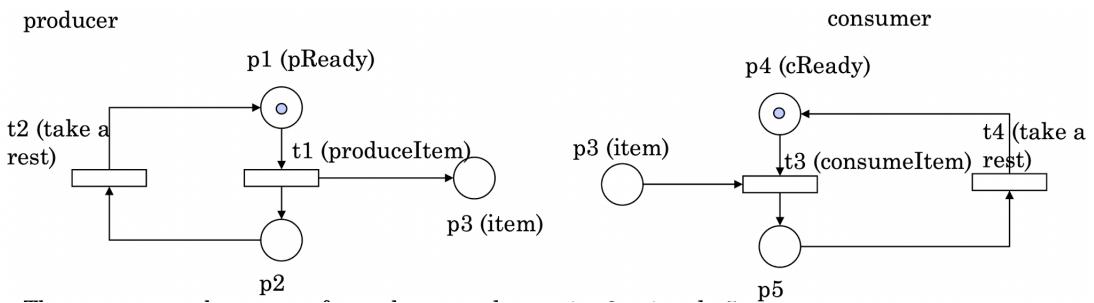
5.5.3 Confusione Simmetrica

Con confusione in realtà intendiamo una scelta libera e in questo caso è una scelta libera simmetrica, il modello è molto esplorativo:



5.6 Producer e Consumer

Il modello di producer-consumer consiste in due cicli, uno per il produttore e uno per il consumatore. Nell'immagine che segue potrebbe sembrare che ci siano due reti differenti in realtà è la stessa rete il cui punto di unione è il posto p3.

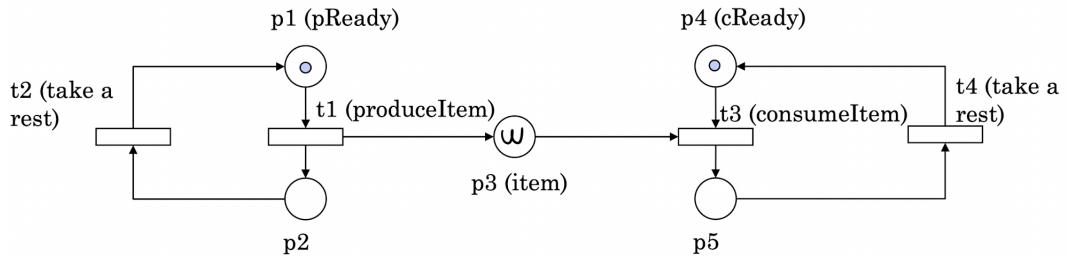


La cosa che è importante notare è che la quantità di token presenti in p3 è strettamente dipendente dalla velocità del producer e del consumer:

- Producer » Consumer: Se il producer è molto più veloce del consumer quello che accade è che il p3 ci sarà un accumulo di Token
- Producer « Consumer: Se il consumer è molto più veloce del producer il posto p3 sarà sempre vuoto perché il token che arriva verrà immediatamente consumato senza dare la possibilità di accumularsi

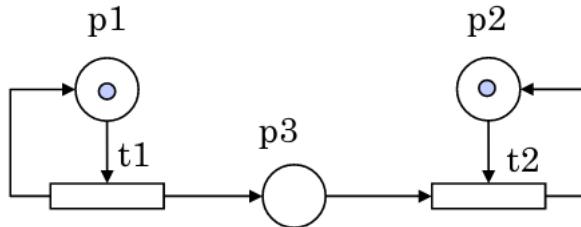
Nel primo caso nel posto p3 andrebbe messo un ω per indicare che il numero di token è molto alto, potenzialmente infinito.

La rete unita è come segue:



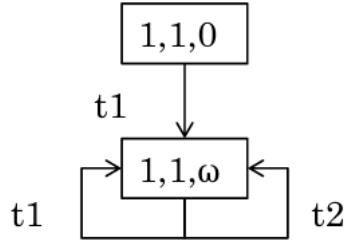
Quando una rete di Petri è illimitata, unbounded, il grafo che si ricava non è delle marcature raggiungibili ma un grafo di copertura, di coverability. Inoltre c'è da notare che la rete non è fortemente connesso perché come si vede dal produttore al consumatore è possibile muoversi mentre al contrario non è possibile farlo.

Per poter studiare il grafo di raggiungibilità effettuiamo una semplificazione alla rete anche se non è particolarmente complessa. La semplificazione consiste nell'eliminare la transizione t2 e t4.



Si noti che sono stati modificati i nomi per rendere i posti e le transizioni successivi. Abbiamo la marcatura iniziale $M_0 = [1, 1, 0]$ e notiamo che facendo scattare, dalla marcatura iniziale, la transizione t1 passiamo alla marcatura $M_1 = [1, 1, 1]$ e notiamo che la marcatura M_1 copre strettamente la marcatura M_0 cioè abbiamo che M_1 contiene M_0 e inoltre facendo scattare nuovamente t1 abbiamo che M_1 continua a coprire M_0 con l'incremento dei token nel posto 3. Quando ci troviamo in una situazione del genere il posto che aumenta quando una transizione scatta è un posto unbounded e viene etichettato con ω .

Da notare che ω vuol dire infinito per cui è valida la notazione $w - 1 = w = w + 1$. Il grafo di copertura è il seguente:



da cui potrebbe sembrare che la rete sia irreversibile perché una volta passati da t_1 si rimane intrappolati nel ciclo attorno alla marcatura $[1, 1, \omega]$ ma in realtà guardando la rete si vede bene che facendo scattare il consumatore infinite volte senza fare scattare il produttore quello che otterremmo è la marcatura iniziale $M_0 = [1, 1, 0]$

5.7 Costruire il grafo di raggiungibilità (copertura)

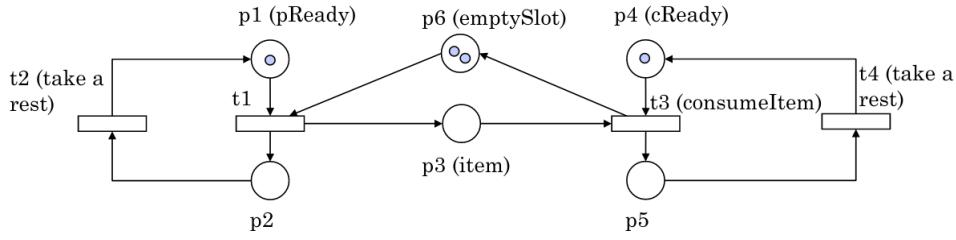
```

while there is a new node, N (whose marking is M), do
    if no transitions are enabled in M then label N dead-end
        (deadlock node)
    else
        for each transition, t, enabled in M do
            compute the marking, M', obtained from M when t
                fires;
            if there is a node, N'', that contains marking
                M' then
                draw an arc labeled with t from N to N';
            else do
                if there is a node, N'', that is an ancestor
                    of N (i.e., on the path from N to the
                    root) and contains a marking, M'', that is
                    strictly covered by M', then
                    set M'[p] to omega for each p for which
                        M'[p]>M''[p];
                    add a new node, N', containing M', and
                    draw an arc labeled with t from N to
                    N'; label N' as new;
            label N as examined.

```

Si noti che il grafo prende il nome di grafo di raggiungibilità nel caso in cui esso non presenti alcuna ω mentre al contrario si chiama grafo di copertura.

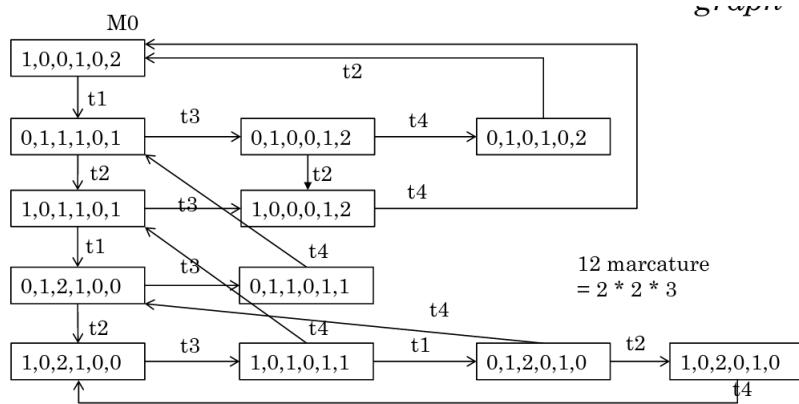
Prima di vedere un esempio di grafo di raggiungibilità vediamo una rete Produttore-Consumatore con una modifica che ci permette di non avere il posto unbounded. Solitamente i posti unbounded sono da evitare perché non si prestano a future implementazione perché implicano grande consumo di risorse:



Con questa soluzione quello che si fa è semplicemente limitare il numero di token che possono trovarsi in p_3 al numero di slot disponibili dal consumatore.

Una considerazione particolare da fare su questa rete è che il numero di token complessivi in p_1 e p_2 e in p_4 e p_5 è sempre 1 per coppia. Questo introduce il concetto in invarianti. Lo stesso identico concetto è applicabile a p_6 e p_3 in cui il numero di token è al più 2. Generalizzando un invarianto è un insieme di posti in cui, al variare della marcatura, il numero di token rimane costante.

Il grafo di raggiungibilità della rete bounded vista in precedenza, partendo dalla marcatura iniziale $M_0 = [1, 0, 0, 1, 0, 2]$ è:



5.7.1 Definizione raggiungibilità

- Date due transizioni M e M' , M' è direttamente raggiungibile da M se esiste una transizione t , abilitata in M , che scattando fa passare lo stato da M a M' .
- Date due transizioni M e M' M' è raggiungibile se esiste una serie di transizioni che scattando portano la rete dallo stato M allo stato M'
- Data una rete $PN=(N, M_0)$ il suo set di raggiungibilità R è l'insieme di tutte le transizioni raggiungibili da M_0 (inclusa).
- Date due transizioni M e M' , M' copre M se e solo se $M'(p) \geq M(p)$ per ogni p in P . M' copre strettamente M se e solo se $M' \supsetneq M$

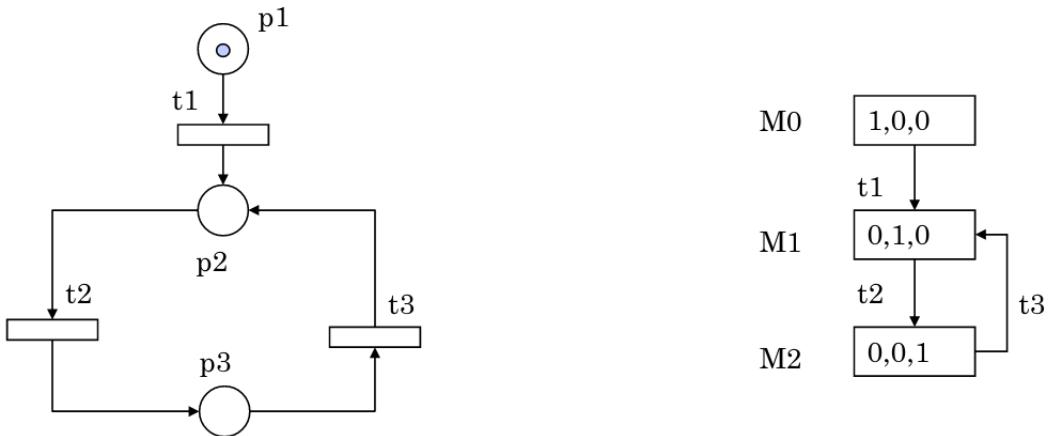
5.8 Proprietà comportamentali

Sono le proprietà della rete, quello che ci aspettiamo che la rete soddisfi

- **Boundedness:** Una rete di Petri è *k-bounded* se e solo se il numero di token in ciascun posto non eccedono un numero k.
- **Safeness:** Se una rete è 1-bounded
- **Liveness:** Una rete è viva se tutte le transizioni sono vive. Una transizione T è viva se e solo se per ogni marcatura raggiungibile M' esiste una marcatura M'' che è raggiungibile da M' e T è abilitata da M''. Una transizione T è *dead* se non è mai abilitata da nessuna marcatura
- **Deadlock-freedom:** Se ogni marcatura raggiungibile abilita qualche transizione
- **Reversibility:** Una rete di Petri è reversibile se e solo se per ogni marcatura raggiungibile M, M_0 è raggiungibile da M. In generale ogni marcatura raggiungibile da ogni altra marcatura è definita come *home-state*. La presenza di un *home-state* denota un sistema periodico

Le proprietà Boundedness, Liveness e Reversibility sono indipendenti quindi dal punto di vista empirico il numero di condizioni da verificare sono 8, 2^3

5.8.1 Esempi



La rete è una rete deadlock-free perché ogni transizione scatta almeno una volta ma non è una rete live perché la transizione t1 scatta soltanto una volta. Nelle slide sono disponibili diversi esempi sulle proprietà comportamentali però qui vengono riportati i formalismi per indicare le proprietà:

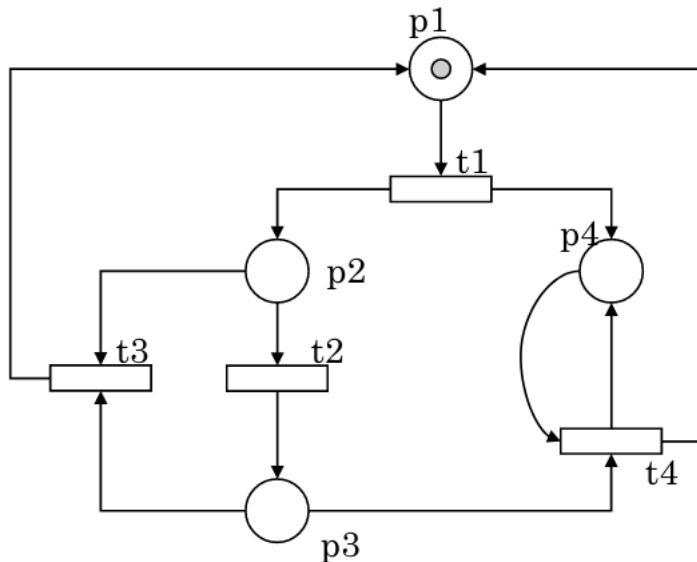
unbounded **B**

not live **L**

not reversible **R**

Inoltre le stesse lettere possono essere utilizzate per indicare le proprietà avvocate, senza la stanghetta orizzontale.

Vi sono poi dei normalismi sulle transizioni che dobbiamo iniziare a utilizzare per poter descrivere bene le reti. Ci aiutiamo con una rete per dare i formalismi:



- La transizione t1 è detta *fork* perché ha una sola freccia entrante ma più frecce uscenti
- La transizione t2 è detta *passante* perché ha una freccia entrante e una uscente
- La transizione t3 è un *join* perché ha più di una freccia entrante e una sola freccia uscente
- La transizione t4 è un *fork-join* perché ha più di una freccia entrante e più di una freccia uscente

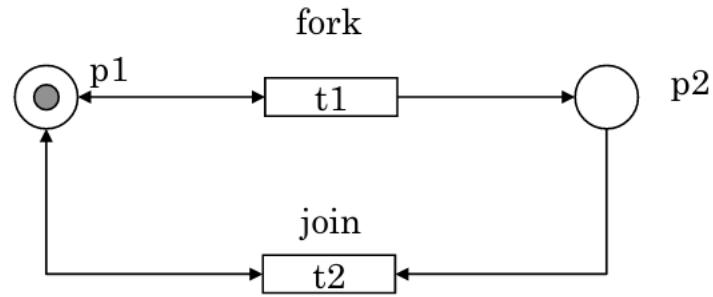
Le proprietà comportamentali si possono analizzare attraverso un grafo di raggiungibilità che però diventa grafo di copertura nel caso di una rete illimitata.

Di seguito sono riassunti i risultati delle analisi effettuate in precedenza:

- una rete di Petri è limitata se non compare ω altrimenti è illimitata
- Una rete è *safe* se e solo se le marcature contengono solo 0 e 1

- Dato un grafo di raggiungibilità M è raggiungibile se e solo se esiste un nodo che contiene M . Se abbiamo un grafo di copertura allora una transizione M è copribile se esiste un nodo che copre M .
- La marcatura corrispondente a un dead-end node denota un deadlock.
- In generale non è possibile dire se una rete è live o meno partendo dal grafo di raggiungibilità o di copertura.

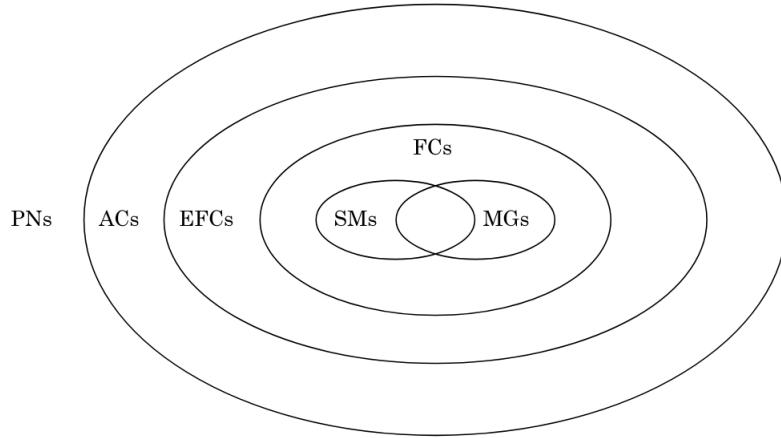
5.8.2 Archi bidirezionali



Come si vede in figura ci sono degli archi bidirezionali che in realtà è solo una rappresentazione grafica di due archi, uno in un verso e uno nell'altro.

5.9 Analisi di sottoclassi delle reti di Petri

Assumiamo, anche se non sarà sempre soddisfatto, che le reti siano strettamente connesse e inoltre i pesi vengono sempre omessi perché sono sempre pari a 1. Nella figura che segue si vede il posizionamento delle sottoclassi delle reti di petri. Al centro si vede SMs e MGS dove SMs sono le macchine a stati che abbiamo fatto in precedenza mentre MGs è un grafo a marcato che vedremo oggi:



5.9.1 Macchine a stati

Le reti di petri sono considerate macchine a stati nel momento in cui ogni transizione hanno esattamente un posto in input e un posto in output. Sono ancora presenti le scelte libere ma non esistono fork o join. Inoltre la marcatura iniziale ha un solo token. Le proprietà di una rete di petri a stati sono quelle che abbiamo visto in precedenza con qualche modifica:

- Una SM è live se contiene almeno un token, poiché è libero di girare
- La rete è safe se è contenuto al massimo 1 token nell'intera rete
- Con più token nella rete si continua a definire ancora bounded ma non è più safe

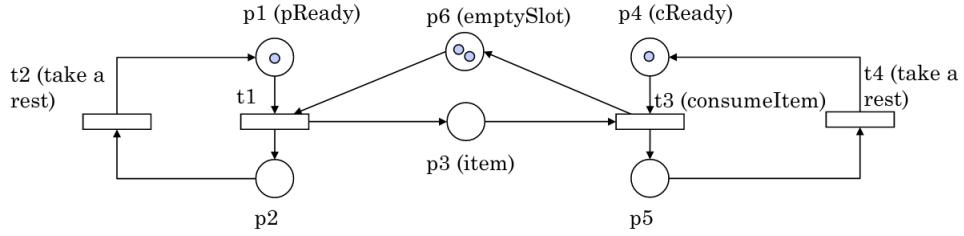
5.9.2 Grafi marcati (MGs)

Un posto può avere una sola transizione in input e una sola in output. Non vi è alcuna limitazione per le transizioni che possono essere fork e join.

I grafi marcati esprimono concorrenza e sincronizzazione poiché non possono essere modellati conflitti e confluenze.

Le proprietà di una MG possono essere studiate facilmente attraverso i suoi circuiti. Un circuito è una sequenza di posti e transizioni(chiamati elementi) e_1, \dots, e_n tale che tutti gli elementi sono distinti e che $e_i + 1$ in e_i^\bullet e e_1 in e_n^\bullet cioè che l'elemento $i+1$ deve trovarsi nel postSet dell'elemento i e il primo elemento deve essere nel postSet dell'ultimo elemento

Esempio di circuiti



Nella figura possiamo individuare 3 circuiti, questa volta totalmente disgiunti ma potrebbero anche non esserlo:

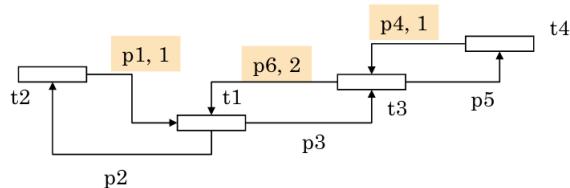
- Partendo da p1: p1, t1, p2, t2
- Partendo da p4: p4, t3, p5, t4
- Partendo da p3: p3, t3, p6, t1

Abbiamo delle notazioni che ci permettono di semplificare la scrittura dei circuiti e sono principalmente due anche se considereremo solo la prima:

- Scriviamo solo i posti per rappresentare il circuito, ad esempio p1, p2 invece di p1, t1, p2, t2
- Scriviamo solo le transizioni per rappresentare il circuito, ad esempio t1, t2 invece di p1, t1, p2, t2

Adesso ci potremmo chiedere se la rete è live e come abbiamo detto in precedenza per studiare le proprietà dobbiamo solo guardare i circuiti infatti per capire se una rete del genere è bounded basta verificare se c'è un token in ogni circuito.

È possibile generare il grafo marcato corrispondente alla rete di petri e il risultato è quello che segue



i numeri evidenziati in giallo accanto al nome dei posti indicano il numero di token. In realtà questa modellazione era stata sviluppata in maniera distinta dalla rete di petri e poi si è trovata questa corrispondenza, noi non utilizzeremo questa modalità di rappresentazione.

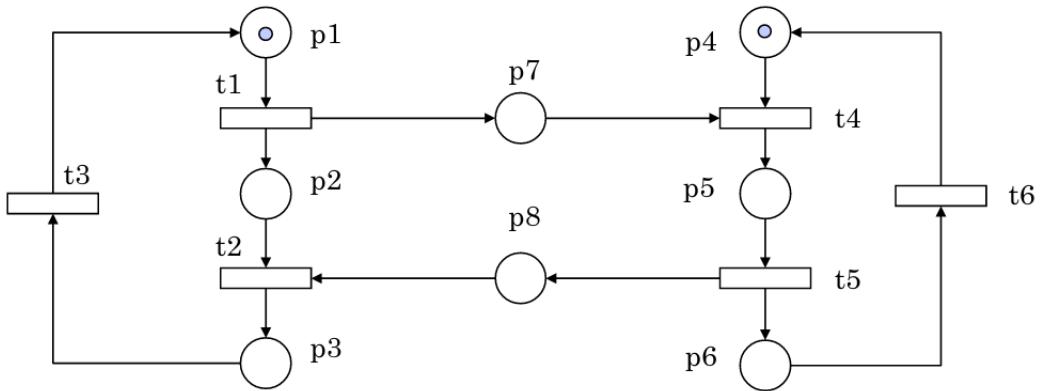
Di seguito sono elencati i risultati per le reti marcate

- Un MG è *bounded* se e solo se è fortemente connesso

- Il numero di token in un circuito è costante ed è pari al numero dei token iniziali
- Come detto in precedenza un MG è *live* se e solo se ogni circuito ha un *token count* > 0
- Un circuito si dice marcato se e solo se il suo *token count* è > 0
- Un posto potrebbe far parte di uno o più circuiti
- Il numero massimo di token che possono trovarsi in un posto è pari al token count del circuito (tra quelli di cui fa parte) che ha il minimo token count.
- Un MG live è *safe* se e solo se per ogni posto si può trovare un circuito (tra quelli di cui fa parte) il cui token count è 1.

Le ultime due affermazioni sono molto importanti da capire perché spesso ci sono errori. Anche se nell'esempio precedente non è evidente ma un posto può essere parte di più circuiti ed è per questo che vi sono le ultime due affermazioni.

Vediamo adesso un esempio più complesso e più completo



I circuiti in questo caso li rappresentiamo con la modalità semplificata e sono:

- p1, p2, p3
- p4, p5, p6
- p1, p7, p5, p8, p3

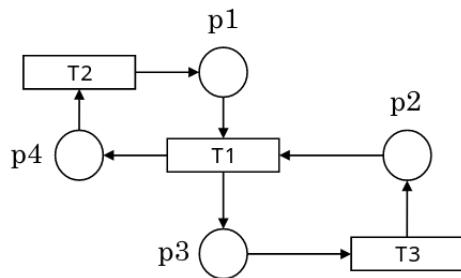
Si noti che anche p7, p5, p8, p3, p1 è valido ma vi è una regola importante per la scrittura dei circuiti che dice che il posto di partenza di un circuito deve essere quello con indice inferiore.

Per quanto riguarda la proprietà possiamo dire sicuramente che vi è almeno 1 token in ogni circuito. Vi sono diverse considerazioni che possono essere fatte su questa rete e bisogna fare attenzione alla marcatura iniziale che potrebbe modificare drasticamente

le proprietà. Inoltre non è possibile marcare tutti i circuiti con 1 token perché vi sono circuiti che condividono lo stesso token e quindi possiamo derivare due gruppi che sono praticamente circuiti disgiunti. Il gruppo 1 è formato dai due circuiti p1, p2, p3 e p4, p5, p6 mentre il secondo gruppo è p1, p7, p5, p8, p3. Considerando questi due gruppi dobbiamo prendere quello con il numero maggiore di gruppi per capire quali sono i circuiti di base. Si noti che i circuiti nel primo gruppo sono totalmente disgiunti.

Di seguito sono elencate le regole a cui attenersi per l'individuazione dei circuiti:

- Si parte ad analizzare dal posto p1 e si individuano i circuiti e poi si procede con i successivi, detti candidati
- In un circuito il posto iniziale deve essere il posto con indice inferiore del circuito
- Un posto candidato è ignorato se i posti immediatamente successivi o precedendo hanno indici inferiori
- In un singolo circuito una transizione non può essere attraversata più di una volta.



