CORSO I.F.T.S.

"TECNICHE PER LA PROGETTAZIONE E LA GESTIONE DI DATABASE"

Matricola 2014LA0033

DISPENSE DIDATTICHE
MODULO DI "PROGETTAZIONE SOFTWARE"

Dott. Imad Zaza

Lezione del 14/07/2014

Progettazione

In dettaglio

Progettazione

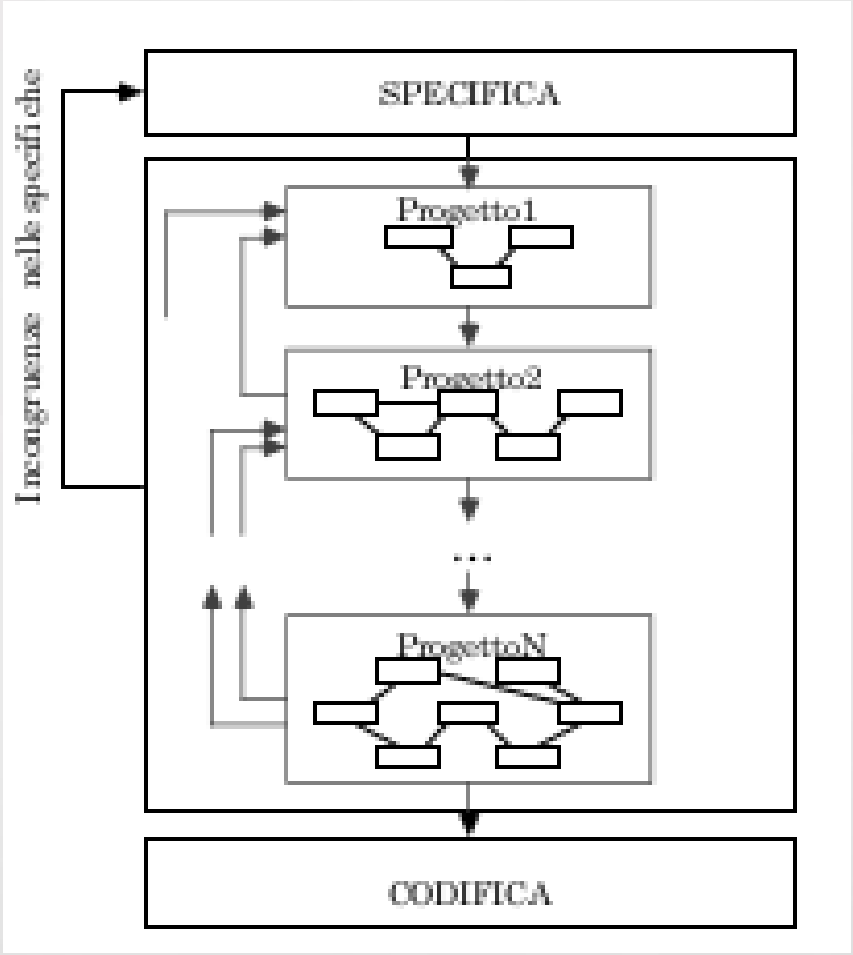
- Attività preliminare alla realizzazione di qualsiasi artefatto complesso.
- Scopi:
  - scomporre l’artefatto in parti
  - assegnare a ognuna una funzione
  - definire le loro interazioni
  - verificare che l’artefatto complessivo sia
- funzionale agli obiettivi

Definzione

- Attività che fa da ponte fra i requisiti e implementazione del software (dal “cosa” al “come”)
  - Definizione della struttura di tutti gli artefatti coinvolti nella produzione di software
    - Es.: il documento dei requisiti richiede una struttura che lo renda facile da capire e modificare

Progettazione SW

- Processo iterativo
  - Porta a scoprire incongruenze delle specifiche
  - Non c'è un livello di dettaglio definito prima di passare alla codifica
  - Si inizia con un'architettura.



Architettura Software

- Mostra la struttura globale e l'organizzazione del sistema
- Definisce:
  - i principali componenti del sistema
  - relazioni fra componenti
  - motivazioni della scomposizione scelta
  - vincoli sul progetto dei componenti

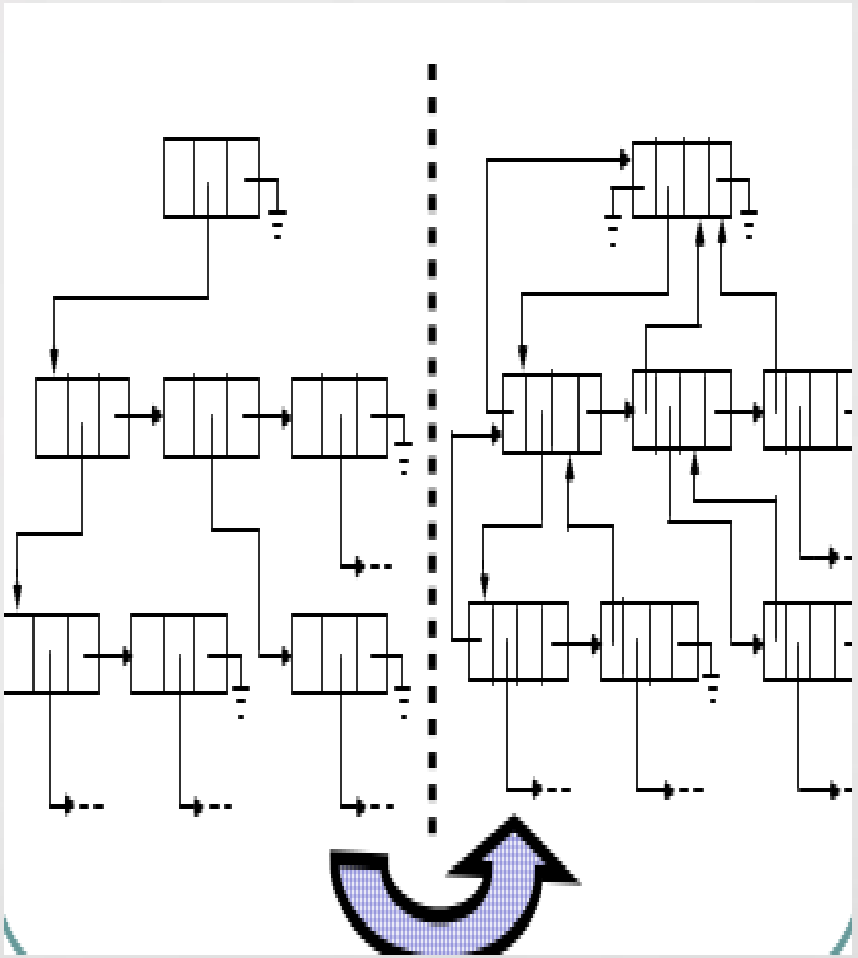
  - Guida la successiva progettazione

Scopo della progettazione

- Progettazione in vista del cambiamento
  - Prevedere i cambiamenti possibili
  - Soddisfare i requisiti attuali, ma in modo adattabile a nuovi requisiti
- Famiglie di prodotti
  - Progettare insieme diverse versioni di un prodotto permette di risparmiare sulla produzione delle parti comuni

Possibili cambiamenti

- Algoritmi
  - Possono essere necessari algoritmi diversi a seconda dell'applicazione (compromesso spazio/tempo, hardware disponibile, etc.)
- Rappresentazione dei dati
  - I cambiamenti hanno una ricaduta notevole sui costi di manutenzione
- Progettare in modo da astrarre dai dettagli



Possibili cambiamenti

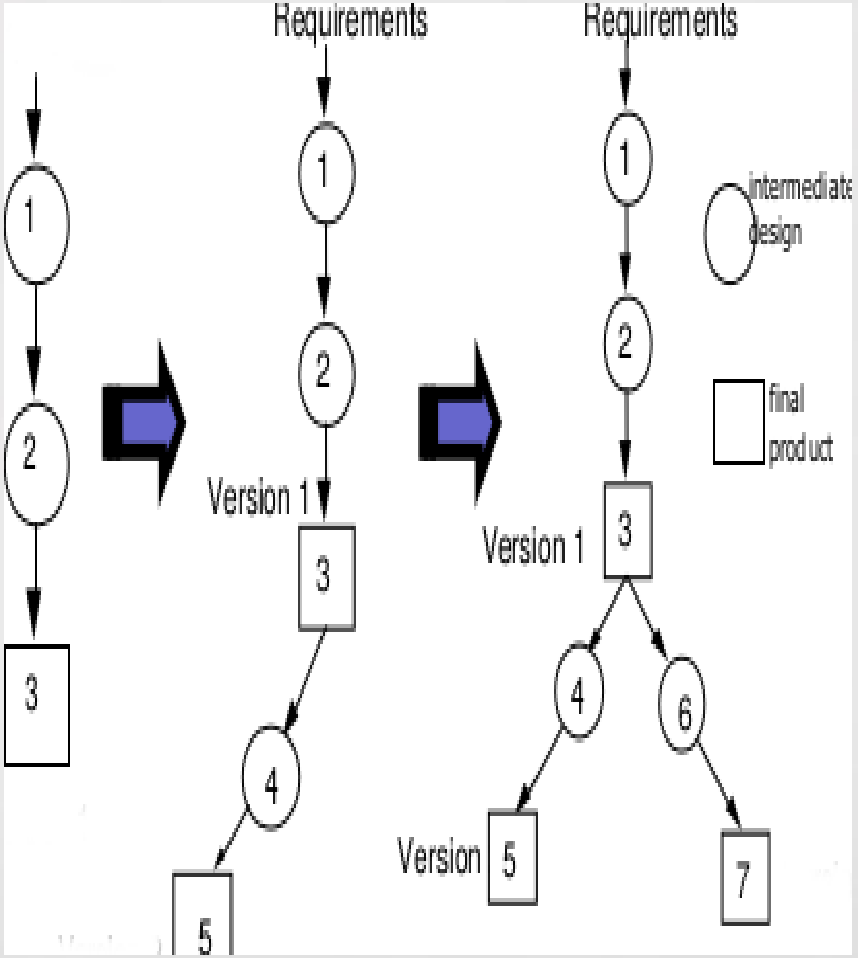
- Macchina astratta
  - Sistema operativo
  - Semantica del linguaggio di programmazione
- DBMS
- Ambiente sociale (leggi, moneta, etc.)
- Cambiamenti dovuti al processo di sviluppo

Famiglie di prodotti

- Versioni diverse dello stesso sistema
- Alcuni requisiti sono comuni, altri variano
- a seconda della versione
  - Es. famiglia di telefoni cellulari
    - Possono differire per standard di rete, lingue supportate, disponibilità di fotocamera, etc.
  - Es. sistema di prenotazioni
    - Hotel e università possono differire per risorse, intervalli, etc.

