La progettazione del software

Prof. Paolo Ciancarini
Corso Ingegneria del Software
CdL Informatica
Università di Bologna

Obiettivi della lezione

- Che cos'è la **progettazione**?
- Quali sono le regole di **buona** progettazione?
- Come si **impara** a progettare?
- Come si impara a progettare sistemi fatti di componenti di buona qualità e riusabili?

Cos’è la progettazione?

2001: Ape and bones
http://www.youtube.com/watch?v=toNuups_j4A

Agenda

- Fondamenti di progettazione
- Progettare sistemi software
- Misure di qualità del progetto software
- Moduli, componenti e gerarchie sw

Discussione

- Cos’è la progettazione?

Che cosè la progettazione?

Progettare = *anticipare* un futuro artefatto mediante problem solving creativo

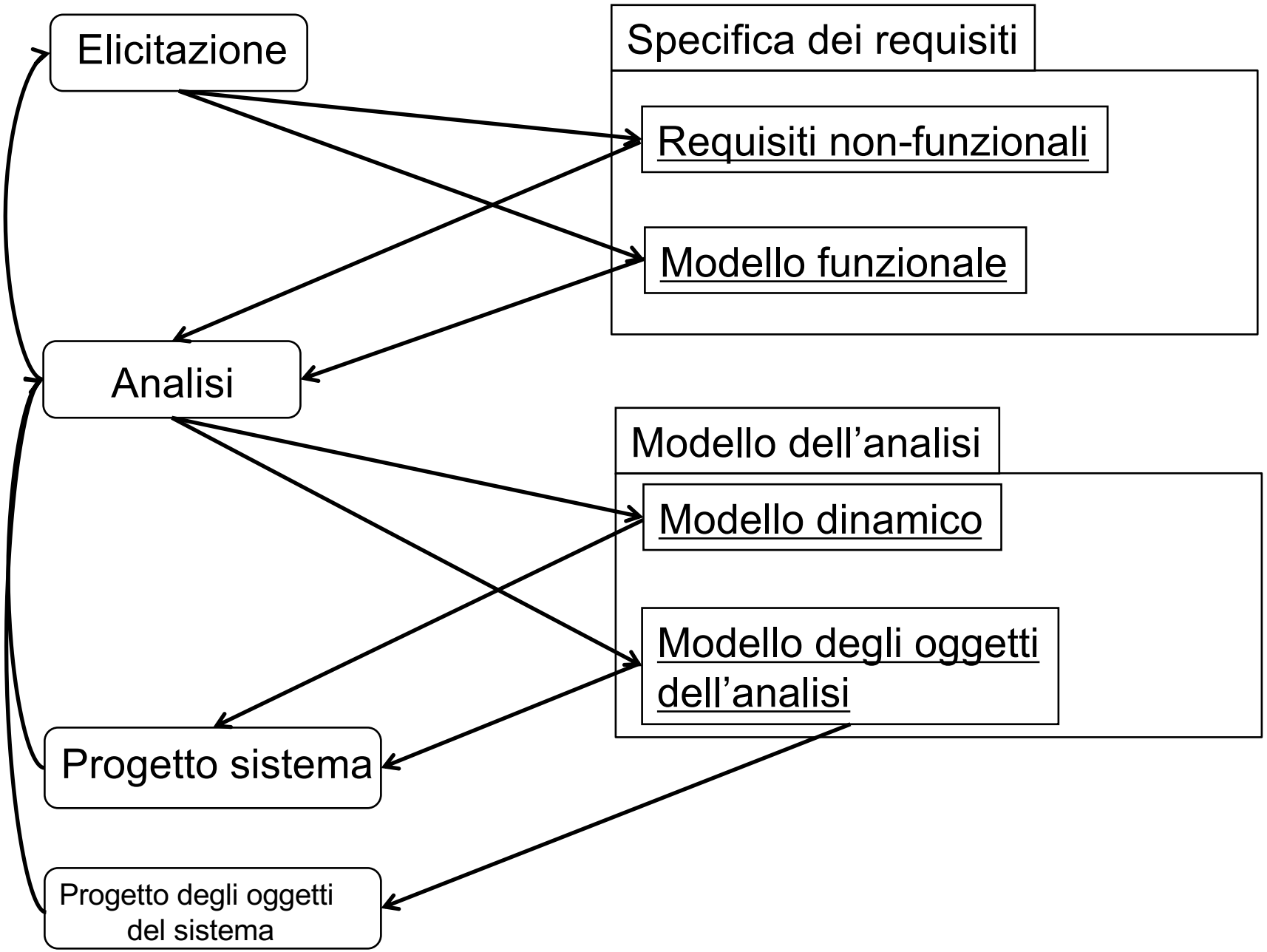
Analisi e progettazione

L'**analisi** si occupa di:

- Capire il problema, e preparare la base per una soluzione mediante un **modello di analisi**

La **progettazione** vera e propria si occupa di

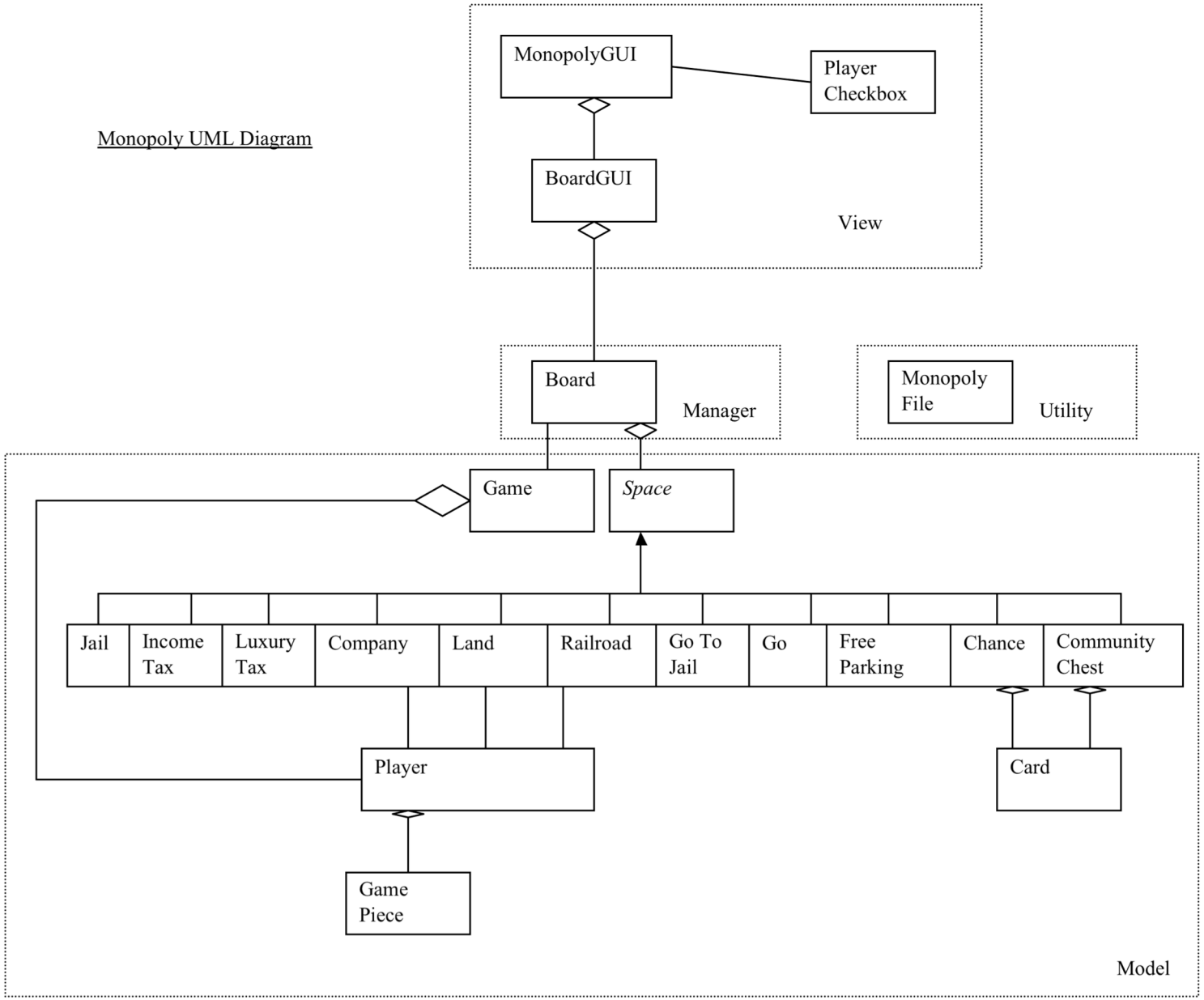
- Descrivere (anticipare) una soluzione al problema mediante un **modello progettuale** che di solito si ispira ad un qualche **paradigma progettuale**


Elicitazione
Specifica dei requisiti
Requisiti non-funzionali
Modello funzionale
Analisi
Modello dell'analisi
Modello dinamico
Modello degli oggetti dell'analisi
Progetto sistema
Progetto degli oggetti del sistema

Modello di un'applicazione sw

- Un *modello* è una rappresentazione semplificata della realtà che contiene informazioni ottenute focalizzando l'attenzione su alcuni aspetti cruciali e ignorando alcuni dettagli
- Un *modello di un'applicazione sw* descrive e rappresenta in modo semplificato la sua struttura ed il suo comportamento

