UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

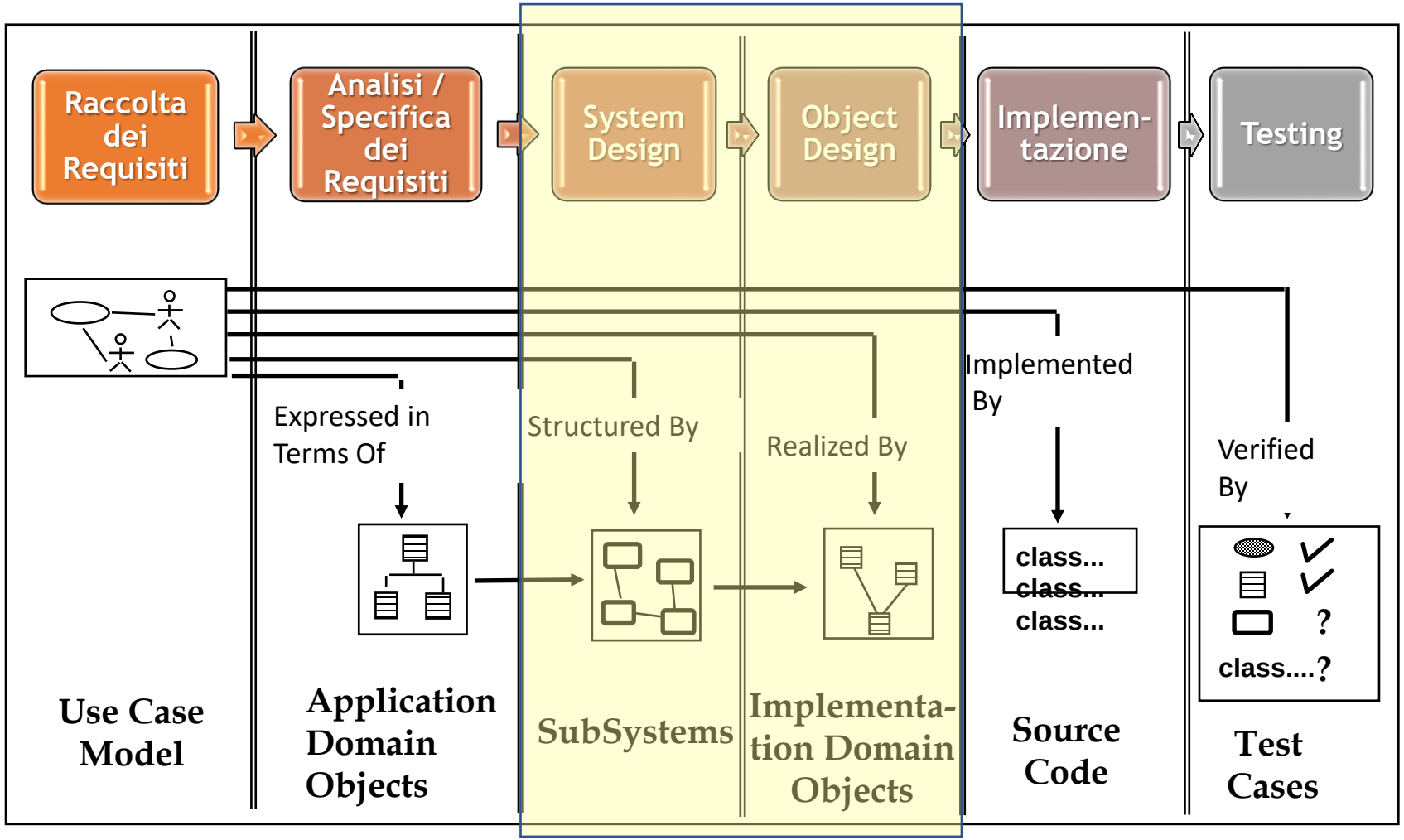
Ingegneria del Software – Software Design

Prof. Sergio Di Martino

Obiettivi della lezione

- Comprendere le attività del Design
  - Strutturazione in sottosistemi
    - Rispettando gli obiettivi del sistema
  - Linee guida per la definizione di artefatti
    - Concetti di Coesione ed Accoppiamento tra artefatti

Software Lifecycle Activities



Progettazione di un sistema

- La progettazione di un sistema (Software Design) è l'insieme dei task svolti dall'ingegnere del software per trasformare il modello di analisi nel modello di design del sistema
- Obiettivo ultimo di System e Object Design è definire un nuovo documento, scritto in linguaggio tecnico/formale, da dare ai programmatori affinchè questi siano in grado di implementare il software descritto nel Documento dei Requisiti Software