Progettazione sequenziale

- Realizzare il primo membro e modificarlo successivamente per ottenere gli altri
- Non soddisfacente:
  - Il progetto della famiglia è condizionato da scelte fatte per il primo membro
  - I progetti intermedi spesso non sono documentati e si procede direttamente sul codice



Progettazione corretta

- Durante la progettazione del primo membro, identificare
  - le parti comuni
  - quelle che caratterizzano i singoli membri
- Successivamente, riprendere dalla progettazione delle parti caratteristiche
- Questo metodo presuppone la modularizzazione

Modularizzazione

- Modulo: componente ben definito di un sistema software
- Fornitore di risorse computazionali o di servizi
  - Dati
  - Funzioni
  - Oggetti

Relazioni tra moduli

- Dipendenza funzionale (USES)
  - evidenzia la dipendenza di un modulo dalle funzionalità esportate da un altro
- Scomposizione (IS_COMPONENT_OF e derivate):
  - indicano come un modulo si suddivide in altri di dimensioni minori

Uses

- E’ utile che USES sia una gerarchia
  - Un sistema strutturato gerarchicamente è più facile da
    - capire
    - implementare
    - testare
- partendo dalle foglie e andando verso l’alto.

Livelli di astrazione

- I moduli di livello x forniscono una macchina astratta ai moduli di livello maggiore di x.
  - Macchina astratta: insieme di servizi a cui si accede astraendo dalla loro implementazione

Interfaccia e implementazione

- La distinzione fra interfaccia e implementazione è un principio fondamentale della progettazione software.
  - Permette a chi implementa il modulo di concentrarsi solo su esso e di astrarre dall'implementazione dei server
    - L'interfaccia è un contratto fra l'implementatore e l'utente del server.
    - L'implementazione può cambiare senz conseguenze per i client.

Come progettare l'interfaccia di un modulo?

- Compromesso fra
  - necessità di esportare più servizi per incrementare la funzionalità
  - necessità di esportare meno servizi per mantenere semplice l'interfaccia
- E' meglio non esportare qualcosa che può cambiare.

Esempio

- Sistema di controllo di un impianto
  - Raccoglie dati fisici e li archivia in uno storico a seconda dei valori restituisce segnali di controllo per l’impianto
  - Si prevede di voler integrare, in futuro, funzionalità di interrogazione e analisi statistica sui dati

Esempio

- Modulo per l’acquisizione dell’input che nasconde il modo in cui i dati vengono raccolti e restituisce i valori
- Modulo per l’accesso allo storico che nasconde la rappresentazione dei dati
- Modulo per l’interpretazione delle interrogazioni che ne restituisce una versione astratta

Modulo per sviluppo incrementale

- Sviluppo incrementale:
  - Identificare subito un nucleo di sottoinsiemi utili (e urgenti) dell'applicazione
  - Definire con precisione le interfacce dell parti che vengono rimandate!
  - Possibilità di sviluppare versioni semplificate di alcune parti

Politiche

- Una stessa operazione può essere effettuata secondo politiche diverse (es. scheduling di processi)
  - Se la politica non è rilevante per il client può essere utile incapsularla, in modo da poterla cambiare se è il caso
  - In certe applicazioni può però essere necessario esporla (RT/scheduling)

Notazione

- Necessità di notazioni non ambigue
- Non è ancora emersa una notazione standard
  - UML
  - Due notazioni Vecchie:
    - TDN, testuale
    - GDN, grafica

TDN

- Ispirata a linguaggi di programmazione modulari come Ada o Modula-2
  - Parzialmente non specificata
  - Un modulo può esportare qualsias risorsa:
    - variabile
    - tipo
    - procedura
    - funzione...

TDN

- I commenti forniscono
  - informazioni di natura semantica sui servizi esportati
  - la definizione del protocollo, ossia delle regole che il modulo client deve seguire per utilizzare i servizi
    - inizializzazione
    - propedeuticità
    - informazioni di natura non semantica

TDN

- uses indica i moduli server
  - exports indica le risorse esportate (interfaccia)
  - implementation definisce ad alto livello l'implementazione: utile per
    - comprendere il funzionamento
    - guidare l'implementazione effettiva
    - commenti informali sulle parti
    - incapsulate e sulla motivazione

Esempio

- **module** X
- **uses** Y, Z
- **exports var** A : **integer**;
- **type** B : **array** (1. .10) of real;
- **procedure** C (D: **in out** B; E: **in integer**; F: **in real**);
- Quello che descritto in linguaggio naturale
- bla bla bla

- **implementation**
- Dettagli implementative sempre in linguaggio naturale
- **is composed off** R, T
- **end** X

GDN

- Notazione grafica corrispondente a TDN
  - Modulo indicato da un rettangolo
  - Le risorse esportate sono indicate su frecce entranti
  - Il rettangolo corrispondente a un modulo contiene i suoi componenti
- Non ci interessa ! Ci piace UML !

Verifica

In dettaglio

Validazione e Verifica

- verifica: il prodotto (o ogni artefatto intermedio) è costruito nel modo giusto (cioè rispetta la sua specifica)
- validazione: il prodotto è quello giusto (cioè rispetta i suoi veri requisiti)?
- Useremo solo il termine “verifica”.

Necessità della verifica

- Ogni fase del processo di produzione del
- software è soggetta a errore.
-
- Quindi è necessario verificare
-
- ogni artefatto (dal documento dei requisiti al
- codice sorgente)
-
- ogni fase del processo, compresa la verifica
- stessa.

Scopo della verifica

- Lo scopo dell'attività di verifica è convincere della bontà del prodotto (finale o intermedio) chi è interessato ad esso.
- Le risorse e le tecniche (formali, sistematiche, informali) da impiegare dipendono dal tipo di prodotto e dal suo valore.

Risultati non binari

- Difficilmente il risultato della verifica è un sì o un no.
  - La certezza assoluta del risultato di una verifica è praticamente impossibile da raggiungere, e non sarebbe comunque sufficiente.
  - Nella prassi, un certo grado di noncorrettezza è accettabile e accettato.

Verifica oggettiva e soggettiva

- Non tutte le qualità sono facilmente
- misurabili.
- Verifica oggettiva:
- osservazione della risposta a input e
- confronto con la risposta corretta
- dimostrazione matematica di una proprietà
- Verifica soggettiva:
- usabilità
- riusabilità

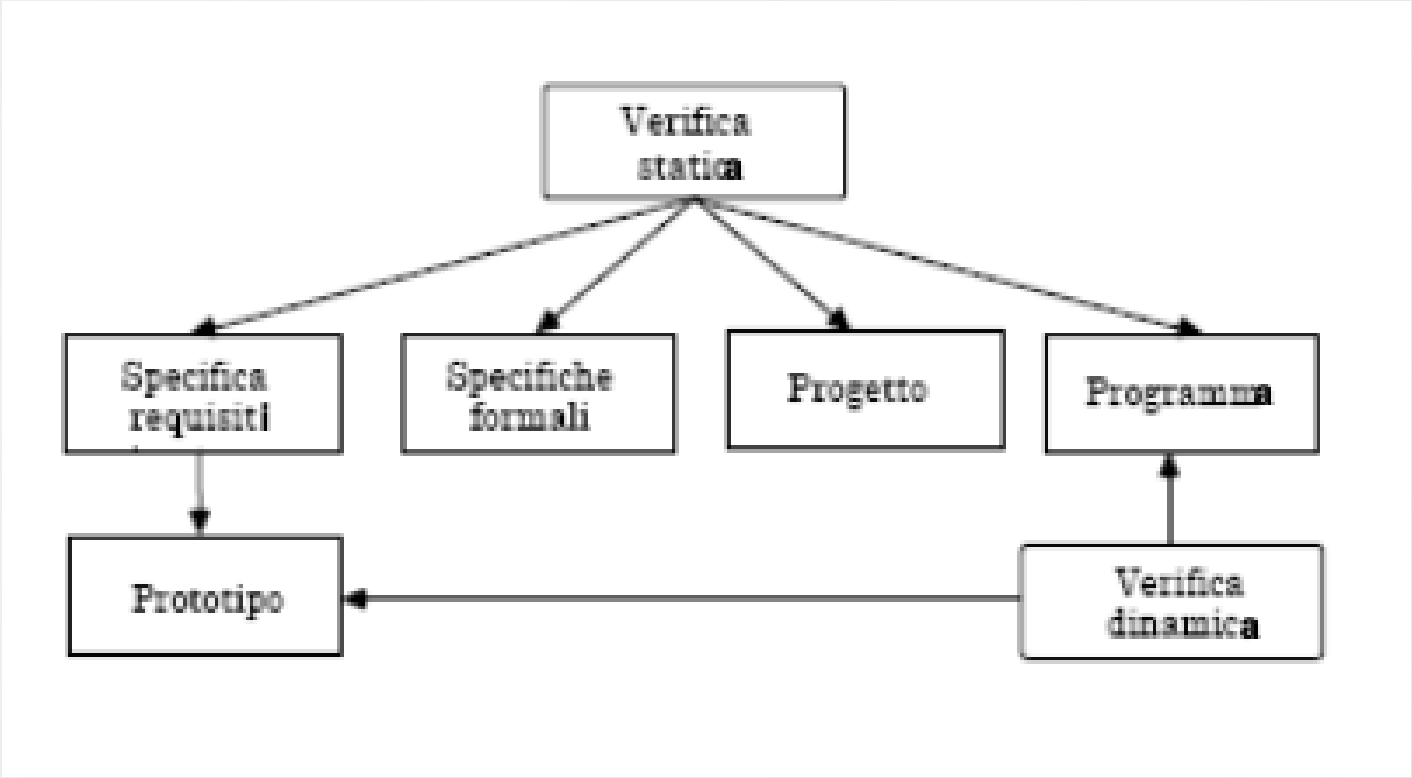
Verifica di qualità implicite

- La bontà di un prodotto consiste anche nel rispetto di requisiti non specificati, perché impliciti.
  - Robustezza
  - Prestazioni
  - Manutenibilità

Approcci alla verifica

- Dinamico (test):
  - Osservazione del comportamento del prodotto e confronto con i suoi requisiti
  - Necessità di approccio sistematico
- Statico (analisi):
  - Osservazione del prodotto e ragionamento sulle sue proprietà
  - Approcci complementari

Verifica statica e dinamica: applicabilità



Test (o analisi dinamica)

- Consiste nell'osservare il comportamento del sistema in un certo numero di condizioni significative
  - Non può (in generale) essere esaustivo
  - Scelta di un insieme finito di dati di test
    - La scelta discende da considerazioni sul programma e/o sul problema

Mancanza di continuità

- Il test in un numero finito di casi non può portare a estrapolare conclusioni sulla correttezza del programma.
  - Mancanza di continuità: un programma che funziona correttamente in un caso può presentare malfunzionamenti anche in un caso leggermente diverso

Test in piccolo e in grande

- Test in piccolo: dei singoli moduli.
  - Approcci:
    - White box (o strutturale o trasparente): test set scelti in base all'implementazione.
    - Black box (o funzionale): test set scelti in base alla specifica.
- Test in grande: organizzazione e suddivisione dell'attività di test in base alla struttura di sistemi complessi.