Modello sw: esempio

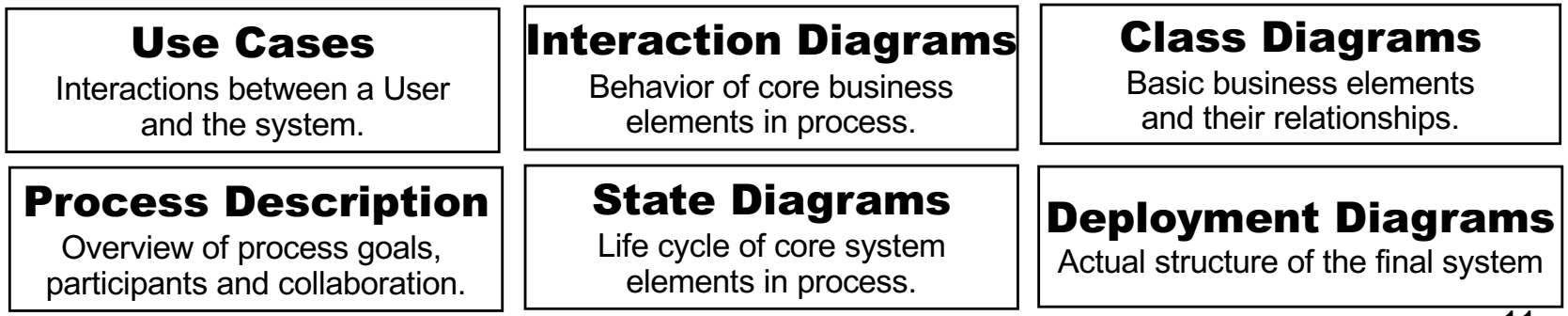


I modelli del sw richiedono più viste

Un buon modello include almeno quattro viste:

- Vista **di contesto** (dell'ambito di uso o scopo)
- Vista **Funzionale** (compiti in relazione allo scopo)
- Vista **Strutturale** (elementi e relazioni tra gli oggetti)
- Vista **Comportamentale** (dinamica e interazioni)

Esempio: in UML abbiamo



Perché più viste

I. La vista **di contesto** descrive ambito e scopo del sistema da progettare (es. “Vision document”))
II. La vista **funzionale** descrive la forma di un sistema in relazione allo scopo (es. “Documento dei Casi d’uso”)
III. La vista **comportamentale** modella il comportamento di un sistema in relazione ai possibili input
IV. La vista **strutturale** modella i componenti di un sistema e le loro relazioni

Le prime due viste sono “**soggettive**” (descrivono le intenzioni del progettista), invece le seconde sono “**oggettive**” (modellano proprietà future del sistema)

Visione e ambito: struttura del documento tipo

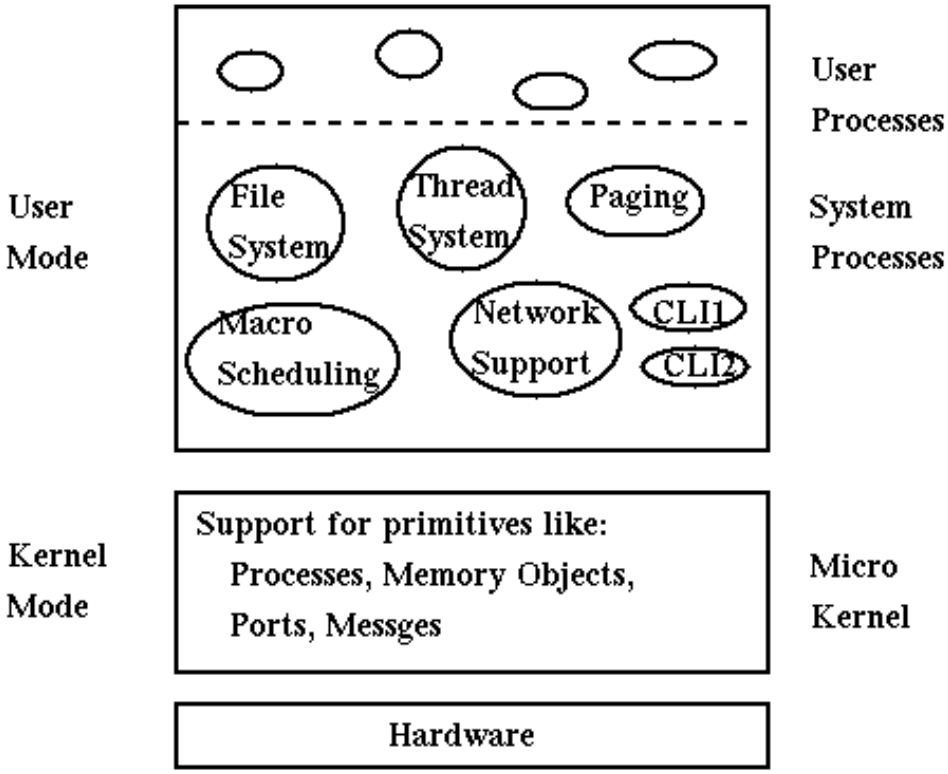
Fasi della progettazione sw

Dopo l'analisi, distinguiamo due fasi di progettazione:

- Progettazione **architetturale**
  - Decomporre il modello del sistema in moduli
  - Determinare le relazioni tra i moduli
- Progettazione **dettagliata**
  - Specifica delle interfacce dei moduli
  - Descrizione funzionale dei moduli

L’architettura permette di controllare la complessità

L’architettura di un sistema software definisce una visione comune che tutte le parti interessate allo sviluppo devono accettare e condividere



Micro–Kernel Architecture

Descrivere l'architettura sw

Non esistono metodi universali per descrivere le architetture sw

- Disegni informali
- Diagrammi UML (vari)
- Linguaggi architetturali

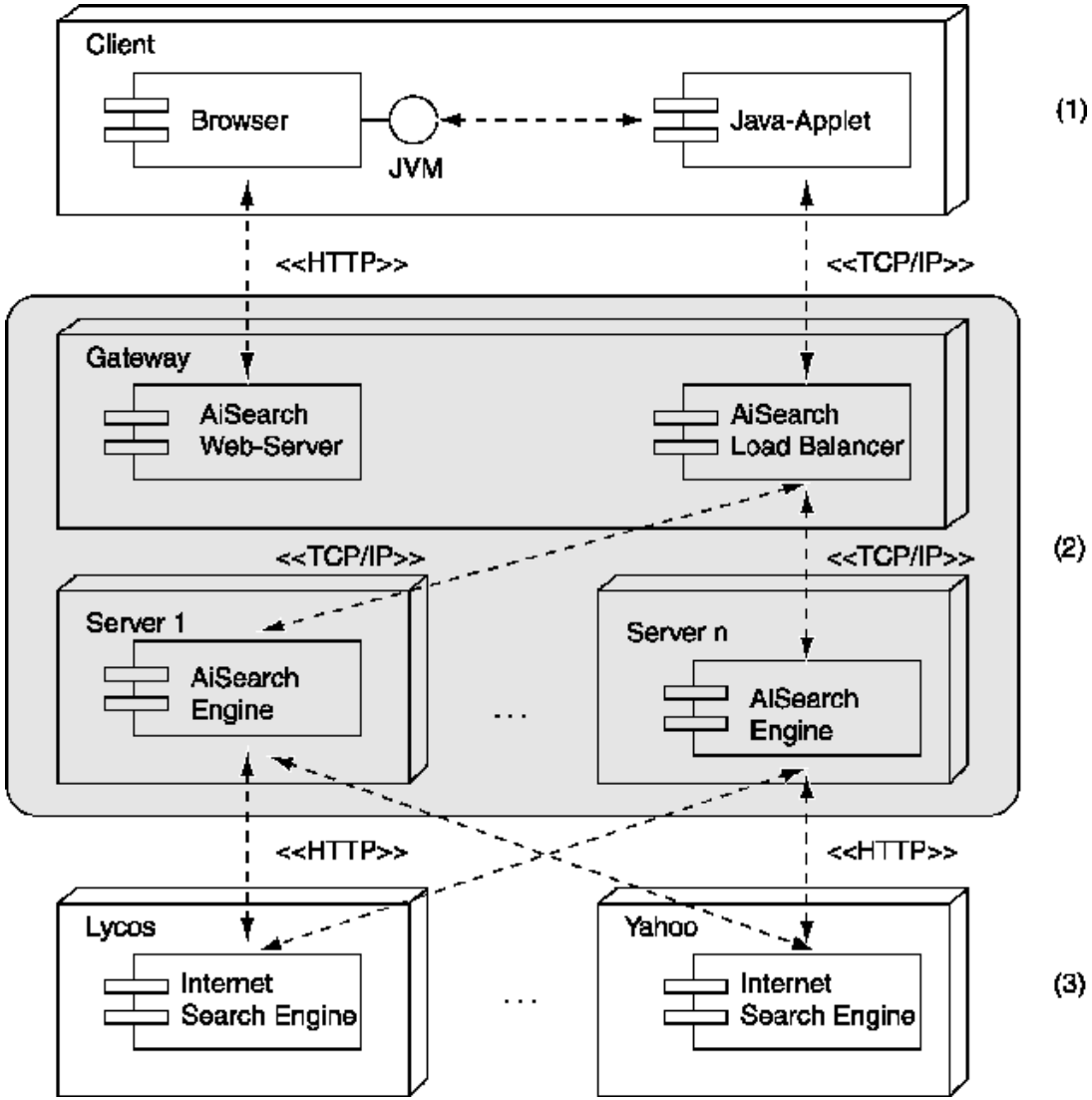


Diagramma di deployment in UML

Livelli di astrazione

- Strutture dati e algoritmo
- Classe (strutture dati e molti algoritmi)
- Package/Moduli (gruppi di classi correlate, magari interagenti via design pattern)
- Moduli/Sottosistemi (moduli interagenti contenenti ciascuno molte classi; solo le interfacce pubbliche interagiscono con altri moduli/sottosistemi)
- Sistemi (sistemi interagenti con altri sistemi - hardware, software ed esseri umani)