Scopo del system design

- Definire gli obiettivi di design del progetto
- Definire **l'architettura** del sistema, decomponendolo in sottosistemi più piccoli che possono essere realizzati da team individuali
- Selezionare le strategie per costruire il sistema, quali:
  - Strategie hardware/software
  - Strategie relative alla gestione dei dati persistenti
  - Il flusso di controllo globale
  - Le politiche di controllo degli accessi
  - …

Scopo dell'object design

- Data l'architettura del Sistema, specificare il dettaglio realizzativo dei sottosistemi più piccoli
- Selezionare le strategie ottimali per l'implementazione
  - Design Pattern

Output del software design

- Un documento contenente:
  - Una chiara descrizione di ognuna delle strategie elencate nella slide precedente
  - un insieme di modelli statici/dinamici del sistema che includano la decomposizione del sistema in sottosistemi
    - Formalizzato con Class Diagram, Sequence Diagrams, StateChart, etc… di UML
  - un insieme di modelli statici/dinamici che descrivano la progettazione interna dei sottosistemi

II System Design

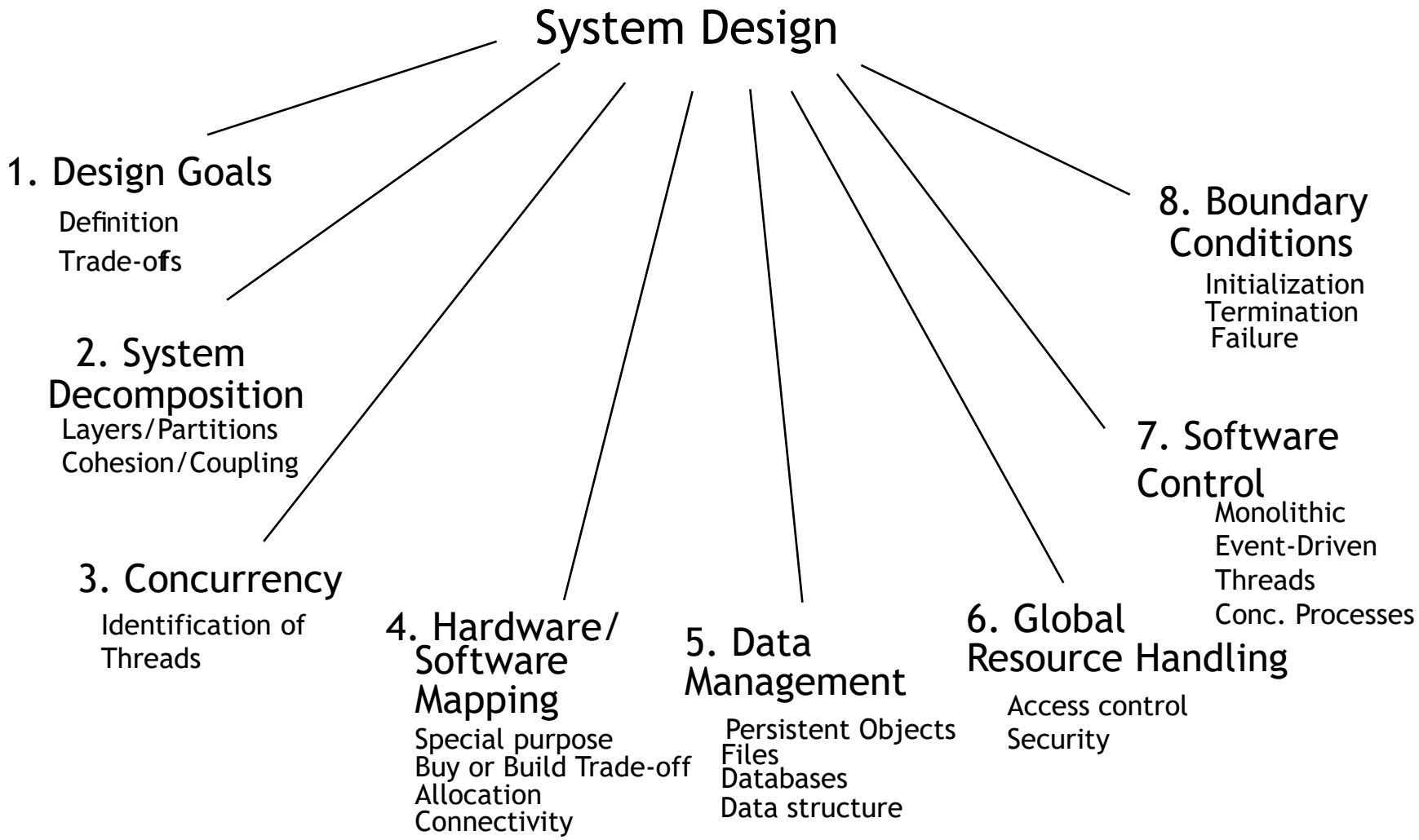
System Design

- Cambia radicalmente punto di vista:
  - Finora abbiamo lavorato per interagire col cliente
    - Linguaggio per profani, poco formale
  - Nel system design i nostri interlocutori sono le squadre di programmatori che dovranno implementare il sistema
    - Chi leggerà i nostri documenti sarà un esperto di informatica
    - Linguaggio per esperti tecnici

Attività di system design

- Il system design è costituito da tre macro-attività:
  1. Identificare gli obiettivi di design:
     - gli sviluppatori identificano quali caratteristiche di qualità dovrebbero essere ottimizzate e definiscono le priorità di tali caratteristiche
  2. Progettazione della decomposizione del sistema in sottosistemi:
     - basandosi sugli use case ed i modelli di analisi, gli sviluppatori decompongono il sistema in parti più piccole. Utilizzano stili architetturali standard.
  3. Raffinare la decomposizione in sottosistemi per rispettare gli obiettivi di design:
     - La decomposizione iniziale di solito non soddisfa gli obiettivi di design. Gli sviluppatori la raffinano finché gli obiettivi non sono soddisfatti.

System Design



Identificare gli obiettivi qualitativi del sistema

- E’ il primo passo del system design
- Identifica le qualità su cui deve essere focalizzato il sistema
- Molti design goal possono essere ricavati dai requisiti non funzionali o dal dominio di applicazione, altri sono forniti dal cliente
- E’ importante formalizzarli esplicitamente poiché ogni importante decisione di design deve essere fatta seguendo lo stesso insieme di criteri