Obiettivi dell’attività di test

- Dimostrare la presenza di errori, non la loro assenza (Dijkstra).
  - Il successo del test non implica la correttezza del programma.
  - Dovrebbe essere eseguita secondo metodi sistematici, e integrata con altre tecniche (analisi statica).

Obiettivi dell’attività di test

- Dovrebbe aiutare a localizzare gli errori, non solo rilevarli, per facilitarne la correzione.
  - Dovrebbe essere ripetibile
  - Influenza dell’ambiente di esecuzione
  - Non-determinismo
  - Dovrebbe essere accurata (utilità di specifiche formali)

Criteri di selezione per test strutturali (white box)

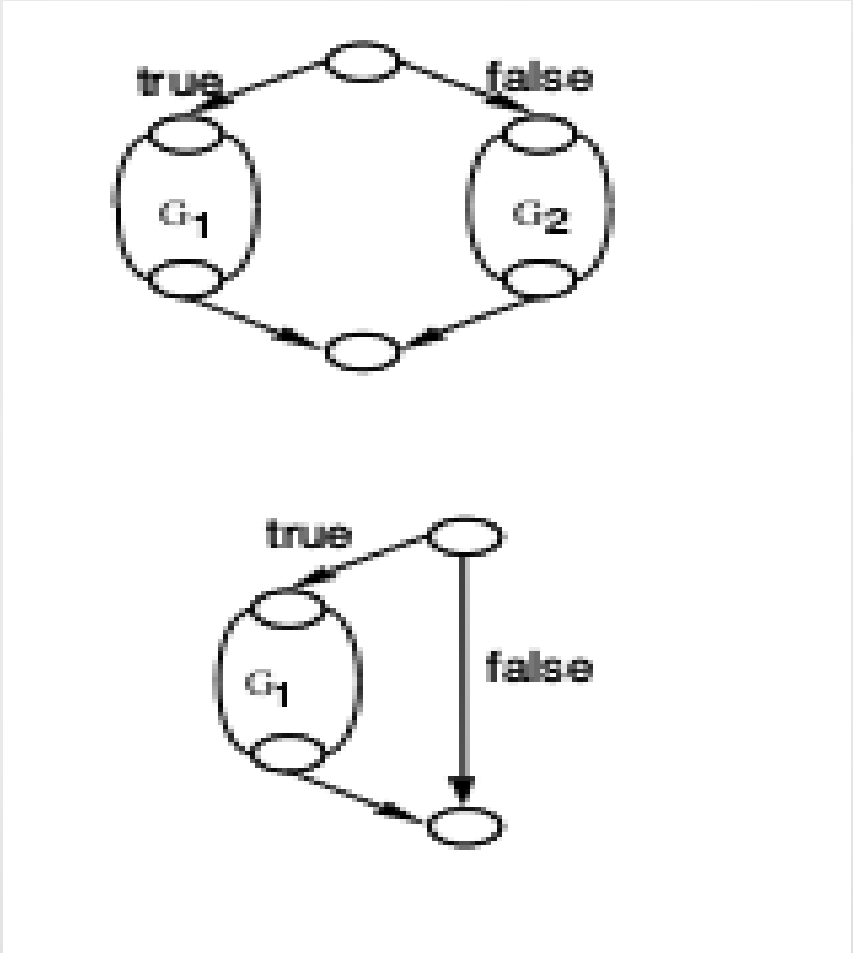
- Copertura delle istruzioni
  - Copertura delle decisioni (o degli archi)
  - Copertura delle condizioni
  - Copertura dei cammini

Grafo di controllo

- Costruito ricorsivamente
  - Ogni istruzione elementare è rappresentata da un nodo
- Siano S1 e S2 due istruzioni, e G1 e G2 i corrispondenti grafi. Allora:

Grafo del If

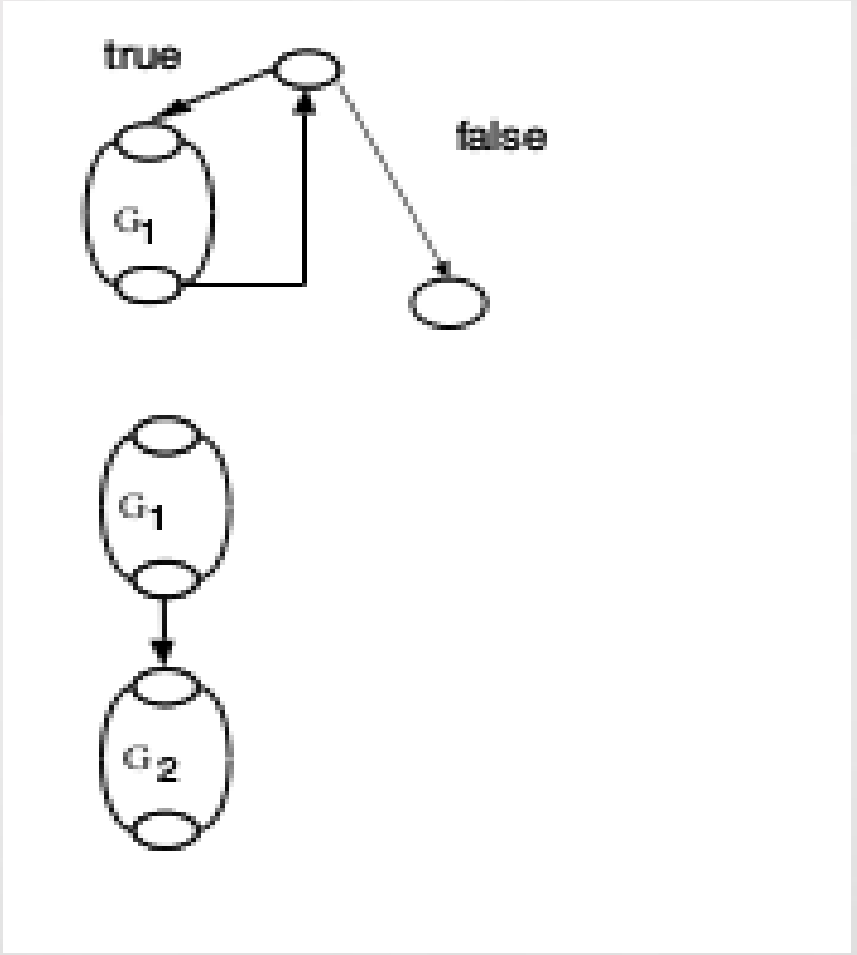
```
if cond then
        s1;
else
        s2;
end if

if cond then
   s1;
end if
```



Grafo del while e sequenza

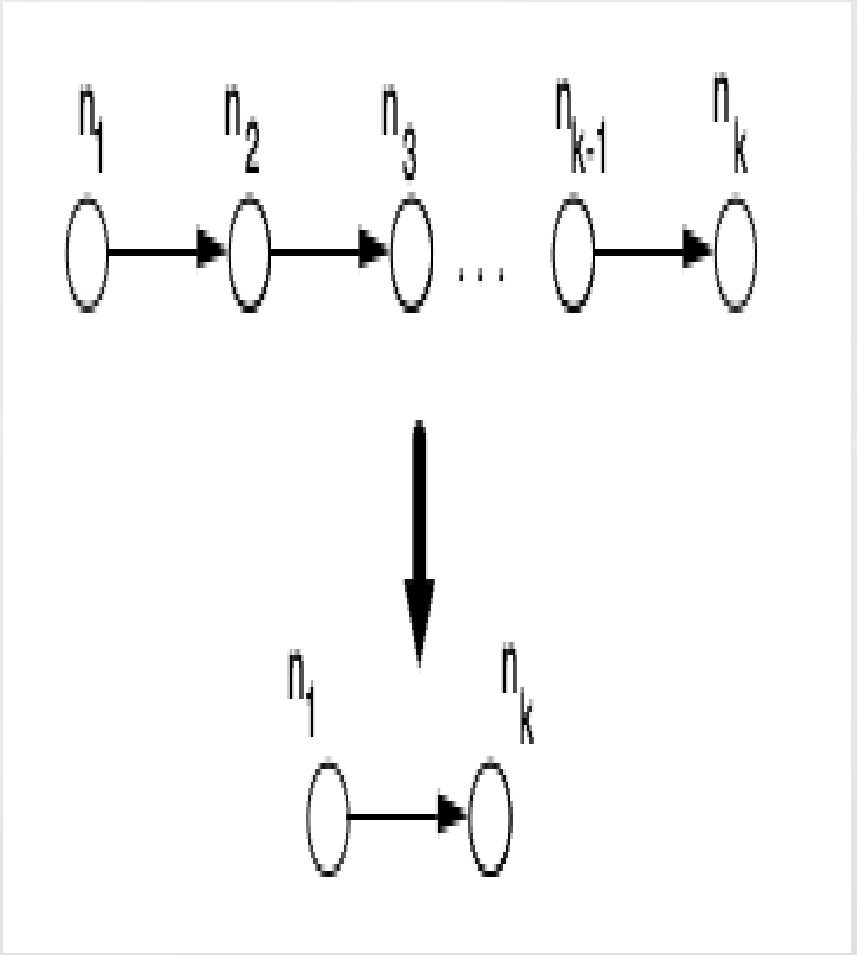
```
while cond loop
   s1;
end loop
```

```
s1;
s2;
```



