Indice

5 – Modelli di sistema 2
5.1 – Sistemi orientati agli oggetti 4
5.1.1 – Oggetti e valori 6
5.1.2 – Classi e metodi 8
5.1.3 – Gerarchie, migrazione e ridefinizione 11

5 – Modelli di sistema

La **modellazione dei sistemi** è il processo che sviluppa modelli astratti di un sistema, dove ogni modello rappresenta una differente vista o prospettiva del sistema.

I modelli sono utilizzati durante il processo di ingegneria dei requisiti per facilitare la deduzione dettagliata dei requisiti per un sistema.

È possibile sviluppare modelli sia di sistemi esistenti sia di nuovi sistemi:

1. I modelli di un **sistema esistente** si usano durante l'ingegneria dei requisiti. Servono a chiarire che cosa fa il sistema esistente, i suoi punti di forza e di debolezza del sistema;
2. I modelli di un **nuovo sistema** si usano durante l'ingegneria dei requisiti per descrivere con maggior chiarezza i requisiti proposti ad altri stakeholder del sistema.

Il modello tralascia deliberatamente i dettagli per rendere più comprensibile il sistema. Un modello è un'astrazione del sistema che si sta studiando, non una rappresentazione alternativa del sistema. Un'astrazione deliberatamente semplifica il progetto di un sistema e ne evidenzia le caratteristiche più salienti.

È possibile **sviluppare vari modelli** per rappresentare il sistema da differenti prospettive, per esempio:

1. Una **prospettiva esterna**; viene modellato il contesto o l'ambiente in cui opera il sistema;
2. Una **prospettiva di interazioni**; vengono modellate le interazioni tra il sistema e il suo ambiente, o tra i componenti del sistema;
3. Una **prospettiva strutturale**; viene modellata l'organizzazione del sistema o la struttura dei dati elaborati dal sistema;
4. Una **prospettiva comportamentale**; vengono modellati il comportamento dinamico del sistema e le sue risposte agli eventi.

Quando si sviluppano i modelli, occorre una certa flessibilità nel modo di utilizzare la notazione grafica. Esistono tre modi in cui i modelli grafici sono comunemente utilizzati:

1. Per **stimolare e focalizzare la discussione su un sistema proposto o già esistente**. Scopo è stimolare e focalizzare la discussione tra gli ingegneri del software che si occupano dello sviluppo del sistema.
2. Per **documentare un sistema esistente**. Se i modelli vengono utilizzati nella documentazione, non devono essere completi, in quanto possono servire per documentare soltanto alcune parti di un sistema.
3. Per **fornire una descrizione dettagliata del sistema** che può essere utilizzata per generare l'implementazione del sistema.

L’UML ha 13 tipi di diagrammi e quindi consente la creazione di vari tipi di modelli di sistemi. Tuttavia, esistono 5 tipi di diagrammi che sono sufficienti a rappresentare gli elementi essenziali di un sistema:

1. **Diagrammi di attività**. Mostrano le attività coinvolte in un processo o nell’elaborazione dei dati.
2. **Diagrammi di casi d’uso**. Mostrano le interazioni tra un sistema e il suo ambiente.
3. **Diagrammi di sequenza**. Mostrano le interazioni tra attori e sistema e tra componenti del sistema.
4. **Diagrammi di classe**. Mostrano le interazioni tra attori e sistema e tra i componenti del sistema.
5. **Diagrammi di stato**. Mostrano come il sistema reagisce agli eventi interni ed esterni.

5.1 – Sistemi orientati agli oggetti

Un **sistema a oggetti** gestisce collezioni di oggetti. Ciascun **oggetto** ha un identificatore, uno stato e un comportamento:

- L'**identificatore** (OID) garantisce l'individuazione in modo univoco dell'oggetto, e permette di realizzare riferimenti tra oggetti;
- Lo **stato** è l'insieme dei valori assunti dalla proprietà dell'oggetto – è in generale un valore a struttura complessa;
- Il **comportamento** è descritto dall'insieme dei metodi che possono essere applicati all'oggetto

Inoltre, un **tipo** descrive la proprietà di un oggetto, cioè la parte statica, e l'interfaccia dei suoi metodi, cioè la parte dinamica.