La **progettazione architetturale** riguarda gli ultimi 2 livelli

Regole di progettazione

- Semplicità
  - Nessun sistema nasce complesso
  - Un design semplice spesso è elegante
  - Il codice semplice è più facile da capire e correggere
- Affidabilità e robustezza
  - *Primum non nocere*: non progettare sistemi che fanno danni
  - Il sistema dovrebbe essere capace di gestire input inattesi
  - Il sistema dovrebbe superare ogni errore senza crash
- Efficacia
  - Bisogna risolvere il problema “giusto”
  - Bisogna risolvere il problema in modo efficiente
  - Prevedere integrazioni e compatibilità con altro software

Discussione

- Cos’è la buona progettazione?

“buona” progettazione

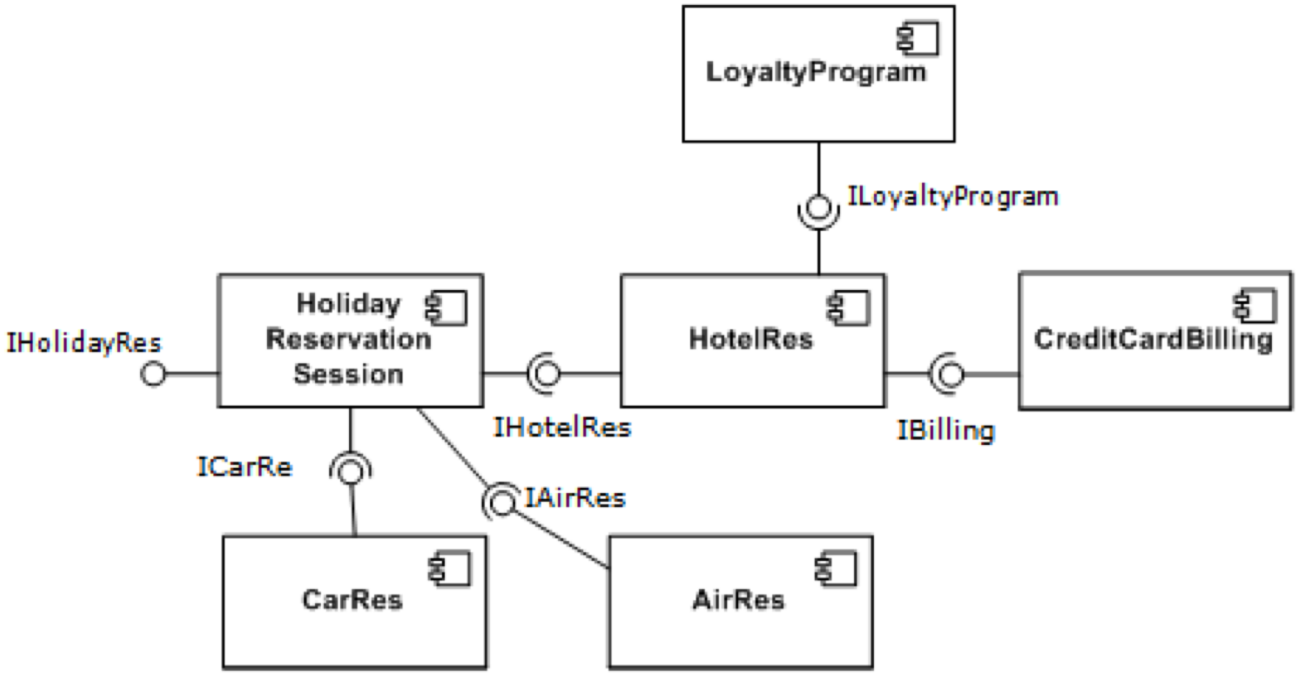
- Astrazione
- Information hiding
- Separazione dei concern

Componenti

- I sistemi software sono fatti di componenti
- Esempio: il Web si basa su un componente chiamato browser (client) e su uno chiamato Webserver; entrambi usano un componente che implementa http
- Il client dipende dal server; entrambi dipendono da http

Componente: elemento a “run time”

- Un ***componente*** di un sistema ha un’*interfaccia* ben definita verso gli altri componenti
- La progettazione dovrebbe facilitare la composizione, il riuso e la riparazione dei componenti, e rendere flessibile la riconfigurazione del sistema



Modulo: elemento a “design time”

- Un modulo software è *unità di progettazione* in quanto *contenitore* di programmi e strutture dati
- I moduli sono anche *unità di incapsulamento* quando permettono di separare l’interfaccia dal corpo
  - L’interfaccia del modulo esprime ciò che il modulo offre o richiede; gli elementi dell’interfaccia sono visibili agli altri moduli.
  - Il corpo del modulo contiene il codice corrispondente agli elementi dell’interfaccia e realizza la semantica del modulo
- I moduli sono anche *unità di compilazione* quando possono essere compilati separatamente
  - Questo facilita lo sviluppo da parte di persone che lavorano in modo indipendente, il riuso, le riparazioni e le riconfigurazioni

Esempio

Questa interfaccia e le classi dovrebbero essere inserite in un unico package perché:

- I tipi sono tutti correlati (coesione): hanno tutti a che vedere con la grafica.
- Chi usa il modulo come libreria ricorderà facilmente cosa può trovarci dentro
- Il package crea un namespace specifico che evita conflitti con altri namespace
- I tipi entro il package hanno visibilità completa sugli altri tipi entro lo stesso package

```
// in the Draggable.java file
public interface Draggable {…}

// in the Graphic.java file
public abstract class Graphic {…}

// in the Circle.java file
public class Circle extends Graphic
    implements Draggable {…}

// in the Rectangle.java file
public class Rectangle extends Graphic
    implements Draggable {…}

// in the Point.java file
public class Point extends Graphic
    implements Draggable {…}

// in the Line.java file
public class Line extends Graphic
    implements Draggable {…}
```

Riusabilità

- La *riusabilità* di un modulo software M è la probabilità che M possa essere usato di nuovo in un sistema diverso da quello per cui è stato progettato inizialmente
- I moduli riusabili
  - sono più facilmente modificabili rispetto a moduli non riusabili
  - aumentano la produttività di chi li riusa
  - sono testati di più
  - costano di più di corrispondenti moduli non riusabili

Riuso

- Il ***riuso*** è l'impiego effettivo di un modulo M, creato per un sistema A, in un sistema B
- La ***riusabilità*** di M è una condizione necessaria per il suo riuso, ma non sufficiente
- (per es. M potrebbe essere riusato per B, ma non succede perché: M non ci appartiene; oppure occorrono informazioni di cui non disponiamo; oppure non è abbastanza coeso; ecc.)
- Dunque non bisogna confondere riusabilità e riuso