Semplificazioni

- Una sequenza di archi si può collassare in un solo arco:



Criterio di copertura delle istruzioni

- Scegliere i casi di test in modo da eseguire, compleplessivamente, tutte le istruzioni elementari del programma almeno per un caso di test.
  - Solitamente si fa riferimento alla grammatica del programma e si considera istruzione tutto ciò che può essere derivato dal costrutto <statement>: ad esempio, nei linguaggi imperativi, assegnamenti, operazioni di I/O e chiamate di subroutine.
  - Misura di copertura C0 = <numero di istruzioni eseguite> / <numero di istruzioni eseguibili>

Esempio

```
leggi(x); leggi(y);
finchè  x != y
   se x > y allora
        x:= x – y;
   altrimenti
        y:= y – x;
gcd := x;
```

- Si ottiene opertura delle istruzioni (C0=1) con il test set {(3,3), (4,3), (3,4)}

Test funzionali (black box)

- Il test white-box non rileva la mancata implementazione di funzionalità.
  - Test basati sulla specifica di un programma, anziché sulla loro implementazione.
  - I test set si individuano in base a casistiche ricavate dalla specifica (formale o informale).

Specifica

- Il programma riceve in ingresso una registrazione che descrive una fattura (viene fornita una descrizione dettagliata del formato della registrazione).
- La fattura deve essere inserita in un file di fatture ordinato per data.
- La fattura deve essere inserita nella posizione corretta del file. Se esistono altre fatture che riportano la stessa data, la nuova fattura deve essere inserita dopo l'ultima già presente.
- Inoltre devono essere eseguite alcune verifiche di coerenza: il programma dovrebbe verificare se il cliente è già presente nel corrispondente archivio dei clienti, se i dati del cliente nei due file corrispondono, etc.

Approccio intuitivo al test

- Si fornisce una fattura la cui data corrisponde alla data corrente
- Si fornisce una fattura la cui data precede la data corrente (illegale?). Sottocasi:
  - Fattura con stessa data di una fattura già presente
  - Fattura con data diversa da tutte quelle già presenti
  - Si forniscono diverse fatture scorrette

Tecniche sistematiche blackbox

- Test guidato da specifiche logiche
  - Test guidato da sintassi
  - Test basato su tavole di decisione
  - Test basato su grafi causa-effetto

Test in grande

- Molte delle tecniche viste finora hanno alta complessità, o non sono automatizzabili.
- Possono quindi essere applicate solo a programmi piccoli, o a porzioni di codice.
- Il test di programmi di grandi dimensioni richiede tecniche per affrontare la complessità, esattamente come la progettazione e la specifica.

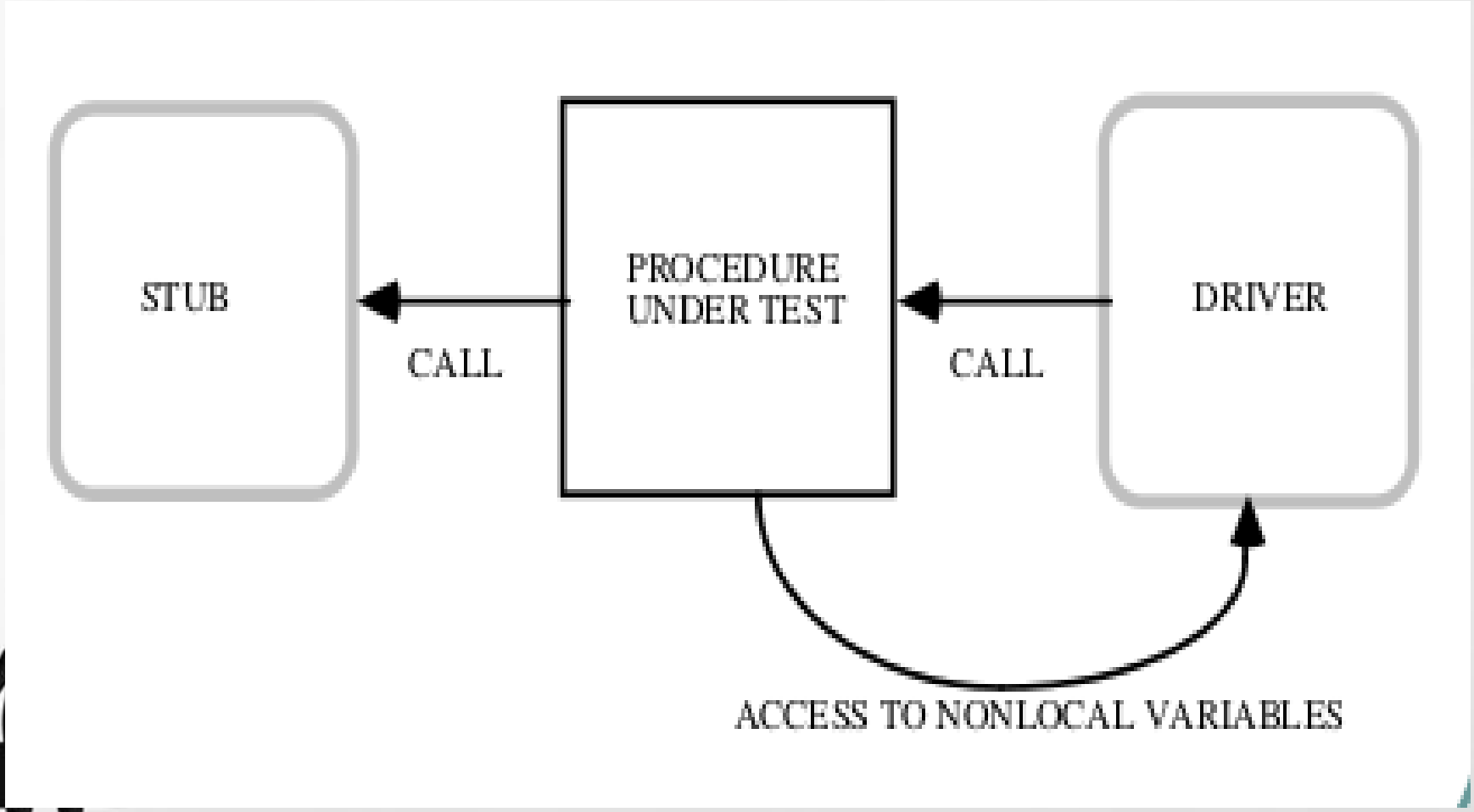
Test e modularità

- La modularità di un progetto guida l'attività di test.
  - Attività:
    - Test di modulo (singolo)
    - Test di integrazione: interazioni fra moduli
    - Test di sistema (complessivo)
    - Test di accettazione: eseguito dal cliente

Test di modulo

- Un modulo non può essere testato senza le funzionalità che importa.
  - Se un modulo Mi richiede Mh e Mk (Mi USES Mh, Mi USES Mk), e questi non sono disponibili, è necessario simulare le loro risorse.
  - Inoltre può essere necessario testare la chiamata di Mi da parte di un altro modulo.
  - Scaffolding (ponteggio): ambiente per il test di modulo.

Test di un modulo funzionale



Stub

- Un modulo necessiti di una funzione non ancora sviluppata
  - Si può costruire uno stub, ossia una funzione con gli stessi parametri di input e output, ma che simula il calcolo utilizzando un oracolo (come un file o un operatore umano).
  - Utilità di specifiche eseguibili (es. logiche)

Driver

- Programma che simula l'invocazione del modulo sotto test.
  - Procede all'inizializzazione del modulo sotto esame e alla chiamata di operazioni significative e in sequenze significative.

Test di integrazione

- Test big-bang:
  - prima tutti i moduli separatamente poi l'intero sistema
  - Si salta la fase di test di integrazione.
  - Difetto: i problemi derivati dall'interazione fra moduli si presentano tutti insieme.

Test incrementale

- Vantaggi:
  - si può anticipare il test di integrazione a quando sono pronti i moduli interessati
  - è più facile localizzare gli errori
  - è opportuno testare insieme moduli correlati
  - un'aggregazione parziale di moduli può costituire una fornitura parziale (anche per test di accettazione)
  - può ridurre la necessità di stub e driver

Test bottom-up e top-down: relazione USES

- Se la relazione USES è una gerarchia:
  - Bottom-up: inizio del test dalle foglie e sviluppo di driver
  - Top-down: inizio del test dalla radice e sviluppo di stub.

Test di software object-oriented

- Problemi specifici
  - Ereditarietà
  - Genericità
  - Polimorfismo
  - Binding dinamico
  - Il test di software OO è un argomento di ricerca ancora non assestato.

Ereditarietà e test

- L’ereditarietà permette il riuso di codice mediante specializzazione.
- Anche i test, quando possibile, andrebbero riusati.
  - In linea di massima sarebbe opportuno:
    - non ripetere il test degli elementi (metodi) invariati rispetto alla superclasse
    - testare i nuovi metodi e quelli ridefiniti
    - Possono però esserci interazioni.

Strategie per il test di una gerarchia

- Considerare ogni classe come unità indipendente?
  - Si perde l'incrementalità
  - Annotare manualmente i casi di test:
    - Test che non va ripetuto per nessun erede
    - Test che va ripetuto per la classe erede X e tutti i suoi eredi
    - Test che va ripetuto con gli stessi input
    - Test che va ripetuto modificando gli input

Test di sistema

- Spesso coinvolge componenti non
- software (hardware, operatori)
- A seconda della struttura del sistema, la
- modifica di un componente può
- comportare o meno un nuovo test di
- sistema
- Tecnica dello scaffolding non limitata al
- software

Altri tipi di test

- Non solo correttezza funzionale
  - Test di tutte le qualità:
  - Test di resistenza al sovraccarico
  - Più in generale, test di robustezza
  - Test di regressione
  - Le modifiche al software possono introdurre nuovi errori o manifestarne di già esistenti
  - Tenere traccia dei test su tutte le versioni del software per poterli ripetere in caso di modifiche.