Nella foto partendo da p1 individuiamo:

- p1, p4
- p1, p3, p2, p4

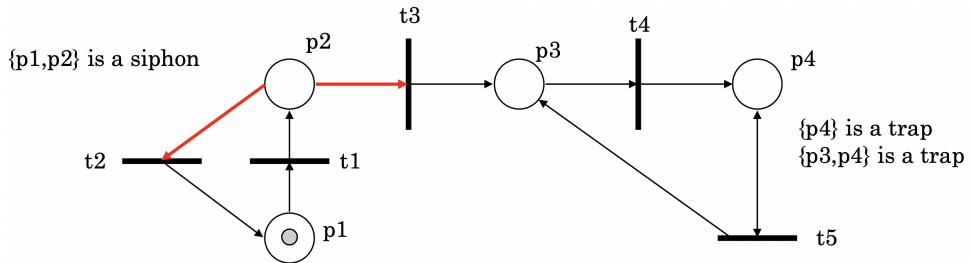
ma attenzione al fatto che il secondo circuito proposto non è valido perché la transizione T1 viene sottesa due volte, vediamolo per esteso: p1, T1, p3, T3, p2, **T1**, p4, T2. Da p2 individuiamo:

- p2, p4, p1, p3
- p2, p3

Il primo circuito proposto non è valido per lo stesso motivo del precedente circuito non valido, T1 viene sottesa 2 volte.

5.9.3 Reti Free-Choice

La caratteristica delle reti FC è che sia posti che transizioni possono avere più input e più output. Ciascun posto che abbia una o più transizioni di output è l'unico posto di input di quelle specifiche transizioni. Anche se è possibile intraprendere una sola direzione partendo da un posto FC dobbiamo fare attenzione a effettuare tutte le combinazioni, una alla volta, per capire se ci sono deadlock nella rete.



Utilizzeremo due concetto nuovi per formalizzare queste tipologie di reti:

- **Sifone:** Se un sifone, nella rete precedente la coppia p_1, p_2 , rimane senza token allora rimarrà senza token per tutto il resto. Nell'esempio precedente si vede bene che il token che gira tra p_2 e p_1 può rimanere a girare tra loro all'infinito ma nel momento in cui esce dalla coppia p_1, p_2 è impossibile che torni indietro
- **Trappola:** Se una trappola è marcata, nell'esempio precedente p_4 è una trappola, è impossibile che perda i token. Nell'esempio precedente bisogna notare che la freccia tra p_4 e t_5 è bidirezionale. Questo vuol dire che t_5 può sempre scattare ma non perde mai i token. Trappole come quelle nella rete precedente, cioè con un arco bidirezionale, è detta *trappola monoposto*.

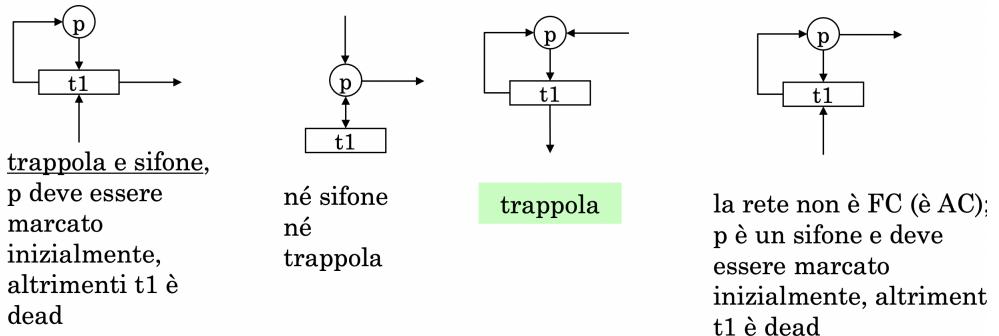
Di seguito la definizione formale di sifone e trappole:

- Preso un subset non vuoto S di posti è un sifone se e solo se una transizione toglie un posto da un token da un posto nel subset considerato allora la stessa transizione aggiunge un token in un posto sempre appartenente al subset S . Come detto in precedenza se un sifone rimane vuoto rimarrà vuoto per tutte le marcature successive
- Preso un subset Q non vuoto di posti è una trappola se e solo se ogni transizione che ha come input un posto in Q e ha anche un output in Q .

Inoltre si nota che l'unione di due o più sifoni è ancora un sifone e allo stesso modo l'unione di due o più trappole è ancora una trappola.

Una rete FC è *live* se e solo se i suoi sifoni contengono(o sono uguali a) un token iniziale nelle trappole.

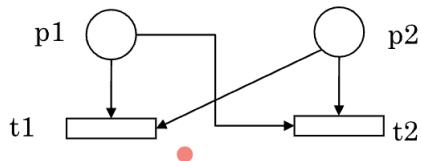
Vi sono dei pattern noti che ci portano a considerare dei subset monoposto. Sono principalmente 4 e tutti e 4 rispettano la regola data in precedenza, cioè che un post ha sia un input che un output dalla stessa transizione:



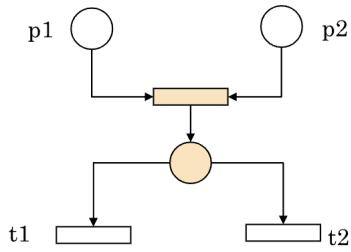
Nelle reti FC possiamo considerare le proprietà di boundedness e di freedom. Per farlo non vorremmo fare il grafo delle marcature per cui le seguenti considerazioni possono essere utili per analizzare questi aspetti:

- Una rete è unbounded se si trova una sequenza di scatti ripetitiva che aumenta il numero di token nella rete: ad esempio una sequenza di transizioni passanti tranne una transizione fork.
- Una rete non è deadlock free se si trova una sequenza di scatti che porta tutti i token della rete nello stesso posto che ha come unico output una transizione join (o join-fork).
- Una rete che non è deadlock free (cioè che ha un deadlock) non è live.
- In una rete in cui non compare l' ω deadlock freedom e liveness coincidono
- Una rete unbounded potrebbe essere DF ma non *live*

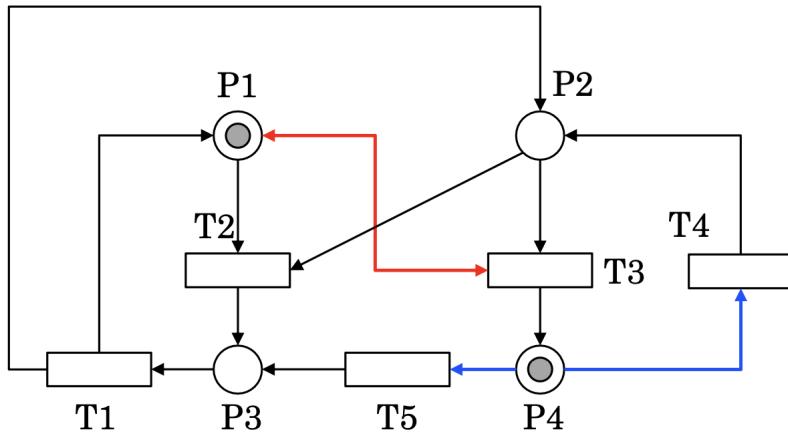
5.9.4 Reti Free Choice Estese



perché la rete precedente è EFC? Per il semplice fatto che qui si tratta di due posti che hanno come transizioni di uscita le stesse 2 transizioni in comune. Si potrebbe fare un esempio con più di due transizioni e due posti ma per semplicità le tratteremo così, con un numero limitato di posti. Anche se noi le tratteremo senza alcuna manipolazione c'è da notare che un EFC potrebbe essere trasformata in una FC.

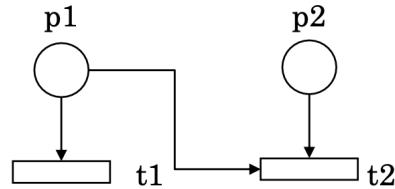


Ma questa dualità ci permette di utilizzare i teoremi delle FC anche sulle EFC ed è per questo che possiamo utilizzare il teorema di Commoner.



- Che tipo di rete è?
- Quali sono i sifoni?
- Quali sono le trappole?
- La rete è live oppure no? Si spieghi perché
- La rete ha dei deadlock o no? Se sì con quale marcatura?
- La rete è bounded? Se no in quali posti?
- La rete è safe o no?
- La rete è reversibile o no e perché?
- Nel grafo delle marcature come sono scritte le marcature che si ottengono con uno scatto di transizione da M0?

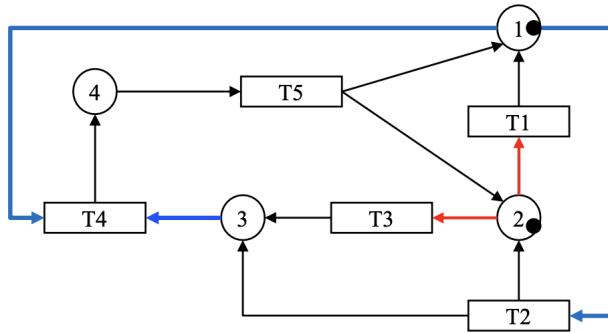
5.9.5 Reti a scelta asimmetrica



La rete potrebbe anche essere chiamata confusione, la logica è esattamente come quella che abbiamo introdotto quando abbiamo parlato delle scelte asimmetriche. La scelta è asimmetrica perché t_1 può scattare solo con un token in p_1 mentre t_2 può scattare solo se si hanno due token in p_1 e p_2 contemporaneamente.

Per lo studio della *Liveness* non è più abbastanza lo studio con il teorema di Commoner che è sufficiente ma non necessario. Bisogna considerare una seconda definizione che è quello di *Place Invariant* che è una combinazione lineare di posti che presenta lo stesso numero di token per qualunque marcatura. Il place invariant garantisce che un sifone presenta un numero di token **SEMPRE** maggiore di 0.

Il motivo per cui il teorema di Commoner non va più bene è per via della presenza dell'asimmetria.



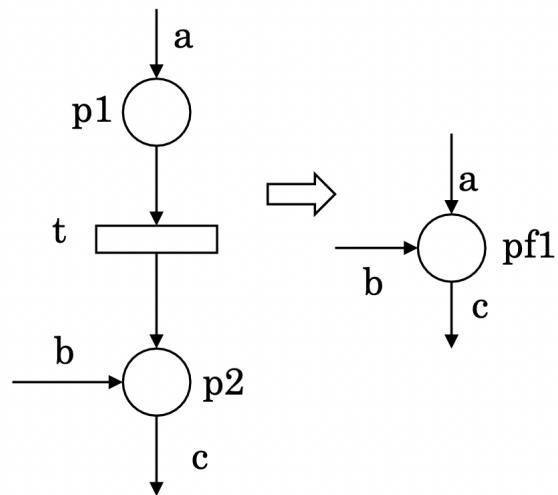
Per riconoscere una rete AC bisogna individuare il postSet di un posto che contiene 2 transizioni e il postSet di un posto che comprende solo una delle due transizioni contenute nel primo postSet. In sostanza il primo postSet include strettamente il secondo postSet.

- Esistono sifoni che non contengono trappole? Se sì, quanti e quali sono?
- Quali sono le trappole?
- La rete è live? Perché?
- La rete è bounded? Se no quali sono i posti unbounded?
- La rete è deadlock free? Se no indicate una marcatura dead end e gli scatti corrispondenti.

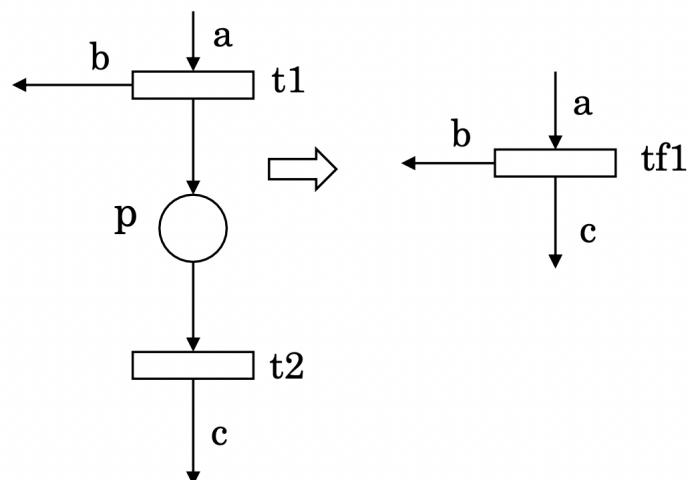
5.10 Riduzione della complessità di una rete

5.10.1 Regole di riduzione in serie

Fusione di posti

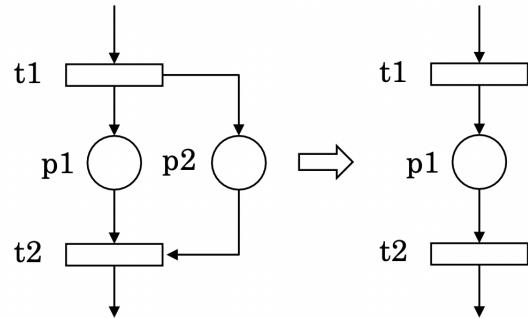


Fusione transizioni

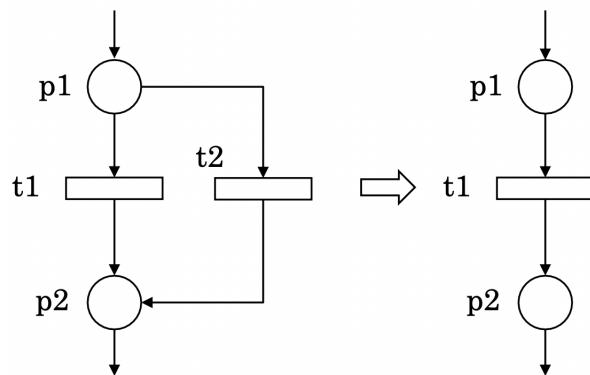


5.10.2 Regole di riduzione in parallelo

Fusione di posti

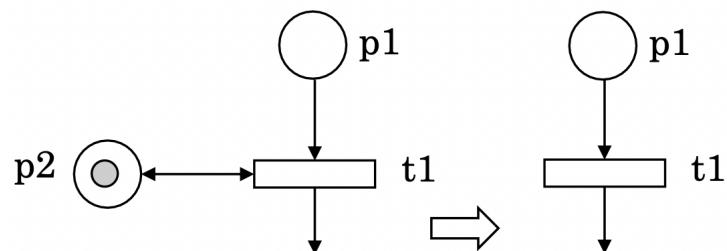


Fusione di transizioni

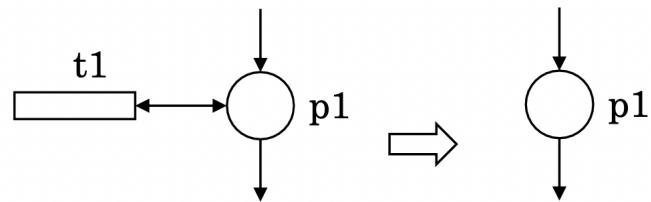


5.10.3 Riduzioni in loop

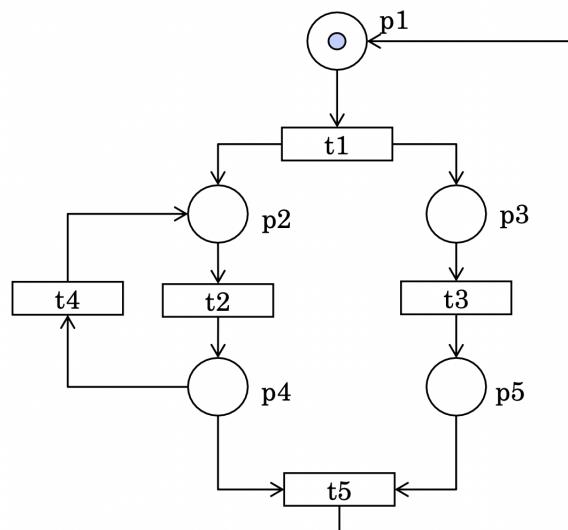
Rimozione di self-loop di posto



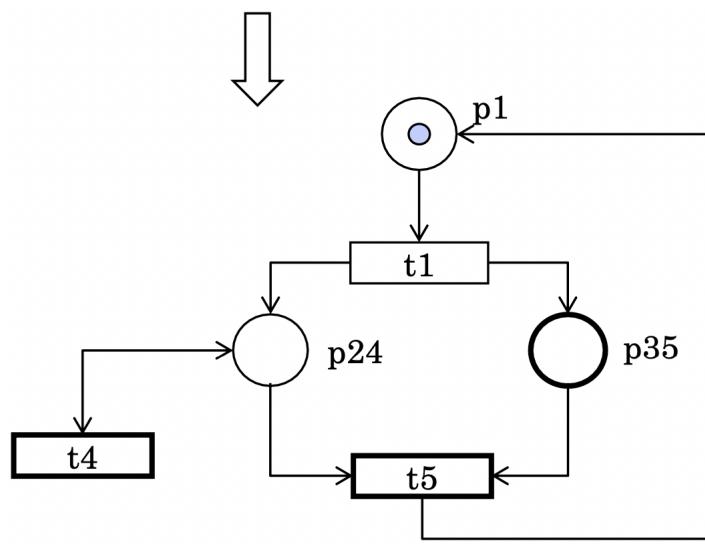
Rimozione di self-loop di transizione



5.10.4 Esempio di riduzione



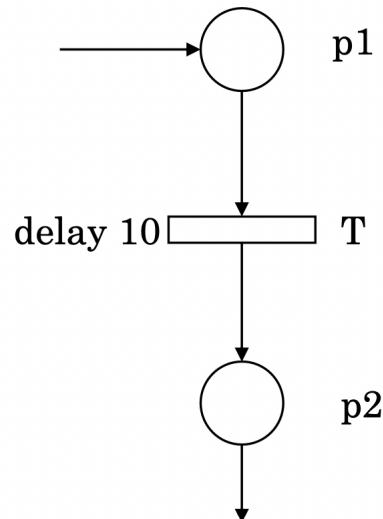
La rete, che è la stessa che abbiamo visto all'inizio della spiegazione delle reti di petri è una rete AC in p_4 e p_5 . Con le regole appena viste possiamo ridurre la rete per avere una rappresentazione più semplice:



Chapter 6

Reti temporizzate

Nelle reti temporizzate le transizioni non sono istantanee ma hanno una durata espressa con un numero intero per semplicità:

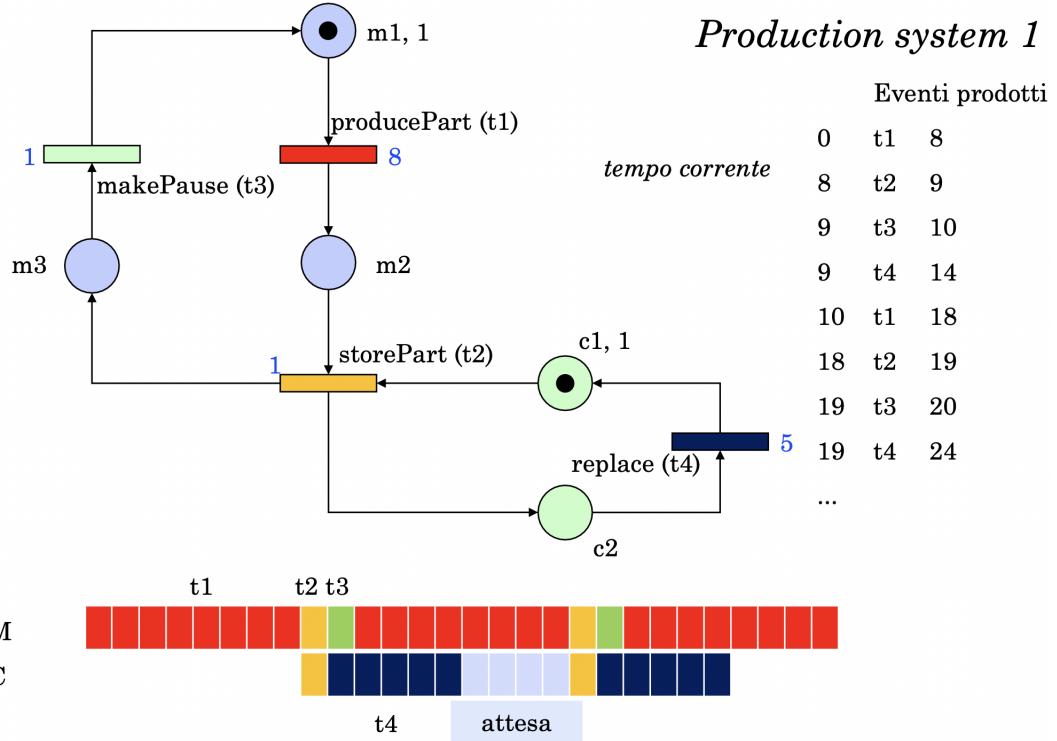


I token si muovono allo stesso modo rispetto alle reti non temporizzate ma quando la transizione scatta consuma i token dai posti in input e dopo un certo periodo di tempo produce i token in output.

6.1 Simulazione ad eventi discreti

Le reti temporizzate le utilizzeremo per modellare delle simulazioni a tempo discreto dove un evento è in sostanza il completamento di uno scatto dopo che il tempo di una transizione è terminato. Gli eventi presenti sono tenuti in una lista ordinata per tempi crescenti.

Il tempo corrente è inizialmente zero e man mano aumenta fino a raggiungere il prossimo elemento nella lista degli eventi.



Nella tabella di destra ci sono le transizioni che scattano con il relativo istante di tempo. Come si può notare dopo un certo periodo, che possiamo chiamare transitorio, ci rendiamo conto che i tempi tra due stesse transizioni sono costanti, per esempio tra due t3 passano 10 unità di tempo, tra due t4 passano 10 istanti di tempo. In realtà questo tempo che intercorre tra due transizioni è uguale per tutte le transizioni e questo intervallo di tempo si chiama **Tempo Ciclo** ed è deterministico, si può ricavare con una formula analitica. Quindi possiamo dire che il tempo ciclo del sistema produttivo preso in esempio è 10.

6.2 Calcolo del tempo ciclo

Si può calcolare attraverso 3 step

- Calcolare per ogni circuito la durata complessiva
- Per ogni ciclo si divide la durata massima per il numero di token presenti nei posti del circuito
- Il tempo ciclo è il quoziente maggiore tra quelli calcolati

Nell'esempio precedente, per ottenere il tempo ciclo dobbiamo sommare i tempi dei due cicli evidenti:

- $m1, t1, m2, t2, m3, t3 = (8 + 1 + 1)/1 = 10$
- $c1, t2, c2, t4 = 1 + 5 = 6/1 = 6$

Chapter 7

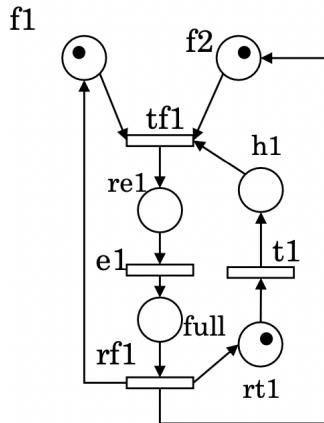
Reti Object-Oriented

Le reti viste fino ad ora sono delle reti che non hanno dati, non trasportano nessun tipo di informazione. Quelle che vediamo adesso invece sono delle reti che puntano ad oggetti per cui riescono a trasportare informazioni. Sono sempre reti di petri che però hanno delle particolarità in più. Si noti che le reti di petri ad oggetti sono più compatte rispetto alle reti tradizionali e inoltre tutto quello che possiamo fare con le reti di petri ad oggetti si può fare anche con le reti di petri tradizionali ma diventa molto più complesso.

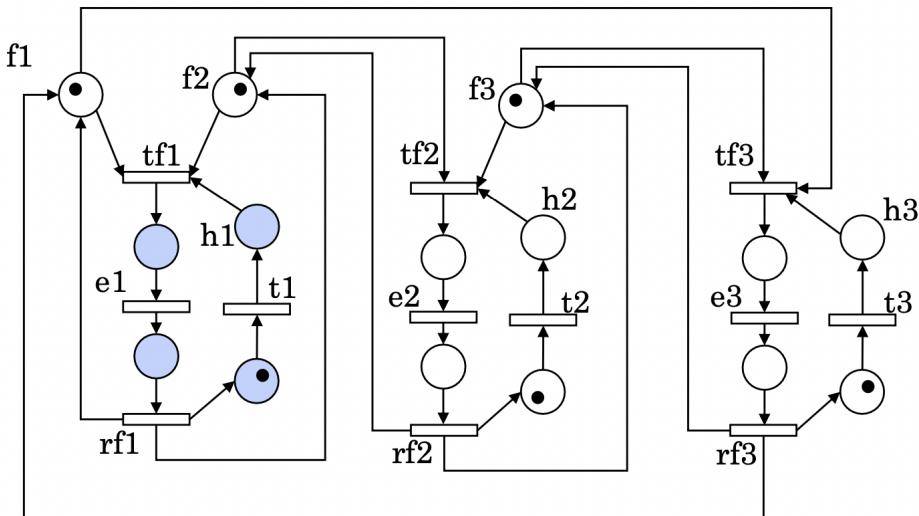
In queste reti i token non sono vuoti ma contengono dei puntatori a dati. I dati potrebbero essere rappresentati in maniera molto formali oppure attraverso dei linguaggi grafici, o almeno i parte.

Per introdurre queste reti introduciamo un problema molto famoso chiamato problema dei 5 filosofi: Ipotizziamo che vi siano 5 filosofi che pensano e ad un certo punto sono affamati e quindi vanno in una stanza dove è presente una tavola rotonda. Si dia il caso che il cibo si trova in un piatto al centro e ogni filosofo a bisogno di due forchette per poter prendere il cibo dal piatto centrale ma ci sono solo 5 forchette. Se i filosofi arrivano contemporaneamente e prendono una forchetta a ciascuno si trovano in uno stato di stallo perché ognuno ha una forchetta ma nessuno può prendere il cibo. Si può mettere il vincolo che ogni filosofo prenda le forchette due insieme in modo che non si vada in deadlock.

Vediamo una rappresentazione della rete senza gli oggetti:



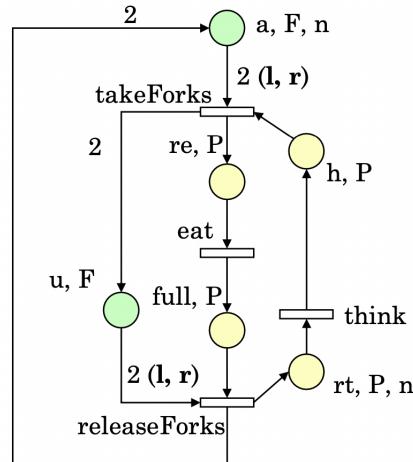
Quello che si nota è che questa rete in realtà modella il comportamento di un solo filosofo ma noi abbiamo più di un filosofo che devono adeguare il comportamento in base al comportamento degli altri. Questo vuol dire che serve una rete come quella appena vista per ogni filosofo, di seguito un esempio con 3 filosofi:



Ipotizzando di voler fare un esempio con 10 filosofi sarebbe impossibile fare una rete di petri con questa metodologia. Possiamo sfruttare le reti di reti ad oggetti per poter modellare questo problema in maniera molto compatta, con una sola rete, in cui ci sono token che non sono vuoti, che sono istanze di Fork e Philosopher.

