

Laboratorio di Progettazione di Sistemi Software

Introduzione

Valentina Presutti (A-L)

Riccardo Solmi (M-Z)

Indice degli argomenti

- **Introduzione all'Ingegneria del Software**
- **UML**
- **Design Patterns**
- **Refactoring**

Bibliografia

- ***UML – Unified Modeling Language***
 - *UML Reference Page (OMG)*
- ***Design Patterns***
 - *“Design Patterns – Elements of Reusable Object-Oriented Software”*, GoF
 - *“Thinking in Patterns with Java”*, Bruce Eckel (scaricabile)
- ***Refactoring***
 - *“Refactoring to Patterns”*, Joshua Kerievsky
 - *“Refactoring – Improving the Design of Existing Code”*, Martin Fowler
- ***Eclipse IDE + UML Plugin***
 - Ambiente di programmazione che supporta refactoring e UML.
- **Trovate tutto nella pagina web dedicata a questo corso:**
 - http://www.cs.unibo.it/~solmi/teaching/labss_2005-2006.html
 - <http://www.cs.unibo.it/~presutti>

Ricevimento

▪ *Metodo 1: il newsgroup*

- Avete a disposizione un newsgroup:
unibo.cs.informatica.paradigmiprogrammazione

▪ *Metodo 2: il wiki*

- Avete a disposizione un wiki:
<http://courses.web.cs.unibo.it/Labpss0506/>

▪ *Metodo 3: subito dopo le lezioni*

- siamo a vostra disposizione per chiarimenti
- in aula e alla lavagna, in modo da rispondere a tutti

▪ *Metodo 4: ricevimento*

- Martedì pomeriggio dalle 15 alle 17

Alcuni concetti

▪ **Ingegneria del Software**

- L'Ingegneria del Software è una disciplina che studia i *metodi* di produzione, gli *strumenti* di sviluppo e misura la *qualità* di sistemi software complessi.

▪ **Sistema Software**

▪ **Dominio**

- Un'area di conoscenza (concetti, terminologia, attività) conosciuta da chi la usa.
- Incorpora la conoscenza necessaria per costruire una famiglia di Sistemi Software.
- Il confine è fissato dai clienti (stakeholders).
- Esempi di domini *verticali* (o business area): prenotazioni aeree, gestione portafoglio, contabilità, inventario, gestione ordini, sistemi medici, ausili didattici, ...
- Esempi di domini *orizzontali*: DBMS, compilatori, GUI, librerie di collezioni (liste, insiemi, ...), algoritmi numerici, librerie grafiche, ...

▪ **Requisiti e Features**

- Richieste e vincoli imposti dai clienti. Una *feature* è una funzionalità osservabile dall'utente.

▪ **Modello (di un dominio)**

- Astrazione costruita su un sottoinsieme del dominio che soddisfa i requisiti. Facilita l'implementazione del Sistema Software. I concetti del dominio vengono qui chiamati *entità*.

Di cosa non ci occupiamo in questo corso

- **Supporto al lavoro in gruppo (cooperazione, coordinamento)**
- **Gestione delle versioni e delle configurazioni**
- **Documentazione**
- **Testing**
- **Patterns ad altri livelli di astrazione**
 - Code Patterns, Architectural Patterns
 - Analysis Patterns, Organization Patterns
 - Domain-Specific Patterns ...

UML – Unified Modeling Language

- **UML è un linguaggio grafico per analizzare, specificare, visualizzare e documentare lo sviluppo dei documenti di progetto di un Sistema (Software)**
- **UML fa uso di 13 tipi di diagrammi.**
- **Noi studiamo solamente:**
Class Diagram, Object Diagram, Sequence Diagram
 - I primi due descrivono la struttura di un modello; il terzo descrive il comportamento.
 - Sono usati nei cataloghi di Design Patterns e Refactoring
 - Supportano meglio la sincronizzazione tra modello e implementazione (roundtrip)

Design Patterns

▪ **Definizione**

- Un Design Pattern descrive un problema ricorrente di progettazione e uno schema di soluzione per risolverlo.
- NB Il termine Design nel nome non significa che riguardano solo la progettazione