Criteri di design

- Possiamo selezionare gli obiettivi di design da una lunga lista di qualità desiderabili.
- I criteri sono organizzati in cinque gruppi:
  - Performance
    - Tempo di risposta, Throughput, Requisiti di Memoria, …
  - Affidabilità
    - Robustezza, Disponibilità, Tolleranza ai fault, Sicurezza, …
  - Costi
    - Sviluppo, Manutenzione, Gestione, …
  - Mantenimento
    - Estendibilità, Modificabilità, Adattabilità, Portabilità, …
  - Usabilità

Design Trade-offs

- Quando definiamo gli obiettivi di design, spesso solo un piccolo sottoinsieme di questi criteri può essere tenuto in considerazione.
  - Es: non è realistico sviluppare software che sia simultaneamente sicuro e costi poco.
- Gli sviluppatori devono dare delle priorità agli obiettivi di design, tenendo anche conto di aspetti manageriali, quali il rispetto dello schedule e del budget.

Design Trade-offs (cont.)

- Esempi:
  - Spazio vs. velocità. Se il software non rispetta i requisiti di tempo di risposta e di throughput, è necessario utilizzare più memoria per velocizzare il sistema (es. Caching, più ridondanza). Se il software non rispetta i requisiti di memoria, può essere compresso a discapito della velocità.
  - Tempo di rilascio vs. funzionalità. Se i tempi di rilascio sono stringenti, possono essere rilasciate meno funzionalità di quelle richieste, ma nei tempi giusti.
  - Tempo di rilascio vs. qualità. Se i tempi di rilascio sono stretti, il project manager può rilasciare il software nei tempi prefissati con dei bug e, in tempi successivi, correggerli, o rilasciare il software in ritardo, ma con meno bug.
  - Tempo di rilascio vs. staffing. Può essere necessario aggiungere delle risorse al progetto per accrescere la produttività.

Concetti di base di buon design

Qualità del codice

Concetti di System Design

- Molti criteri di Design sono strettamente dipendenti dal problema trattato
  - La scelta del miglior bilanciamento tra i vincoli posti è un compito dell'analista, che opera in base alla propria esperienza/sensibilità
- Esistono però dei Criteri di Design generici, che valgono per qualunque software O-O
  - Questi criteri, se ben applicati, migliorano notevolmente la qualità interna del codice, senza introdurre particolari svantaggi

Pensare in avanti

- Quando progettiamo una classe dovremmo sforzarci di pensare a quali cambiamenti potranno essere richiesti in futuro
- “Progettare per ANTICIPARE IL CAMBIAMENTO”
- Le nostre scelte iniziali devono facilitare l’evoluzione futura
  - E’ una delle richieste fondamentali, nonché fonte di valutazione per il progetto!