Relativamente alla **parte statica**, i tipi vengono costruiti a partire da un insieme di tipi atomici:

- Numeri;
- Stringhe;
- OID;
- Booleani;
- Tipi enumerativi;
- Ecc.

Per rappresentare il valore nullo si usa nil, un valore polimorfo. Ogni definizione di tipo associa un nume (etichetta) a un tipo.

Con i **costrutti di tipo** si definiscono tipi complessi. I costruttori di tipo, tra loro ortogonali, sono:

- $record\text{-}of(A_1{:}T_1, \dots, A_n{:}T_n)$ o $tuple(A_1{:}T_1, \dots, A_n{:}T_n)$
- $set\text{-}of(T)$
- $bag\text{-}of(T)$
- $list\text{-}of(T)$

Dato un tipo complesso T, un oggetto che ha per tipo T si dice istanza di T. Non tutti i sistemi offrono tutti i costruttori.

Un **esempio di tipo complesso**:

```
Studente: record-of(
   Matricola: string, Nome: string,
   Universita: record-of(
      Nome: string,
      SedeRettorato: string,
      Sedi: set-of(
         record-of(
            Nome: string, Citta: string,
            NumDocenti: integer))),
   Eta: integer, AnnoCorso: integer,
   InsegnamentiSuperati: set-of(record-of(
                              Esame: string,
                              Voto: string)))
```

Analogamente, un **esempio di valore complesso** (è possibile definire dei valori complessi compatibili con un tipo complesso):

```
<S1, [Matricola: "VR001", Nome: "Mario Rossi",
      Universita: [
         Nome: "UNIVR", SedeRettorato: "Palazzo Giuliari",
         Sedi: {
            [
               Nome: "Ca Vignal", Citta "Verona",
               NumDocenti: 120
            ],
            [
               Nome: "EconomiaVI", Citta "Vicenza",
               NumDocenti: 30
            ]
         }
      ],
      Eta 25, AnnoCorso: 2,
      Insegnamenti superati: {
         [Esame: "Ingegneria del Software", Voto: "30 e lode"],
         [Esame: "Architetture hardware", Voto: "30"]
      }
   ]
>
```

5.1.1 – Oggetti e valori

L'**uso** di **tipi** e **valori complessi** consente di associare ad un singolo oggetto una struttura qualunque. Al contrario, nel modello relazionale alcuni concetti devono essere rappresentati tramite più relazioni.

Tuttavia, la rappresentazione proposta per Studente negli esempi precedenti non è "normalizzata": per farlo è necessario utilizzare dei **riferimenti tra oggetti**.

Un **oggetto** è una coppia $< \text{OID}, \text{Valore} >$ in cui **OID** (*object identifier*) è un valore atomico definito dal sistema e trasparente all'utente e **Valore** è letteralmente un valore complesso.

Il valore assunto da una proprietà di un oggetto può essere l'OID di un altro oggetto, in questo caso viene chiamato **riferimento**.

Un **esempio** di riferimenti:

```
Studente:   record-of(
        Matricola: string, Nome: string,
        Universita: *EnteUniv,
        Eta: integer, AnnoCorso: integer,
        InsegnamentiSuperati: set-of(record-of(
                                    Esame: string,
                                    Voto: string))
)

EnteUniv:   record-of(
        Nome: string, SedeRettorato: string,
        Sedi: set-of(*SedeUniv)
)

SedeUniv:   record-of(
    Nome: string, Citta: string,
    NumDocenti: integer
)
```