I posti nelle reti ad oggetti devono avere delle informazioni aggiuntive:

- Nome del posto
- Tipo dei token che può contenere
- Numero di token iniziali



La prima cosa da notare è che gli archi sono pesati in modo che le transizioni scattino solo quando 2 forchette sono disponibili. I nomi degli oggetti sono i nomi del posto in cui si trovano. I nomi dei posti sono:

- a: Available
- re: Ready To Eat
- rt: Ready To Think
- full: Full after eat
- u: Unavailable

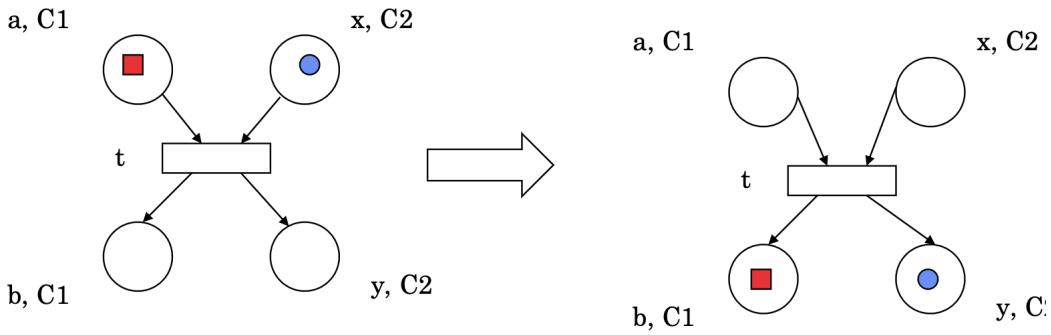
si possono anche definire delle *guardie* in corrispondenza delle transizioni per fare in modo che la transizione scatti solo se la guardia è soddisfatta, due esempi sono:

- takeForks: g: l.i == h.i and r.i == h.i % n + 1
- releaseForks: g: l.i == full.i and r.i == full.i % n + 1

che garantiscono che le forchette che ogni filosofo prende sono esattamente quelle alla sua destra e alla sua sinistra.

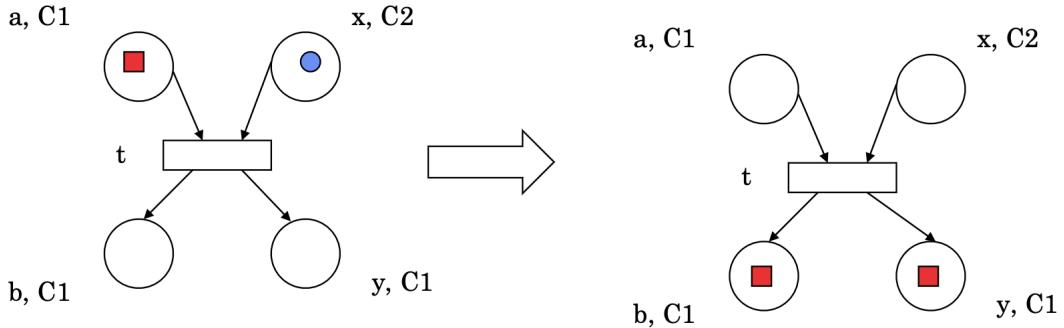
7.1 Funzionamento

Un posto in una rete ad oggetti può contenere uno o più token ma tutti dello stesso tipo e il nome di un posto include anche il nome della classe che può contenere e il numero di token iniziali

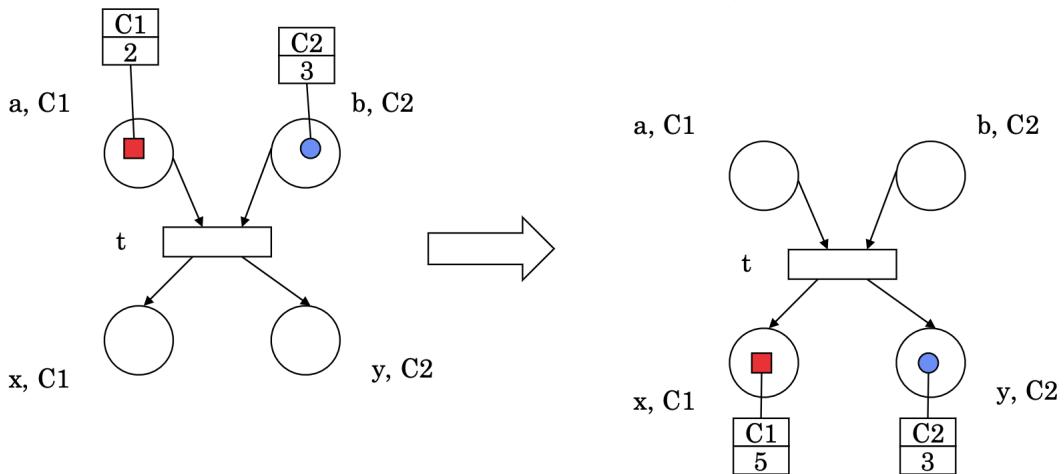


Solitamente i token di input e output da una transizione sono di tipo differente però quando gli oggetti che passano da una transizione essi vengono smistati in modo corretto nei posti corretti, come si vede dall'immagine.

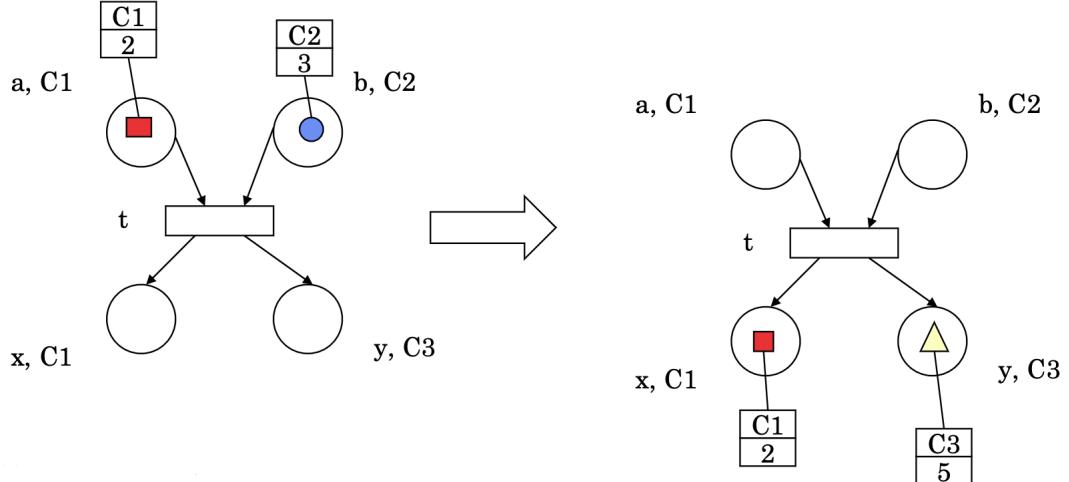
In alternativa se i posti di output sono dello stesso tipo quello che succede è che si effettua una copia dell'oggetto che i posti di output possono ospitare mentre eventuali token che non hanno alcuna corrispondenza nella rete vengono scartati:



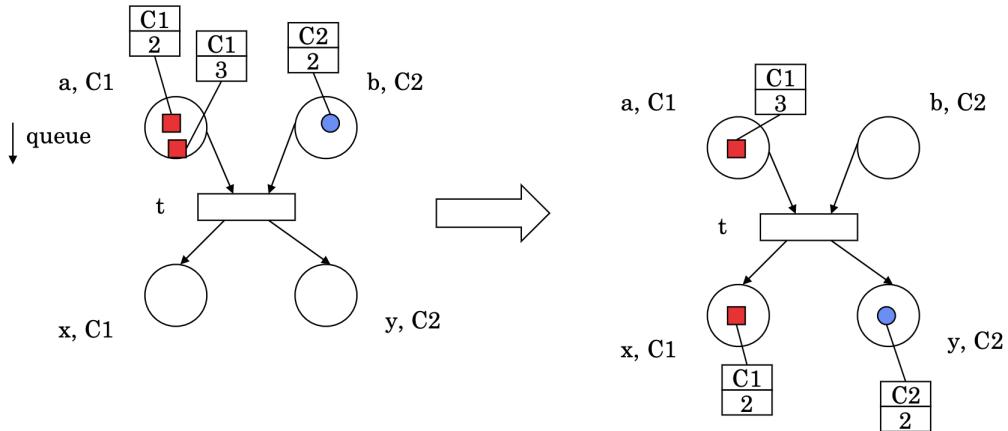
Sui token possiamo anche effettuare delle azioni nel momento in cui passano per una transizione:



L'azione in questione è: $t: a.v1 += b.v2$. L'approccio nello svolgimento di questo è di tipo procedurale che in seguito abbandoneremo a favore dell'approccio dichiarativo. Con lo stesso meccanismo è possibile generare dei token:



in cui si genera il token a triangolo con la procedura: $t: y = new\ C3\ (a.v1 + b.v2)$. Inoltre è possibile prendere dei token in input con dei differenti criteri da quello di default. Il comportamento di default dice di prendere i token che sono primi nella coda di token nel posto, questo introduce un nuovo concetto che è quello di coda che si viene a creare quando più di un token si trovano in un singolo posto:



N.B.: una transizione può avere, oltre al nome, anche un numero in seguito al nome che rappresenta la priorità della transizione rispetto alle altre transizioni attaccate allo stesso posto di input.

Chapter 8

Modelli di processi

Il formalismo utilizzato è sempre quello delle reti di petri però il significato di posti e transizioni è notevolmente esteso. I posti contengono token che si riferiscono ad entità presenti nel sistema informativo sottostante. Le transizioni rappresentano **Task** che possono essere eseguiti sia dal sistema in maniera automatica che da persone coinvolte nel processo.

I partecipanti sono divisi in ruoli e i task umani indicano il ruolo richiesto dalla persona che lo esegue. Per esplicitare vincoli ed effetti vengono utilizzati pre o post condizioni, come fatto in precedenza negli argomenti precedenti.

Esiste una suddivisione dei task sulla base degli effetti che hanno.

Esistono due tipi di scenari:

- Centralizzato
- B2B

8.1 Scenario centralizzato

Il procedimento di analisi in un contesto centralizzato è diviso in alcune parti fondamentali:

- Analisi dei requisiti
- Definizione del modello informativo
 - entità
 - attributi
 - relazioni
 - invarianti
- Definizione del processo

Vediamo subito un esempio per chiarire il contesto:

Esempio

Un’azienda tratta gli ordini immessi dai suoi clienti. Un ordine si riferisce ad un tipo di prodotto. Gli ordini hanno un importo; se è ≤ 1000 , l’ordine è accettato automaticamente, altrimenti è esaminato dall’accountMgr (ruolo interno) associato al cliente. L’account manager può accettarlo o respingerlo e il risultato è scritto nello stato dell’ordine. Poi il cliente legge l’ordine per conoscerne lo stato. Clienti, account manager e tipi di prodotto sono già registrati nel sistema informativo.

Dal testo possiamo estrapolare le tipologie di entità che dobbiamo modellare:

- Cliente
- Ordine
- Tipo (di prodotto)
- AccountMgr

in cui bisogna notare che Cliente, Tipo, AccountMgr sono di contesto perché sono già presenti nel sistema informativo mentre Ordine è l’entità che dobbiamo gestire in questo processo che si occupa proprio di gestire gli ordini. Definiamo quindi gli ordini:

- Cliente: immette ordine, legge ordine
- AccountMgr: accetta o respinge ordine (esamina ordine)
- System: accetta ordine

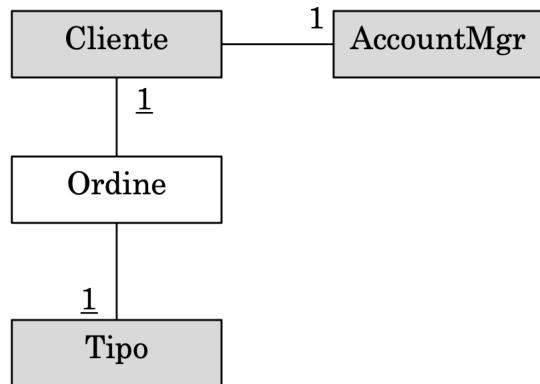
I nomi dei task sono generalmente costituiti da un verbo e un complemento oggetto.

Adesso, nell’ordine in cui li abbiamo elencati in precedenza procediamo con le tre fasi:

Analisi dei requisiti

Un’azienda tratta gli ordini immessi dai suoi clienti. Un ordine si riferisce ad un tipo di prodotto. Gli ordini hanno un importo. Un ordine è collegato necessariamente al cliente che l’ha immesso e al tipo al quale si riferisce. Un ordine è trattato dall’accountMgr del cliente. Gli ordini hanno un importo; se è ≤ 1000 , l’ordine è accettato automaticamente, altrimenti è esaminato dall’accountMgr associato al cliente.

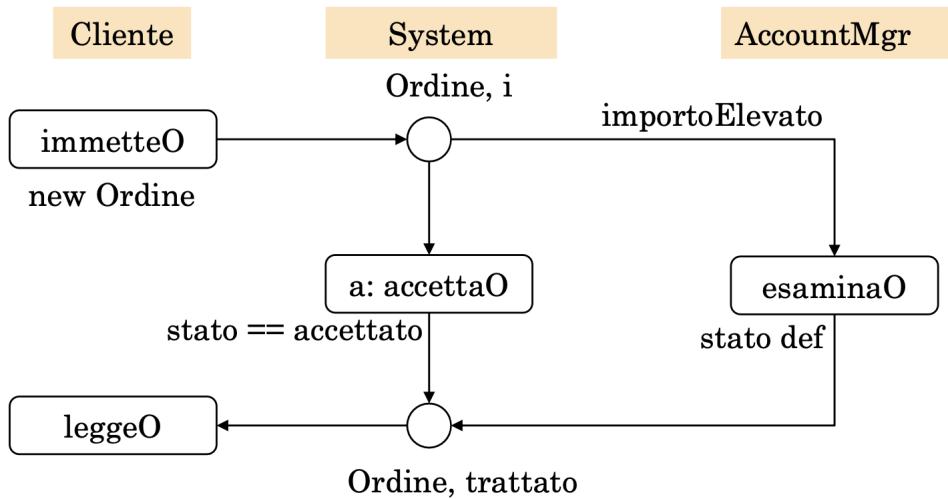
Modello informativo



Con gli attributi:

- Ordine: int importo; stato (accettato, respinto);
- boolean importoElevato = importo > 1000. // serve nel processo

Processo



Il task `immetteO` può essere eseguito da qualsiasi cliente e si noti pure che la dicitura `new Ordine` non è una dicitura procedurale ma dichiarativa. Quando `immetteO` si conclude il risultato è che nel sistema è presente un nuovo Ordine e lo si capisce dalla presenza della dicitura procedurale `new Ordine`.

Il posto `Ordine, i` sta ad indicare che in quel posto entrano degli oggetti di tipo `Ordine` che si trovano nello stato `i` che sta per *iniziale*.

Il task `esaminaO` ha come risultato che lo stato dell'ordine viene definito. Per cui si ha che l'ordine entra in quel task con `stato == null` ed esce con uno stato definito

tra *Accettato* e *Rifiutato*. Si noti che l'accountMgr che effettua l'analisi dell'ordine e che decide se rifiutare o meno l'ordine è l'accountMgr assegnato a quel cliente. Un solo accountMgr è assegnato ad un cliente mentre un accountMgr può avere più clienti da seguire.

Nel caso in cui *importoElevato* sia false si esegue *a: accettaO()* che è una procedura automatica fatta dal sistema che ha come risultato *stato == accettato* quindi dopo questo task si ha che l'ordine ha uno stato definito pari ad *accettato*.

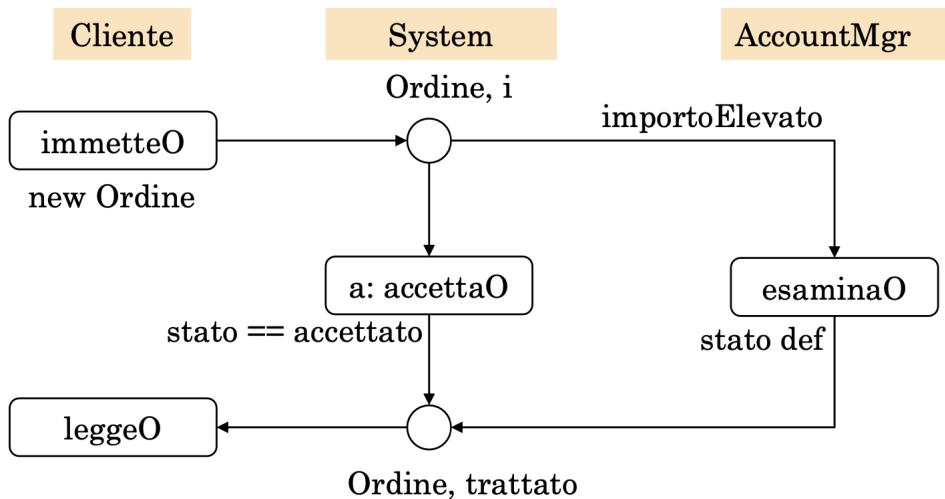
Il task *leggeO* può essere eseguito da un Cliente ma non da uno qualsiasi ma solo da quello che lo ha creato.

8.2 Tipi di task: entry task, exit task

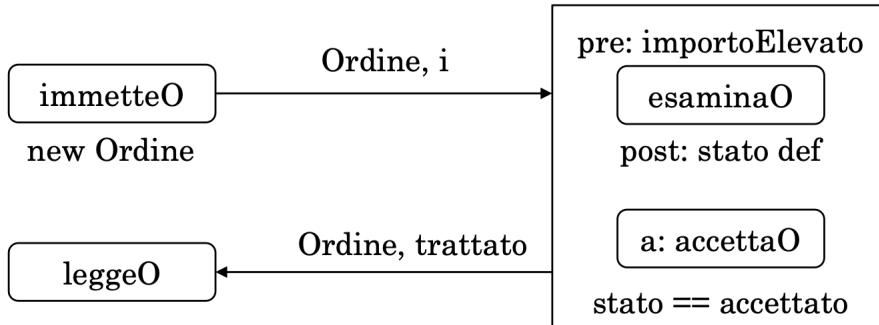
- **Entry Task:** Un entry task non ha input e ha il compito di immettere una nuova entità
- **Exit task:** Non ha un output e ha il compito di togliere una entità dal processo

8.3 Scelte strutturate e non strutturate

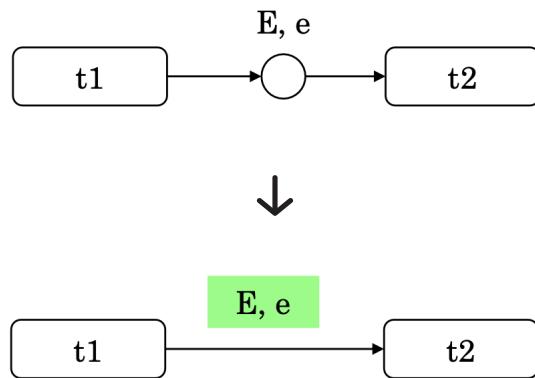
Abbiamo già visto la scelta non strutturata che per semplicità riportiamo qua sotto:



ma vi è anche una scelta strutturata che consiste nel racchiudere in blocchi dei task contestualizzati e ordinati con delle precedenze:

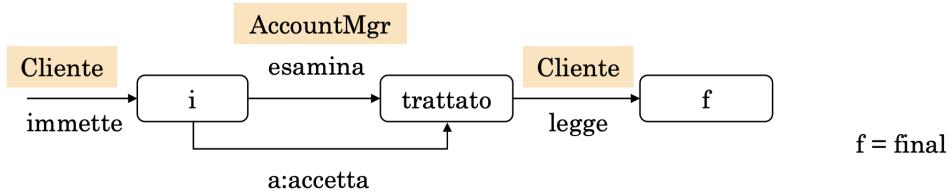


Quello che si nota è che i posti *Ordine, i* e *Ordine, trattato* sono stati inglobati nei link che collegano i task con il box, questo processo si chiama assorbimento:



8.4 Life cycle degli ordini

Il ciclo di vita degli ordini è possibile gestirlo attraverso dei modelli di stato. Esso mostra come si passa dallo stato iniziale allo stato trattato:



Per implementare un processo si utilizza un elemento software chiamato Process Engine. Questo process engine interagisce con le entità software che implementano i task sia automatici che umani.

I task umani sono gestite attraverso liste di task separate per ogni utente chiamate **working list**. Ogni voce della working list corrisponde con un task che un utente può fare o deve fare a seconda se il task è opzionale o obbligatorio. Al click su una entry viene aperta una web activity con cui l'utente interagisce per poter effettuare delle scelte e portare a termine un task. Vediamo subito degli esempi di complessità crescente:

Esempio: Gestione Proposte

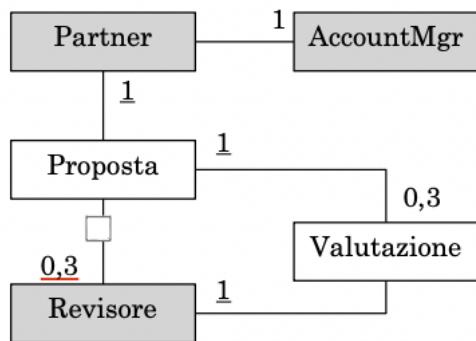
Un'organizzazione tratta le proposte inserite dai partner. Una proposta è esaminata dall'account manager relativo al partner: può accettarla, respingerla o sottoporla a 3 revisori che sceglie. I revisori forniscono una valutazione nel range 1..10.

In modo automatico, se la media è ≥ 6 la proposta è accettata, altrimenti è respinta. L'esito della proposta (accettata o respinta) è scritto nello stato che il partner può poi leggere. Nel sistema informativo sono registrati partner, revisori e accountMgr (con i legami con i partner).

Ruoli: Partner, AccountMgr, Revisore.

Entità: Proposta, Valutazione.

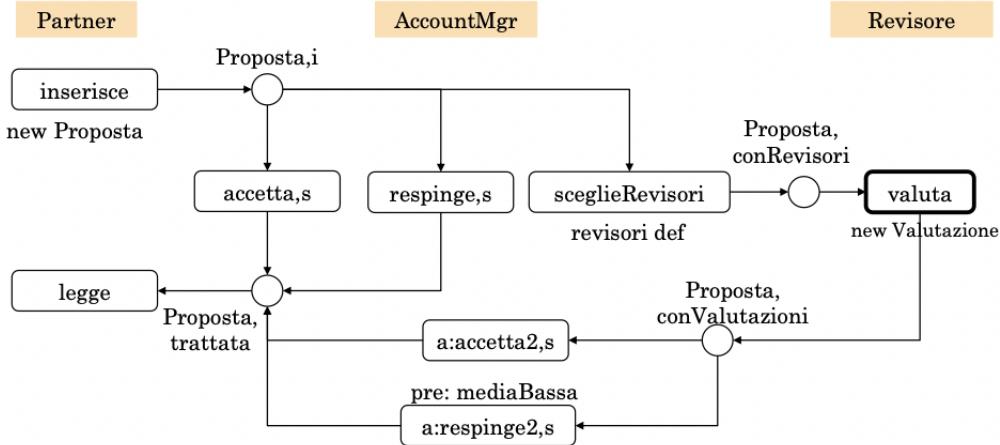
La prima considerazione va fatta sui partner, revisori e accountMgr che sono dei ruoli che già sono presenti nel sistema, magari gestiti da altri processi pre esistenti. Il modello informativo è quello che segue:



La cardinalità $0..3$ che è anche sottolineata non rappresenta un range ma indica che i revisori sono 3 oppure nessuno. Non possono essere 1 o 2. Stessa cosa vale per le *Valutazione*. Gli attributi del modello informativo sono quelli che seguono:

- **Proposta:** stato(accettata, rifiutata) inizialmente null, boolean mediaBassa = valutazione.avverage(val) < 6
- **Valutazione:** int val (1..10)

Il processo, come anche i requisiti, somigliano all'esempio fatto in precedenza per la gestione ordini:



Nel posto *Proposta, i* abbiamo una vera free choice perché il non è presente una condizione per cui vuol dire che quella è una scelta umana per cui l'account manager ha la possibilità di scegliere cosa fare. A seconda della scelta dell'AccountMgr si ha una situazione differente in particolare nell'ultimo caso a destra vengono scelti 3 revisori che devono effettuare la revisione eseguendo il task *valuta*. Quel task deve essere effettuato da 3 revisori differenti per cui è chiamato *Task Multi-Performer*. Il posto subito dopo al task *valuta* esplicita il fatto che la proposta sia con le valutazioni. Sulla base della valutazione dei tre revisori la proposta viene accettata o meno grazie alla condizione successiva che lavora sull'attributo booleano *mediaBassa*.

Una precisazione sul *Task Multi-Performer* è che il process engine prima di proseguire con il data flow aspetta che tutti i performer finiscano di eseguire il task per cui in questo caso si va avanti solo dopo che tutti e 3 i revisori effettuano il task e non prima.

Come vediamo nella condizione i task iniziano con *a:* e questa cosa cambia una piccola sfumatura. Con *a:* la scelta è automatica, sulla base della condizione, e il task è automatico mentre senza *a:* la scelta rimane automatica, sempre sulla base della condizione, mentre il task è umano, non più automatico.

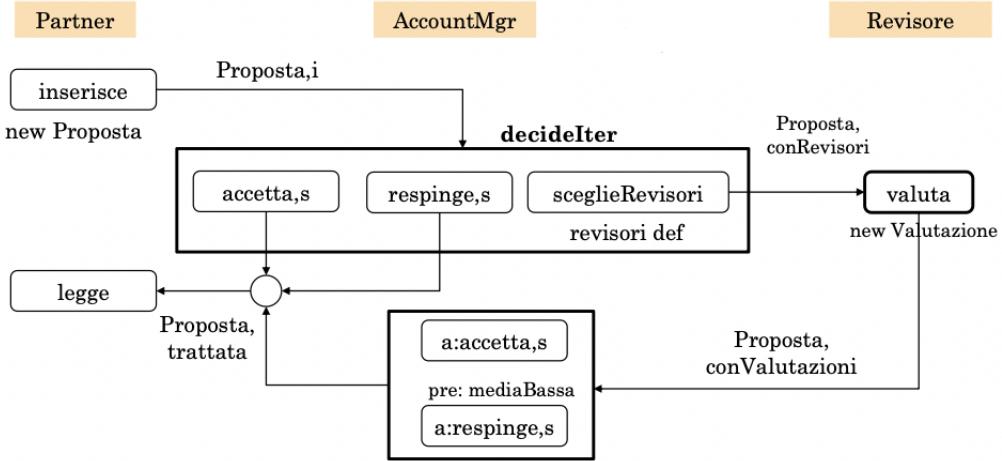
Modifica dello stato

Alcuni task nel processo sono espressi con una *s* nel nome, come ad esempio: *accetta, s*, *respinge, s*. Questa è solo una modalità di rappresentazione della modifica dello stato. In pratica invece di scrivere la post condizione *stato == accettata* con questa dicitura, *s* si sta ad indicare che lo stato viene modificato nello stato più lessicalmente simile al nome del task, nel nostro caso *accetta* e *respinge*. È utilizzato solo per semplificare la dicitura, potrebbe non piacere.