La Decomposizione

- Per ridurre la complessità della soluzione, decomponiamo il sistema in parti più piccole, chiamate **sottosistemi**.
  - Un sottosistema tipicamente corrisponde alla parte di lavoro che può essere svolta autonomamente da un singolo sviluppatore o da un team di sviluppatori.
- Decomponendo il sistema in sottosistemi relativamente indipendenti, i team di progetto possono lavorare sui sottosistemi individuali con un minimo overhead di comunicazione.
- Nel caso di sottosistemi complessi, applichiamo ulteriormente questo principio e li decomponiamo in sottosistemi più semplici.

La Decomposizione

- P= problema, C=complessità di P, E=sforzo per la risoluzione di P
- Dati 2 problemi P1 e P2, se C(P1) > C(P2), allora E(P1)> E(P2).
- Ma è stato dimostrato che C(P1+P2)> C(P1) + C(P2), e quindi si ha che E(P1+P2)> E(P1) + E(P2)

Modellazione di sottosistemi

- Subsystem (UML: Package)
  - Collezioni di classi, associazioni, operazioni e vincoli che sono correlati

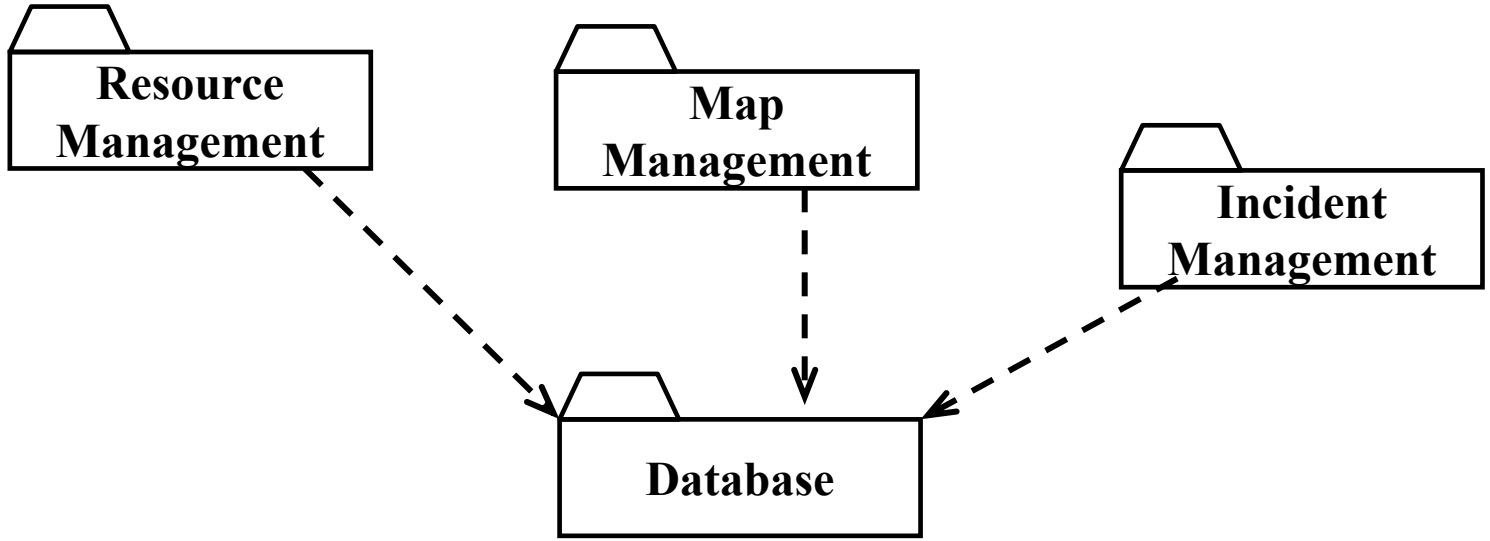
- JAVA fornisce i package che sono costrutti per modellare i sottosistemi
  - C++ / C# hanno il costrutto di namespace per indicare lo stesso concetto

Servizi e interfacce di sottosistemi

- Un sottosistema è caratterizzato dai **servizi** che fornisce agli altri sottosistemi
- L'insieme di operazioni di un sottosistema che sono disponibili agli altri sottosistemi forma **l'interfaccia** del sottosistema
  - Include il nome delle operazioni, i loro parametri, il loro tipo ed i loro valori di ritorno.
- Il system design si focalizza sulla definizione dei servizi forniti da ogni sottosistema in termini di:
  - Operazioni
  - Loro parametri
  - Loro comportamento ad alto livello