Mentre, un **esempio** di oggetti compatibili con i tipi definiti:

```
<OID1, [Matricola: “VR001”, Nome: “Mario Rossi”,
                Universita: OID2,
                Eta: 25, AnnoCorso: 2,
                InsegnamentiSuperati: {
                        [Esame: “Ingegneria del Software”, Voto: “30 e lode”],
                        [Esame: “Architetture hardware”, Voto: “30”]}>

<OID2, [Nome: “UNIVR”, SedeRettorato: “Palazzo Giuliari”,
                Sedi: {OID3, OID4}]>

<OID3, [Nome: “Ca Vignal”, Citta: “Verona”, NumDocenti: 120]>

<OID4, [Nome: “EconomiaVI”, Citta: “Vicenza”, NumDocenti: 30]>
```

Tra gli oggetti sono definite le seguenti **relazioni**:

- **Identità** ($Oa = Ob$), richiede che gli oggetti abbiano lo stesso identificatore;
- **Uguaglianza superficiale** ($Ob == Oc$), richiede che gli oggetti abbiano lo stesso stato, cioè lo stesso valore per proprietà omologhe;
- **Uguaglianza profonda** ($Ob === Od$), richiedere che le proprietà che si ottengono seguendo i riferimenti abbiano gli stessi valori (l’uguaglianza dello stato non è richiesta):
    - $Oa =< OID1, [a,\ 10,\ OID4] >$
    - $Ob =< OID1, [a,\ 10,\ OID4] >$
    - $Oc =< OID2, [a,\ 10,\ OID4] >$
    - $Od =< OID3, [a,\ 10,\ OID5] >$
    - $Oe =< OID4, [a,\ b] >$
    - $Of =< OID5, [a,\ b] >$

Il concetto di riferimento presenta analogie con quello di “puntatore” nei linguaggi di programmazione, e con quello di “chiave esterna” in un sistema relazione. Tuttavia, esistono delle differenze:

1. I **puntatori** possono essere corrotti a causa di una gestione errata da parte del programmatore (dangling). A differenza dei riferimenti a oggetti che vengono invalidati automaticamente in caso di cancellazione di un oggetto referenziato.
2. Le **chiavi esterne** sono visibili, in quanto realizzate tramite valori. A differenza degli identificatori di un oggetto che non sono associati a valori visibili dall’utente.
3. Modificando gli attributi di una **chiave esterna**, è possibile perdere riferimenti. A differenza della modifica di un oggetto referenziato che continua ad esistere.

5.1.2 – Classi e metodi

Gli oggetti sono associati a un **tipo** (intensione) e a una **classe** (implementazione). Per **tipo** si intende una astrazione che permette di descrivere lo stato e il comportamento di un oggetto. Per **classe** si intende la descrizione di un'implementazione di un tipo, ovvero la struttura dei dati e l'implementazione di metodi tramite programmi.

La **definizione di una classe** è normalmente separata in due parti:

1. La definizione del tipo degli oggetti appartenenti alla classe;
2. L'implementazione, che descrive il codice dei metodi e talvolta le strutture fisiche usate per la memorizzazione.

Gli oggetti vengono **raggruppati in collezioni**, chiamati **estensioni**.

Verrà utilizzato un modello semplificato che utilizza le seguenti supposizioni: la classe descrive sia l'implementazione sia l'estensione di un tipo; mentre, i tipi sono astrazioni che descrivono sia lo stato che il comportamento.

Per cui, il modello utilizzato avrà:

- Tipi e classi distinti;
- Ogni classe è associata a un solo tipo;
- Non esiste un concetto apposito per l'estensione.

I concetti di **classe** e di **estensione non** sono **identici**.

Infatti, una **classe** è una implementazione di un tipo, quindi uno stesso tipo può avere implementazioni diverse. Questo è particolarmente importante non per assegnare semantiche diverse a oggetti di uno stesso tipo (**da evitare!**), ma ad esempio per implementare una semantica unica di un tipo in riferimento a piattaforme architetturali diverse (nel caso di un sistema a oggetti distribuito).