Task Multi-Performer

Scelta strutturata

Si può modificare il processo assorbendo i posti nei link ed effettuando la scelta in maniera strutturata:



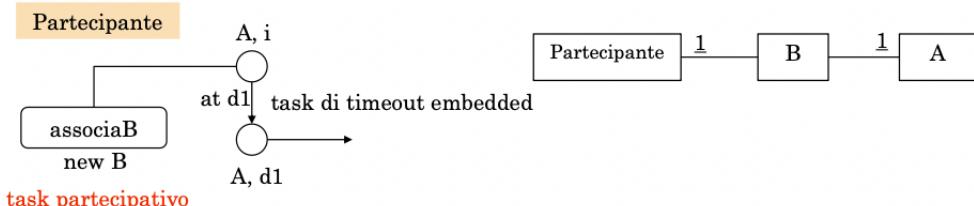
Notiamo che i posti *Proposta, i*, *Proposta, conRevisori*, *Proposta, conValutazioni* sono stati inglobati nei link e la scelta è stata trasformata in una scelta strutturata di nome *decideIter*.

8.5 Tipi di task

Esistono diversi tipi di task e li vediamo di seguito con le descrizioni opportune.

8.5.1 Task Partecipativo

Certe volte ci troviamo a dover modellare delle entità che devono essere agganciate, entro una scadenza, ad altre entità. Un esempio potrebbe essere una conferenza che permette a chi presenta qualcosa a quella conferenza di mandare gli articoli per quella conferenza. Abbiamo due tipologie di entità quindi, in generale A e B e nel nostro esempio Articoli e Conferenza.

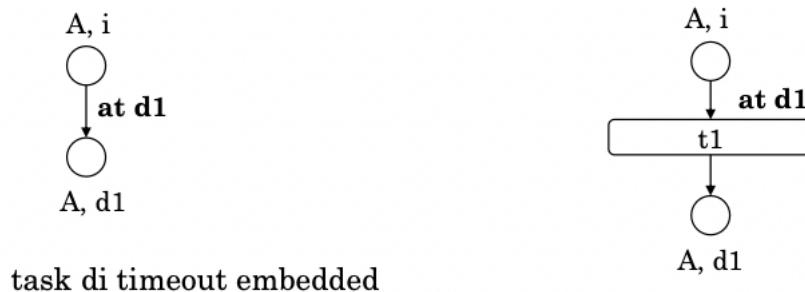


L'entità A si trova nello stato i per un certo lasso temporaneo, fino ad una data d1 per esempio. Mentre A si trova in questo stato è possibile agganciare entità di tipo B

all'entità A. Per esprimere la scadenza è stato sufficiente scrivere *at d1* per rappresentare che quel link viene attraversato al raggiungimento di d1. Il link tra *associaB* e A, i è senza freccia perché è un link particolare per il task partecipativo che sta ad indicare questa particolare rappresentazione. Quindi quel link particolare indica che i B generati vengono attaccati ad A secondo il modello informativo che vediamo a destra. Una volta che A passa nel posto A, d1 sarà possibile accedere alle entità B passando da A con una navigazione che rispecchia il modello informativo a destra. Il task *associaB* viene eseguito da un partecipante e per i nostri studi il task viene eseguito sempre da un umano perché deve essere fatta una scelta per poterlo eseguire, nel nostro esempio deve essere scelta la conferenza a cui sottomettere l'articolo.

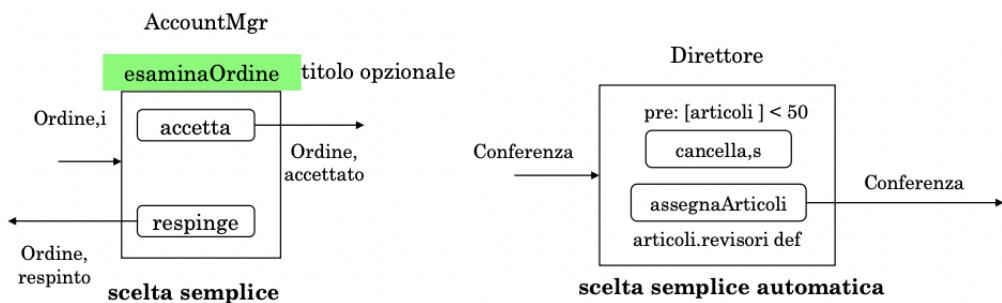
8.5.2 Task di timeout

Come abbiamo già visto prima un task di timeout è un task che scatta in maniera automatica al raggiungimento di un timeout, per esempio *at d1*. Considerando A: Date d1;



8.5.3 Scelta semplice

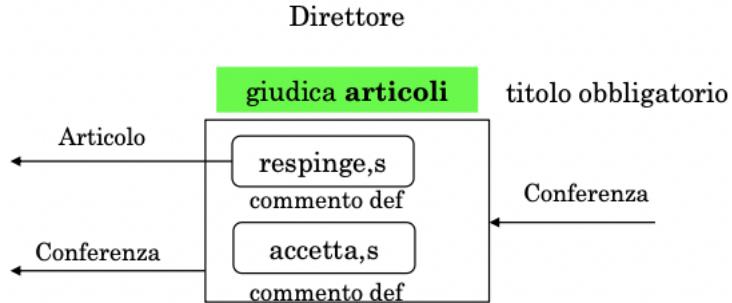
Vista con una scatola racchiude i task che possono essere eseguiti in maniera esclusiva e esprime anche tutti i suoi output. Può anche essere automatica esprimendo una condizione.



La scelta automatica rappresentata presenta dei task umani perché essi non sono preceduti da *a*:

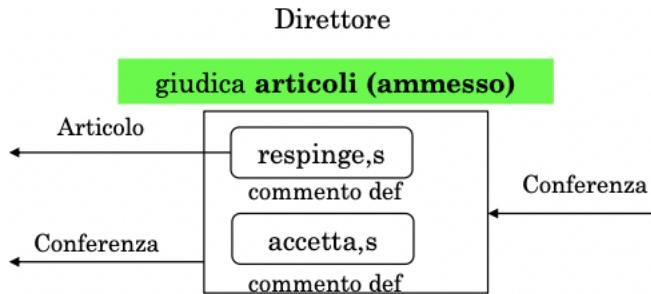
8.5.4 Scelta composta

La differenza sostanziale con la scelta semplice è che la scelta lavora su delle entità che sono diverse da quelle che riceve in input.



Come si vede quello che la scelta riceve in input è una conferenza ma poi i task dentro la scelta comporta agiscono, sempre in mutua esclusione, sugli articoli sottomessi per la conferenza. Se ci fossero 50 articoli bisognerebbe effettuare 50 volte la condizione. L'output può essere dell'intero blocco oppure del singolo task. Per esempio l'output del task respingi potrebbe essere utilizzato per mandare una comunicazione all'autore dell'articolo mentre l'output del blocco potrebbe essere usato per andare avanti con il processo.

8.5.5 Scelta composta con restrizioni



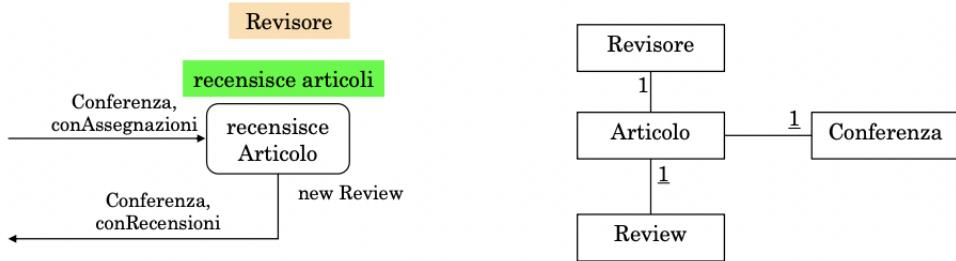
Come si vede nella parte evidenziata in verde tra parentesi abbiamo una restrizione, in questo caso (*ammesso*) cioè nella condizione verranno elaborati solo gli articoli ammessi.

8.5.6 Test composto associativo



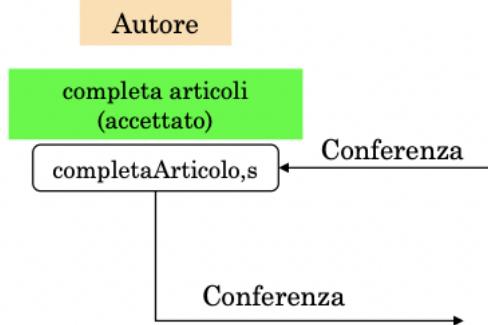
È un task che prende come input una conferenza, quindi un contenitore, e il task lavora sul singolo articolo e lo si capisce anche dal nome del task che è al singolare però la parte evidenziata in verde ci fa capire che quel task fa parte di una elaborazione più generale di articoli della conferenza.

8.5.7 Task composto generativo



Come il precedente prende in input un contenitore, nel nostro esempio Conferenza, ma rappresenta un task che viene eseguito sul singolo articolo. Si chiama generativo perché alla fine del task il risultato nel sistema è la generazione di una entità *Review* però il task composto, *recensisce Articolo*, termina solo quando tutti gli articoli sono stati recensiti.

8.5.8 Task composto con restrizione

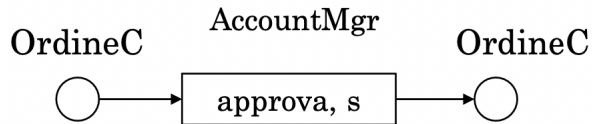


Come per i precedenti lavora sui singoli articoli ma il task si ritiene completato solo quando tutti gli articoli sono stati trattati. La particolarità è che gli articoli trattati da questo task sono limitati dalla restrizione che vediamo tra parentesi, in questo caso *accettato* per cui tutti gli articoli accettati verranno trattati da questo task mentre quelli non accettati verranno scartati da questo task.

8.5.9 Effetti dei task e dataflow

In generale i task mediate le post condizioni producono effetti sul **sistema informativo**.

Task passante modificativo



OrdineC = OrdineCliente

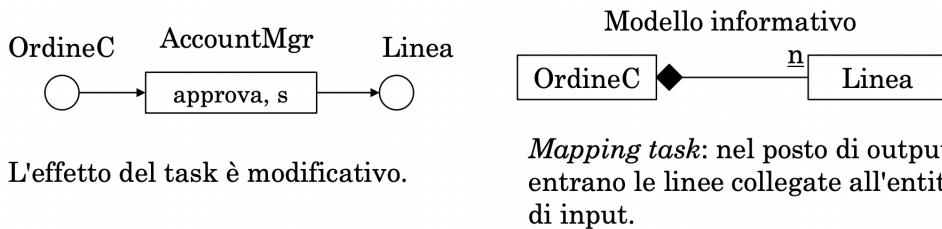
L'ordine che entra ne esce modificato, per questo parliamo di task modificativo.

Task di mapping generativo



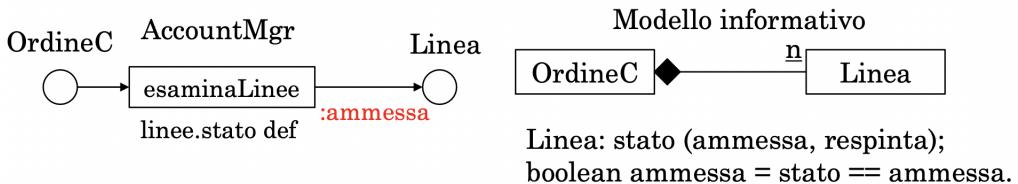
Entra una richiesta che non esce poiché essa viene trasformata, mappata, nella richiesta di output. Inoltre, avendo generato la *RichiestaP* il task prende il nome di task generativo.

Task modificativo di mapping



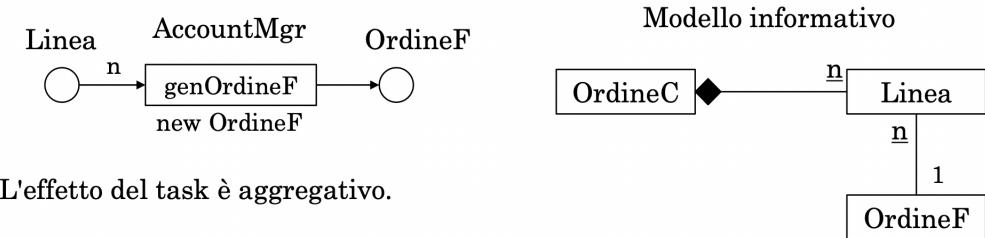
L'effetto del task è che entra un *OrdineO* e il suo output sono tutte le linee di ordine contenute nell'ordine iniziale e in più l'ordine ha cambiato stato

Task modificativo di mapping con restrizione



È esattamente come il precedente ma si trova una restrizione sullo stato della linea. Per cui solo le linee con stato *ammessa* vengono spostate nel posto *Linea*.

Task aggregativo



Prende in input un certo numero di linee e ha come output un oggetto ordine che naturalmente conterrà le linee che sono state precedentemente prese in input dal task. Per questo motivo è un task aggregativo.

Task Riduttore

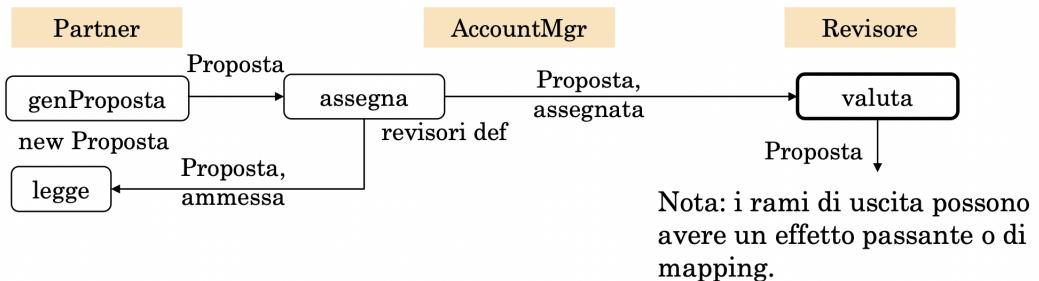


In sostanza si riduce la granularità degli oggetti in ingresso. La differenza con il precedente è che non si genera un oggetto proposta. Si può anche esplicitare un numero n da raggiungere prima di ridurre



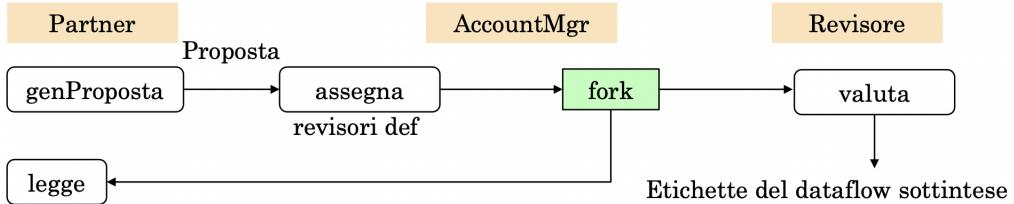
La differenza sostanziale con il task aggregatore è che in pratica l'aggregazione avviene senza alcun collegamento. Il task riduttore ha anche l'effetto di sincronizzare.

Fork Task



Il task *assegna* che rappresenta una transizione è una fork che ha come post condizione che i revisori siano definiti e l'output di quella transizione arriva a *valuta* e *legge*. Il fork in questo caso è un task manuale ma non è detto che sia sempre così, è possibile anche optare per un fork automatico:

Si può usare anche un task fork automatico

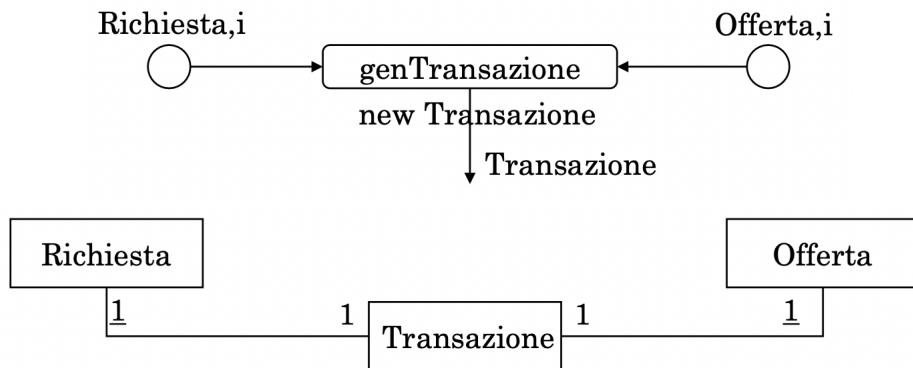


Join Task

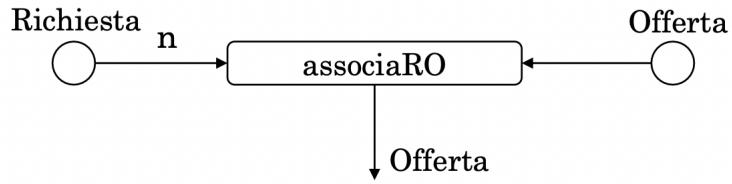
Il Join ha 2 posti di input e uno di output e ha 3 effetti differenti:

- Aggregativo
- Associativo
- Di sincronizzazione

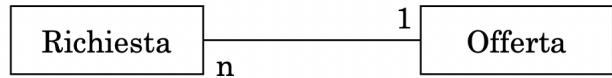
Il ramo in uscita dal Join può avere un effetto passante o di mapping.



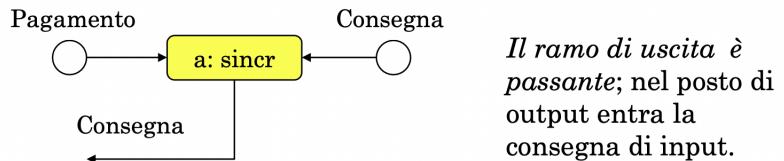
Come abbiamo già visto in precedenza con le reti di petri è possibile avere delle transizioni, in questo caso chiamati task, che formano delle Fork-Join che non sono altro che l'unione dei due costrutti che abbiamo appena studiato. Il join visto in precedenza ha un effetto aggregativo, quello che segue ha un effetto associativo:



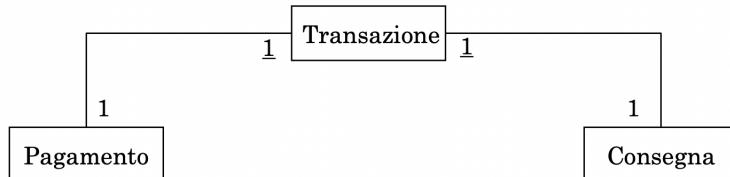
`associaRO: post: richieste.offerta == offerta.`



Inoltre vi è anche un esempio del Join con effetto di sincronizzazione:



`sincr: pre: pagamento.transazione == consegna.transazione.`



8.6 B2B Systems

Nei sistemi B2B vi sono 3 modelli di cui 2 già sono noti perché li abbiamo usati precedentemente

- Collaborazioni con relativi modelli informativi
- Modello informativo del processo considerato
- Modello del processo

Un processo B2B è costituito da un certo numero di organizzazioni che cooperano per raggiungere uno scopo comune. La cooperazione avviene lo scambio di messaggi con protocoli pre definiti. Per trattare questo scambio di messaggi dobbiamo introdurre il modello di collaborazione che sono modelli binari e devono essere definiti prima dell'implementazione dei processi di business.

8.6.1 Assunzioni

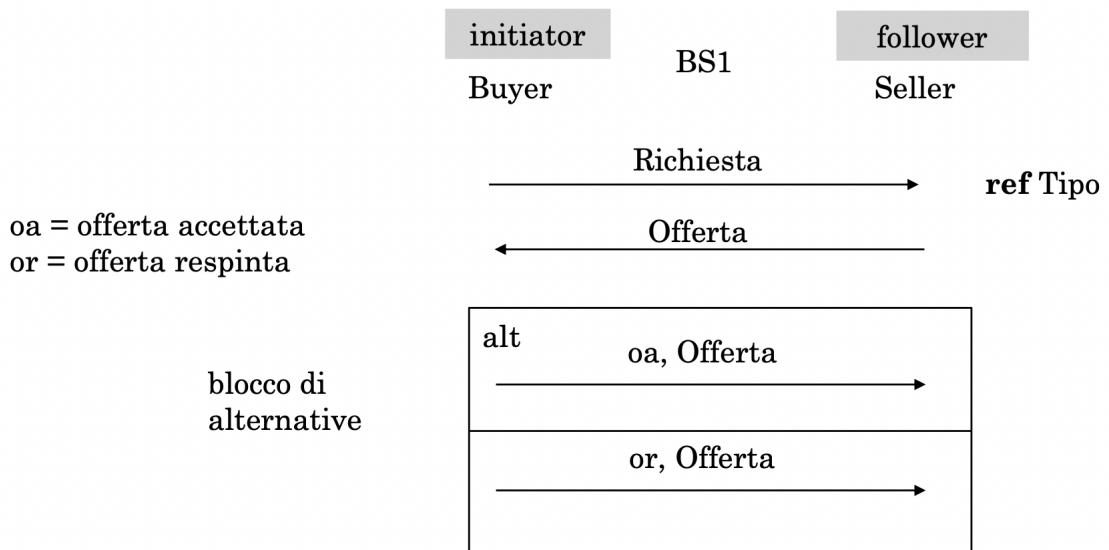
I messaggi sono fortemente correlati all'entità di business. Possiamo assumere che un messaggio di output ha una sua entità nell'organizzazione che la invia e che potrebbe essere come input di un ricevente che usa quel dato per modificare una propria entità. Inoltre le organizzazioni che hanno sistemi totalmente differenti possono utilizzare delle entità globali.

Si assume inoltre che il mapping tra messaggio e business entity sia automatico

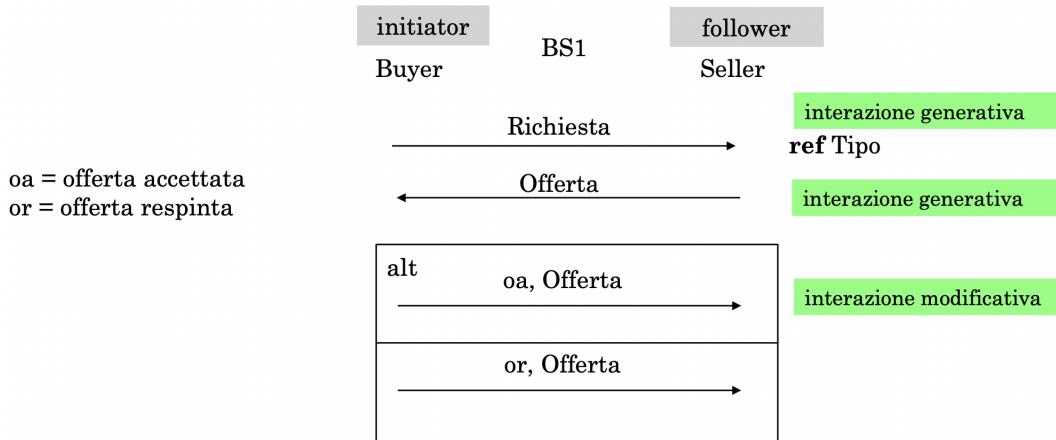
8.6.2 Modello di collaborazione

È un modello che definisce il modo in cui due organizzazioni devono interagire in modo da raggiungere un obiettivo comune. Vediamo subito un esempio di collaborazione tra un venditore e un acquirente:

Il buyer manda al seller una richiesta di offerta relativa ad un tipo di prodotto offerto dal seller. Il seller risponde con un'offerta. Il buyer può accettare o rifiutare l'offerta (comunicando la decisione al seller). I tipi di prodotti sono noti al buyer.

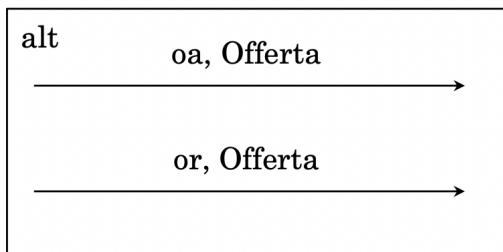


Dove ci sono diversi elementi da analizzare. Vediamo che tipo di interazioni sono in gioco in questo schema:

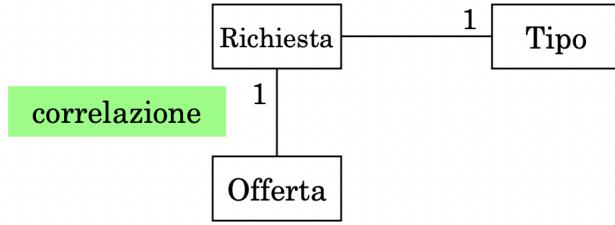


- **Interazioni generative:** Trasportano nuove entità con tanto di attributi di collaborazione mediante dei messaggi. Un messaggio include un header, che racchiude le informazioni del mittente e del destinatario, e un payload che contiene gli attributi dell'entità. Il nome dell'interazione, come Richiesta, è il nome del tipo dell'entità trasportata. Chi riceve il messaggio genera una nuova entità grazie agli attributi contenuti nel payload del messaggio. L'interazione iniziale di una collaborazione è sempre generativa.
- **Interazione modificativa:** Riguarda l'entità già trasmessa in precedenza e contiene delle modifiche agli attributi che il mittente vuole rendere note al destinatario. Il primo nome è il nome dell'interazione, spesso una sigla, il secondo nome è il nome dell'entità modificata.