UML e OO

Notazioni e Nozioni

Perché modelliamo

Riassunto

I **modelli**

- ci aiutano a “visualizzare” un sistema come è o come vorremmo che fosse
- ci permettono di specificare la struttura o il comportamento di un sistema
- ci forniscono un “template” che ci guida nella costruzione di un sistema
- documentano le decisioni che abbiamo preso

Perché modelliamo

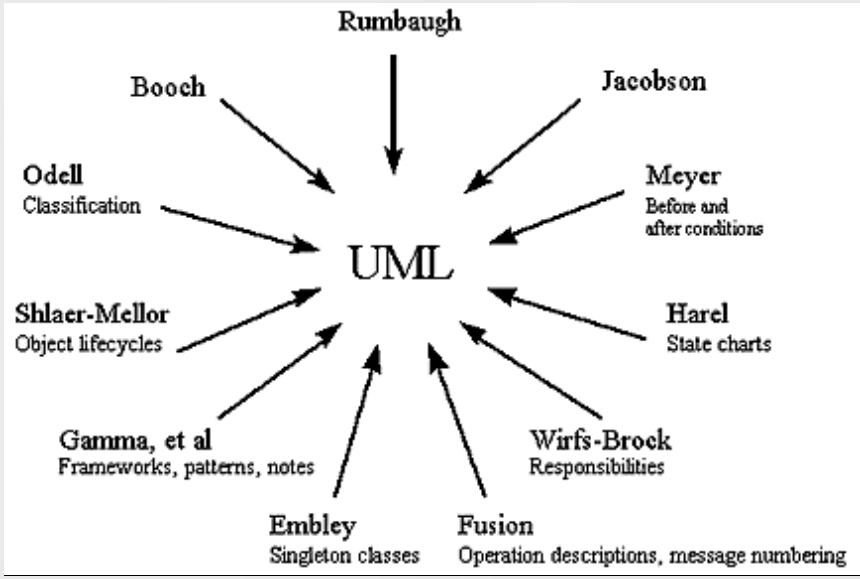
- *Divide et impera*: tramite i modelli ci focalizziamo su un solo aspetto alla volta
- Ogni modello può essere espresso a differenti livelli di precisione
- per un sistema non banale:
  - ⇒ non un solo modello
  - ⇒ ma un piccolo insieme di modelli, che possono essere costruiti e studiati separatamente, ma che sono strettamente interrelati

Unified Modeling Language

- un linguaggio (e notazione) universale, per la creazione di modelli software
- nel novembre ‘97 è diventato uno **standard** approvato dall’**OMG** (Object Management Group)
- numerosi i co-proponenti: Microsoft, IBM, Oracle, HP, Platinum, Sterling, Unysis …..

Unified Modeling Language

- è l'unificazione dei metodi:
  - Booch-93 di **Grady Booch**
  - OMT di **Jim Rumbaugh**
  - OOSE di **Ivar Jacobson**



- ha accolto inoltre le idee di numerosi altri metodologie

Storia

- 1994 Prima versione
- 1997 Accettato da Object Management Group (OMG) come release UML 1.1.
- 2001 Versione UML 1.4
- 2004 Versione UML 2
  - Interchange
  - OCL2 Specification
  - UML 2.0 Infrastructure
  - UML 2.0 Superstructure

UML

- Cos'è:
  - Linguaggio di modellazione visuale, non proprietario
    - sintassi e semantica
    - Sintesi di notazioni usate in passato (ER, State chart, OMT, DFD, ..)

UML - linguaggio universale

- linguaggio per specificare, costruire, visualizzare e documentare gli artefatti di un sistema
- universale: può rappresentare sistemi molto diversi, da quelli web ai legacy, dalle tradizionali applicazioni Cobol a quelle object oriented e a componenti

UML non è un metodo

- è un linguaggio di modellazione, non un metodo, né una metodologia
- definisce una notazione standard, basata su un meta-modello integrato degli "elementi" che compongono un sistema software
- non prescrive una sequenza di processo, cioè non dice "prima bisogna fare questa attività, poi quest'altra"

UML e Processo Software

> "un unico processo universale utilizzabile per tutti gli stili di sviluppo non sembra possibile e tanto meno desiderabile"

In realtà UML assume un processo:

- basato sui Casi d'Uso (*use case driven*)
- incentrato sull'architettura
- iterativo e incrementale

I dettagli di questo processo di tipo generale vanno adattati alle peculiarità della cultura dello sviluppo o del dominio applicativo di ciascuna organizzazione

Modellare in UML

- Un modello UML è fatto di diagrammi (di diverso tipo)
  - esprimono entità e relazioni tra le diverse entità (oggetti del mondo reale, costrutti software, comportamenti..)
  - di norma un tipo di diagramma corrisponde a una vista

I modelli UML

- Per catturare e identificare esattamente i requisit ed il dominio dimodo da mettere in condizione la compresione da parte di tutti gli elementi coinvolti (stackholders)
- Di pensare in termini di progetto di sistema
- Di catturare decisioni progettuali in maniera immutabile e separata dai requisiti
- Di organizzare, trovare, filtare, esaminare e modificare le informazioni inerenti a grandi sistemi
- Di vagliare una miriade di soluzioni in maniera economica
- Di gestire sistemi complessi

Assestment

- In UML c’è tutto e di più: può essere usato in più modi secondo fini diversi
- Praticamente ogni cosa in UML è opzionale
- Massima libertà su come usare i costrutti del linguaggio
- Estensibile (stereotipi, profili)
- Può essere usato per generare codice e casi di test (MDA)

Viste e Diagrammi

- Una vista è un sottoinsieme di costrutti UML che mette in evidenza un aspetto del sistema:
  - Vista statica
    - diagramma delle classi, di deployment, ..
  - Vista dinamica
    - macchine a stati, diagramma di sequenza,..
  - Vista dei casi d'uso
    - diagramma dei casi d'uso
  - Vista fisica
    - diagr. componenti, diagr. deployment
  - Vista logica (progettuale)
    - diagr. classi, sequenza, attività, ...

Diagrammi UML

livello "logico":

dei casi d'uso	**Use Case Diagram**
delle classi	**Class Diagram**
di sequenza	**Sequence Diagram**
di collaborazione	**Collaboration Diagram**
di transizione di stato	**Statechart Diagram**
delle attività	**Activity Diagram**

livello "fisico":

dei componenti	**Component Diagram**
di distribuzione dei componenti	**Deployment Diagram**

Diagramma dei casi d’uso

Mostra:

- le modalità di utilizzo del sistema (casi d’uso)
- gli utilizzatori e coloro che interagiscono con il sistema (attori)
- le relazioni tra attori e casi d’uso

Un **caso d’uso**

- rappresenta un possibile “modo” di utilizzo del sistema
- descrive l’interazione tra attori e sistema, non la “logica interna” della funzione

una funzionalità dal punto di vista di chi la utilizza

Perche' costruire modelli dei casi d'uso

- Primo momento nel ciclo di vita del software in cui costruiamo delle rappresentazioni (talvolta semi-formali) del software.
- Motivazioni:
  1. Esplicitare e comunicare a tutti gli stakeholder i requisiti funzionali del sistema– In generale, analizzare, identificare, descrivere gli usi tipici del sistema da parte dei suoi utilizzatori. Considerare anche tutta l'eventuale logica derivante dalla gestione di errori, eccezioni, flussi alternativi.
  2. Validare i requisiti utente – Essendo il modello dei casi d'uso piuttosto semplice (pochi costrutti, semantica precisa, ma intuitiva), diventa un valido strumento per assicurarci di aver sviluppato un sistema software corrispondente alle vere necessita' dell'utente.

Cosa è un use case

- Un use case cattura chi (attori) fa cosa (interazione) con il sistema. Con quale scopo (goal), senza considerare ciò che è dentro il sistema

- Un use case
  - Raggiunge un singolo, discreto, completo, significativo, e ben definito task di interesse per un attore
  - È un pattern di comportamento tra alcuni attori ed il sistema — una collezione di possibili scenari
  - È scritto usando il vocabolario del dominio applicativo
  - Definisce scopo ed intento (non azioni concrete)
  - È generale e indipendente dalla tecnologia

Use case diagram e requisiti

- I casi d'uso servono a:
  - chiarire i requisiti del committente in termini comprensibili
  - trovare aspetti comuni (riuso)
  - individuare gli attori del sistema
  - individuare gli eventi a cui il sistema deve rispondere

- Un requisito può dare origine a più casi d'uso
- Ad un caso d'uso possono venire associati più requisiti

Casi d'uso, attori, scenari e descrizioni

- Quattro concetti fondamentali:
  1. Caso d'uso – rappresenta una specifica interazione tra un attore e il sistema.
     In UML rappresentato mediante un'ellisse etichettata col nome del caso d'uso.
  2. Attore – rappresenta un *ruolo* che caratterizza le interazioni tra utente e sistema. L'utente non è necessariamente umano; In UML rappresentato mediante un omino.
  3. Scenario – sequenza di *azioni* che definisce una particolare interazione tra attore e sistema.
  4. Descrizione – *testo* che descrive lo scenario, l'ordine temporale della sequenza di azioni, l'eventuale trattamento degli errori, gli attori coinvolti.