Una **estensione** è una collezione di oggetti avente lo stesso tipo. Un oggetto può appartenere a più collezioni, essere rimosso da una collezione, ecc.

Un **metodo** è una procedura utilizzata per incapsulare lo stato di un oggetto, ed è caratterizzato da una **interfaccia** (o segnatura) e una **implementazione**:

- L'interfaccia comprende tutte le informazioni che permettono di invocare un metodo (il tipo dei parametri)
- L'implementazione contiene il codice del metodo.

Il **tipo** di un oggetto comprende, oltre alle proprietà, anche le interfacce dei metodi applicabili a oggetti di quel tipo.

Si possono immaginare i metodi come delle funzioni, ovvero che possono avere più parametri di ingresso ma un solo parametro di uscita.

Immaginando sempre l'approssimazione dei metodi come funzioni, esistono i seguenti tipi:

1. **Costruttori**, per costruire oggetti a partire da parametri di ingresso. Il valore di ritorno è l'OID dell'oggetto costruito.
2. **Distruttori**, per cancellare oggetti, ed eventuali altri oggetti ad essi collegati.
3. **Di accesso**, funzioni che restituiscono informazioni sul contenuto degli oggetti.
4. **Trasformatori**, procedure che modificano lo stato degli oggetti, e di eventuali altri oggetti ad essi collegati.

Infine, un metodo può essere **pubblico** o **privato**.

Un **esempio** di metodi:

```
add method Init(                                    // Costruttore
        Matricola_par: string,
        Universita_par: string,
        Nome_par: string,
        Eta_par: string): Studente
       in class Studente is public;

add method Voto(Insegn_par: string): string     // di accesso
       in class Studente is public;

add method public Assegna_voto(Insegn_par: string, Voto_par: string)
                                                  // trasformatore
       in class Studente is public;     // void
```

Dato che si è ipotizzato che una classe raccolga tutti gli oggetti di uno stesso tipo, si immagina che una classe sia un contenitore di oggetti, che possono essere dinamicamente aggiunti o tolti alla classe (tramite costruttori e distruttori).

Ad una classe è associato un solo tipo, quindi gli oggetti in una classe sono tra loro omogenei (hanno le stesso proprietà e rispondono agli stessi metodi).

Una classe, inoltre, definisce anche l'implementazione dei metodi, che va specificata in qualche linguaggio di programmazione.

Un **esempio** di implementazione e invocazione di metodi:

```
Ref<Studente> Studente::Init
        (string Matricola_par,
         string Nome_par,
         string Universita_par,
         integer AnnoCorso_par)
{ self->Matricola = Matricola_par;
   self-> Nome = Nome_par;
   self->Universita = Universita_par;
   self->AnnoCorso = AnnoCorso _par;
   return(self); }

Ref<Studente> X;  //dichiarazione di variabile
X = new(Studente);
X -> Init("VR001", "Mario Rossi", "UNIVR", 2);
```

Un altro esempio:

```
void Studente::Assegna_voto(Insegn_par: string, Voto_par:string)
{self->InsegnamentiSuperati->add(record_of(Insegn_par: string, Voto_par:
  string))}

Ref<Studente> X;
X = new(Studente);
 (X -> Init("VR001", "Mario Rossi", "UNIVR",2))-> Assegna_voto("Ingegneria del
   SW", "30 e lode");
```

5.1.3 – Gerarchie, migrazione e ridefinizione

Tra i tipi (e le classi) di un sistema a oggetti, è possibile definire una gerarchia di ereditarietà. Una sotto-classe **eredita** lo stato e il comportamento della super-classe e può, in aggiunta, rendere più specifici lo stato e il comportamento.

Tutti gli oggetti delle sotto-classi appartengono automaticamente alle sopra-classi (*sub-typing*).