Un blocco di alternativa può anche avere una forma "compatta":

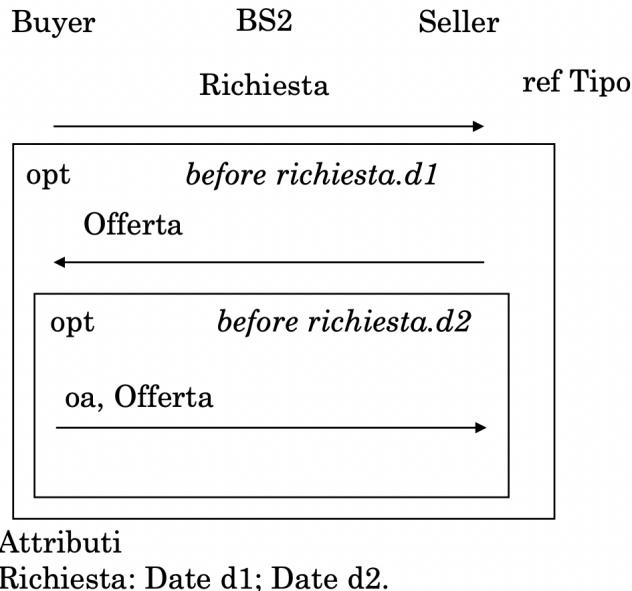


Associato al modello di collaborazione, per mettere in relazione le informazioni che vengono trasmesse dobbiamo creare un modello informativo:



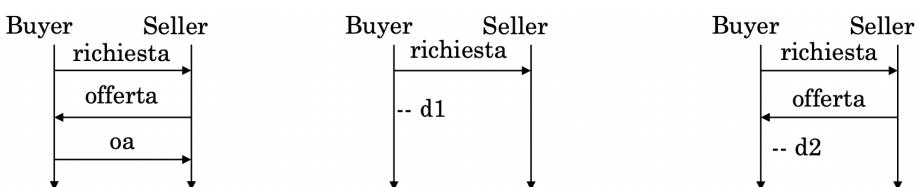
Il motivo per cui è presente una correlazione è perché l'offerta è l'effetto della richiesta e questo comporta che in un'offerta è presente l'id univoco della richiesta.

Trattiamo una variante del modello di collaborazione visto in precedenza: La richiesta include due deadline, d1 and d2. L'offerta va inviata entro d1 e l'accettazione entro d2, altrimenti la collaborazione si considera conclusa. Se l'accettazione non è inviata nei termini stabiliti l'offerta si considera rifiutata.



Attributi
Richiesta: Date d1; Date d2.

Con i blocchi *opt* indichiamo dei blocchi che possono essere saltati e ci permettono di soddisfare i nuovi requisiti. In generale quello che succede è che se siamo ancora in tempo per mandare l'offerta allora avremo la possibilità di ricevere un'Offerta con lo stato modificato, oa, in risposta ma solo se siamo entro la seconda deadline d2. Le casistiche possibili sono 3 distinte:



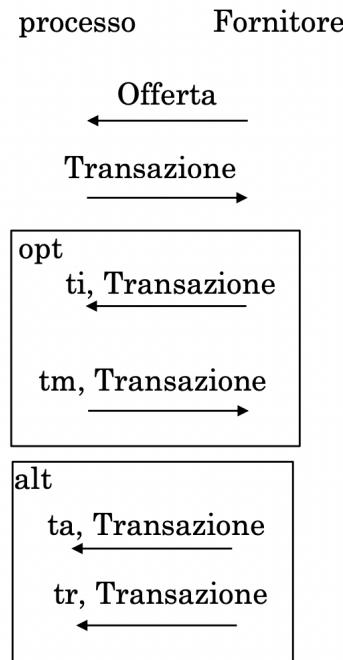
I blocchi *opt* e *alt* sono in realtà dei frammenti di flusso UML.

8.6.3 Blocco Opt e Loop

Possono essere di tre tipi differenti:

- event-driven
- time-driven
- data-driven

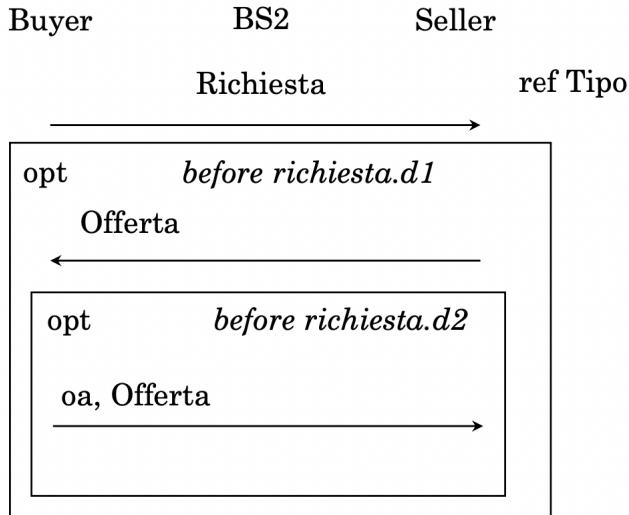
Opt event-driven



L'evento è il modo in cui la transazione viene generata, con quale stato.

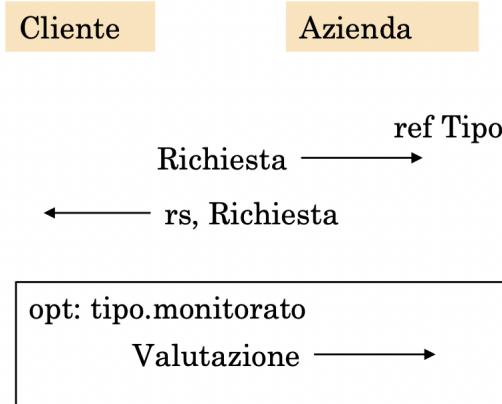
Opt time-driven

Quello che abbiamo visto in precedenza:



Opt data-driven

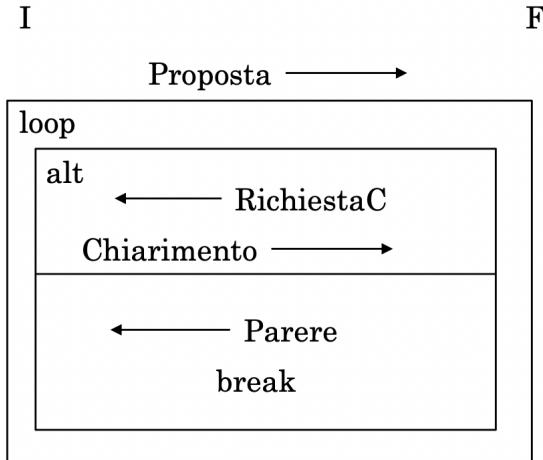
Sulla base di un dato già presente nel diagramma di comunicazione è possibile fare un opt:



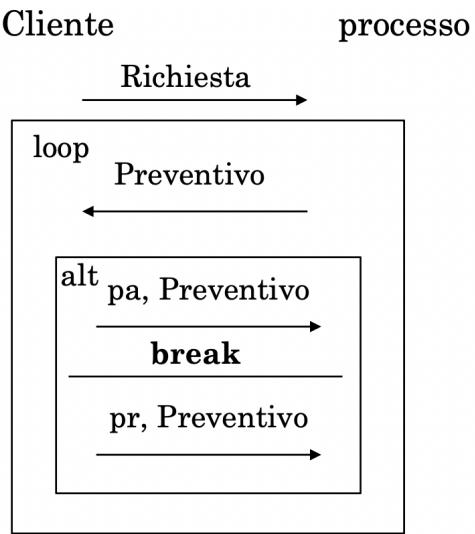
Attributi
Tipo: boolean monitorato.

Loop

Il blocco loop permette di eseguire tante volte un blocco. Come ogni blocco ha il suo nome nell'angolo in alto a sinistra:



Da notare che per uscire dal gruppo quello che deve succedere è che si deve passare dal *break* nella seconda parte dell'*alt*. Il *break*, esattamente come il *continua*, funziona come nei linguaggi di programmazione. Vediamo un altro loop:



In questo caso abbiamo una prima interazione fuori dal blocco *alt*. La differenza tra i due è che il primo viene eseguito solo se il messaggio che riceviamo è una *RichiestaC* oppure un *Parere* altrimenti non viene eseguito mentre nel secondo loop esso viene eseguito sempre perché è presente l'interazione *Preventivo* che viene eseguite indipendentemente dall'alternativa.

Il loop ha un operatore *par* che indica che l'esecuzione del loop è concorrente. Nell'esempio che segue, scritto in forma testuale, quello che succede è che il loop si ferma una volta che riceve un Progetto con lo stato *pa* per ogni richiesta di chiarimento.

```

Studio      processo
-> Progetto
loop par
alt
  <- RichiestaC
  -> rs, RichiestaC
---
  <- pa, Progetto
  break

```

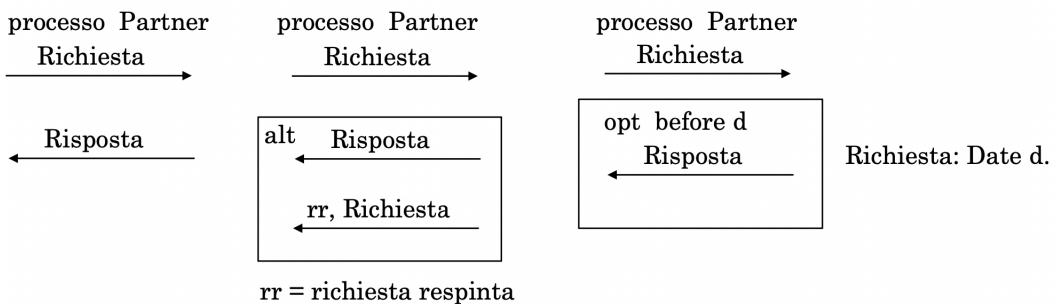
rs = richiesta soddisfatta
pa = progetto accettato

8.6.4 Task d'interazione

Vediamo i task d'interazione tramite degli esempi: Nel primo esempio il processo manda una richiesta al partner che:

- risponde con una risposta
- risponde con una risposta o dichiara respinta la richiesta
- può rispondere con una risposta entro la scadenza d della richiesta (quindi la risposta è opzionale)

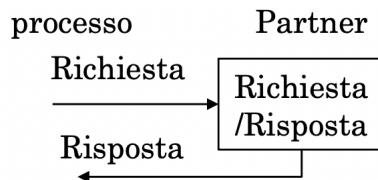
possiamo descrivere queste 3 casistiche con i diagrammi di collaborazione:



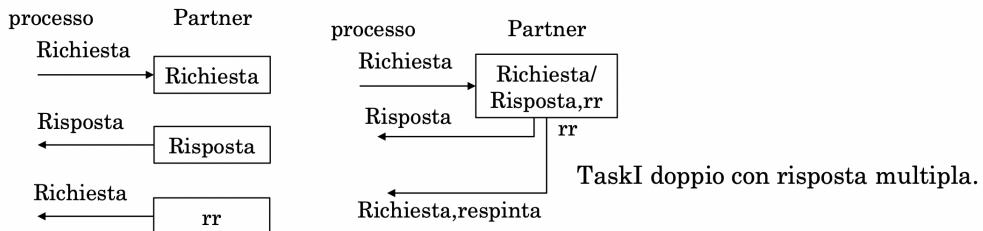
Nel processo nella swimlane del Partner troveremo un task di input e uno di output:



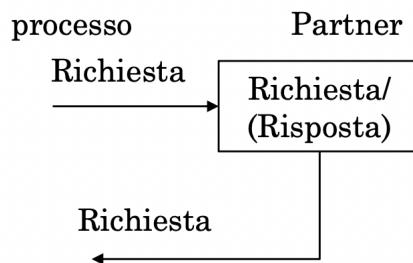
Il task di interazione effettua una serie di operazioni in maniera del tutto automatica, per esempio il task che accetta la richiesta, che è un task di input, in maniera implicita effettua la creazione di una richiesta da immettere nel sistema ed è per questo che non è esplicitata la post condizione *new Richiesta*. In modo alternativo possiamo anche mettere un task doppio invece che due singoli:



Le risposte da un task possono anche essere più di una:

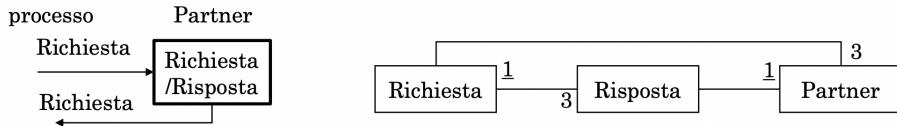


Inoltre le risposte potrebbero essere opzionali, non obbligatorio, magari sulla base di certi input non bisogna generare delle risposte:



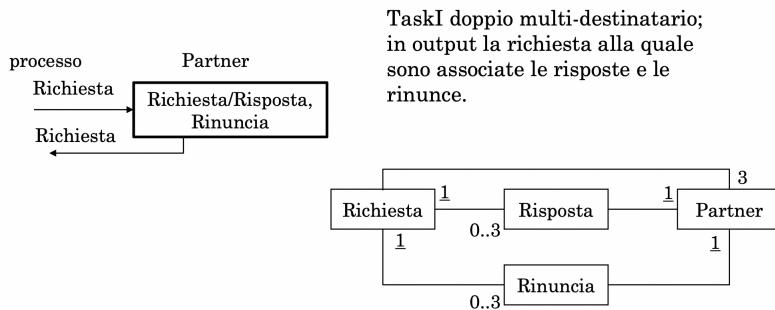
Come si vede essendo la risposta facoltativa, messa tra parentesi tonde, il ramo di uscita presenta la risposta invece che la richiesta questo perché poi, essendo la risposta collegata alla richiesta, potrà essere recuperata in seguito.

C'è anche la possibilità di interagire con più partner, tramite dei task multi-destinatario:



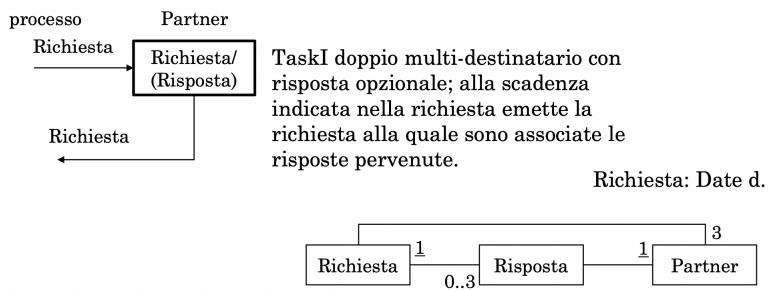
In questo caso la risposta viene mandata a 3 partner, come si vede dalla cardinalità. Nel ramo di uscita del task *Richiesta/Risposta* è la richiesta per lo stesso motivo di prima, perché la richiesta è associata alla risposta.

Il partner potrebbe rispondere alla Richiesta con una Risposta oppure con una Rinuncia:



Nel caso di rinuncia si può elaborare la rinuncia a posteriori facendo riferimento alla richiesta in output che è collegata alla risposta vera e propria, Richiesta o Rinuncia.

Nel caso di risposte multi-destinatario è possibile che le risposte siano facoltative:



8.6.5 Uso dei riduttori

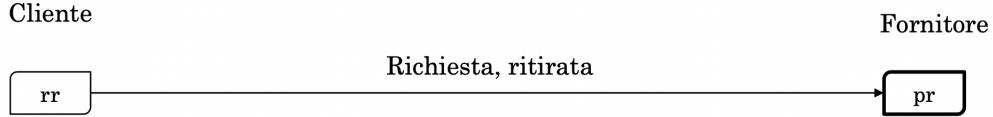
Nell'esempio che segue il riduttore è di tipo time-driven che allo stendere della richiesta racchiude tutte le risposta nella richiesta:



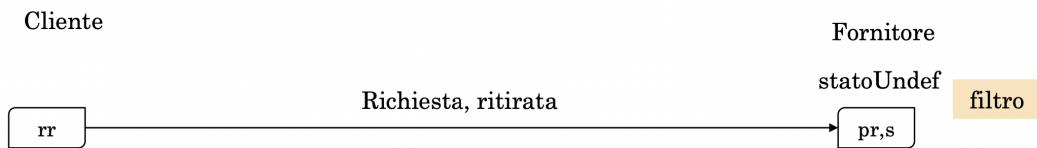
riduttore: $t = \text{richiesta.d.}$

R , t indica *Riduttore*, *time-driven* e *riduttore*: $t = richiesta.d$ imposta la scadenza del riduttore.

È possibile avere un posto di mapping in modo da trasformare un'entità in un'altra:



In pratica al fornitore non interessa la richiesta respinta ma il preventivo respinto per cui il task pr che si occupa di effettuare questo mapping. Si potrebbe anche applicare un filtro al Fornitore per far sì che vengano scartate le richieste respinte:



Chapter 9

SOA - Service Oriented Architecture

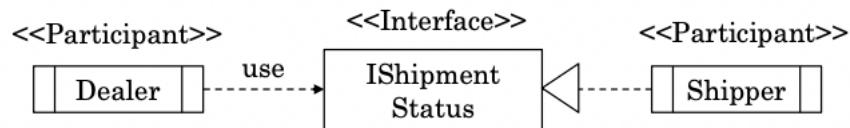
Gli attori del sistema interagiscono con dei servizi che sono funzionalità accessibili attraverso protocolli di rete. Ma queste interazioni sono sincrone o asincrone? Nei casi semplici potrebbero essere sincrone ma in generale sono asincrone.

SoaML offre un approccio alla modellazione utilizzando i diagrammi UML. Quello che fa SoaML è offrire un profilo UML che altro non è che un insieme di stereotipi che caratterizzano delle classi.

Chapter 10

Servizio sincrono

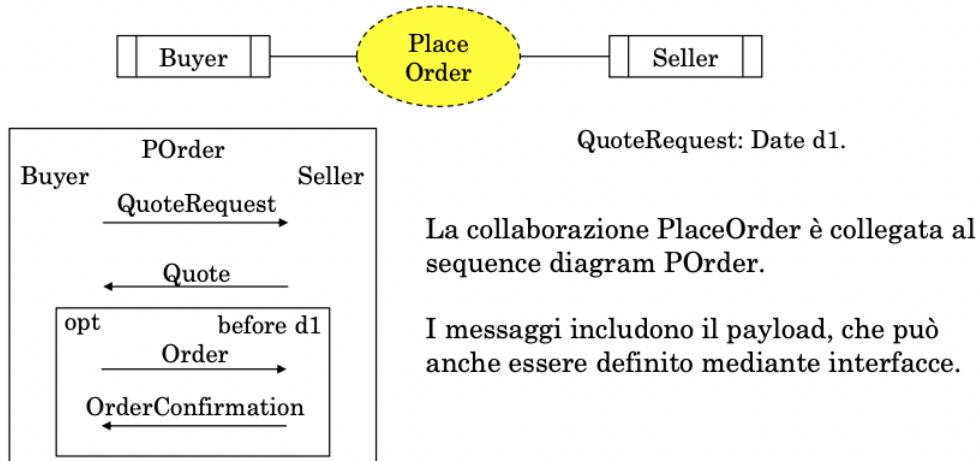
Prendiamo come esempio il diagramma che segue



Diversamente dagli esempi precedenti Dealer e Shipper hanno le doppie stanghette perché sono classi attive.

10.1 Collaborazione

Una collaborazione definisce un'interazione complessa che è modellata attraverso dei sequence diagram:



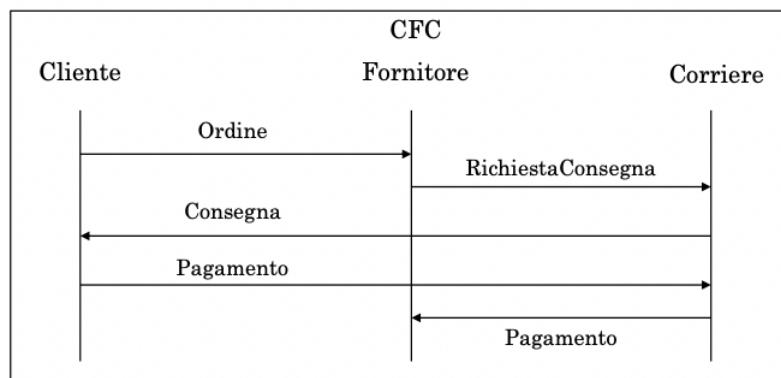
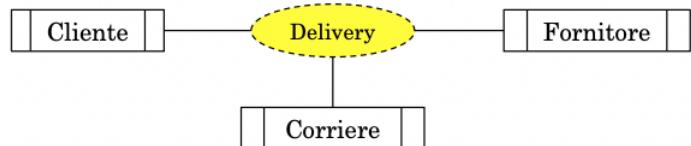
In pratica il cerchio tratteggiato giallo è un riferimento al Sequence Diagram che descrive la collaborazione tra le due entità.

10.2 Servizi

Del diagramma precedente possiamo estrapolare i servizi. In linea di massima ogni freccia è un servizio:

- **Buyer:** Quote, OrderConfirmation
- **Seller:** QuoteRequest, Order

10.2.1 Collaborazione Multipla



Chapter 11

DDD - Domain Driven Design

Vi sono alcuni contesti introdotti con questo approccio che serve nel momento in cui il modello di dominio è molto grande e comprende molti team.

11.1 Bounded context

È una porzione di un model domain costituita da classi strettamente correlate e con pochi collegamenti con altri bounded context. Le classi in generale si dividono in

- **Entity:** Entità che evolvono nel tempo, statefull
- **Value Object:** Elementi che non evolvono e sono generalmente immutabili
- **Aggregate Root:** Gestisce la consistenza degli elementi di un bounded context, come una Facade.
- **Repository:** Definisce le operazioni per la persistenza e le interrogazioni.

11.2 Ubiquitous Language

In questo contesto è sinonimo di *Linguaggio Comune*. Sta ad indicare che bisogna trovare un linguaggio comune in modo tale che tutti siano capaci di capire di cosa si sta parlando

11.3 Capability

La capacità di operare in un certo ambito. Per esempio un *Bounded Context* viene assegnato ad un team che è capace di fare tutto quello che serve per portare avanti quella determinate porzione del modello di dominio. Se per esempio il bounded context parla di ordini e acquirenti ci devono essere nel team persone esperte di acquisti e ordini.

Chapter 12

Microservizi

Quello che abbiamo detto in precedenza è utile per introdurre e capire i Microservizi. In breve sono dei servizi non troppo grandi che riguardano una sola porzione del sistema che è utile mantenere separata dalle altre per diverse motivazioni. Ogni microservizio dovrebbe essere deployable in maniera del tutto indipendente al punto tale che tra i vari microservizi è possibile utilizzare linguaggi differenti e strategie di persistenza differenti. Ciascun microservizio è eseguito da qualche processo. Ogni microservizio può essere aggiornato in modo autonomo e indipendente dagli altri microservizi. Un microservizio è gestito da un team cross-funzionale.

I Microservizi sono molto più flessibili delle strategie monolitiche però sono meno performanti. Se c'è di mezzo un DB allora le operazioni potrebbero essere decisamente più complicate perché siamo in un ambiente distribuito.

12.1 Evoluzione da monoliti a microservizi

12.1.1 Integrazione

Si può avere un core monolito a cui i microservizi si interfacciano quando devono realizzare delle richieste. Questa pratica è molto utilizzata da banche che pur avendo la necessità di dover aggiornare i propri servizi non possono permettersi di riscrivere il core della banca perché sarebbe un lavoro troppo grande

12.1.2 Sostituzione

Un monolito esistente potrebbe essere sostituito quando inizia a dare problemi o suddiviso in modo da farlo diventare una serie di microservizi.

12.1.3 Nuova applicazione basata su microservizi

Si parte da zero a trasformare un sistema monolito a microservizi.

Chapter 13

Business Process

Un business process ha l'obiettivo di raggiungere un obiettivo orchestrando le unità di lavoro, chiamati task. Un processo è un insieme di task che si svolgono nel tempo e nello spazio definiti con degli input e degli output chiari. Business Process Management è una disciplina che coinvolge diversi aspetti, controllo, automazione, esecuzione ecc ecc.

13.1 BP Models

Sono modelli definiti da una notazione ben specifica. Ci sono due categorie identificate:

- BP come procedura standard: cioè l'orchestrazione è fatta in automatico secondo delle regole date in precedenza
- BP come un gestore di casi: utilizzato per gestire delle casistiche che necessitano di modalità specifiche e custom. L'orchestrazione è totalmente o parzialmente a carico dei partecipanti al caso.

La notazione utilizzata è la BPMN che è stata standardizzata dalla OMG.

13.2 Workflow Pattern

Sarebbe la struttura del control flow nei modelli di business. Le componenti sono divise in due categorie *Basic* e *Advanced*

13.2.1 Basic control patterns

Sequence, parallel split, synchronization, exclusive choice, simple merge

13.2.2 Advanced control and synchronization patterns

Multi-choice, synchronization merge, ...

13.3 Control Flow e Data Flow

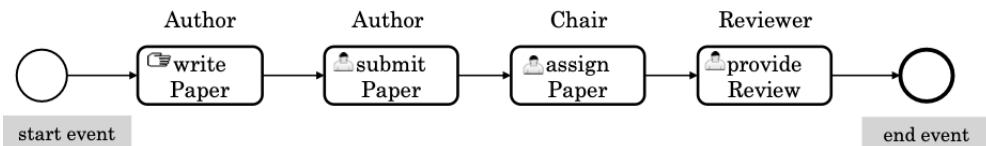
Nei processi di business sono separati:

- **Control Flow:** Specifica la precedenza tra task, quando un task è completato vengono emessi un certo numero di control-flow token che attiveranno i task che raggiungeranno nella rete
- **Data Flow:** L'input di un task è l'output del task precedente. Definisce una dipendenza di dati tra i task.

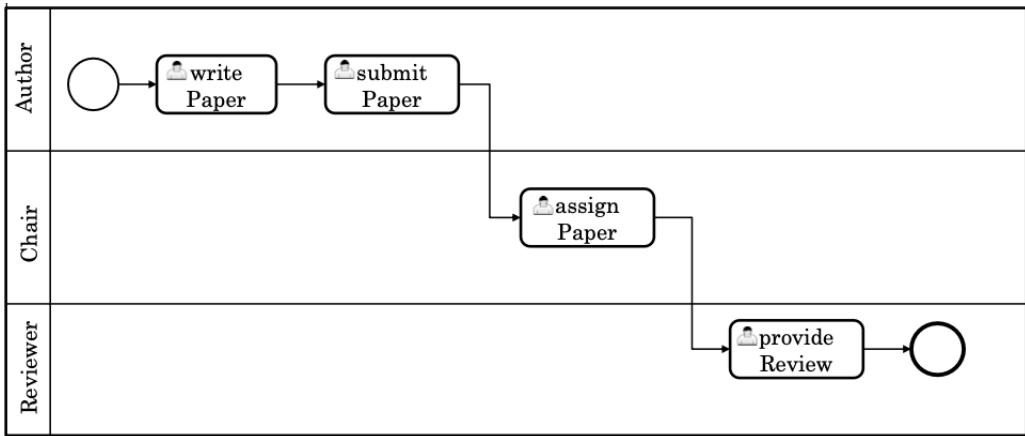
I dati di processo nei BP Models vengono conservati in variabili che vengono accostate al processo.