Attori

Gli attori sono rappresentati tramite il **ruolo** che giocano nel caso d'uso e sono normalmente indicati tramite l'icona:

Relazione tra attori

- E’ possibile definire gerarchie di attori
- Associare un attore ad uno o più attori specializzati

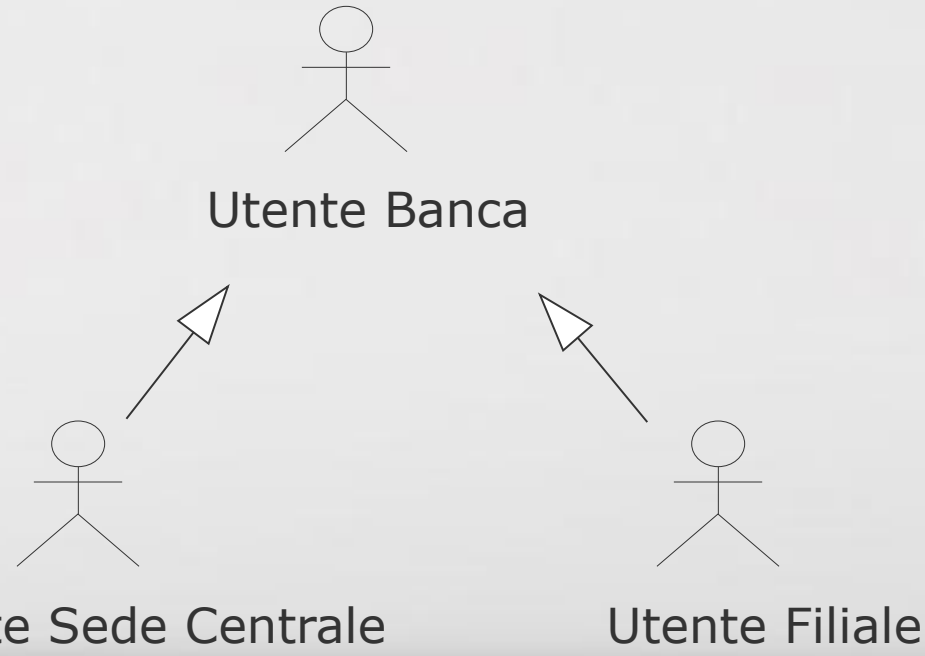
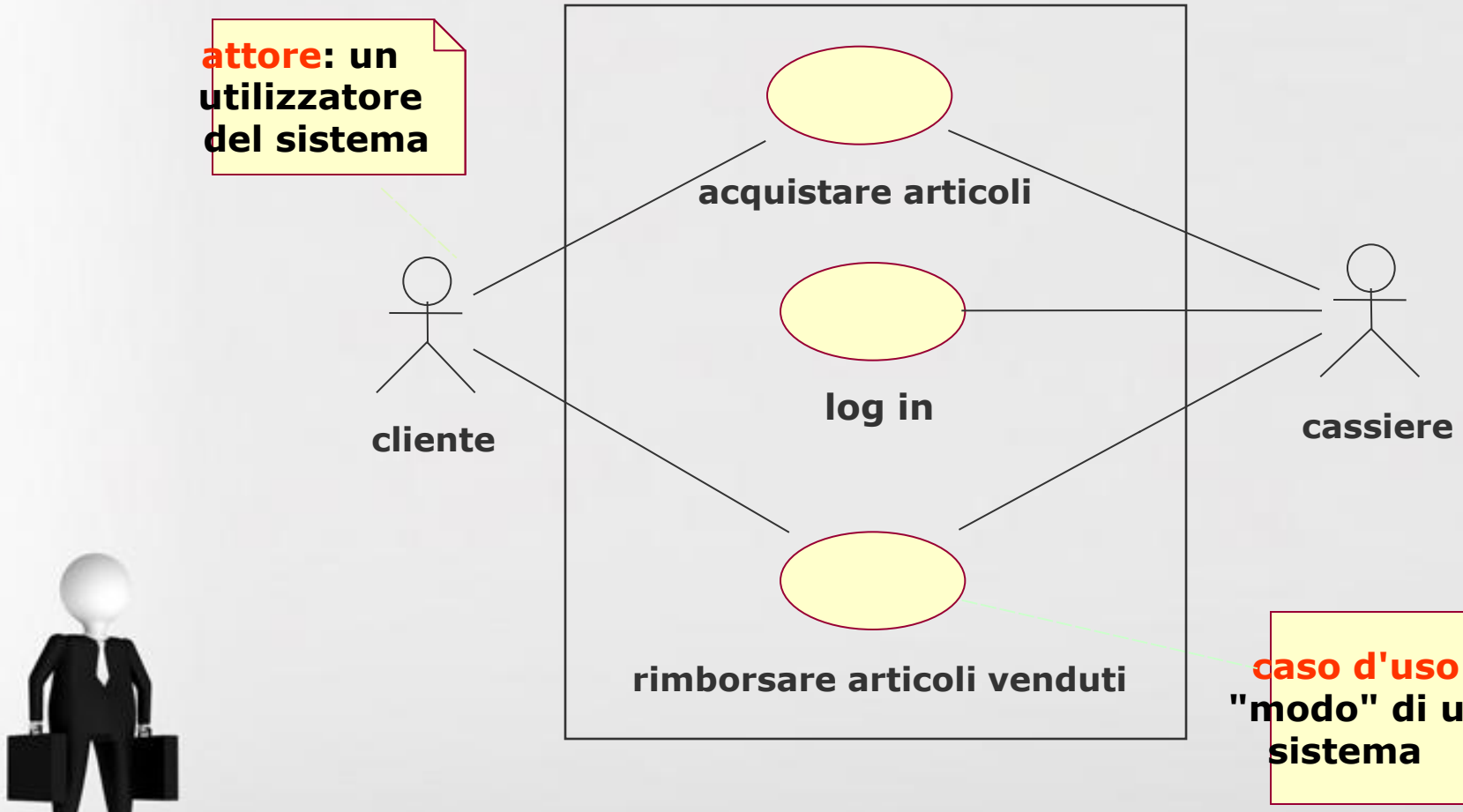


Diagramma dei casi d'uso



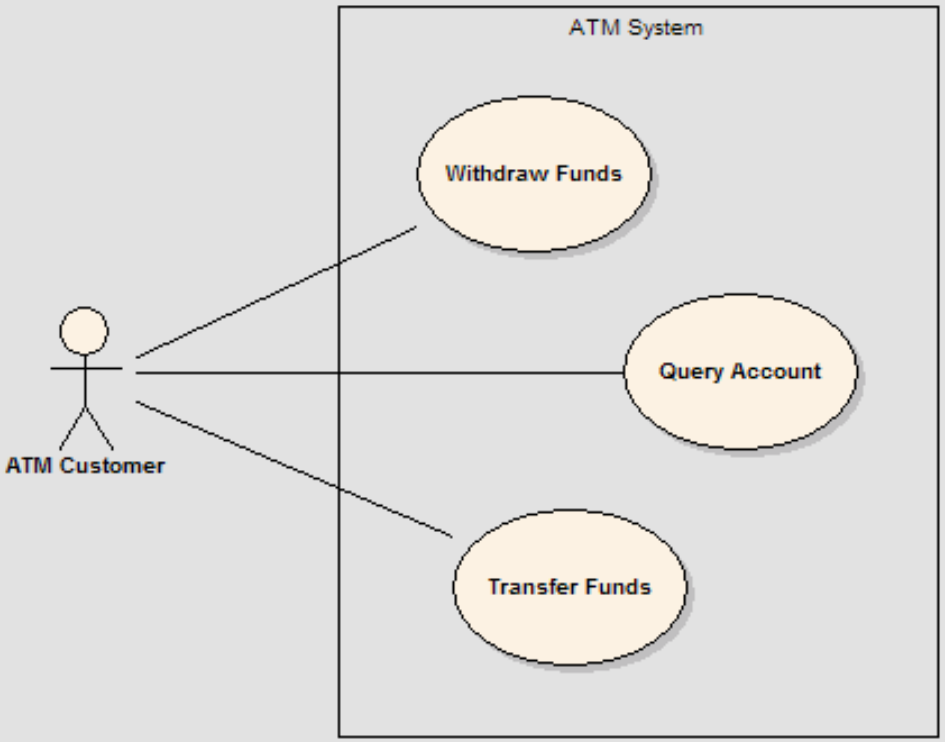
Template per la documentazione di un caso d'uso

TEMPLATE DI BASE

- *Nome caso d'uso* – Ogni caso d'uso deve avere un nome; il nome esprime il goal dell'utente nell'utilizzo del sistema.
- *Goal (summary description)* – descrizione della funzionalità fornita dal sistema e che soddisfa una necessità dell'utente, ossia che è percepita dallo stesso utente come "valore".
- *Attori* – persona, dispositivo o altra entità esterna al sistema che interagisce con il sistema. Per ogni caso d'uso esiste sempre un attore primario che è colui che inizia il caso d'uso stesso.
- *Precondizioni* – Condizioni che devono essere soddisfatte all'inizio del caso d'uso. Rappresentano le "garanzie minime" che devono essere soddisfatte per poter attivare lo scenario di utilizzo del sistema (Garanzie fornite dagli attori al sistema).
- *Trigger* – evento trigger che attiva il caso d'uso
- *Descrizione (main success scenario o scenario principale)* – descrizione della sequenza di interazioni più comune tra gli attori e il sistema. In particolare viene descritta la sequenza principale che porta alla conclusione del caso d'uso con successo. La descrizione è definita in termini di input forniti dall'attore e di risposta del sistema. Il sistema è trattato secondo un modello di tipo black-box, concentrandosi su cosa esso fa in risposta agli input, e non su come internamente queste risposte vengano prodotte.
- *Alternative (estensioni)* – descrizioni delle variazioni dalla sequenza di passi tipica del main success scenario. Tali alternative estendono lo scenario principale. La gestione delle eccezioni è un esempio tipico di tali estensioni. Non tutte le alternative portano necessariamente ad un fallimento del caso d'uso.

- *Postcondizioni* – Condizioni sempre soddisfatte al termine del caso d'uso (Garanzie forni sistema agli attori)

Esempio di scenario: Prelievo dal B



Nome caso d'uso: Withdraw Funds.

Scope: ATM system

Goal (summary): L'utente dell'ATM preleva una
valido.

Dipendenze con altri casi d'uso: nessuna

Attori: Cliente del terminale ATM.

Trigger: inserimento di una carta di credito nel lettore del terminale ATM

Precondizioni: Il terminale dell'ATM è in attesa (idle) e visualizza un messaggio di "Benvenuto".