Legge di Joel

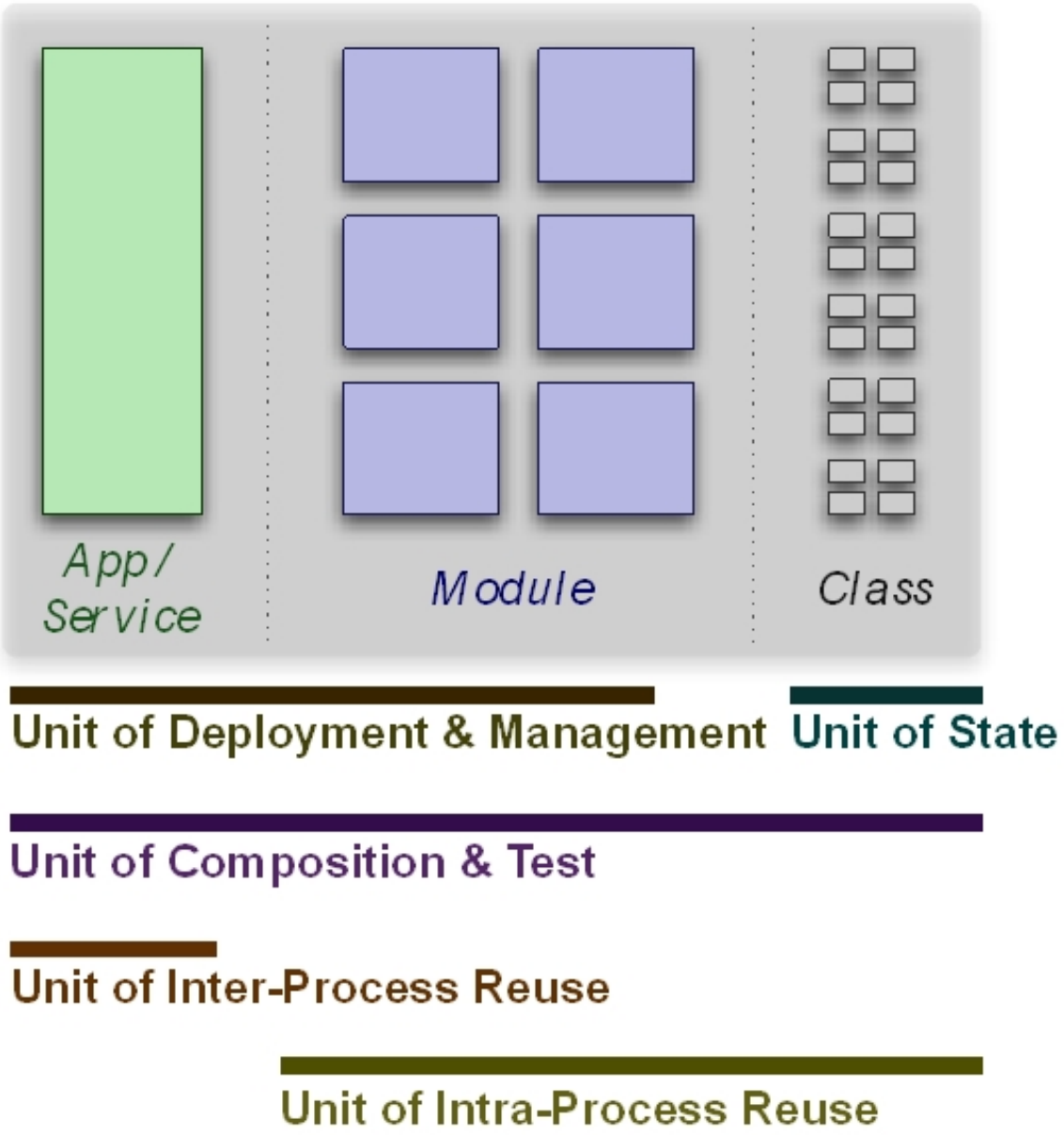
- Leggere codice è più difficile che scriverlo
- Conseguenza: sarete spesso tentati di riscrivere da zero un software che non capite, ma non avete alcuna garanzia che il nuovo tentativo andrà meglio del precedente

http://www.joelonsoftware.com/articles/fog0000000069.html

Cosa si può riusare

- Linee di codice
- Moduli (es. procedure o classi)
- Schemi di classi (*design patterns*)
- Schemi architetturali (*architectural patterns*)
- Componenti (es. webserver)
- Artefatti vari: dei requisiti, dei test, del processo, ecc.

Moduli, classi, e tipi di riuso



Progetto modulare

- La principale proprietà di un sistema ben progettato è la modularità
- Un sistema modulare è divisibile in parti più piccole (moduli) che possono essere create indipendentemente e usate in sistemi diversi
- Esempi: automobili, computer

Gerarchia di moduli

- La gerarchia dei moduli di un sistema sw riduce le dipendenze vincolando la topologia delle relazioni tra i moduli
- La **struttura gerarchica** dei moduli forma la **base di progetto**
  - Decompone il dominio del problema
  - Facilita lo sviluppo parallelo
  - Isola le ramificazioni di modifica e versionamento
  - Permette la prototipazione rapida

Alcune relazioni gerarchiche tra moduli

- `X usa Y` (dipendenza funzionale, es. *call*)
- `X is_a Y` (ereditarietà)
- `X has_a Y` (X contiene Y, ovvero Y è parte di X)
  - Es. “Un programma *has* 10 procedure”
- `X è_composto_da` {Y,Z,W}
  - Es. “Un programma *è_composto_da* 2 package”

  (decomposizione architetturale: solo le foglie sono “vero” codice)