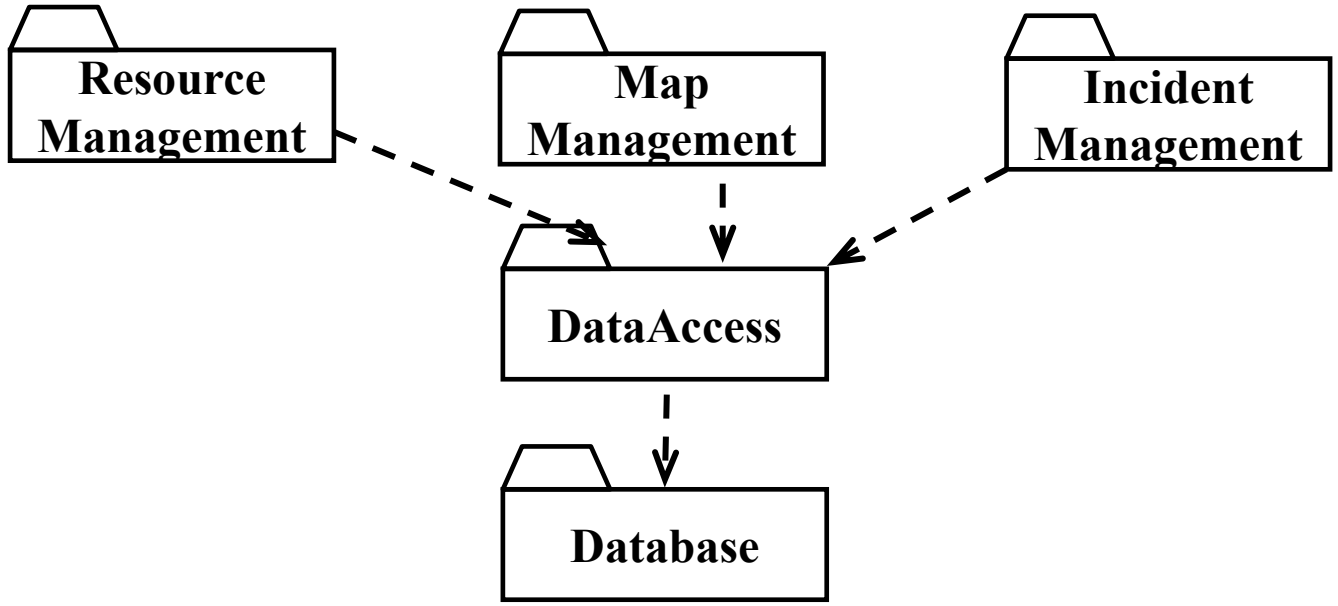
Esempio di scelta di design: accesso diretto al database

- Non riesco a specificare chiaramente le responsabilità di ogni modulo (Gestisce, Salva?)
- Tutti i sottosistemi accedono al database direttamente, rendendoli vulnerabili ai cambiamenti dell'interfaccia del sottosistema Database
  - Se cambia il DBMS, devo modificare il codice in tutte le classi che accedono al DB!



Esempio: accesso al database attraverso un sottosistema di Storage

- Chiariamo le responsabilità:
- Introduco il sottosistema DataAccess che gestisce l'I/O da db
  - Se cambia il DBMS, devo modificare solo il sottosistema Storage!



Osservazioni

- Nell’esempio proposto abbiamo ridotto le relazioni fra i quattro sottosistemi, ma …
- Abbiamo aumentato la complessità!
- Con l’obiettivo di ridurre le relazioni si rischia di aggiungere dei livelli di astrazione che consumano tempo di sviluppo e tempo di elaborazione.

Indipendenza Funzionale

- La suddivisione in sottosistemi può essere guidata da due fattori qualitativi fondamentali:
  - Coesione (cohesion)
  - Accoppiamento (coupling)

- L'obiettivo è **l'indipendenza funzionale** dei sottosistemi, che si massimizza con Alta Coesione e Basso Accoppiamento

Regole di buon Software Design

La Coesione

Coesione

- La Coesione è una misura di quanto siano fortemente relate e mirate le *responsabilità* (o servizi offerti) di un modulo.
- Se ciascuna unità è responsabile di un singolo compito, diciamo che tale unità ha una **alta coesione**.
- Un'alta coesione è una proprietà desiderabile del codice, poiché permette:
  - Di comprendere meglio i ruoli di una classe
  - Di riusare una classe
  - Di manutenere una classe
  - Di limitare e focalizzare i cambiamenti nel codice
  - Utilizzare nomi appropriati, efficaci, comunicativi