Descrizione (main success scenario o scenario principale):

1. Il cliente inserisce nel lettore del terminale ATM la sua carta di credito.
2. Il sistema riconosce la carta e ne legge il numero (card number).
3. Il sistema visualizza a terminale la richiesta di inserimento del PIN utente.
4. Il cliente inserisce il suo PIN.
5. Il sistema verifica che la carta non è scaduta, né è stata rubata o risulta smarrita.
6. Il sistema verifica che il PIN inserito dal cliente corrisponda con quello della carta.
7. Il sistema controlla che il conto corrente sia accessibile mediante l'utilizzo della carta.
8. Il sistema visualizza il conto corrente del cliente e visualizza le possibili operazioni che il cliente può effettuare (Prelievo, Ultimi Movimenti, Saldo Disponibile, Altri Servizi).
9. Il cliente seleziona il conto corrente desiderato (nel caso il cliente ne abbia più di uno) e seleziona l'operazione Prelievo.
10. Il sistema visualizza la richiesta dell'ammontare da prelevare.
11. Il cliente inserisce l'ammontare da prelevare.
12. Il sistema verifica che il conto corrente contenga sufficienti fondi per consentire la conclusione dell'operazione e che non sia stato superato il limite massimo giornaliero per il prelievo.
13. Il sistema autorizza il prelievo.
14. Il sistema emette il denaro, registrando la transazione sul conto corrente.
15. Il sistema stampa la ricevuta mostrando il numero della transazione, il tipo di operazione effettuata, la quantità di denaro prelevata e il saldo del conto corrente.
16. Il sistema espelle la carta.
17. Il sistema ritorna nello stato iniziale di attesa (idle), visualizzando il messaggio di "Benvenuto"

Esempio di scenario (Cont.)

Alternative (estensioni):

2a. Se il sistema non riconosce la carta dell'utente, quest'ultima viene espulsa dal lettore.

5a. Se la carta risulta scaduta, il sistema la confisca.
5b. Se la carta risulta smarrita, il sistema la confisca.
5c. Se la carta risulta rubata, il sistema la confisca.

6a. Se il cliente ha inserito un PIN che non corrisponde con quello della carta, il sistema visualizza una nuova richiesta di inserimento del PIN. Se il cliente inserisce per tre volte un PIN errato, il sistema confisca la carta.

7a. Se il sistema verifica che il numero di conto non è valido, viene visualizzato un messaggio di errore e la carta viene espulsa dal lettore.

12a. Se non ci sono sufficienti fondi nel conto corrente, il sistema visualizza un messaggio di errore ed espelle la carta. Il terminale ATM viene riportato allo stato di attesa (idle) e viene visualizzato il messaggio di "Benvenuto".
12b. Se il limite giornaliero massimo di prelievo è stato già raggiunto, il sistema visualizza un messaggio di errore ed espelle la carta. Il terminale ATM viene riportato allo stato di attesa (idle) e viene visualizzato il messaggio di "Benvenuto".

14a. Se lil terminale ATM non ha sufficienti fondi per completare la transazione, il sistema visualizza un messaggio di errore, espelle la carta, il terminale ATM viene riportato allo stato di attesa (idle) e viene visualizzato il messaggio di "Benvenuto".
14b. Se il cliente seleziona da terminale il pulsante Cancel, il sistema cancella la transazione ed espelle la carta. Il terminale ATM viene riportato allo stato di attesa (idle) e viene visualizzato il messaggio di "Benvenuto".

Postcondizioni: L'utente ha ottenuto la somma di denaro che aveva richiesto di prelevare.

Questioni aperte: nessuna.

Relazioni tra casi d’uso

- Relazione di inclusione «include»
- Relazione di generalizzazione
- Relazione di estensione «extend»

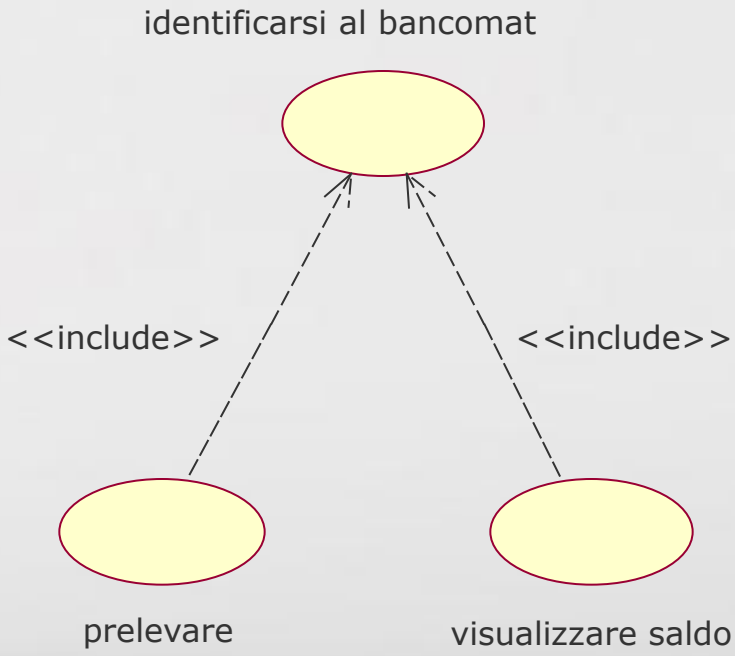
- Poche relazioni: modello semplice, senza grossa complessità sintattica

Relazioni tra casi d’uso: inclusione

- Rappresenta l’invocazione di un caso d’uso da parte di un altro.
- Simile alla chiamata di una funzione
- Serve per scomporre casi d’uso complessi in casi d’uso più semplici
- Esprime il riuso di singoli casi d’uso

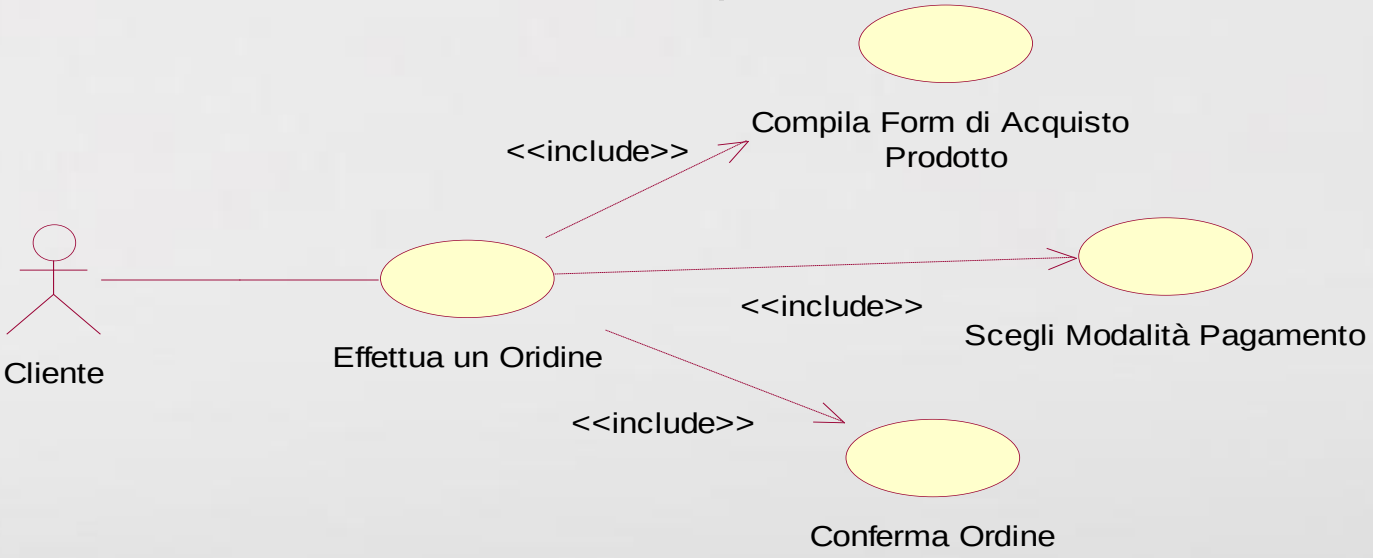
Relazioni tra casi d’uso: include

<<include>> : può mostrare anche il **comportamento comune** a uno o più casi d’uso

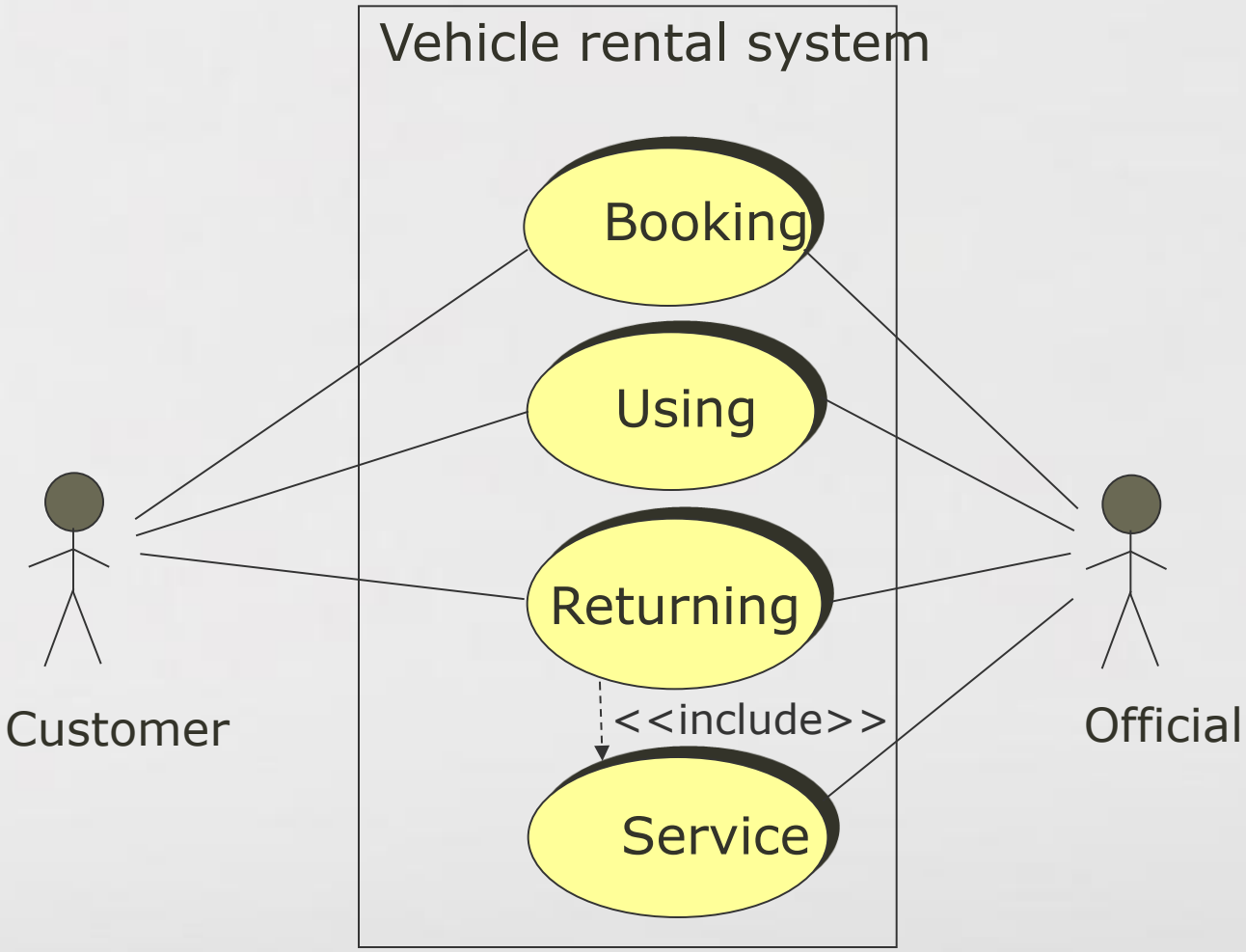


Esempio

- Effettuare un ordine di acquisto include (sempre!) i seguenti passi:
  1. Compilazione di un form di acquisto
  2. Scelta della modalità di pagamento;
  3. Conferma dell'ordine di acquisto