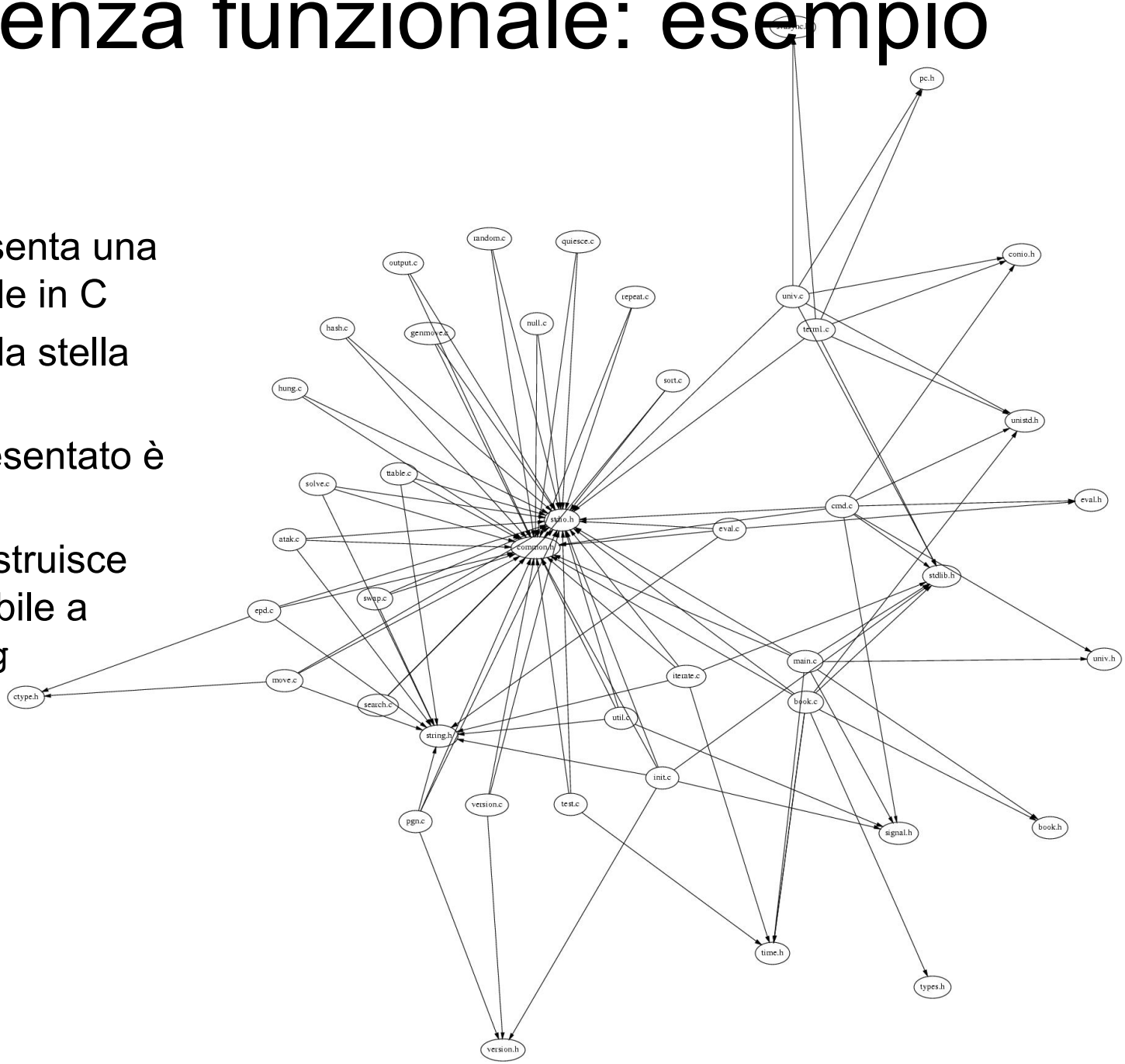
Dipendenza funzionale: esempio

Questo grafo rappresenta una dipendenza funzionale in C

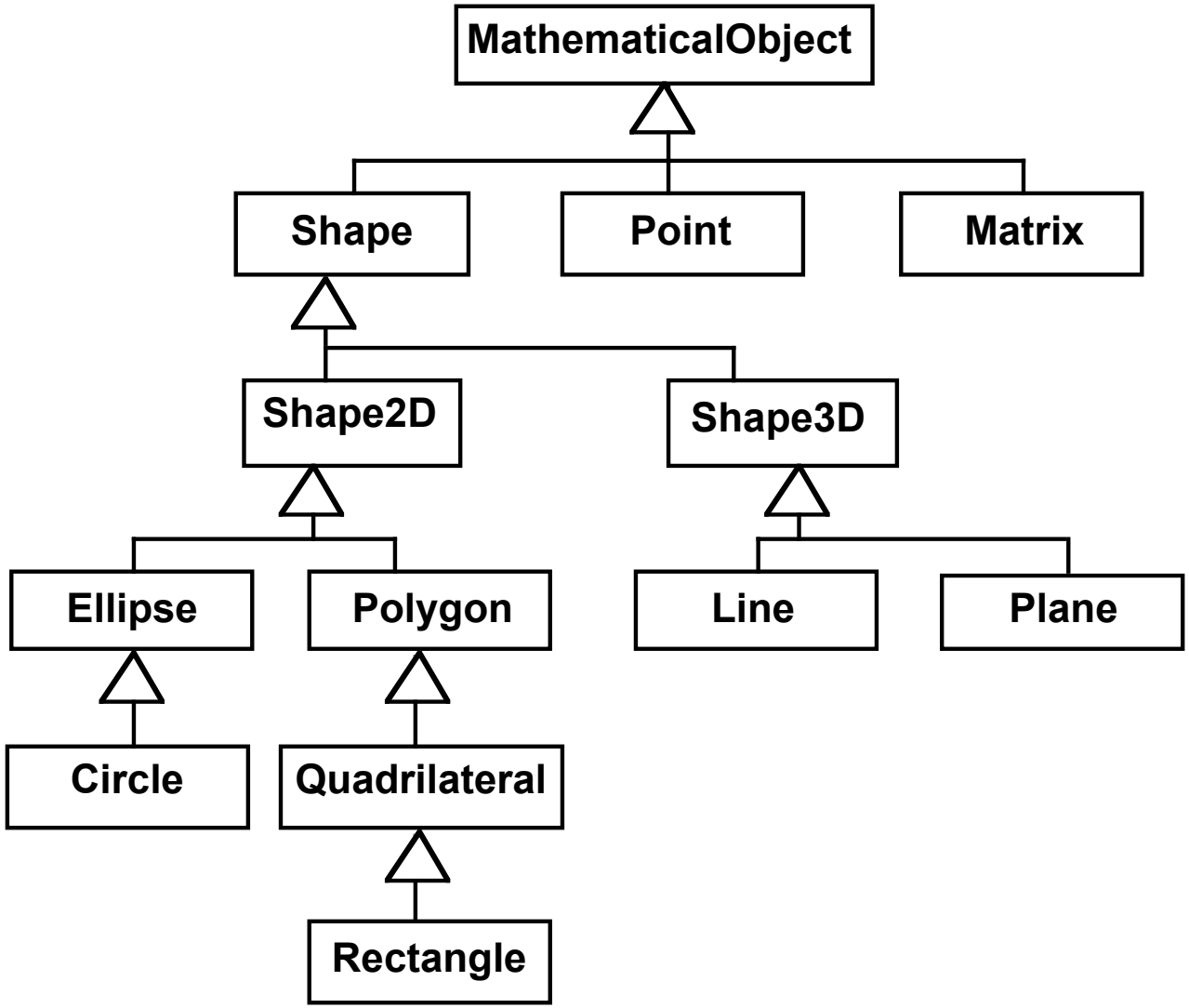
I moduli al centro della stella sono degli #include

IL programma rappresentato è GNUChess5.0

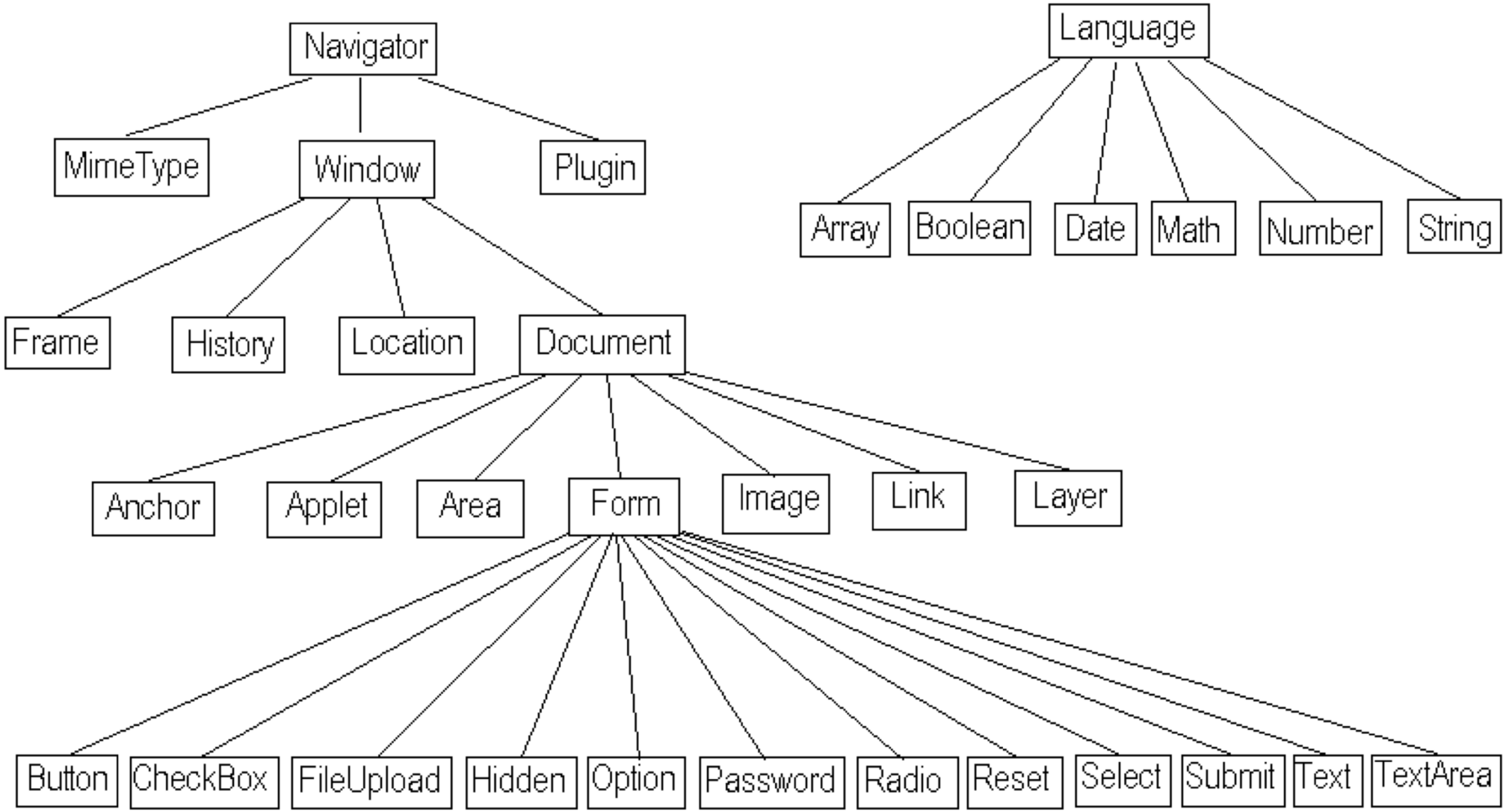
Il programma che costruisce questi grafi è disponibile a `www.graphviz.org`



Ereditarietà: esempio



JavaScript Object Hierarchy

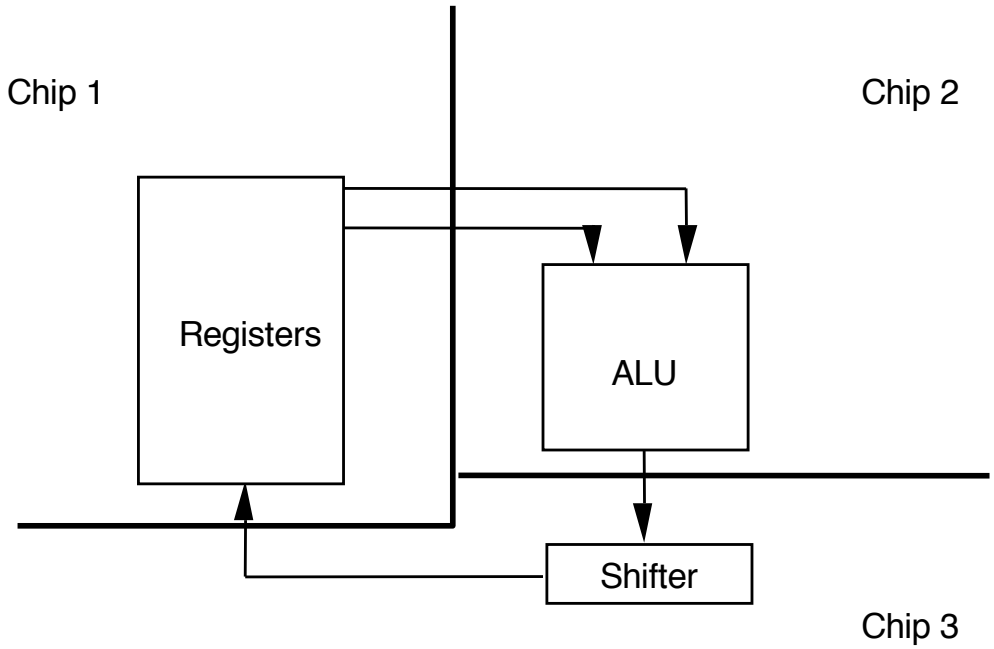


Dipendenze

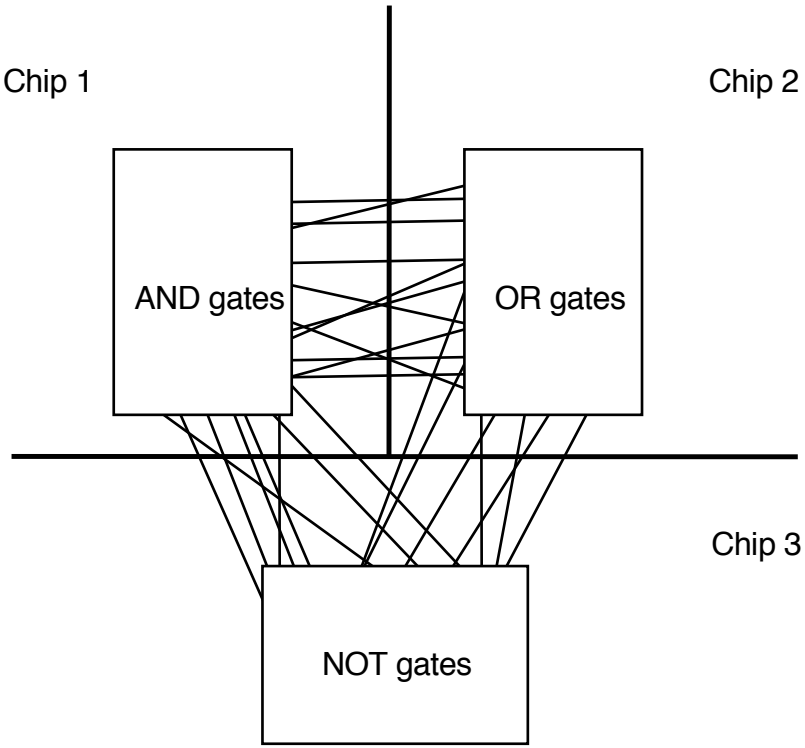
- Le dipendenze che espone un componente ostacolano il suo riuso
- Se il componente A dipende dai componenti B,C,D, allora:
  - riusare A implica dover riusare B,C,D
  - per capire come funziona A occorre capire come funzionano B,C,D
  - se B o C o D si guastano di solito si guasta anche A
  - se B o C o D cambiano di solito occorre cambiare anche A

Progetto semplice o complesso?