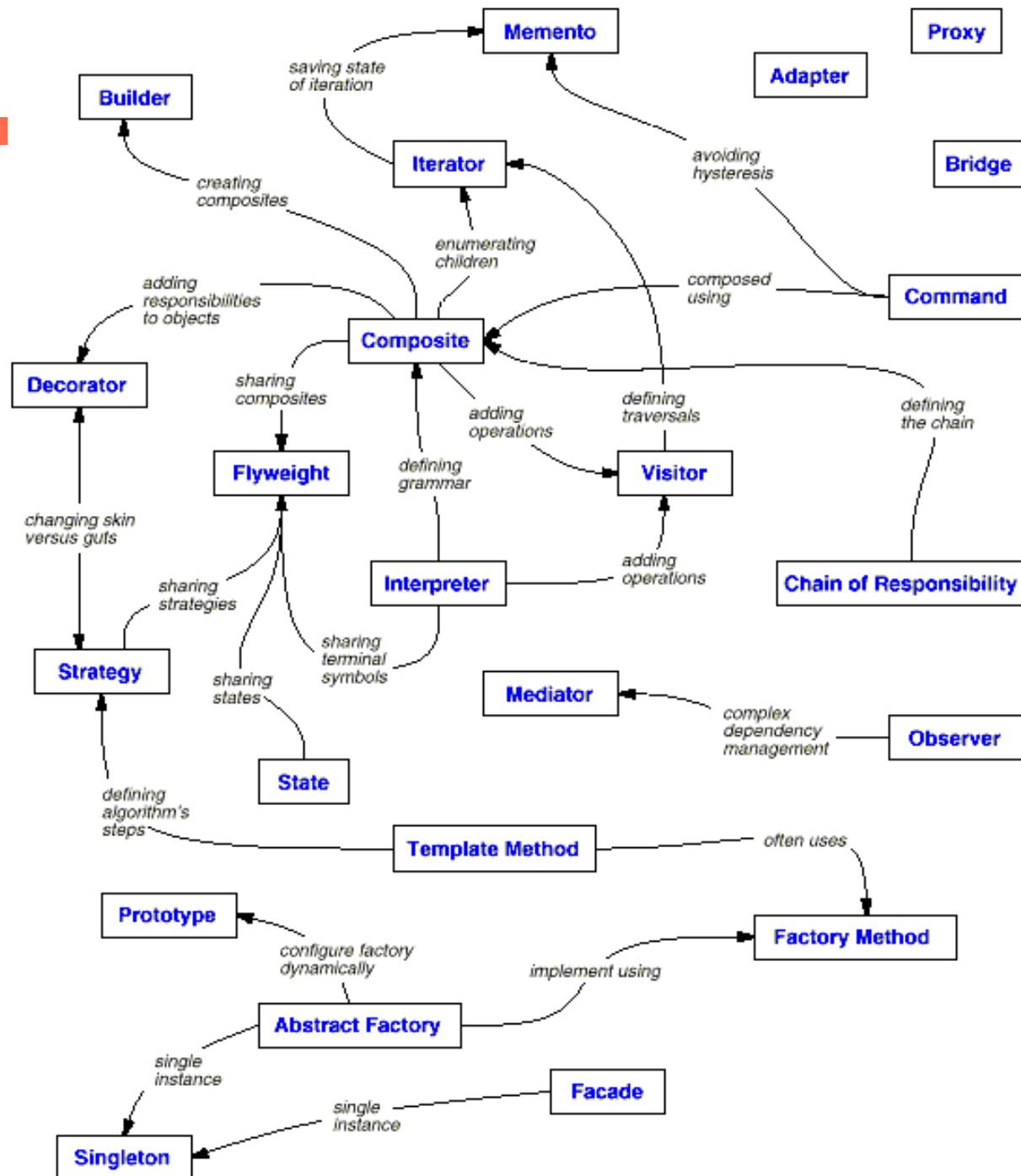
▪ **Serve a ...**

- Progettare e implementare il Software in modo che abbia la flessibilità e le capacità di evolvere desiderate
- Dominare la complessità di un Sistema Software

▪ **Catalogo**

- Ogni Design Pattern ha un *nome*, un campo di *applicabilità*, uno schema di *soluzione* e una descrizione delle *conseguenze* della sua applicazione.

Esempi di Design Patterns



Refactoring

▪ **Definizione**

- Un Refactoring è un processo di *trasformazione* di un Sistema Software che non altera il comportamento osservabile ma migliora la struttura interna (design).

▪ **Serve a ...**

- Migliorare/incorporare il Design
- Mantenere/rendere il codice leggibile
- Facilitare la scoperta degli errori (bugs)
- Mantenere la velocità di sviluppo

▪ **Catalogo**

- Ogni Refactoring descrive la motivazione e i dettagli implementativi di una trasformazione di codice.

Esempi di Refactoring

- **Rinomina**
 - Package, classi, interfacce, metodi, campi, variabili, parametri
- **Sposta**
 - Package, classe, membro
- **Estrai**
 - Superclasse, interfaccia, metodo, variabile
- **Introduci**
 - Variabile, campo, costante, parametro
- **Cambia signature**
 - Aggiungi, elimina, riordina.
- **Incapsula campo**

Importanza del vocabolario di progettazione

- **Catalogo di Design Patterns e Refactorings**
 - Non sono da imparare a memoria ma ...
 - Devono entrare a far parte del vostro linguaggio
- **Sapere scrivere il codice non significa sapere i Design Pattern**
 - Se anche riconoscete alcune soluzioni che avete già usato anche voi non significa che sapete già quei DP
- **Ragionare ad un livello di astrazione superiore**
 - Imparare ad esprimere le proprie *intenzioni* usando i Design Patterns e le operazioni di Refactoring
 - Scendere a livello di righe di codice solo quando necessario
 - Imparare a riconoscere i Design Pattern nei sorgenti

Design Patterns vs Refactoring

■ **Design Patterns: progettare anticipando i cambiamenti**

- *Prima* si progetta il Software poi lo si implementa.
- Massimizzare il riuso (senza ricompilazione)
- Progettare il Sistema in modo da anticipare i nuovi requisiti e i cambiamenti dei requisiti attuali.