Esempio

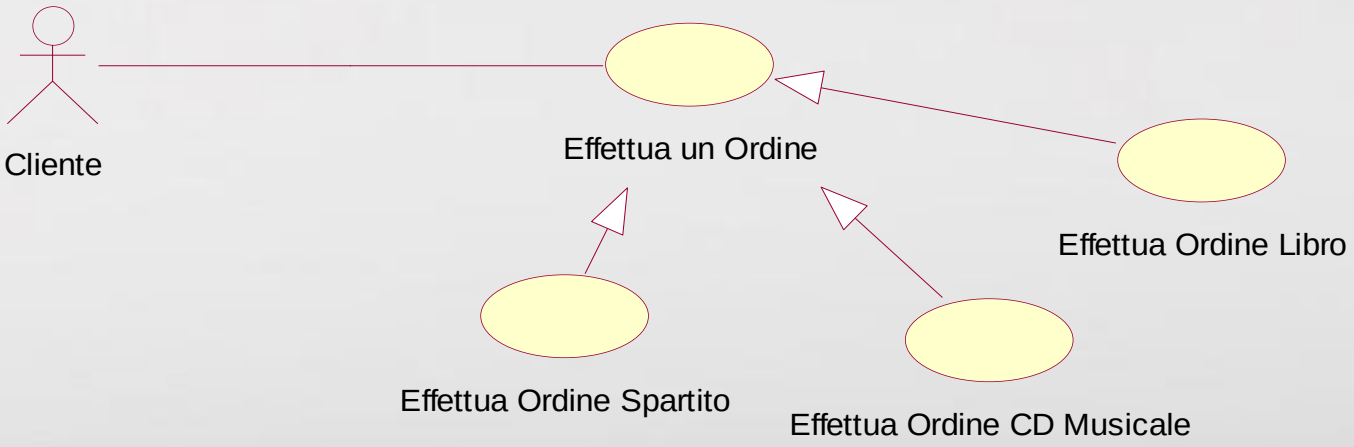

Vehicle rental system
Booking
Using
Returning
<<include>>
Service
Customer
Official

Relazioni tra casi d'uso: generalizzazione

- Simile all'ereditarietà tra classi (che vedremo)
- Relazione che lega un caso d'uso generico a casi d'uso che sono particolari specializzazioni (realizzazioni) del primo
- Logica del caso d'uso derivato simile a (ma diversa da) quella del caso d'uso di base
- Viene riscritta (specializzata) la sequenza delle azioni di base oppure viene elaborata una sequenza alternativa

Relazioni tra casi d'uso: generalizzazione

- Effettua un Ordine è un caso d'uso generale che può essere specializzato nei seguenti:
  1. Effettua un ordine di un cd musicale
  2. Effettua un ordine di un libro
  3. Effettuare un ordine di uno spartito

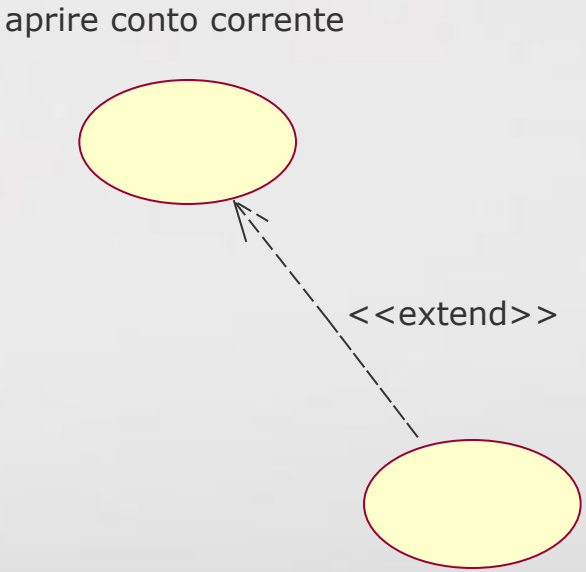


Relazioni tra casi d’uso: estensione

- Rappresenta l’estensione della logica di base di un caso d’uso.
- Il caso d’uso estensione continua il comportamento del caso d’uso di base inserendovi delle azioni alternative da un certo punto in poi (chiamato punto di estensione)
- Il caso d’uso di base dichiara tutti i possibili punti di estensione
- Simile alla gestione degli interrupt hardware (gestione eccezioni)

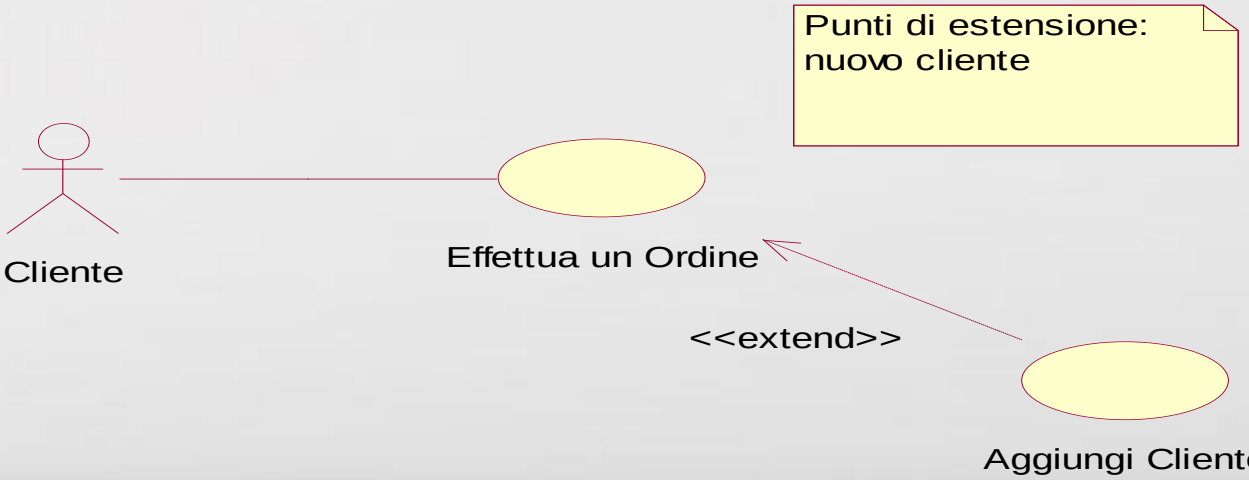
Relazioni tra casi d'uso: extend

<<extend>>: mostra il **comportamento opzionale** (alternativo o relativo al trattamento di **condizioni anomale)**



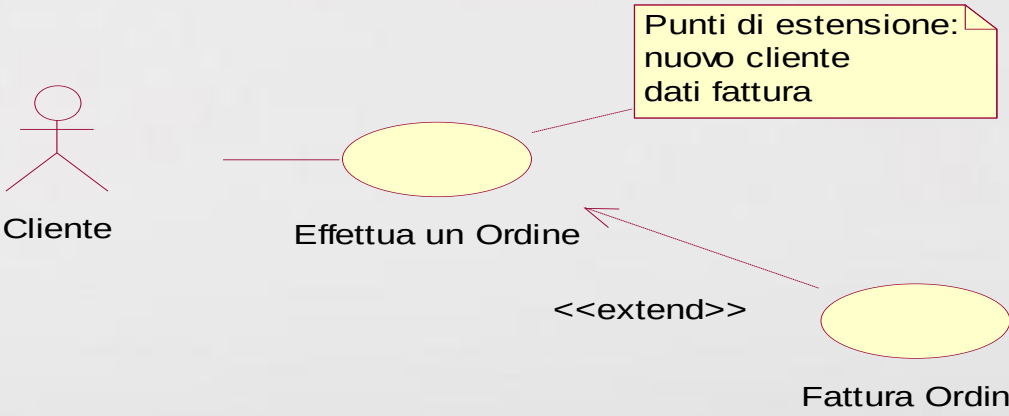
Relazioni tra casi d'uso: estensione

- Effettua un Ordine presuppone che l'utente sia già registrato nel sistema informatico
- Se il cliente è al suo primo ordine e non si è precedentemente registrato? Eccezione nel caso d'uso base!



Relazioni tra casi d'uso (estensione)

- Effettua un Ordine prevede una procedura d'acquisto di default che non invia la fattura
- E se il cliente richiede una fattura? Eccezione nel caso d'uso base



Esempio: Sportello Bancomat

- Il Sistema sarà eseguito su uno sportello bancomat automatico
- L'utente deve essere in grado di depositare assegni sul suo conto
- L'utente deve essere in grado di prelevare i soldi dal suo conto
- L'utente deve poter interrogare il Sistema sul saldo del suo conto
- Se lo richiede, l'utente deve poter ottenere la ricevuta per la transazione.
- I tipi di transazione sono ritiro o deposito.
- La ricevuta deve indicare la data della transazione, ilnumero del conto, il saldo precedente e successivo la transazione
- Dopo ogni transazione, il nuovo saldo deve essere visualizzato all'utente

Esercizio: Soluzione