CPU con 3 chip (classi)



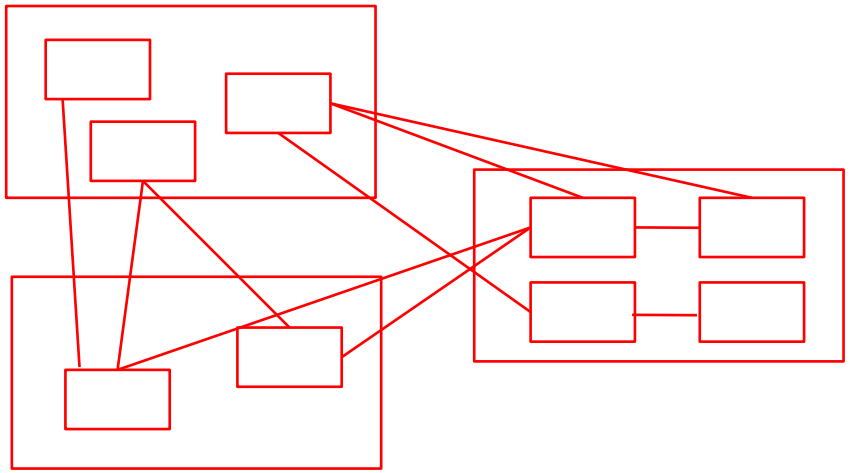
CPU con 3 chips (classi)



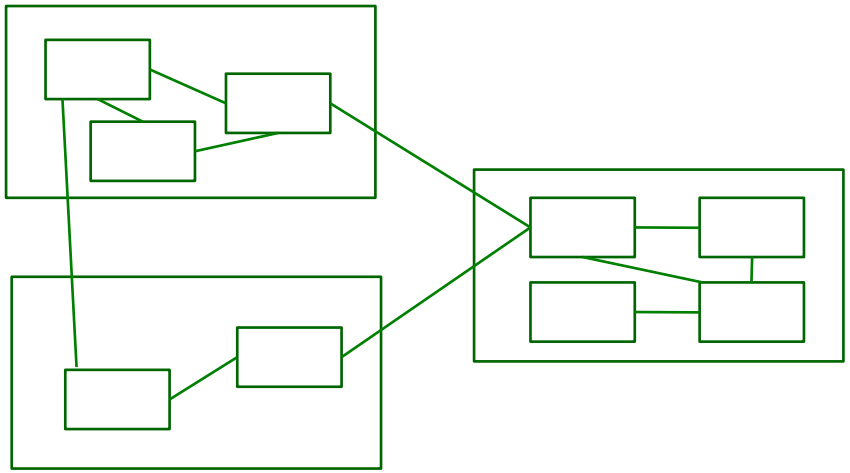
Progetto semplice o complesso?

- I due progetti sono funzionalmente equivalenti, ma quello di destra è
  - Difficile da capire
  - Difficile da correggere
  - Difficile da estendere o migliorare
  - Difficile da riusare
  - Costoso da manutenere
- Il progetto di sinistra è migliore:
  - Massimizza le relazioni intraclasse (**coesione**)
  - Minimizza le relazioni interclasse (**accoppiamento**)

Accoppiamento



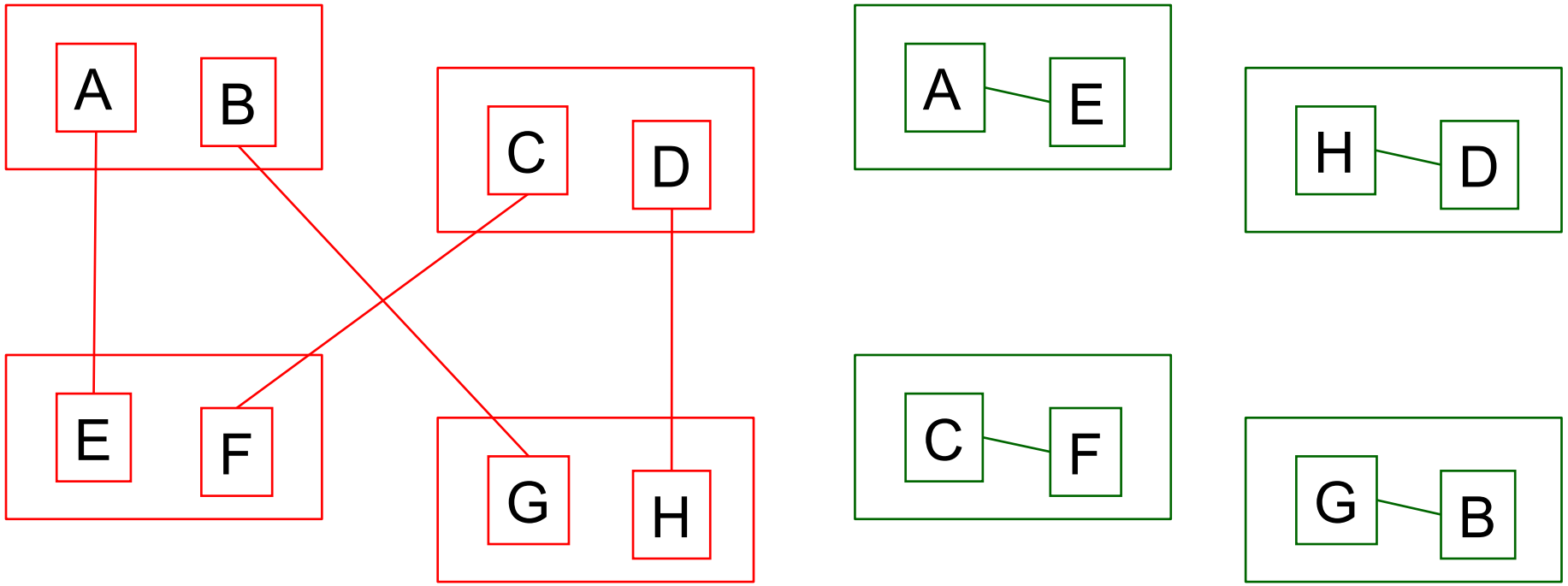
Accoppiamento stretto



Accoppiamento lasco

dipendenza

Coesione



Bassa coesione

Alta coesione

relazione funzionale

Principi di progettazione

La buona progettazione minimizza le dipendenze tra i moduli

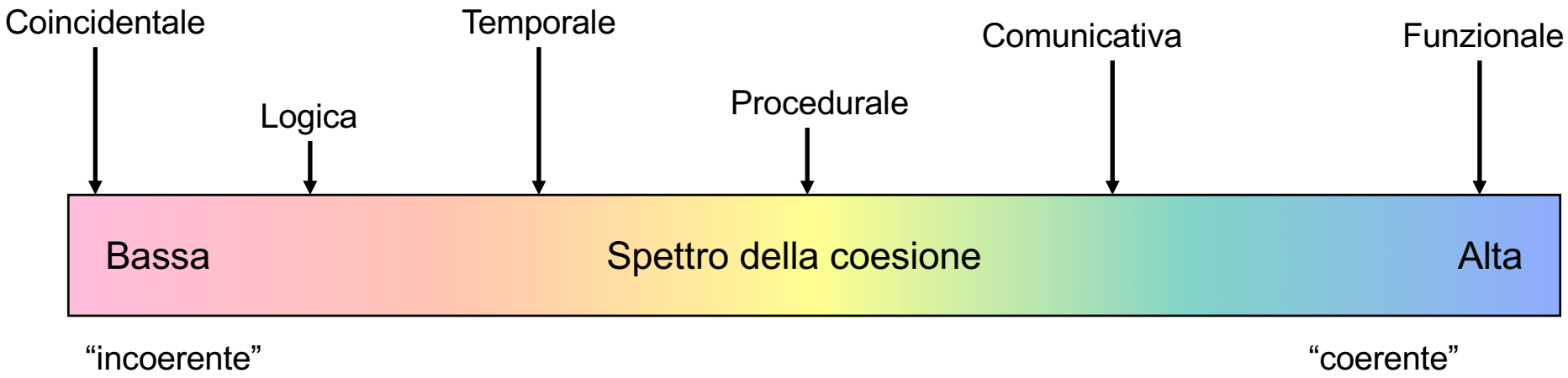
Per minimizzare le dipendenze tra moduli:

- Ogni modulo dovrebbe essere coeso
- Ogni modulo dovrebbe essere disaccoppiato

Coesione di un modulo

- **Coesione**: misura della coerenza funzionale di un modulo software
- Ogni modulo dovrebbe avere **alta coesione**
- Un modulo ha massima coerenza funzionale se fa una cosa sola - e la fa bene
- Un modulo ha buona coesione se le cose che fa sono funzionalmente correlate
- Conseguenza: le classi incluse in grossi moduli dovrebbero essere funzionalmente correlate

Tipi di coesione



Coincidentale: Molteplici azioni o funzioni completamente scorrelati
Logica: serie di azioni o funzioni correlate (e.g. libreria di funzioni di IO)
Temporale: serie di azioni o funzioni simultanee (e.g. moduli di inizializzazione)
Procedurale: serie di azioni che condividono sequenze di passi
Comunicativa: coesione procedurale sugli stessi dati
Sequenziale: coesione comunicativa in cui si opera sui dati sequenzialmente
Funzionale: modulo contenente una sola azione o funzione

Esempio: coesione coincidentale

- L'oggetto non rappresenta una singola nozione o racchiude codice scorrelato

```
class esempio   {
   public static int findPattern(String text, String pattern)
         { // blah}
   public static int average( Vector numbers )
         { // blah}
   public static OutputStream openFile(String fileName)
         { // blah}
   }
```

Esempio: coesione logica

- Il modulo contiene funzioni correlate;
  se ne sceglie una mediante un parametro di invocazione

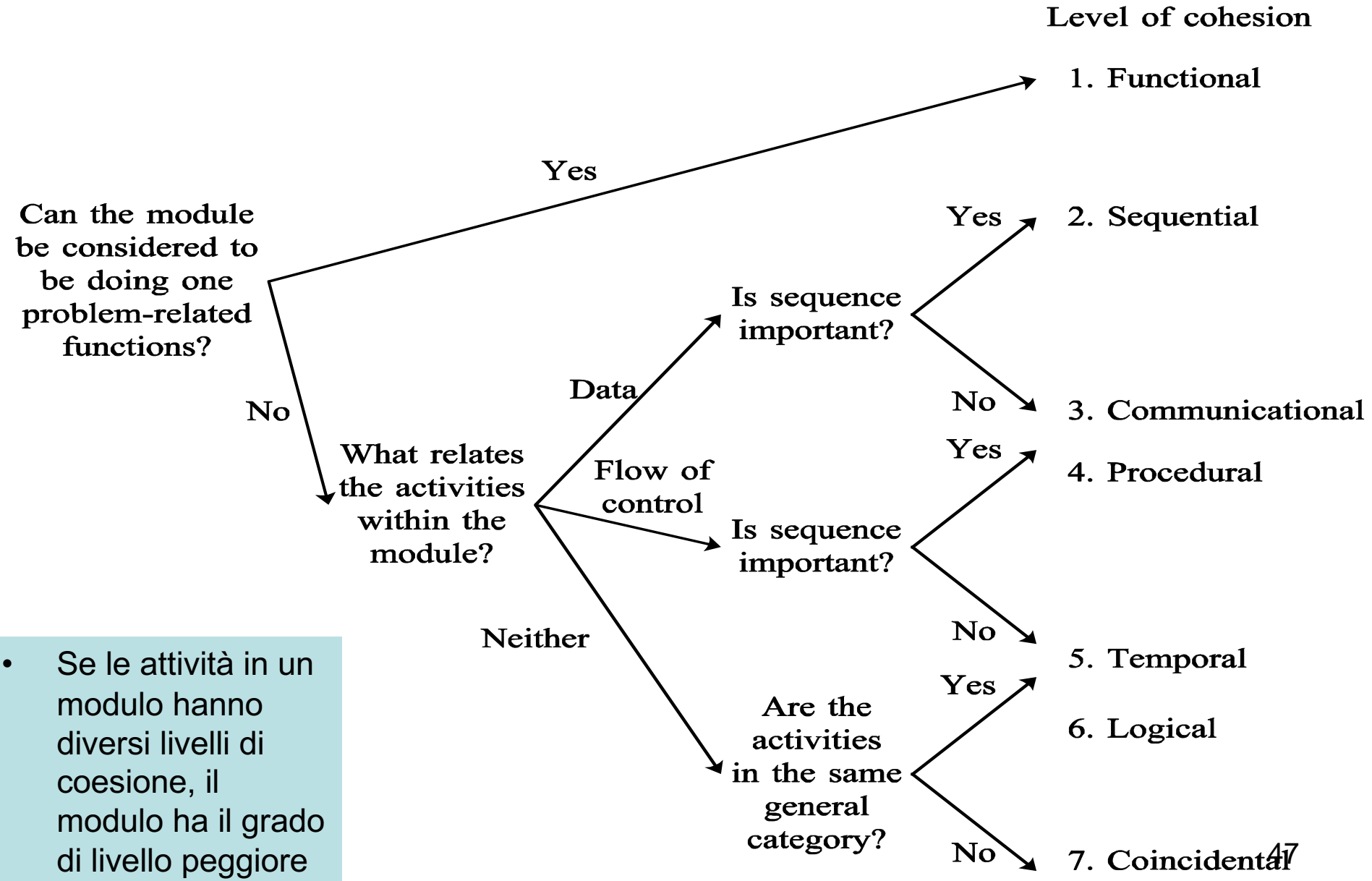
```
public void esempio( int flag )   {
   switch ( flag )       {
        case ON:         // codice ON          break;
        case OFF:        // codice OFF          break;
        case CLOSE:      // codice CLOSE          break;
        case COLOR:      // codice COLOR          break;
}   }
```

Coesione temporale

- Il modulo raccoglie comandi che vengono elaborati in un certo breve periodo di tempo
- Esempio tipico: modulo di inizializzazione

```
procedure initializeData()   {
     font = "times";
     windowSize = "200,400";
     foo.name = "Not Set";
     foo.size = 12;
     foo.location = "/usr/local/lib/java";
}
```

Decidere il livello di coesione



- Se le attività in un modulo hanno diversi livelli di coesione, il modulo ha il grado di livello peggiore

Accoppiamento dei moduli

- **Accoppiamento**: Misura del grado di dipendenza di un insieme di moduli software
- I moduli di un sistema dovrebbero esibire un **basso accoppiamento** (detto anche **accoppiamento lasco**)
- Questo si ottiene quando ci sono solo poche *dipendenze* tra i moduli, tutte funzionalmente rilevanti e necessarie
- Il software ad alto accoppiamento è fatto male (difficile da capire, mantenere, correggere, ecc…)

Ridurre l’accoppiamento

L’accoppiamento tra i moduli si riduce:

- eliminando relazioni non necessarie
- riducendo il numero delle relazioni necessarie
- rilassando le relazioni necessarie

Quale codice è meno accoppiato?

- Stampa

```
void ciao(){
  cout << “Ciao!”;
}
```

- Funzione stringa

```
string ciao(){
  return “Ciao!”;
}
```

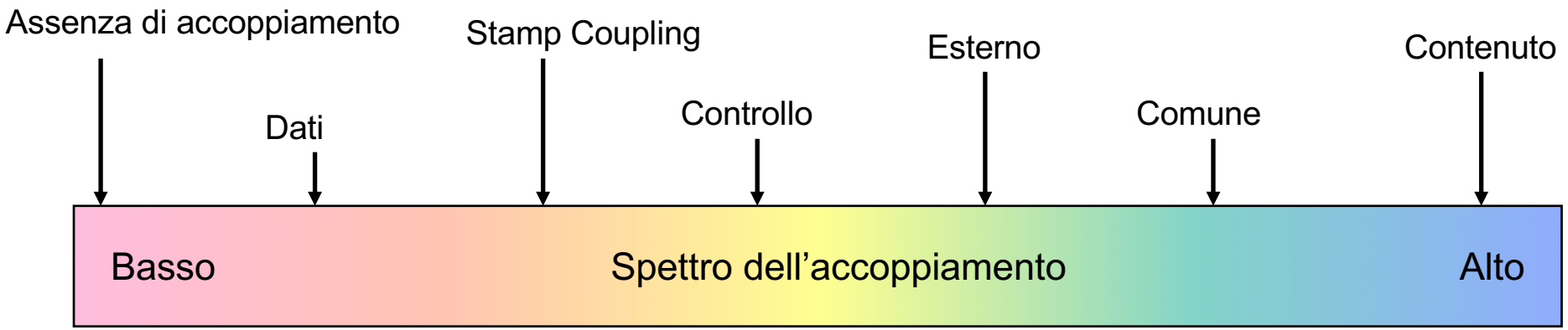
La stampa dipende dallo schermo; la funzione stringa dalla definizione di “string”

La funzione stringa è migliore, perché il codice di stampa è dipendente dalla console, e non può stampare in altri modi (su file, in una finestra grafica, ecc.)