13.4 Esempio

Il modello di un BP può essere espresso sia come diagramma libero che come diagramma con swimlanes. Quello che segue è quello libero

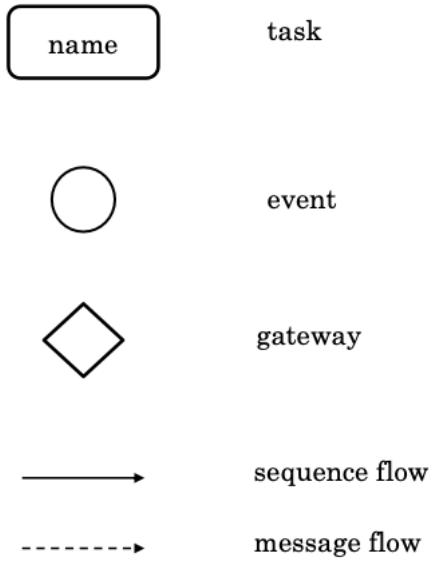


Il successivo è quello con swimlane che in questo caso sono orizzontali:



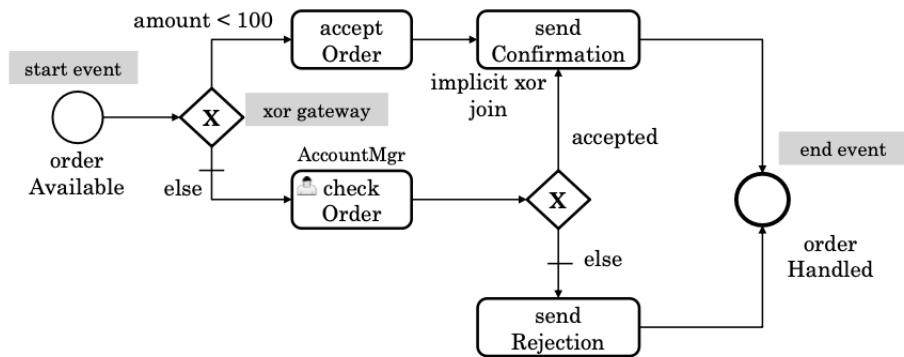
13.5 Elementi grafici dei BP Models

Gli elementi grafici di un BP model sono tutti quelli che seguono:

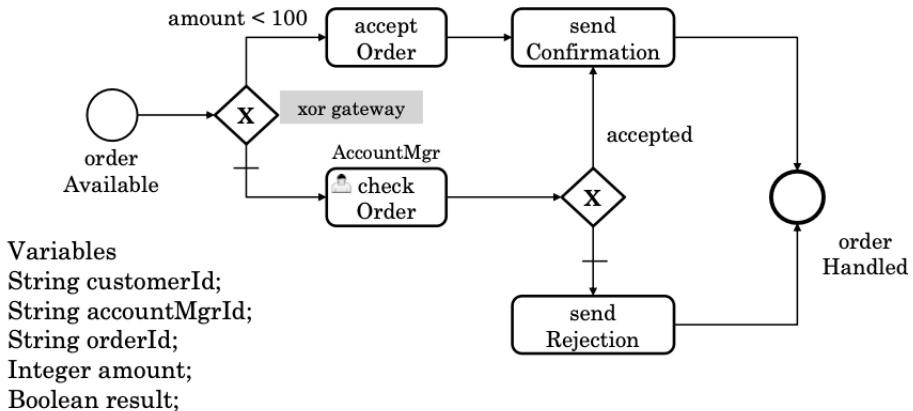


13.6 Esempio 2

This process handles the life cycle of a customer order. The information passed to the process includes the customerId, the accountMgrId, the orderId and the order amount. The result is true if the order has been accepted, false otherwise. If the amount is < 100 the order is accepted automatically; otherwise it has to be checked by an accountMgr, who may accept or reject it.



I nomi del control flow non devono essere interpretati come quelli che conosciamo dai precedenti modelli. Per esempio in questo caso l'elemento *implicit xor join* non è il join che conosciamo noi ma esclusivo, scatta solo se uno o l'altro ramo emette un token. Come detto in precedenza i dati del processo vengono conservati in variabili, vediamo le variabili per questo esempio:

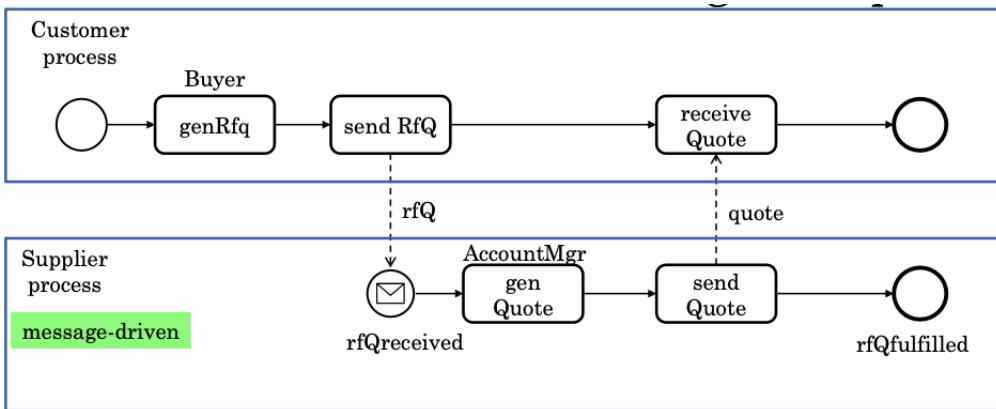


When the process is instantiated the new instance receives the values of the first four variables.

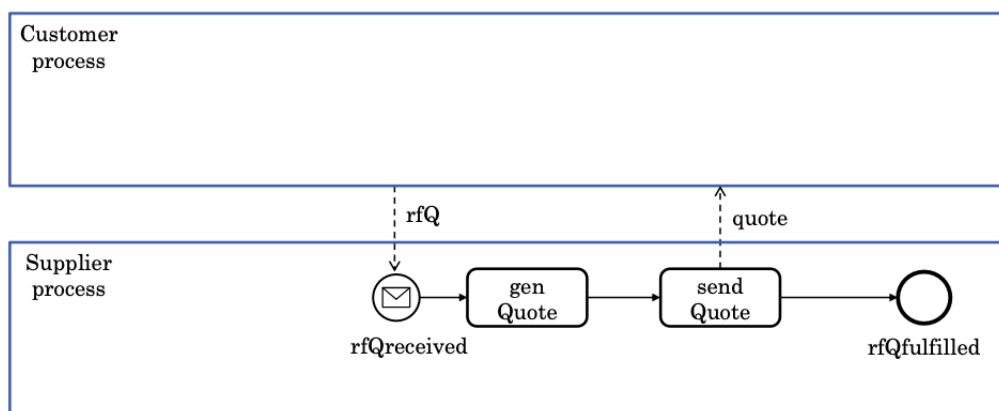
The outcome is contained in variable *result*.

Come vediamo sopra alcuni task viene specificato un ruolo che è il ruolo che può eseguire quel task. A run time il process engine si occuperà di assegnare i task ai giusti ruolo.

La cosa interessante di questi processi è che sono processi eseguibili. È inoltre possibile modellare processi B2B e lo si fa attraverso l'utilizzo di costrutti che vengono chiamate *pool* che è un rettangolo che racchiude i task dello stesso processo:

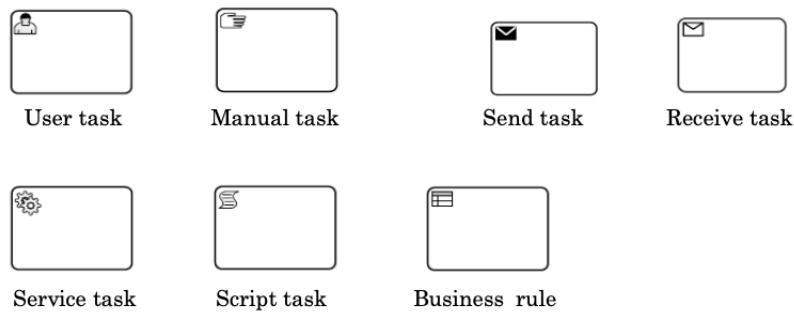


È inoltre possibile non entrare nel dettaglio di un processo, per esempio in questo caso del Customer Process:

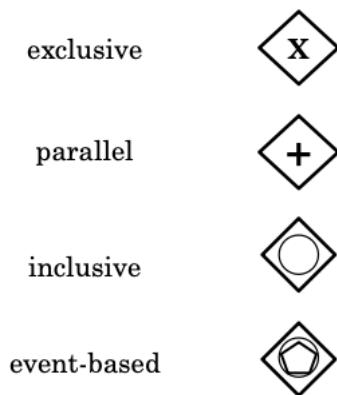
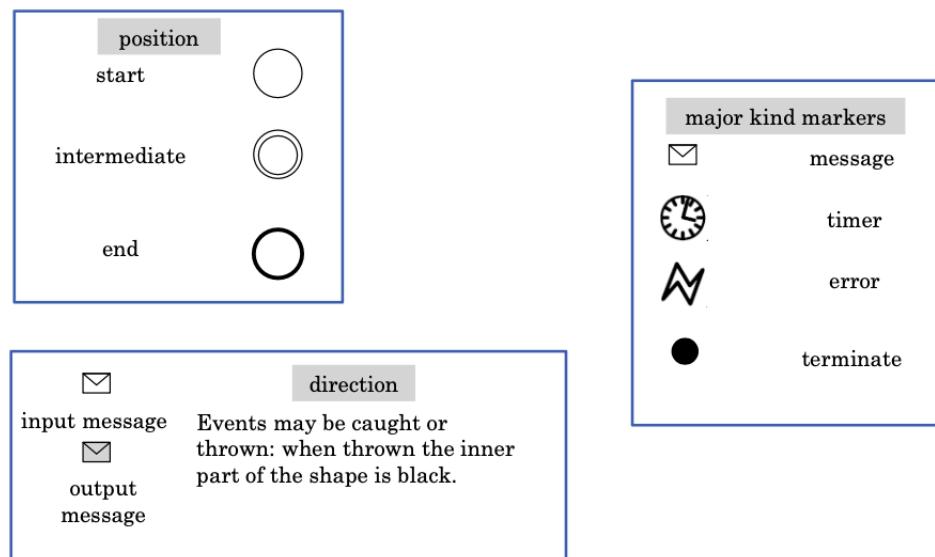


Come si vede dai modello appena visti ci sono davvero tanti simboli grafici. Effettivamente vi sono circa un centinaio di simboli che possiamo utilizzare. Di seguito alcuni simboli:

13.6.1 Task



13.6.2 Events

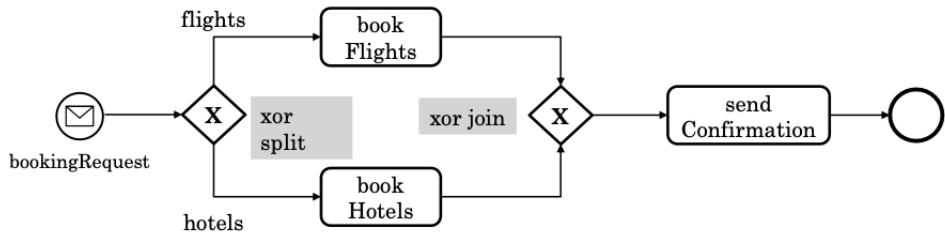


Esempi di gateways

Un'agenzia di viaggio desidera ricevere prenotazioni con diverse modalità:

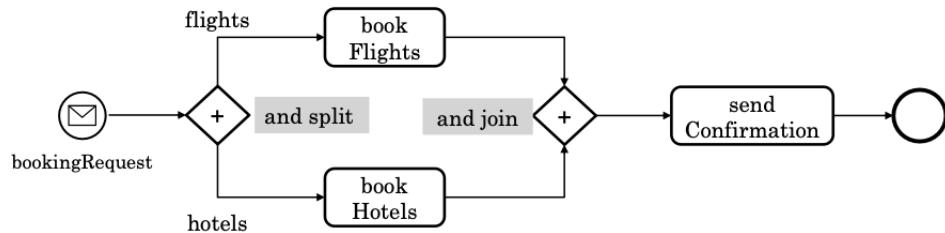
- Voli o Hotel
- Voli e Hotel
- Voli e/o Hotel

Per modellare la prima casistica possiamo usare un exclusive gateway:



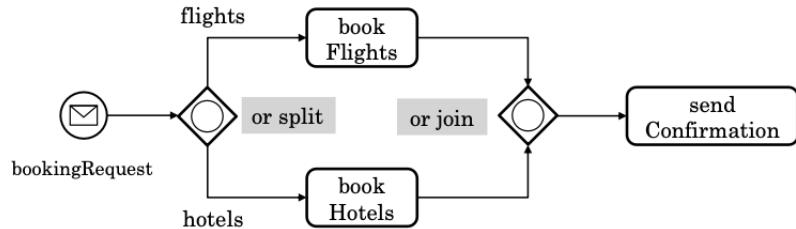
xor split manda nel ramo opportuno il token, *xor join* ha lo stesso effetto di una confluenza, non bisogna aspettare entrambe.

Per la seconda casistica possiamo usare un parallel gateway:



Funzionano esattamente come fork e join.

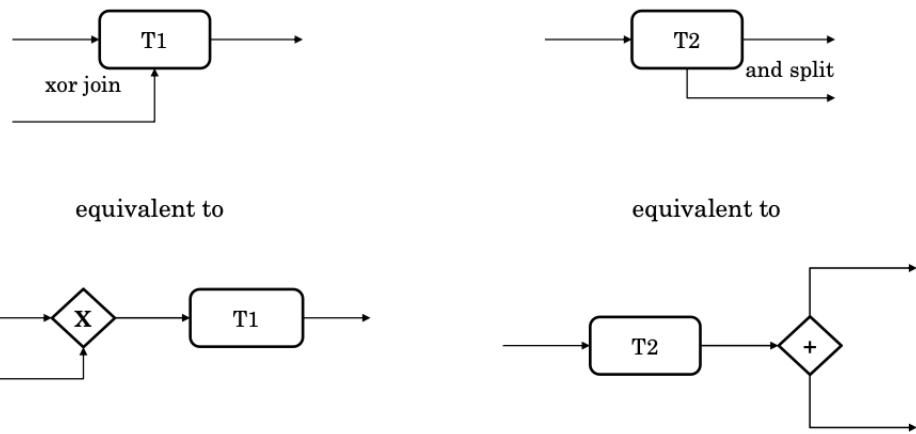
Per gestire la terza casistica possiamo usare un inclusive gateway:



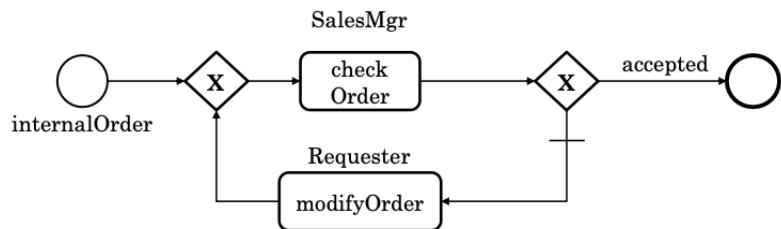
Nel momento in cui arriva un token viene istradato nel ramo corretto. Il *or join* aspetta che vengano emessi due token dall'*or split* collegati. Quando viene fatta una prenotazione per volo e hotel vengono generati due token che hanno degli id specifici e che sono collegati tra loro per cui l'*or join* aspetta che vi sia la coppia insieme per poter effettuare il join.

13.7 Altri elementi grafici dei modelli

13.7.1 Task con più input/output

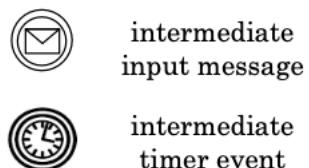


13.7.2 Loop

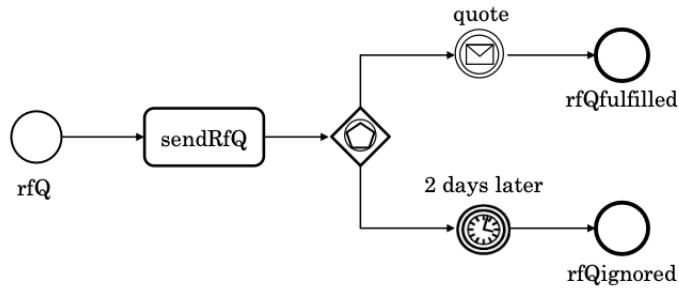


13.7.3 Event-based gateway

Prima di tutto si usano due simboli aggiuntivi



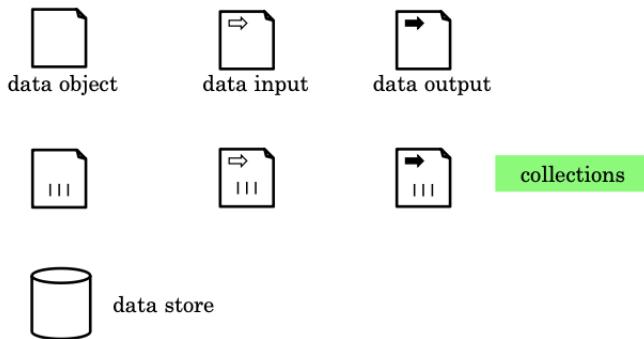
e si possono usare come segue:



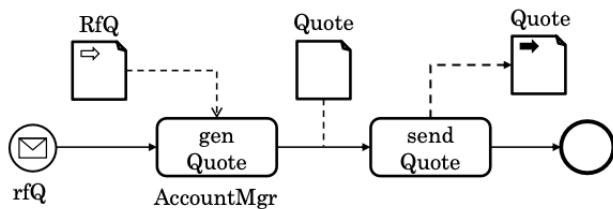
Quello che succede è che se arriva un messaggio al gateway esso si ferma in attesa di un evento esterno che potrebbe essere una quote oppure la scadenza dopo 2 giorni. Anche se non è mostrato la quote arriva dall'esterno. Quindi il token bloccato nel gateway viene sbloccato e mandato nel ramo che ha generato l'evento solo quando l'evento viene generato.

13.7.4 Oggetti Dati

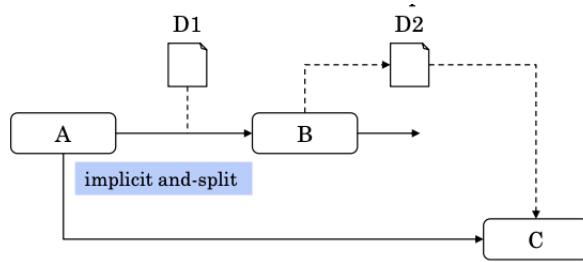
Ci sono degli oggetti che rappresentano i dati e delle collection di dati sia in input che in output:



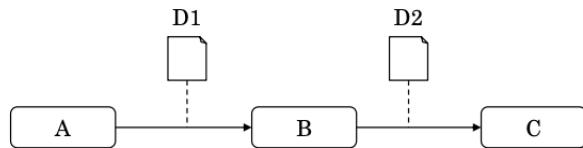
Per capire come usarli vediamo degli esempi:



In sostanza quello che succede è che il token fa sempre la stessa strada, cioè attraversa tutti i task, però viene mostrata in maniera esplicita quali sono gli input e gli output dei task. Inoltre, secondo lo standard, è possibile definire dei constraint con i data object. Si valuti l'esempio seguente per capire di cose si sta parlando

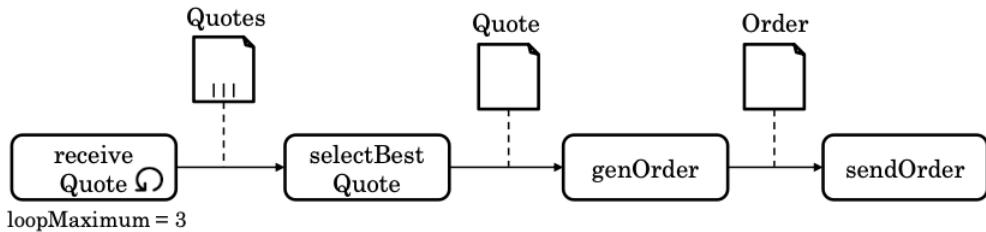


Come si vede B e C aspettano che A finisca la sua esecuzione per ricevere i token di controllo ma una volta che A termina in realtà solo B può passare in esecuzione perché C ha una dipendenza di dato che viene soddisfatta da B. In realtà questo esempio non è proprio ben fatto, potrebbe essere modificato come segue:



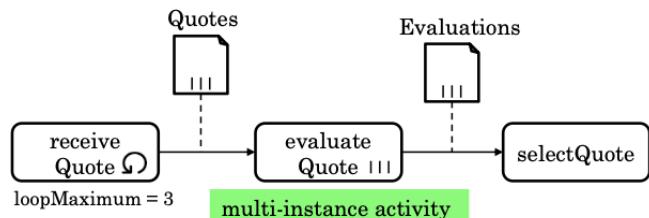
Questa equivalenza tra i due modelli è vera perché purtroppo i modelli BP hanno il control flow e il data flow separato.

L'esempio che segue fa capire come usare delle collezioni:



Come si vede il task *receive Quote* viene eseguito 3 volte, e questo viene indicato dalla freccia circolare e dal numero specificato dalla variabile *loopMaximum*. Ad ogni iterazione la quota raccolta viene messa nella collection *Quotes* definita subito dopo.

13.7.5 Multi-instance activities

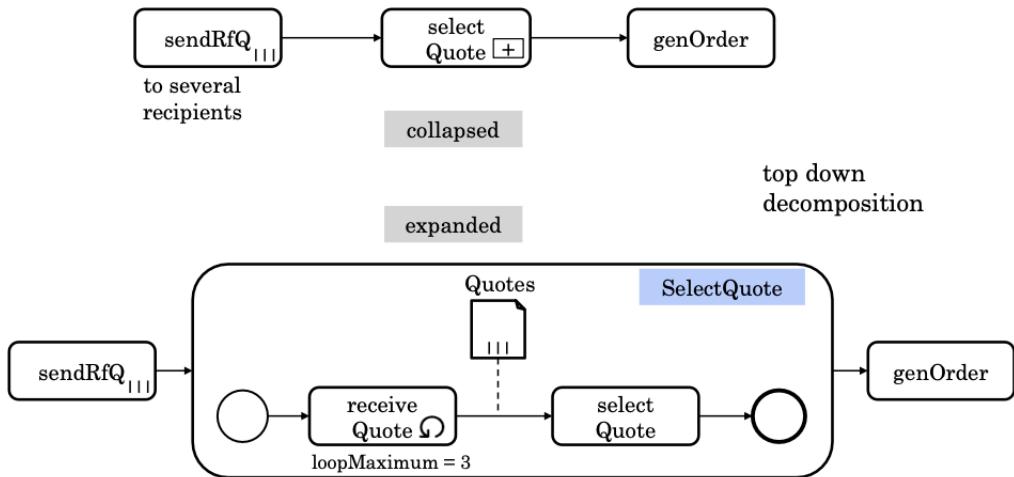


Il task *evaluate Quote* viene eseguito in parallelo e questo lo si può notare dal fatto che a valle del task vi è una collection e non un oggetto singolo e anche dal simbolo |||.

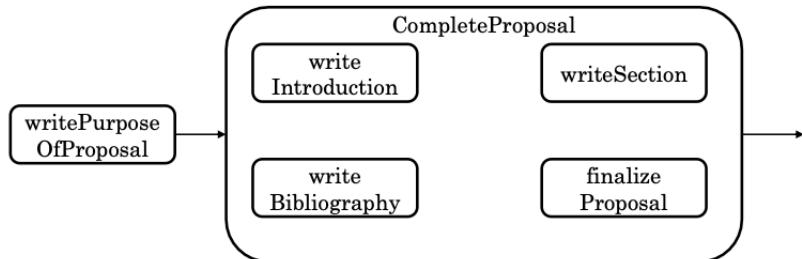
Allo stesso modo, se il simbolo fosse invece tre linee orizzontali starebbe ad indicare che la valutazione avviene in maniera sequenziale e non parallela.

13.7.6 Sotto processi

C'è la possibilità di definire sotto processi:

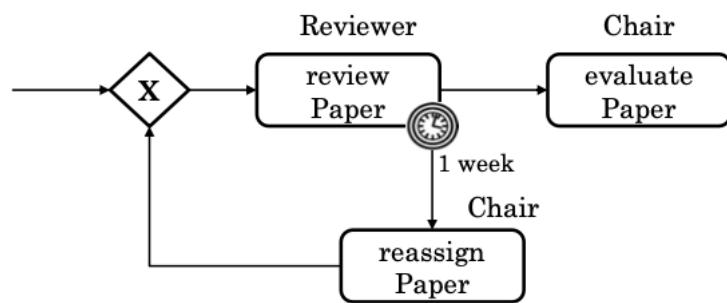


Inoltre è possibile definire un secondo tipo di sotto-processo in cui i task possono essere eseguiti in qualsiasi ordine e anche non eseguiti:



13.7.7 Boundary Events

Sono eventi intermedi capaci di gestire delle anomalie nel processo come ad esempio il caso in cui il task non venga eseguito entro i limiti di tempo.



Alla scadenza della deadline viene cambiato revisore.

13.8 Difficoltà con BPMN

Anche se molto flessibili comunque ci sono delle difficoltà di modellazione.

Chapter 14

Verifica del software

Abbiamo già parlato di queste cose all'inizio del corso, rivediamo che cosa bisogna chiedersi e cosa fare quando si fa verifica del corso:

- **Verification:** Stiamo costruendo bene il prodotto? Lo si fa attraverso meccanismi di testing, white e black box.
- **Validation:** Stiamo costruendo il giusto prodotto?
- **Configuration Management:** Definisce gli aspetti del sistema, controllare le release e i cambiamenti
- **Project Management:** Pianificazione, staffing, monitoraggio, controllo e leading del progetto.
- **Quality Assurance:** Essere certi che standard e procedure richieste siano tenuti in conto per ottenere un prodotto di qualità

14.1 Verifica e Validazione

- **Verifica:** È il processo di un sistema o di un componente per determinare che, indipendentemente dalla fase di sviluppo, soddisfi le condizioni imposte all'inizio della fase.
- Processo che ha lo scopo di valutare un sistema o un componente, una volta sviluppato, rispetti i requisiti forniti dal customer.