■ **Refactoring: affrontare i cambiamenti**

- Per migliorare il Design *dopo* che il codice è stato scritto.
- La compatibilità all'indietro non è un obiettivo.
- Assume di avere il controllo su tutto il codice sorgente.

Rischi e limiti dei Design Patterns

■ **Esistono dei trade-off**

- Non esiste una combinazione di Patterns che mi dà il massimo di flessibilità e di facilità di evolvere.
- Un Design Pattern facilita alcuni sviluppi futuri a scapito di altri.
- Cambiare una scelta progettuale può essere costoso

■ **Rischio di sovra-ingegnerizzare**

- Il codice diventa più flessibile e sofisticato di quanto serve.
- Quando scelgo i Patterns prima di iniziare ad implementare.
- Quando introduco i Patterns troppo presto.

■ **Costo computazionale**

- La flessibilità di una soluzione costa ed è accompagnata da una maggiore complessità.
- Usare un Design Pattern per esigenze *future* è un *costo*

Rischi e limiti del Refactoring

■ **Limiti di applicabilità**

- Non si riesce sempre a trasformare un codice scadente in uno equivalente scritto bene
- Qualche volta conviene riscriverlo da capo
- Il codice deve essere quasi funzionante per poter essere ristrutturato (con strumenti automatici)
- Le operazioni di Refactoring disponibili sono a basso livello

■ **Rischio di sotto-ingegnerizzare**

- Il Design del Sistema Software decade.
- Quando dedico poco tempo alla progettazione perché tanto con i Refactoring si può migliorare dopo.
- Quando ho fretta di aggiungere delle funzionalità e non ho “tempo da perdere” per migliorare anche il design.

Rischi e limiti del Refactoring /2

■ **Costo computazionale**

- Diversi Refactoring introducono delle indirezioni nei programmi (in genere vale la pena).

■ **Necessita Tool automatico/interattivo**

- Una operazione di Refactoring richiede una lunga sequenza di modifiche sparse su diversi sorgenti.
- Un supporto da parte di un IDE è quasi necessario per potere godere di tutti i vantaggi.
- Utile interazione con il programmatore nella applicazione di un Refactoring. (effetto psicologico: i sorgenti sono *miei*)

■ **Spesso modificano le interfacce**

- E' un problema solo per le API

Design Patterns + Refactoring

- **Design e Refactoring continuo**
 - Design e Refactoring sono complementari
 - Si riesce a progettare meglio il Sistema implementandolo
- **Uso Design Patterns per le esigenze attuali**
 - Si riduce enfasi sulla progettazione iniziale
 - Mi basta progettare una soluzione ragionevole non la migliore soluzione.
 - Continuo a chiedermi come evolverà il sistema ma mi basta sapere di poter aggiungere un certo tipo di flessibilità con il Refactoring
- **Uso Refactoring per preparare il Sistema ad accogliere nuove o modificate funzionalità**
 - Prima eseguo Refactoring per avere una struttura conveniente
 - Poi aggiungo/modifico la funzionalità
- **Uso del catalogo di Patterns per descrivere il Sistema attuale e quello che si vuole ottenere e del catalogo di Refactorings per descrivere le trasformazioni.**

Design Patterns + Refactoring /2

■ Incorporare Design nei sorgenti

- Le scelte progettuali e le dipendenze devono essere il più possibile espresse nei sorgenti
- I commenti non sono un deodorante per rendere accettabile un codice illeggibile

■ Velocità

- Le ottimizzazioni possono rendere il codice meno leggibile e più difficile da modificare.
- La maggior parte del codice (90%) viene usato raramente; ottimizzarlo è un lavoro sprecato e per di più mi può impedire di migliorare ulteriormente altre parti.
- Prima scrivo privilegiando il Design
- Poi con l'aiuto di un Profiler, ottimizzo le parti che veramente lo richiedono (10%)

Obiettivi del corso

- **Apprendere un vocabolario di Patterns e Refactorings**
- **Analizzare e comunicare un Sistema Software a livello di Design Patterns**
 - Imparare a riconoscere i Design Pattern nei sorgenti
 - Comprendere le *dipendenze strutturali* che si introducono; in modo da scegliere consapevolmente i Design Pattern.
 - Scendere a livello di righe di codice solo quando necessario
- **Analizzare e comunicare le modifiche da fare ad un Sistema Software in termini di operazioni di Refactoring**
 - Imparare a vedere un Sistema Software come un oggetto modellabile (con alcune parti rigide e altre elastiche)
 - I sorgenti non sono solo miei. Anche gli strumenti di Refactoring mi aiutano a migliorarli.