L'accoppiamento è una dipendenza

- Perché è male che un modulo A dipenda da un altro modulo, B?
- Se il modulo A dipende da B, allora
  - Se B cambia, anche A può dover cambiare
  - A può funzionare solo con B, il che rende A meno riusabile
  - A è più difficile da capire perché occorre capire anche B

Tipi di accoppiamento



Misura dell'indipendenza funzionale di un insieme di moduli

Contenuto: un modulo che referenzia il contenuto di un altro modulo
Comune: due moduli hanno accesso agli stessi dati globali
Esterno: dati comuni tra due moduli con accesso strutturato
Controllo: un modulo passa un elemento di controllo ad un altro modulo
Stamp: un modulo fornisce parte di una struttura dati per un altro modulo
Dati: un modulo produce una struttura dati per un altro modulo che la consuma

Accoppiamento sui dati

- Output di un modulo è input di un altro
- Esempio: A passa X a B, dunque X e B sono accoppiati, perché una modifica all'interfaccia di X richiede una modifica all'interfaccia di B

```
class Receiver   {
  public void message( MyType X )
  {       // qui c'è del codice
    X.doSomethingForMe( Object data );
            // altro codice       }
  }
```

Coesione e accoppiamento

Coesione	Accoppiamento
È una relazione tra elementi dello stesso modulo	È una relazione tra moduli diversi
Mostra la coerenza funzionale del modulo	Mostra l'interdipendenza tra i moduli
È una qualità del sorgente del modulo che misura il grado di correlazione tra gli elementi del modulo	È una qualità del sorgente dei moduli che misura la quantità di connessioni tra i diversi moduli

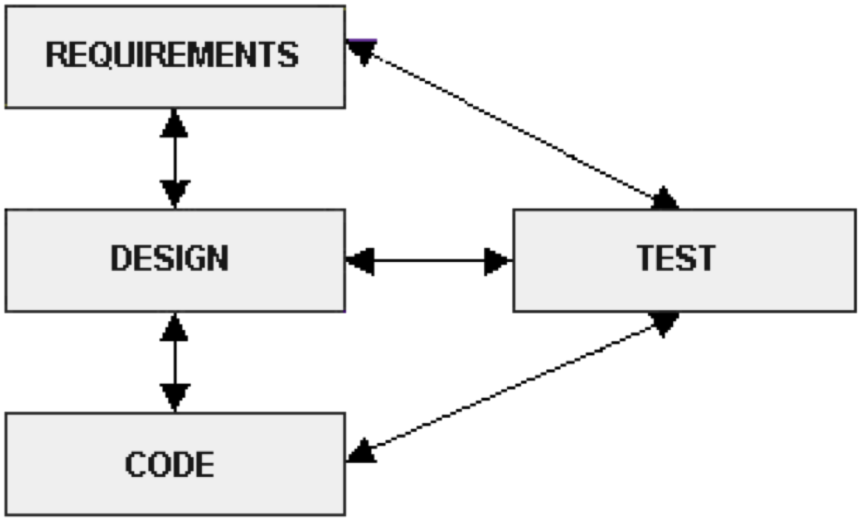
Standard IEEE1016

- Lo standard IEEE 1016-1998 *Software Design Description* specifica la forma del documento di progetto di un sistema software
- Un diagramma importante di questo standard è la Requirements Traceability Matrix (RTM)

IEEE1016

1. INTRODUCTION
   - Design Overview
   - Requirements Traceability Matrix
2. SYSTEM ARCHITECTURAL DESIGN
   - Chosen System Architecture
   - Discussion of Alternative Designs
   - System Interface Description
3. DETAIL DESCRIPTION OF COMPONENTS
   - Component 1
   - Component n
4. USER INTERFACE DESIGN
   - Description of the User Interface
     - Screen Image
     - Objects and Actions
5. ADDITIONAL MATERIAL

Matrice di tracciabilità dei requisiti



ID	USER REQUIREMENTS	FORWARD TRACEABILITY
U2	Users shall process retirement claims.	S10, S11, S12
U3	Users shall process survivor claims.	S13

ID	FUNCTIONAL REQUIREMENTS	BACKWARD TRACEABILITY
S10	The system shall accept requirement data.	U2
S11	The system shall calculate the amount of retirement.	U2
S12	The system shall calculate point-to-point travel time.	U2
S13	The system shall calculate the amount of survivor annuity	U3

Il design del sw nel SWEBOK

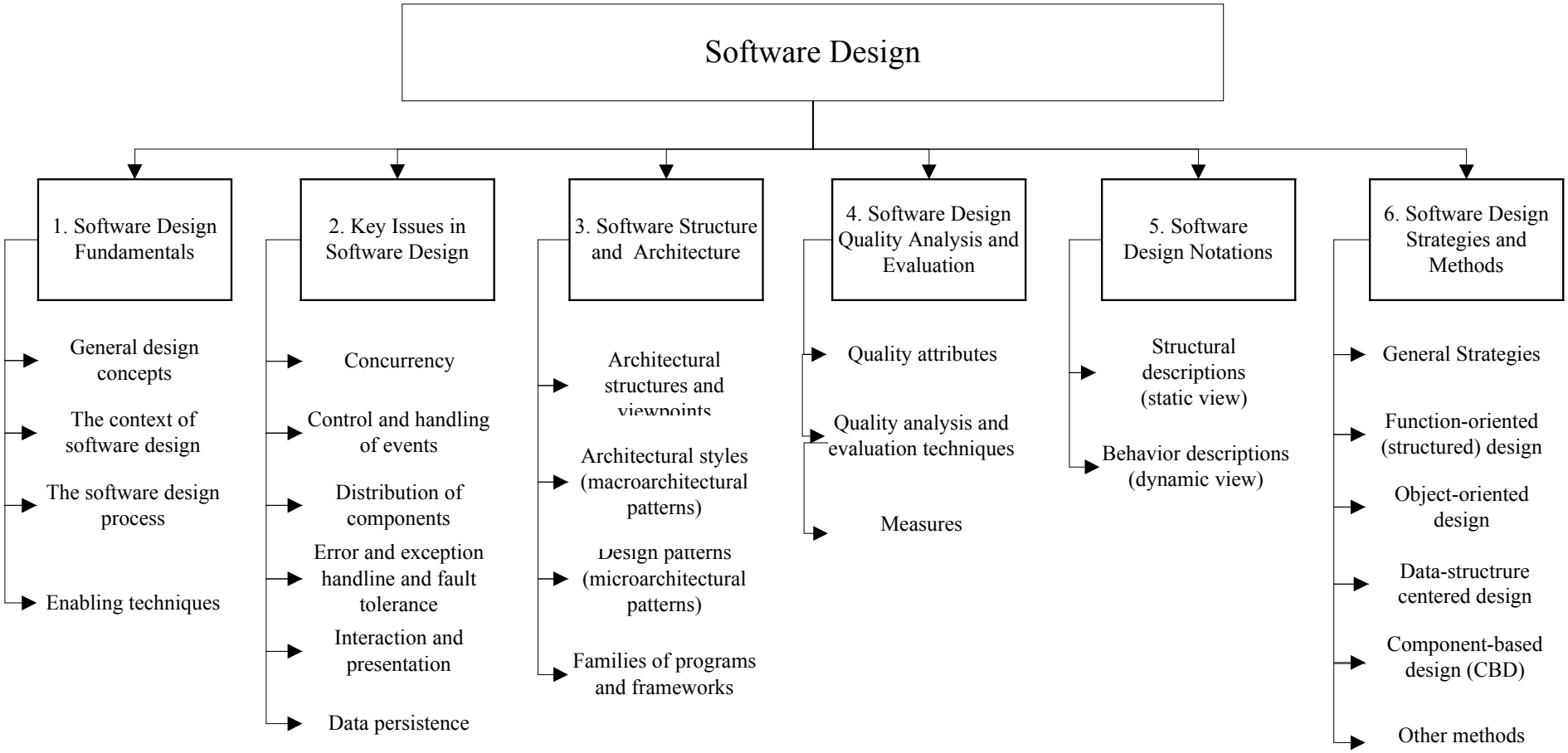


Figure 1 Breakdown of topics for the Software Design KA

Sommario

- La progettazione di un sistema avviene mediante la costruzione di un modello
- Esistono parecchie viste possibili sui modelli del software
- I modelli risulteranno ben progettati se composti da moduli coesi e disaccoppiati
- Le gerarchie sui moduli sono un modo di controllare la complessità progettuale

Domande di autotest

- Cos'è un modello di un'applicazione sw?
- Cos'è un modulo software?
- Come si misura il grado di coesione di un modulo?
- Come si misura il grado di accoppiamento tra due moduli? E tra tre moduli?
- Quali sono le principali gerarchie presenti in un codice sorgente software?

Lettura raccomandata

- Taylor & van der Hoek, Software Design and Architecture: The once and future focus of software engineering, *Int. Conf on the Future of Software Engineering*, 2007

Riferimenti

- Larman, *Applying UML and patterns*
- Capitolo 3 del SWEBOK “Software design”
- IEEE Recommended Practice for Software Design Descriptions (IEEE 1016-1998)
- Winograd, *Bringing Design to Software*, AW 1996
  `hci.stanford.edu/publications/bds/`
- Norman, *La caffettiera del masochista*, Giunti 1997

Siti

- www.booch.com
- www.joelonsoftware.com
- neverletdown.net
- blog.johanneslink.net/2009/03/06/a-unified-theory-of-software-design/

Pubblicazioni di ricerca

- IEEE Transactions on Software Engineering
- ACM Transactions on Software Engineering and Methodology

- International Conference on Software Engineering
- European Conference on Software Engineering
- ACM Conf. on Fundamentals of Software Engineering
- Automated Software Engineering Conference
- Empirical Software Engineering Conference

Domande?