Granularità della Coesione

- Coesione dei metodi
  - Un metodo dovrebbe essere responsabile di un solo compito ben definito
- Coesione delle classi
  - Ogni classe dovrebbe rappresentare un singolo concetto ben definito

Coesione

- Indicatori di un’alta coesione (è il nostro obiettivo)
  - Una classe ha delle resposabilità moderate, limitate ad una singola area funzionale
  - Collabora con altre classi per completare dei tasks
  - Ha un numero limitato di metodi, cioè di funzionalità altamente legate tra loro
  - Tutti i metodi sembrano appartenere ad uno stesso insieme, con un obiettivo globale simile
  - La classe è facile da comprendere

Regole di buon Software Design

Il Coupling

Accoppiamento

- L’accoppiamento è una misura su quanto fortemente una classe è connessa con /ha conoscenza di/ si basa su altre classi.
- Due o più classi si dicono accoppiate quando è impossibile riusare una senza dover riusare anche la/le altra/e
- Se due classi dipendono strettamente e per molti dettagli l’una dall’altra, diciamo che sono fortemente accoppiate.
  - In caso contrario diciamo che sono debolmente accoppiate
- Per un codice di qualità dobbiamo puntare ad un basso accoppiamento

Basso Accoppiamento

- Un basso accoppiamento è una proprietà desiderabile del codice, poiché permette di:
  - Capire il codice di una classe senza leggere i dettagli delle altre
  - Modificare una classe senza che le modifiche comportino conseguenze sulle altre classi
  - Riusare facilmente una classe senza dover importare anche altre classi
- Un basso accoppiamento migliora la manutenibilità del software

Basso Accoppiamento

- Un certo livello di accoppiamento è comunque insito nel concetto di scambio di messaggio del paradigma O-O
  - E’ quindi necessario per il funzionamento del sistema
  - Deve essere limitato ove possibile
- Linea Guida: definire le responsabilità delle classi in modo da ottenere un basso accoppiamento
  - Legge di Demetra (vedi slides successive)

Tipi di Accoppiamento in O-O

- Date 2 classi X e Y:
  1. X ha un attributo di tipo Y
  2. X ha un metodo che possiede un elemento (es: parametro, variabile locale, etc…) di tipo Y
  3. X è una sottoclasse (eventualmente indiretta) di Y
  4. X implementa un’interfaccia di tipo Y

- Lo scenario 3 è quello che porta al massimo accoppiamento in assoluto.

Esempio

```
class Traveler
{
    Car c=new Car();
    void startJourney()
    {
       c.move();
    }
}

class Car
{
  void move()
  {
     // logic...
  }
}
```

- Cosa succede se vogliamo riusare la nostra classe Traveler con un altro mezzo di trasporto che non sia Car?
- Cosa succede se vogliamo cambiare i metodi di Car?
- Traveler ha un attributo di tipo Car, e quindi un forte accoppiamento.

Esempio (cont.)

```
class Traveler
{
    Vehicle v;
    public void setV(Vehicle v)
    {
      this.v = v;
    }

    void startJourney()
    {
       v.move();
    }
}

Interface Vehicle
{
   void move();
}
```

```
class Car implements Vehicle
{
    public void move()
    {
         // logic
    }
}
```

```
class Bike implements
Vehicle
{
    public void move()
    {
         // logic
    }
}
```

Spiegazione

- Nell'esempio, l'introduzione di un'interfaccia Vehicle ha spezzato completamente l'accoppiamento tra Traveler e Car.
- Conseguenze:
  - Traveler è ora accoppiato con un'interfaccia.
  - Tutte le implementazioni dell'interfaccia funzionano con Traveler, senza richiederne modifiche al codice
  - Sarà possibile usare in futuro implementazioni di Vehicle non ancora realizzate oggi (*anticipare il cambiamento*)
  - E' necessaria un'entità esterna che faccia **l'inject** della reale implementazione dell'interfaccia
  - E' aumentata la complessità del codice

Thinking about Interfaces

- Someone on the Java development team who understood interfaces decided to make them a big part of the Java SDK. There are hundreds of examples in the J2SE SDK of the correct way to use interfaces.
- Here are two key benefits derived from using interfaces:
  - An interface provides a means of setting a standard. It defines a contract that promotes reuse. If an object implements an interface then that object is promising to conform to a standard. An object that uses another object is called a consumer. An interface is a contract between an object and its consumer.
  - An interface also provides a level of abstraction that makes programs easier to understand. Interfaces allow developers to start talking about the general way that code behaves without having to get in to a lot of detailed specifics.