14.2 Tecniche di Verifica

14.2.1 Inspection

È una tecnica di analisi statica che consiste in una esaminazione visuale del prodotto al fine di individuare errori, violazioni di standard di sviluppo e altri problemi.

Esaminare vuol dire che dei reviewer esaminano il sorgente del programma con l'obiettivo di trovare problemi. Si verificano in particolare punti critici dei programmi come ad esempio i loop.

Una tecnica famosa è quella di **Fangan** ormai di molti anni fa. Gli elementi in giorno sono tre: Team, process, documents.

Ruoli nel team:

- **Author:** Introduce il proprio lavoro e risponde a domande
- **Moderatore:** Gestisce il team
- **Reader:** Colui che legge il codice alla ricerca di errori
- **Inspector:** Punti di difetti
- **Recorder:** Colui che documenta i punti di difetti individuati.

Il processo di Fangan ha 6 stage:

- **Planning:** Il moderatore verifica che il prodotto da valutare corrisponda a dei criteri di input e gestisce i meeting per poter portare avanti l'analisi del prodotto
- **Overview:** Introduzione da parte dell'autore del prodotto. Dovrebbe spiegare in questa fase quale sia l'obiettivo del prodotto
- **Preparation:** In questa fase si ha a che fare con una *check list* dove sono presenti i tipi di ispezioni da effettuare nel prodotto. La check list potrebbe essere una standard o personalizzata per lo specifico progetto
- **Inspection:** Il reader legge il codice e potrebbe sottolineare i difetti ma non deve in alcun modo trovare una soluzione o fornire un'alternativa.
- **Rework:** I difetti individuati nella fase precedente devono essere sistemati.
- **Follow-up:** Si verifica che le modifiche effettuate non abbiano causato degli effetti collaterali al resto del codice.

Esempio

```
float f (float x, float y)
```

```
spec: if x > 0 returns y/x else returns y/(x + 10)
```

```
float f (float x, float y)
```

```
if (x <= 0) x += 10;
```

```
return y/x;
```

Is this code correct?

La prima parte in alto è una piccola specifica mentre la parte sotto è il codice da verificare. Il codice ha evidentemente un problema perché se x è negativo ed è pari a -10 si effettua una divisione per 0 che darebbe un errore a run time.

Ispezione e testing sono complementari, non occorre sceglierne solo 1 dei due e inoltre l'ispezione potrebbe ridurre il costo del testing. I risultati dell'ispezione non devono essere utilizzati per punire o premiare l'autore del codice a seconda dell'andamento dell'ispezione stessa.

14.3 Tipi di testing

Esistono generalmente due tipologie di testing:

- **Black box testing:** Non sappiamo i meccanismi del programma però sappiamo i dati che si aspetta in input e sappiamo l'output per un determinato input allora possiamo testare il corretto funzionamento mandando tanti input e verificando gli output
- **White box testing:** Abbiamo accesso al codice del programma e quindi tiene conto dei meccanismi interni del sistema.

14.3.1 White Box Testing

Come detto tiene conto della struttura e prende in considerazione di 4 criteri per poter stabilire la **coverage** dei test che esprime quanto del codice presente è stato testato:

- **Nodi:** % di istruzioni eseguite dai test
- **Link:** % delle uscite dai nodi decisionali percorse dai test; cioè % delle espressioni booleane, nei nodi decisionali, per le quali sono state testate entrambi i risultati true e false

- **Percorsi:** % percorsi testati. Un percorso porta da un nodo iniziale ad un nodo finale. Problematico se sono presenti loop
- **Condizioni Multiple:** % delle sotto-espressioni booleane per quali sono state testati indipendentemente i risultati true e false.

Esempio

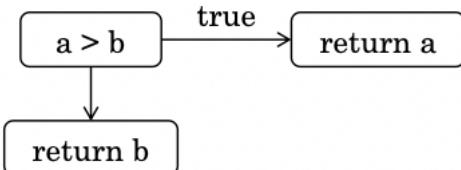
La specifica è:

int f1 (int a, int b) fornisce il maggiore tra a e b.

Il codice che rispetta la specifica è:

```
if (a > b) return a;
else return b;
```

Il grafo che viene generato:

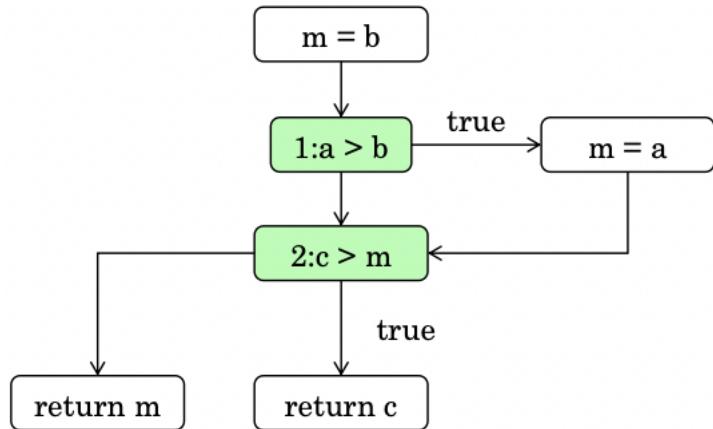


2 test per la copertura dei nodi; t1(20,10): 20, t2(10,20): 20. Almeno tanti test quante sono le uscite.

Gli stessi test per link e percorsi.
Copertura del 100%.

Indicazione dei test, es. t1(20,10): 20
t1 = nome del test
(...) valori degli input
:20 risultato atteso

Anche se nell'esempio precedente non è stato fatto bisogna numerare le condizioni nell'ordine in cui appaiono nel codice. Nei temi di esame troveremo le condizioni numerate nel codice per semplicità:



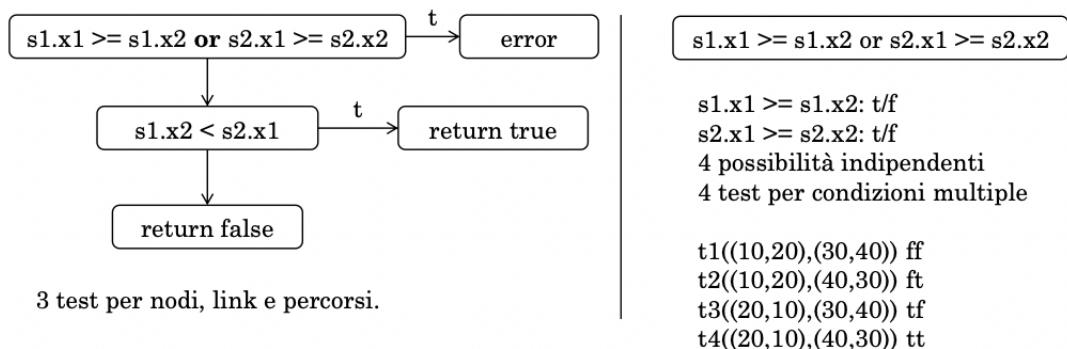
Inoltre i percorsi, grazie a questa numerazione delle condizioni, è possibile indicarli come combinazioni di V e F numerati. ad esempio: 1T 2T, 1T 2F, 1F 2T, 1F 2F che poi si possono scrivere in maniere molto compatte: 1T (2T, 2F), 1F (2T, 2F) = 1T (2), 1F (2) = (1) (2) = 1 2. Si intendano le parentesi come moltiplicazioni, quando compare solo un numero o un numero tra parentesi si intende entrambi i valori di quella specifica condizione, per esempio: 1F (2) = 1F (2T, 2F) = 1F 2T, 1F 2F

Condizioni multiple

Sono le semplici condizioni booleane composte, più complesse di quelle normali ma seguono la stessa logica:

```

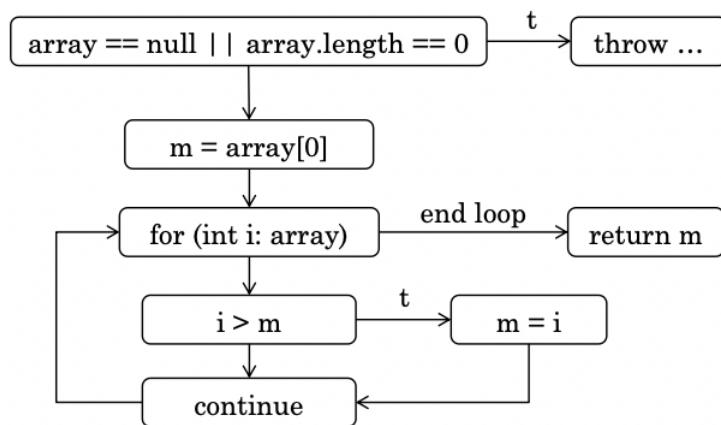
Type Segment: int x1, int x2;
boolean g1 (Segment s1, Segment s2)
fornisce true se s1 precede strettamente s2; dà errore se in un segmento x1 >= x2
if (s1.x1 >= s1.x2 or s2.x1 >= s2.x2) error;
if (s1.x2 < s2.x1) return true; else return false;
  
```



Quello che bisogna considerare è che per una condizione multipla sono necessari più casi di test per poter coprire tutti i casi.

Programma con loop

```
public static int f3 (int[] array) throws Exception {
    if (array == null || array.length == 0) throw new Exception("input error");
    int m = array[0];
    for (int i: array) if (i > m) m = i;
    return m; }
```



Il numero di percorsi potrebbero essere un numero molto alto però convenzionalmente per poter testare un ciclo solitamente si effettuano 3 casi di test:

- Test che non fa eseguire il loop
- Test che fa eseguire il loop 0 volte
- Test che fa eseguire il loop 1 volta

Analizzando la condizione multipla di questo programma vediamo che le possibili combinazioni sono 4 ma in realtà la copertura è di 75% perché non esiste il caso di array nullo con lunghezza. Per cui i casi rimasti sono

- Array nullo
- Array non nullo e lunghezza != 0
- Array non nullo e lunghezza == 0

Per quanto riguarda i loop, come detto in precedenza dobbiamo fare solo 3 casi di test che sono i seguenti:

- t1(null)
- t1([])
- t1([1])

Noi non tratteremo programmi con loop all'interno.

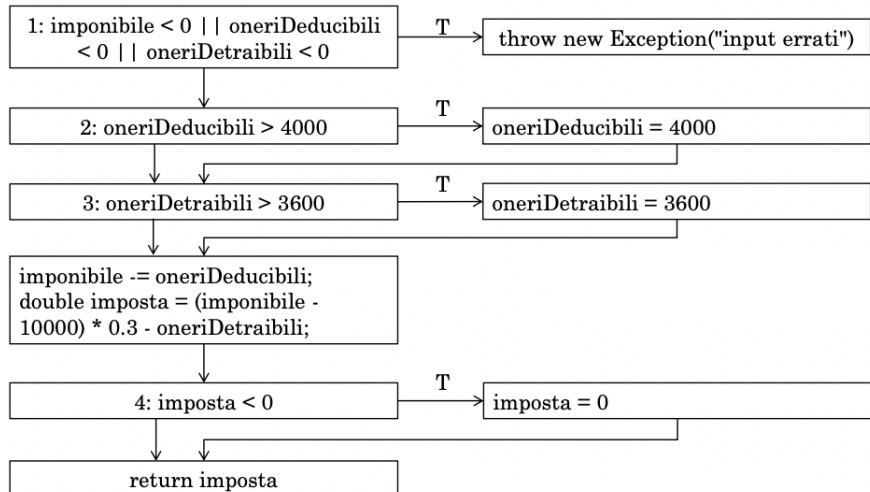
WBT1

Questo esempio presenta una serie di elementi che tratteremo praticamente sempre nei nostri esercizi.

```
static double imposta (double imponibile, double
    oneriDeducibili, double oneriDetraibili) throws Exception{
    if (imponibile < 0 || oneriDeducibili < 0 ||
        oneriDetraibili < 0)
        throw new Exception("input errati");
    if (oneriDeducibili > 4000) oneriDeducibili = 4000;
    if (oneriDetraibili > 3600) oneriDetraibili = 3600;
    imponibile -= oneriDeducibili;
    double imposta = (imponibile - 10000) * 0.3 -
                    oneriDetraibili;
    if (imposta < 0) imposta = 0;

    return imposta;
}
```

Lo scopo è di definire il flow chart e considerare Nodi, Link e percorsi:



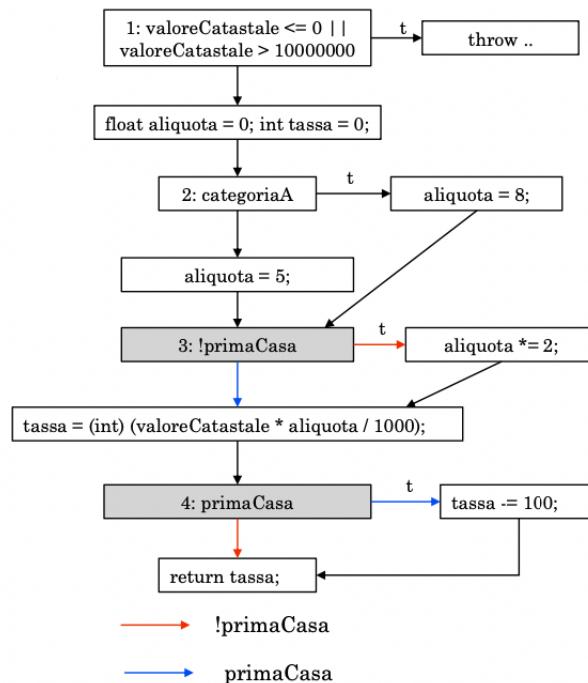
- **Nodi:** 2 percorsi: 1T, 1F 2T 3T 4T

- **Link:** 3 percorsi: + 1F 2F 3F 4F
- **Percorsi:** 9 percorsi: 1T, 1F (2) (3) (4)
- **Condizioni multiple:** 8 casi di test

Il numero minimo di percorsi per soddisfare tutti i criteri sono 15.

14.3.2 Correlazioni

Quando delle condizioni in un programma delle condizioni sono correlate tra loro in qualche modo. Per esempio come nel flow chart che segue:



La correlazione diminuisce il numero di percorsi possibili perché come si vede in corrispondenza del valore di **primaCasa** i percorsi delle due condizioni sono condizionati, uno l'opposto dell'altro.

14.3.3 Black box testing

Non si ha la possibilità di visionare le funzioni da testare, si conoscono solo gli input da mandare al programma. Bisogna verificare se gli output sono corretti. Si dice che gli output sono predetti da un *Oracolo* che è un programma simile o un ragionamento fatto sulla descrizione del programma.

Gli input sono divisi in classi di equivalenza con l'assunzione che tutti gli input della stessa classe si comporti nello stesso modo per cui per testare in modo completo basta un campione per classe.

Un esempio potrebbe essere `int abs(int x)` che ha la descrizione `if x >= 0 return x; else return -x.` Le classi di equivalenza qui sono due, numeri positivi e numeri negativi.

Oltre che ai due campioni per classe di equivalenza vi sono anche degli altri casi di test chiamati **Boundary values** che sono i casi limite, per esempio **MININT**, **0**, **MAXINT**. I valori limite potrebbero essere problematici, per esempio sappiamo che il MININT non può essere portato positivo per cui bisogna pensare bene come testare i valori al bordo.

Un altro esempio più complesso è: il premio assicurativo di un'assicurazione dipende dall'età del contraente e dal tipo di veicolo. Le fasce di età sono 18..25, 26..41, 41..60, 61..80 e vi sono 3 veicoli: auto, moto, van.

- Gli input sono due, età e veicolo
- Le classi di equivalenza sono quelle di età (18..25, 26..41, 41..60, 61..80) e quelle di veicolo (auto, moto, van)
- Vi sono degli errori: 0 (<18), 5 (>80); classe 0 e classe 5

I casi di test sono divisi in livelli di copertura:

- **Copertura minimale:** (19, moto), (27, auto), (42, van), (62, van). Si noti che nell'ultimo test, (62, van), in realtà potrebbe essere utilizzato qualsiasi altro veicolo, si è scelto van in modo del tutto casuale. In pratica con la copertura minimale si cercano di scrivere test solo per tutti i possibili valori, senza considerare le combinazioni
- **Copertura minimale con classi di errore:** (17, moto), (19, moto), (27, auto), (42, van), (62, van), (81, altro). Come il precedente ma si aggiungono i casi di errore
- **Copertura robusta:** (19, moto), (19, auto), (19, van) ecc. Si valutano tutte le combinazioni possibili dei valori
- **Copertura robusta con casi di errore:** Come il precedente ma in più si aggiungono tutte le possibili combinazioni di valori che generano errore, ad esempio: (17, moto), (17, auto), (17, van), (17, altro), (19, moto)
- **Baunday values:** Sono in pratica i limiti di ogni fascia di età per i test normali e per i casi di errore si ha 17, 81, MININT, MAXINT

Si possono visualizzare i livelli di test in una griglia:

| | minimale | | | | | robusta | | | |
|------|----------|---|---|---|---------------|---------|---|---|---|
| van | | | x | x | | x | x | x | x |
| auto | | x | | | | x | x | x | x |
| moto | x | | | | | x | x | x | x |
| | 1 | 2 | 3 | 4 | classi di età | | | | |

| | minimale con errori | | | | | | robusta con errori | | | | |
|--|---------------------|---|---|---|---|--|--------------------|---|---|---|---|
| | | | | | x | | x | x | x | x | x |
| | | | x | x | | | x | x | x | x | x |
| | | x | | | | | x | x | x | x | x |
| | x | x | | | | | x | x | x | x | x |

Per casi più complicati e altri esempi fare fede alle lezioni o alle slide sezione 4.

14.3.4 Obiettivi di testing

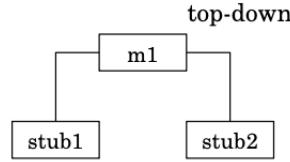
- **Unit:** test fatti dallo sviluppatore stesso di un componente in maniera white box
 - **Integration:** Testing di un sotto-sistema; si usano white box e black box seguiti dagli sviluppatori dell'integrazione.
 - **System:** Eseguito da una squadra di testing, sarà un testing black box
 - **Acceptance:** Eseguito dal committente prima di accettare il prodotto sviluppato. Sicuramente è un black box
 - **Regression:** si eseguono test per verificare che nuove modifiche non abbiano intaccato le funzionalità già testate in precedenza. Può essere white box o black box in maniera congiunta
 - **Beta:** Il prodotto viene testato prima di essere rilasciato. Il prodotto viene dato in prova a degli utenti interessati e questi utenti forniscono feedback. Ci sarebbe anche l'alpha testing però la differenza è che l'alpha testing viene effettuato dagli sviluppatori, in casa di chi sviluppa poiché in fase alpha il test potrebbe essere molto approssimativo mentre il beta testing è più accurato con un prodotto quasi terminato.

I primi 3 tipi di test sono fatti dagli sviluppatori. L'acceptance test viene eseguito dal committente.

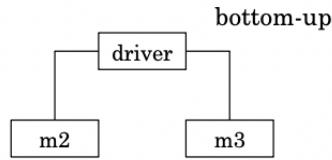
Integration/system testing

Può avvenire su tutto il sistema oppure in modo incrementale con delle metodologie:

- **Top-Down:** I moduli più di basso livello vengono sostituiti da moduli chiamati *stub*.



- **Bottom-Up:** Si fa il contrario dell’approccio precedente cioè si sostituiscono i moduli di alto livello con dei driver. Si perde la visione d’insieme:



14.3.5 Test-driven development

Quello che si fa è scrivere i casi di test prima della scrittura del codice da testa. Inizialmente i casi falliscono ma poi con la scrittura del codice essi iniziano ad avere successo. In maniera iterativa si possono raffinare i test e successivamente anche il codice.

I vantaggi sono:

- Feedback immediato del codice, in positivo e in negativo
- È un approccio che si sposa bene con l’approccio agile

mentre gli svantaggi sono

- Si perde la visione d’insieme
- Si rischia di trascurare l’attività progettuale

Chapter 15

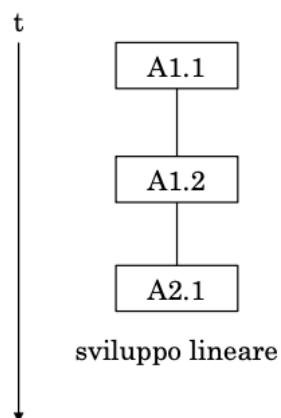
Gestione della configurazione

È uno strumento per tenere traccia dell’evoluzione di un sistema e dei suoi componenti. In particolare permette di tenere traccia delle versioni dei componenti e dei suoi cambiamenti nel tempo.

15.1 Version management

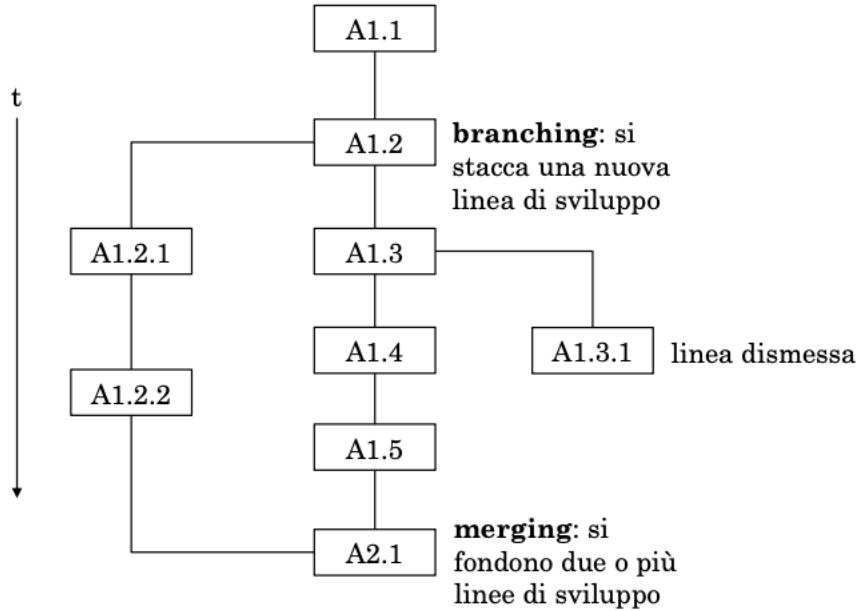
Quando si sviluppa un sistema esso è composto da componenti che si evolvono nel tempo attraverso differenti versioni. Lo sviluppo delle versioni può essere lineare, ad albero e a grafo.

15.1.1 Versionamento lineare



A è il nome del componente e il numero esprime la versione, in questo caso suddivisa in sotto-versioni. L’insieme delle versioni è chiamata codeline.

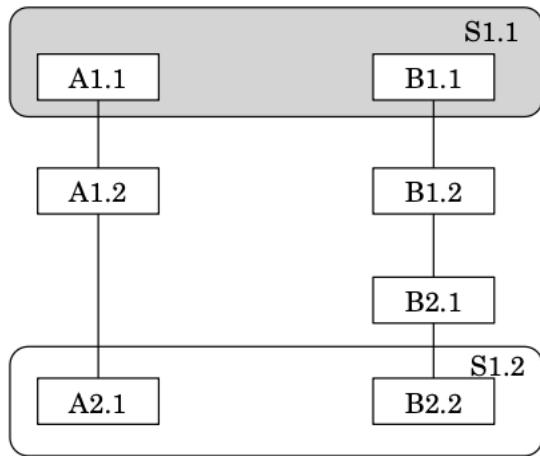
15.1.2 Versionamento a grafo



Si hanno versionamenti a grafo quando si ha a che fare con dei diversi branch del versionamento. Un branch è un punto in cui si decide di intraprendere una nuova strada di sviluppo del prodotto, parallela a quella principale, che quindi ha delle versioni a se stanti. In un secondo momento, dopo il branching, si potrebbe decidere di effettuare il merging per riunificare le linee di versioni in modo da unire le modifiche di due rami. Come vediamo nell'immagine precedente si è unito il ramo A1.2.* al ramo A1.*.

15.2 Sviluppo di un sistema

I componenti di un sistema possono essere molteplici e questi componenti potrebbero dare la versione all'intero sistema:



In pratica la versione di un sistema si compone dalla versione dei suoi componenti. La versione di un sistema si chiama **baseline** e contiene delle versioni dei componenti che lo compongono. L'insieme delle versioni, quindi delle baseline, in sequenza temporale prende il nome di **mainline**.

La gestione della configurazione permette anche di gestire in maniera efficiente le versioni del codice sorgente ottimizzando lo spazio. Per la versione più recente si tiene la versione per intera mentre per le versioni precedenti si tiene un delta tra le versioni. In questo modo partendo dalla versione più recente si può ricostruire qualsiasi versione precedente. Un configuration management offre due operazioni principali

- **Check out:** L'utente riceve una copia locale di una data versione
- **Check in:** L'utente inserisce una versione indicando il rapporto con le precedenti.

15.3 Change management

È un aspetto da considerare perché potrebbe essere un processo. Supponendo di lavorare in team i cambiamenti potrebbero essere gestiti da un processo particolare, per esempio di approvazione delle modifiche. Per esempio potrebbero esserci gli sviluppatori che sottosmettono le richieste di modifica, qualcuno che verifica per valutare i costi/benefici.

Una richiesta di modifica contiene varie informazioni

- Data e richiedente
- Descrizione della richiesta

Un valutatore aggiunge una valutazione alla richiesta assegnando una priorità, difficoltà e tempo di esecuzione.

Un product manager può approvarla o rimandarla. Può raggruppare più richieste. I richiedenti delle modifiche sono informati dell'esito.

15.4 Issue Tracking System

Il sistema è simile alla segnalazione di un'anomalia da parte di un utente ad un help desk. Viene aperto un ticket che segnala delle anomalie che viene gestita da qualcuno, tecnico o meno, e il ticket passa attraverso diversi stati: Open, Discarded, Pending, Assigned, Resolved, ...

15.5 Git

È un sistema open source per il controllo di versione. Le parti importanti di git sono i repository che sono due:

- **Repository Remota:** Cioè una repository su un server che viene utilizzata da tutti i componenti del Team
- **Repository Locale:** Cioè una copia della repository remota su cui può lavorare in comodità direttamente sul proprio file system

Per iniziare si deve avere un client locale di Git che offre una serie di comandi. I principali comandi sono

- init
- add
- commit
- branch
- merge
- checkout
- remote
- add
- fetch
- push

In pratica per iniziare non si fa altro che creare una cartella sul proprio file system e si lancia il comando **init** in quella cartella in modo da inizializzare una repository locale.