Vale la proprietà transitiva: se $C1$ è una sotto-classe di $C2$ e $C2$ è sotto-classe di $C3$, implica che $C1$ è sotto-classe di $C3$.

Un **esempio** di gerarchia:

```
add class StudentePartTime
        inherits Studente
        type record_of(MaxCrediti: integer)

add class StudenteErasmus
        inherits Studente
        type record_of(UniversitaProv: EnteUniv)

Ref<StudentePartTime> X;
X = new(StudentePartTime);
X->Init("VR002","Maria Verdi","UNIVR",3);
X-> MaxCrediti = 30;
```

Nel corso della propria esistenza, un oggetto deve mantenere la propria identità, in modo che sia possibile riferisti ad esso in modo univoco. Tuttavia, è possibile che un **oggetto cambi tipo**: per esempio una persona diviene uno studente, poi uno studente lavoratore, poi un lavoratore, poi una persona sposata, ecc.

Quindi è necessario un meccanismo di acquisizione e perdita di tipi. Solitamente vengono eseguite operazioni di:

1. **Specializzazione**, ovvero si diventa membri di una sotto-classe;
2. **Generalizzazione**, si perde l'appartenenza a una sotto-classe.

Alcuni sistemi richiedono che un oggetto appartenga a una sola classe più specializzata, altri non pongono questo **vincolo**.

Inoltre, in alcuni sistemi è possibile che una **classe erediti da più super-classi**. Per esempio:

```
add class StudentePartTimeErasmus
        inherits StudentePartTime, StudenteErasmus
        type record-of(aziendeVisitate:set(string))
```

Ogni oggetto di StudentePartTimeErasmus appartiene sia ad StudentePartTime che a StudenteErasmus.

Ci possono essere oggetti che appartengono sia a StudentiPartTime sia a StudenteErasmus, ma che non appartengono a StudentePartTimeErasmus.

Attenzione! L'ereditarietà multipla può generare **conflitti di nome**, qualora due o più sopra-classi posseggano proprietà o metodi con lo stesso nome.

Alcune soluzioni sono:

1. Non consentire definizioni con conflitti;
2. Definire dei meccanismi per la risoluzione del conflitto (esempio, ordinamento statico o valutazione dinamica);
3. Ridefinizione locale di proprietà e metodi.

Alcune **tecniche di ridefinizione dei metodi**:

- *Overriding*, consiste nel ridefinire il corpo di un metodo nell'ambito di una sotto-classe (per esempio il metodo *display*);
- *Overloading*, si possono avere diverse versioni dello stesso metodo con firme diverse;
- *Late binding*, ovvero la scelta del metodo da invocare dipende dalla classe cui appartiene l'oggetto; se la classe non è nota al momento della compilazione, è necessario il *late binding*.

Nonostante le tecniche appena presentate, bisogna **tenere in considerazione alcuni fattori**:

- La modifica dell'interfaccia dei metodi richiede molta attenzione;
- Se si ridefinisce un metodo in una sotto-classe, i suoi parametri possono essere definiti in due modi:
  - *co-variante*, ovvero i parametri sono sotto-tipi dei parametri della sopra-classe;
  - *contro-variante*, ovvero i parametri sono sopra-tipi dei parametri della sopra-classe;

Attenzione! La soluzione di definire i parametri in modo co-variante è la più diffusa, ma comporta dei problemi nei parametri d'ingresso.

Tuttavia, si presenta di seguito una **definizione** di *co-variante*:

```
add class Programmatore inherits Utente

add class File ...
method init(Nome: string, Owner: Utente)

add class Sorgente inherits File ...
method init(Nome: string, Owner: Programmatore)
```

Un problema che si crea con l'uso della *co-variante* è: **non** è possibile **garantire** a compile-time che l'invocazione del metodo su *File* sia corretta (il file può essere un Sorgente, ma il parametro *Owner* passato non è un Programmatore).