L'esempio del Paperboy

```
public class Customer {
 private String firstName;
 private String lastName;
 private Wallet myWallet;
 public String getFirstName(){
       return firstName;
 }
 public String getLastName(){
       return lastName;
 }
 public Wallet getWallet(){
       return myWallet;
 }
}
```

```
public class Wallet {
 private float value;
  public float getTotalMoney() {
        return value;
  }
  public void setTotalMoney(float
newValue){
        value = newValue;
  }
  public void addMoney(float
deposit) {
        value += deposit;
  }
  public void subtractMoney(float
debit) {
        value -= debit;
  }
}
```

L'esempio del Paperboy

```
code from some method inside the Paperboy
class...

payment = 2.00; // “I want my two dollars!”


Wallet theWallet = myCustomer.getWallet();

if (theWallet.getTotalMoney() > payment) {

        theWallet.subtractMoney(payment);

} else {

        // come back later and get my money

}
```

Why Is This Bad?

- Let's translate what the code is actually doing into real-language:
- Apparently, when the paperboy stops by and demands payment, the customer is just going to let the paperboy take the wallet out of his back pocket, and take out two bucks. I don't know about you, but I rarely let someone handle my wallet. There are a number of 'real-world' problems with this, not to mention we are trusting the paperboy to be honest and just take out what he's owed.
- If our future Wallet object holds credit cards, the paperboy has access to those too... but the basic problem is that "the paperboy is being exposed to more information than he needs to be".
- That's an important concept... The 'Paperboy' class now 'knows' that the customer has a wallet, and can manipulate it. When we compile the Paperboy class, it will need the Customer class and the Wallet class. These three classes are now 'tightly coupled'. If we change the Wallet class, we may have to make changes to both of the other classes.

Why Is This Bad?

- There is another classic problem that this can create... What happens if the Customer's wallet has been stolen? Perhaps last night a Thief picked the pocket of our example, and someone else on our software development team decided a good way to model this would be by setting the wallet to null, like this:
- victim.setWallet(null);
- This seems like a reasonable assumption... Neither the documentation nor the code enforces any mandatory value for wallet, and there is a certain 'elegance' about using null for this condition. But what happens to our Paperboy? Go look back at the code... The code assumes there will be a wallet! Our paperboy will get a runtime exception for calling a method on a null pointer.
- We could fix this by checking for 'null' on the wallet before we call any methods on it, but this starts to clutter the paperboy's code... our real-language description is becoming even worse...
- “If my customer has a wallet, then see how much he has... if he can pay me, take it”…

Improving The Original Code

```
public class Customer {
 private String firstName;
 private String lastName;
 private Wallet myWallet;
 public String getFirstName(){
        return firstName;
 }
 public String getLastName(){
        return lastName;
 }
```

```
  public float getPayment(float bill) {
   if (myWallet != null) {
    if (myWallet.getTotalMoney() > bill)
        {

theWallet.subtractMoney(payment);
              return payment;
     }
    }
  }
}
```

The Customer no longer has a 'getWallet()' method, but it does have a getPayment() method.

Improving the original code

```
// code from some method inside the Paperboy class...
    payment = 2.00;  // “I want my two dollars!”
    paidAmount = myCustomer.getPayment(payment);
    if (paidAmount == payment) {
        // say thank you and give customer a receipt
    } else {
        // come back later and get my money
    }
```

Why is this better?

- That's a fair question... Someone who really liked the first bit of code could argue that we have just made the Customer a more complex object. That's a fair observation, and I'll address that when I talk about drawbacks, below.
- The first reason that this is better is because it better models the real world scenario... The Paperboy code is now 'asking' the customer for a payment. The paperboy does not have direct access to the wallet.
- The second reason that this is better is because the Wallet class can now change, and the paperboy is completely isolated from that change.
  - If the interface to Wallet were to change, the Customer would have to be updated, but that's it...
  - As long as the interface to Customer stays the same, none of the client's of Customer will care that he got a new Wallet.
  - Code will be more maintainable, because changes will not 'ripple' through a large project.

Why is this better?

- The third, and probably most 'object-oriented' answer is that we are now free to change the implementation of 'getPayment()'. In the first example, we assumed that the Customer would have a wallet. This led to the null pointer exception we talked about. In the real world though, when the paper boy comes to the door, our Customer may actually get the two bucks from a jar of change, search between the cushions of his couch, or borrow it from his roommate.
- All of this is 'Business Logic', and is of no concern to the paper boy... All this could be implemented inside of the getPayment() method, and could change in the future, without any modification to the paper boy code.