Uno sviluppatore che lavora nella sua repository locale quando decide che le modifiche sono stabili procede a lanciare il comando **add <filename>** che mette il file in uno stato detto *staged* e questi file vengono inseriti nel database, basato su mappe su filesystem, del repository locale.

Successivamente si può fare **commit** per far passare i file nello stato committed. Gli stati quindi sono:

- Untracked
- Staged
- Committed

Un commit punta al commit precedenti e all'albero delle cartelle che compone il commit.

15.5.1 Contenuto del repository

I file sono tenuti in una struttura di coppie chiave - valore (database). La chiave è un HASH SHA-1 di 40 caratteri.

Gli oggetti tree definiscono la struttura delle versioni del progetto. Un progetto è costituito dalla directory principale e dalle subdir

Gli oggetti commit indicano l'evoluzione del progetto. Un oggetto commit punto al commit precedente e ad un oggetto tree. Inoltre un oggetto commit ha dei metadati.

Il file staging area contiene i file che saranno inseriti nel prossimo commit.

Un file potrebbero avere delle ramificazioni e per convenzione il ramo principale si chiama master.

Anche gli oggetti tree e gli oggetti commit sono codificati. Con un tag è possibile indicare le versioni più importanti del progetto.

15.5.2 Diramazioni

Per aprire un nuovo ramo si può farlo con il comando **branch** che produce un nuovo ramo che punta al commit corrente. Per passare al nuovo ramo si usa il comando **checkout** e in questo modo il contenuto del ramo a cui si passa viene copiata nella working directory.

15.5.3 Convergenza

Ad un certo punto possiamo fare una unione dei diversi rami per farli convergere nel ramo principale. Per farlo si usa il comando **merge <nomeramo>**. Nei casi più semplici il merge viene fatto in maniera automatica invece in altri casi potrebbero esserci dei conflitti, cioè dei punti in cui i file sono stati modificati nello stesso punto in entrambi i rami che si stanno unendo. In caso di conflitti si deve provvedere a risolvere i conflitti a mano e poi con il comando add si portano nella staging area e quindi si può procedere al commit in modo da apportare le modifiche. Nel caso di modifiche si dice che si sta facendo un merge-commit invece che un semplice merge.

15.5.4 Repository remoto

È possibile utilizzare dei server git, come ad esempio github. Alla repository locale è possibile aggiungere l'url di una repository remota con il comando **remote add [remote-name] [url]** dove solitamente si usa origin come remote-name. Con il comando **fetch** viene copiata la repository remote.

15.5.5 Eventi

È possibile effettuare delle operazioni quando un evento avviene nel server, ad esempio un commit. Questo permette di fare facilmente integrazione continua, testing automatico ecc.

15.5.6 GitHub

Fa hosting di progetti, soprattutto open source. Nel 2019 si contavano 28 milioni di repository pubblici molti dei quali mostravano soluzioni proposte in articoli e blog. Nel 2018 è stata acquistata da Microsoft.

Chapter 16

Project Management

È una disciplina che viene usata in tutto lo sviluppo perché definisce che cosa fare, come farla e chi deve fare le cose. Comprende fase di

- Pianificazione
- Schedulazione
- Esecuzione
- Monitoraggio
- Controllo

Gli aspetti critici sono:

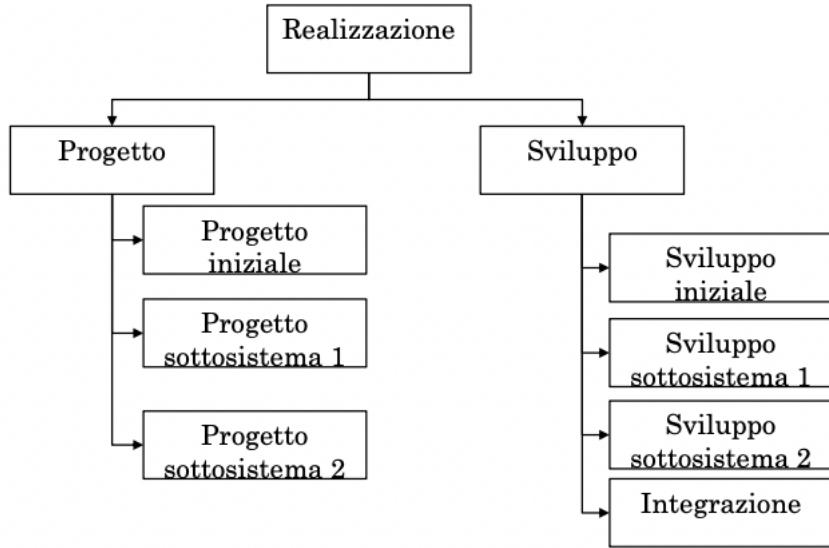
- Complessità del progetto
- Tempo per ottenere il risultato
- Costo

Ci sono degli strumenti di analisi che utilizzeremo come **WBS - Work Breakdown Structure**, **CPM - Critical Path Method**, **PERT - Project Evaluation and Review Technique** e **Gantt chart**. Per il CPM, in particolare per trovare il path critico utilizzeremo le reti di petri con la definizione di percorso più lungo.

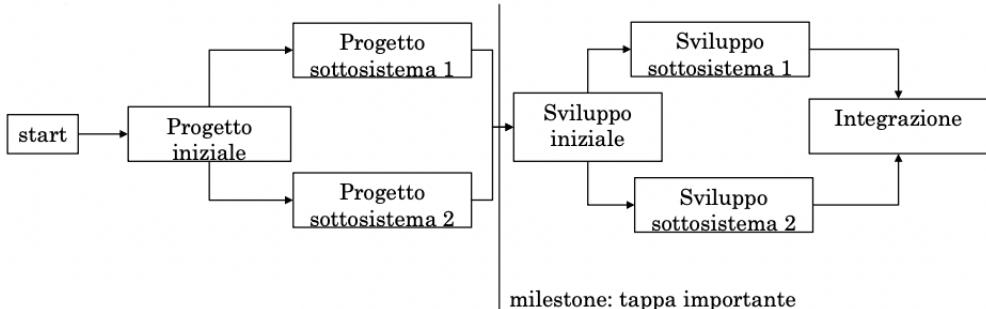
Per quanto riguarda lo sviluppo del software il PM è molto importante per definire i tempi di sviluppo

16.1 Pianificazione

Vediamo un esempio di pianificazione. Supponiamo di dover sviluppare un progetto software che è composto da 2 sotto-progetti che possono essere sviluppati in maniera indipendente. Di seguito il WBS della pianificazione



Le precedenze sono date dalle frecce. Impostiamo quindi il progetto con gli elementi definiti nello schema precedente.

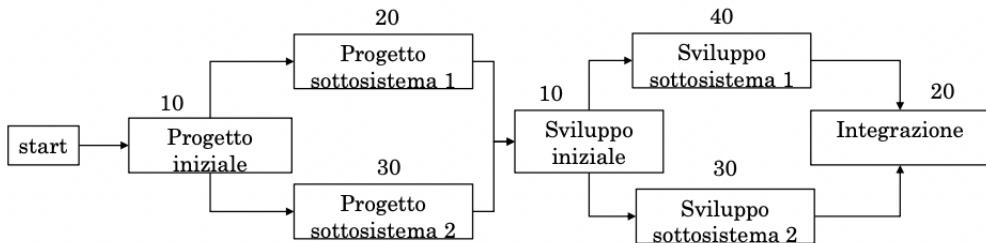


Il punto di partenza è il progetto iniziale che poi si divide in due strade differenti, che sarebbero i due progetti dei due differenti sottosistemi. In seguito i due progetti si riuniscono per iniziare gli sviluppi che a sua volta si dirama in due parti che sarebbero gli sviluppi dei due sottosistemi per poi finalmente arrivare all'integrazione

Abbiamo 4 tipologia di precedenza in questi schemi:

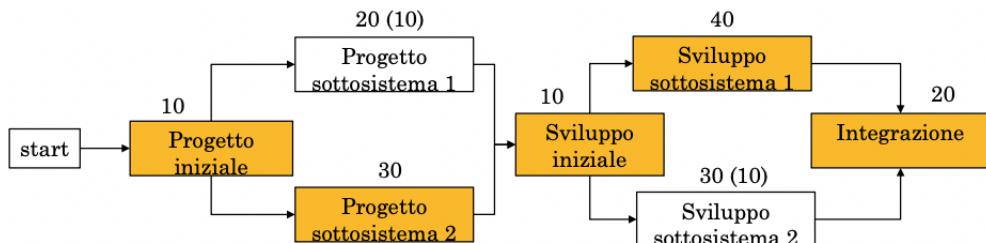
- finish to start - quella usata nel diagramma precedente
- start to start: B inizia dopo che è iniziato A
- finish to finish: A finisce dopo che è finito B
- start to finish: A finisce dopo che è iniziato B

Per quanto riguarda le durate è possibile esprimerele con una unità di tempo desiderata



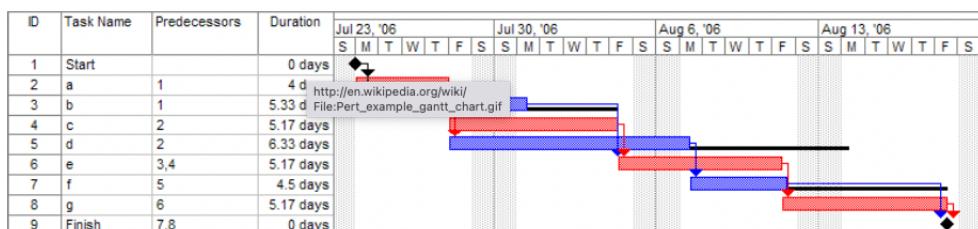
Sulla base di quale analisi utilizziamo tra quelle citate in precedenza si esprimono in maniera differente, per esempio nel PERT vengono combinate 3 diverse durate O - Ottimistica, P - pessimistica e M - La più probabile. La combinazione per il PERT è durata = $(P + 4M + O) / 6$. Invece per il CPM si deve indicare semplicemente quella più probabile.

Con le durate possiamo trovare il percorso critico che sarebbe quello che ha la durata complessiva più lunga. I percorsi sono esattamente quelli che facevamo nel white box testing, nello schema precedente lo sviluppo critico è



Gli altri task che non fanno parte del percorso critico possono essere ritardati perché non fanno parte del percorso critico per cui non modificherebbero la durata complessiva del progetto. La quantità di slittamento viene chiamata **Slack Time**. Il valore di slack appartiene al percorso, non al singolo task per cui se si slitta un task di un valore x non è necessario slittare tutti gli altri successivi di x. So può però distribuire lo slack tra i vari task.

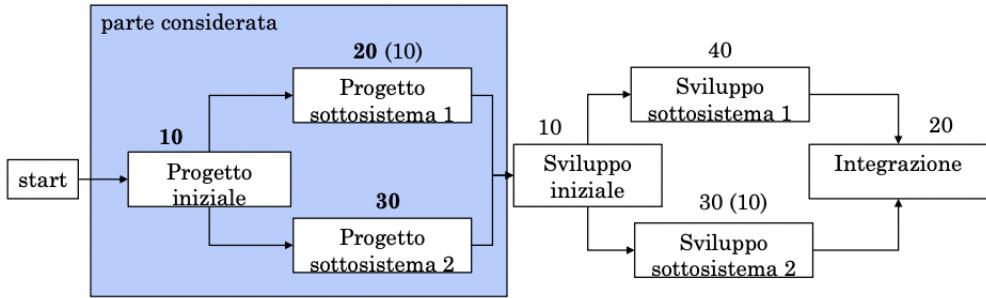
Un altro modello che possiamo considerare per la fase di pianificazione è il modello di Gantt è una sorta di calendario che traccia gli slittamenti e le durate



Le durate vengono calcolate con la media che abbiamo visto in precedenza. I primi due task, la rossa e la blu, sono in parallelo. Ad un certo punto, il terzo task rosso richiede

come task terminati i primi due per cui si vede che il secondo task lo abbiamo fatto slittare fino all'inizio del terzo task rosso.

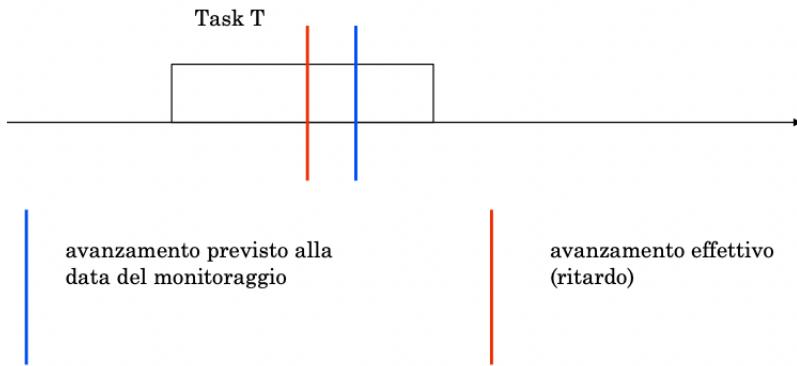
Con le tecniche che abbiamo visto abbiamo solo fatto il planning ma adesso dobbiamo fare la schedulazione del progetto. Supponiamo di avere solo due risorse richieste, programmatore e progettisti. Consideriamo la prima parte dello schema precedente



Supponiamo che il progetto iniziale impieghi 20 giorni di lavoro complessivi, allora servono 2 progettisti per terminarlo in 10 giorni. Per il progetto 1 basta 1 solo progettista se si suppone che il tempo complessivo sia di 20 giorni effettivi. Invece supponendo che il progetto 2 necessiti di 40 giorni allora si richiede di 2 progettisti ma non a tempo pieno. Si potrebbe allocare il progettista del progetto 1, che termina dopo 20 giorni, al progetto 2. Due varianti di allocazione dei progettisti potrebbero essere:

| | PT1 | PT2 | PT1 | PT2 |
|--|-------------------|-----------------|-------------------|-----------------|
| | Progetto iniziale | Progetto 1 P2 | Progetto iniziale | P2 Progetto 1 |
| | Progetto iniziale | Progetto 2 (P2) | Progetto iniziale | Progetto 2 |

Adesso dobbiamo partire con quello che si chiama Monitoraggio che è la fase in cui si verifica che tutto vada come previsto. Quindi si verifica che i task non siano in ritardo, come segue

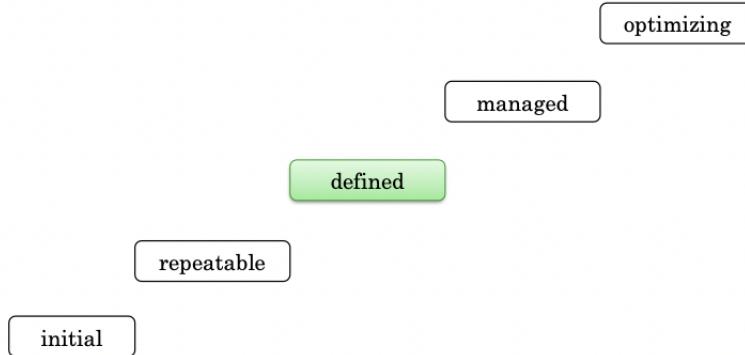


In una situazione di ritardo il Project Manager potrebbe decidere di allocare risorse, rivedere il planning o cose del genere.

16.2 Software process

16.3 CMM

Capability Maturity Model: suddivide i processi in 5 livello di maturità:



Sulla base del livello si ha una qualità maggiore

- **Initial:** Il processo è categorizzato come ah-hoc, nuovo e molto caotico. Pochi processi sono definiti
- **Repeatable:** C'è un po' di gestione del progetto in modo da poter gestire i costi, i tempi ecc. Si applica la consapevolezza di progetti precedenti simili
- **Defined:** Il processo è definito sia per management che per attività di ingegnerizzazione. È documentato, standardizzato e integrato in uno standard software process dell'organizzazione
- **Managed:** Misurazioni dettagliate del processo e della qualità. Sia la parte di processo che di prodotti sono stati capiti e controllati
- **Optimizing:** Si hanno dei feedback però si utilizza del tempo per provare delle tecnologie innovative da inserire nel progetto

16.3.1 CMMI

Presenta dei modelli che non sono altro che collezioni di best practices per migliorare i processi delle organizzazioni che ne fanno uso.

16.3.2 Process areas

Ogni livello che abbiamo visto in precedenza copre delle aree di processo differenti, in particolare copre tutto quello dei livelli precedenti più qualche altra cosa.

Chapter 17

Qualità del software

Esiste uno standard promosso dall'ISO/IEC 25010. Vi sono 8 caratteristiche. Con qualità di intende il grado di soddisfacimento che il sistema offre alle esigenze esplicite o implicite degli stakeholders

- functional suitability
- reliability
- performance efficiency
- usability
- security
- compatibility
- maintainability
- portability

17.1 functional suitability

- Completezza
- Correttezza
- Appropriatezza

Cioè definisce quanto il prodotto software sia conforme a quello richiesto

17.2 performance efficiency

- Comportamento temporali
- Utilizzazione delle risorse
- Capacità

17.3 compatibility

- co-esistenza
- Interoperabilità

17.4 Usability

Racchiude tutti i parametri per effettuare una valutazione dell'usabilità del prodotto considerando anche l'accessibilità, la facilità d'uso, ecc.

17.5 Reliability

- Maturità
- Disponibilità
- Fault Tolerance
- Recupero dei guasti

17.6 Security

- Confidenzialità
- Integrità
- Non ripudio
- Accountability
- Autenticità

17.7 Maintainability

- Modularità
- Riusabilità
- Capacità di analisi
- Modificabilità
- Testabilità

17.8 Portability

- Adattabilità
- Installabilità
- Rimpiazzabilità

17.9 Una nona caratteristica: dependability

Aggiunta dal processore, in pratica indica la *tranquillità* riguardo ai progetti software da cui il nostro software dipende. Cioè i software su cui dipendiamo dovrebbero soddisfare una buona parte dei requisiti precedenti

17.10 Misure

Per poter valutare i parametri precedenti bisogna effettuare delle misurazioni e talvolta si devono anche rispettare certi limiti. Per esempio è possibile che si richieda una availability del 99.9% e bisogna garantirlo perché si potrebbe incorrere a sanzioni.

Le metriche sono di due tipologie

17.10.1 Metriche interne

LOC - Line Of Code

Halstead ha definito una metrica più complessa che dice n_1 = numero di operatori e keyword distinti; N_1 = n. totale (con ripetizioni, non solo quelli distinti). n_2 = numero di operandi distinti; N_2 = n. totale. $n = n_1 + n_2$; $N = N_1 + N_2$

In un linguaggio di programmazione ad oggetti ci possiamo sbizzarrire, per esempio di potrebbe fare il numero medio per classe, il Fan-in/Fan-out (metodi chiamanti/chiamati), ecc.

Chapter 18

Organizzazione dello sviluppo del software

18.1 Agile Development

Secondo il manifesto, che risale al 2001, l'obiettivo è di muoversi in maniera agile dando libertà allo sviluppatore e diminuendo la parte burocratica. Si predilige l'interazione individuale più che interazioni tramite processi e tools. Si dice che si dovrebbe lavorare più sul software che sulla documentazione. Il cliente dovrebbe collaborare. Ci sono 12 principi del manifesto

1. *Customer satisfaction* by early and continuous delivery of valuable software
2. Welcome *changing requirements*, even in late development
3. Working software is *delivered frequently* (weeks rather than months)
4. Close, daily *cooperation between business people and developers*
5. Projects are built around *motivated individuals*, who should be trusted
6. *Face-to-face conversation* is the best form of communication (co-location)
7. *Working software* is the principal measure of progress
8. *Sustainable development*, able to maintain a constant pace
9. Continuous attention to *technical excellence and good design*
10. *Simplicity* - the art of maximizing the amount of work not done - is essential
11. Best architectures, requirements, and designs emerge from *self-organizing teams*
12. Regularly, *the team reflects* on how to become more effective, and adjusts accordingly

18.2 Extreme Programming

Sviluppo agile con pianificazione settimanale, requisiti basati su user stories, pair programming (programmare in coppia), test driven development, continuous integration,

refactoring.

18.2.1 Planning game

Per pianificare una release bisogna avere dei requisiti descritti in user stories con il coinvolgimento del committente. Analisi benefici/tempi/costi. Si devono scegliere, per ogni release, i requisiti da aggiungere in una release.

Poi abbiamo l'iteration planning che sarebbe la definizione dei task, allocazione dei task agli sviluppatori e implementazione con TTD.

User stories

Gli utenti scrivono le esigenze su schede con un formato molto semplice. Per esempio:
Come utente della biblioteca vorrei poter segnalare i miei interessi per ricevere aggiornamenti sui nuovi libri disponibili

Come amministratore vorrei poter avere informazioni di correlazione tra i libri presi in prestito in un dato periodo

Come utente della biblioteca vorrei poter prenotare il prestito di un libro dato in prestito

si noti che i casi d'uso visti a inizio corso con gli schemi UML sono generalmente più complessi in quanto descrivono le interazioni tra l'utente e il sistema.

Pai Programming

Si programma sempre in due. Ci sono dei benefici:

- Si aumenta la coesione del gruppo perché i partner cambiano, non sono sempre gli stessi
- Conoscenza distribuita
- Review informale ma continua

TTD e Integrazione continua

Per TTD si scrivono prima i test e poi il codice. Sappiamo già come funziona. Per integrazione continua si intende di integrare giornalmente il lavoro di tutti gli sviluppatori in una baseline condivisa(si intende la baseline definita nella spiegazione di del configuration management).

Refactoring

È la pratica di migliorare un componente software che a seguito di tante modifiche diventa disordinato e confusionario. Il refactoring punta proprio a migliorare quei componenti. Se si omette il refactoring si arriverà ad un punto per cui bisogna ricominciare da capo.

18.3 Scrum

È una metodologia che risale agli anni 90. Esiste una guida, Scrum Guide, disponibile in varie lingue. Il metodo illustra i principi e l'organizzazione del processo in termini di requisiti, fasi e ruoli.

Il prodotto evolve per incrementi detti **sprint**, gli sprint hanno lo scopo di rilasciare un incremento del prodotto potenzialmente rilasciabile. Uno sprint dura 1 mese nella versione standard.

18.3.1 Product Backlog

Nel PB si trovano i requisiti su cui poi si baseranno gli sprint.

18.3.2 Sprint Backlog

Sono i task che compongono lo sprint, scelti alla definizione dello sprint dal Product Backlog

18.3.3 Elemento fatto

I membri del team devono avere una comprensione condivisa di ciò che si intende per lavoro fatto. La definizione di lavoro fatto (DONE) può evolvere nel tempo. Cioè tutti i membri devono sapere quando un lavoro è definito DONE. Per esempio un elemento potrebbe essere fatto solo a seguito dei test di quel componente.

18.3.4 Sprint

Uno sprint prevede delle fasi

- **Sprint Planning:** Pianificazione della sola parte iniziale dello sprint
- **Daily Scrum:** Ogni membro del team illustra il lavoro del giorno precedente e quello che farà nel giorno corrente
- **Sprint Review:** Alla fine dello sprint, si discutono le difficoltà e se sono rimasti dei task non completati allora essi vengono rimessi nel Product Backlog
- **Sprint Retrospective:** Si fa una retrospettiva per evitare di effettuare nuovamente gli stessi errori negli sprint successivi e nei prossimi progetti

18.3.5 Ruoli e competenze

- **Product Owner:** Gestisce il product backlog, stabilisce l'ordine dei requisiti
- **Team:** Auto-organizzato e cross-funzionale, la dimensione varia tra 3 e 9
- **Scrum Master:** Si assicura che i principi di SCRUM(trasparenza, ispezione, adattamento) siano compresi. Supporta sia il product owner che il team.

Trasparenza: approccio noto a tutti, in particolare il significato di lavoro fatto

Ispezione: dei prodotti per rilevare anomalie o deviazioni dal piano.

Adattamento: correzioni per portare il prodotto entro limiti accettabili.

Chapter 19

DevOps

È un approccio che permette di unificare il team di sviluppo con quello di operation. Il team di operation si occupa di gestire l'infrastruttura su cui sarà messo in esecuzione il software. Occorre una tecnologia per automatizzare i processi che prima erano manuali.

19.1 Continuous Integration e Delivery

Si vuole comporre il sistema complessivo mediante l'integrazione di più sottosistemi sviluppati da più team. Se l'integrazione avviene dopo un lungo periodo di tempo esso potrebbe essere dispendiosa di tempo. Con DevOps l'integrazione si fa giornalmente attraverso test automatizzati e analisi statiche. L'integrazione è una pipeline di attività che procede solo quando lo stage corrente finisce con successo. Per esempio si potrebbe avere una fase della pipeline che fa gli unit test che nel caso in cui falliscano blocca tutti i successivi stage.

La pipeline viene allungata con la fase di **Continuous Delivery**. Il codice viene trasferito in uno staging environment che riproduce le caratteristiche dell'ambiente di produzione in cui vengono effettuati test sulle performance prima di portarla in produzione. Infine il programma viene portato in produzione.

Un esempio di prodotto potrebbe essere **Jenkins** che è open source scritto in Java e permette di effettuare l'esecuzione di una pipeline.

19.2 Deployment

Per portare il software in esecuzione si fa uso di strumenti come **Docker** e **Kubernetes**. Un container docker è una sorta di macchina virtuale che però condivide il kernel della macchina host. L'esecuzione di un docker non cambia anche al cambiare dell'ambiente di esecuzione. Ma dove finiscono le immagini docker? Per poter eseguire le immagini docker è necessario un docker engine che ha principalmente 2 parametri per rendere eseguibili una macchina:

- **build** serve per creare l'immagine docker

- **run** serve per eseguire l'immagine creata

Il docker engine in pratica è un demone di sistema che si occupa di eseguire i container lanciati con il comando **run**.

Le applicazioni possono essere anche multi-container e sparsi su più server. In questo caso, quando si ha a che fare con uno **swarm**, si può:

- Eseguire tutti i container in un solo server, tramite un solo comando
- Eseguire tutti i container su più server dove uno dei server è il master e gli altri sono sub-ordinati al master

Le principali caratteristiche dei container sono **scalabilità** e **load balancing**

19.3 Kubernetes

Offre una piattaforma molto articolata, configurabile, che orchestra tutti i contenitori, docker e altri. ..

ecc ecc

19.4 DevOps e Microservizi

DevOps offre un sistema naturale per i microservizi. Si può costruire una pipeline per ogni microservizio, più semplice di una pipeline di un'applicazione monolitica.