What Are The Drawbacks?

- Is the Customer REALLY becoming more complex? In our first example, the paperboy code using the Customer also had to know about and manipulate the Wallet. This is no longer the case...
- If Customer has become MORE complex, then why are the clients now LESS complex? All the same complexity still exists, it is just better contained within the scope of each object, and exposed in a healthy way.
- By containing this complexity, we get all the benefits discussed above. So yes, we can concede that Customer has gotten more complex. The truth is, it's complexity that existed before in the code, and it may have existed in several places. **Now it occurs once**, and can be maintained in the same place that the changes that may break it occur.

Regole di buon Software Design

Altre regole

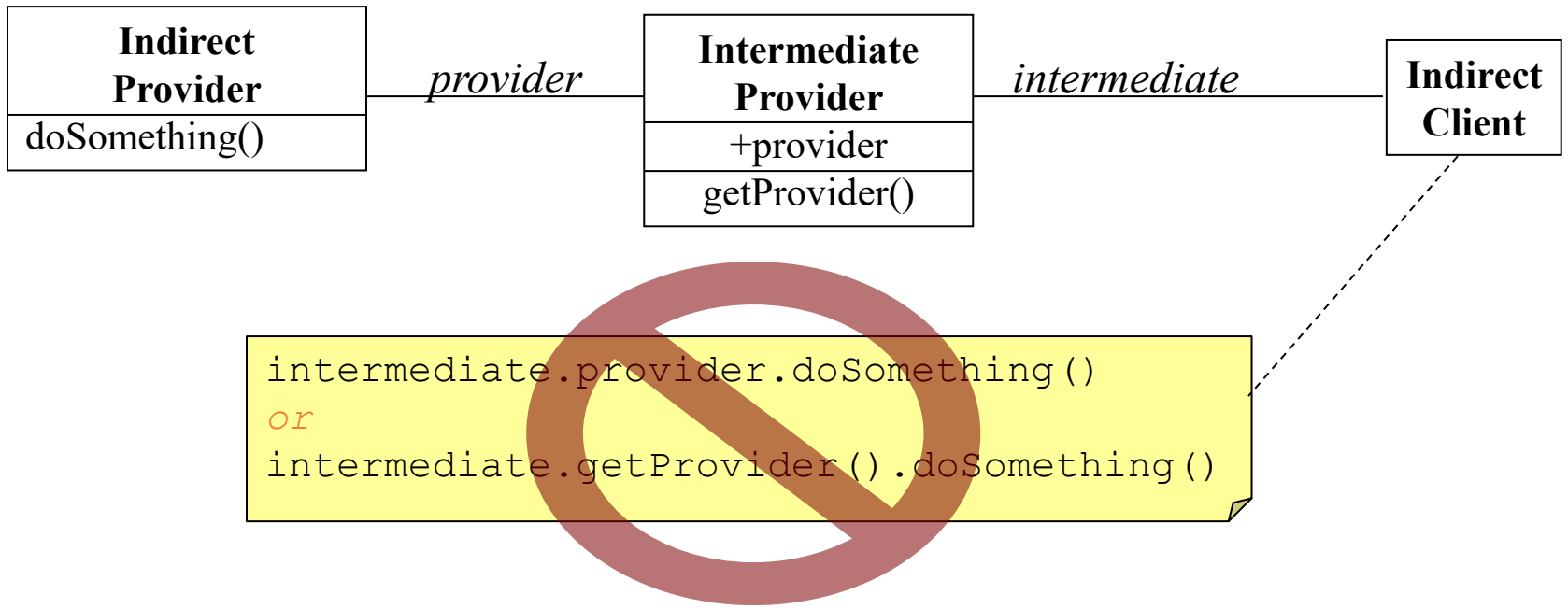
La Legge di Demetra

- E’ un’altra linea guida chiave per avere Basso Accoppiamento
- Utilizzata nel progetto Demetra, uno dei primi grossi sistemi O-O, sviluppato all’univ. di Boston.
- Concetto di base: Spingere al massimo l'information hiding
  - No situazioni tipo: Foo.getA().getB().getC()…..
  - Più è lungo il percorso di chiamate attraversato dal programma, più esso è fragile a cambiamenti

La Legge di Demetra

- Un metodo dovrebbe mandare messaggi solo a:
  1. L’oggetto contenente (this)
  2. Una variabile di istanza di this
  3. Un parametro del metodo
  4. Un oggetto creato all’interno del metodo

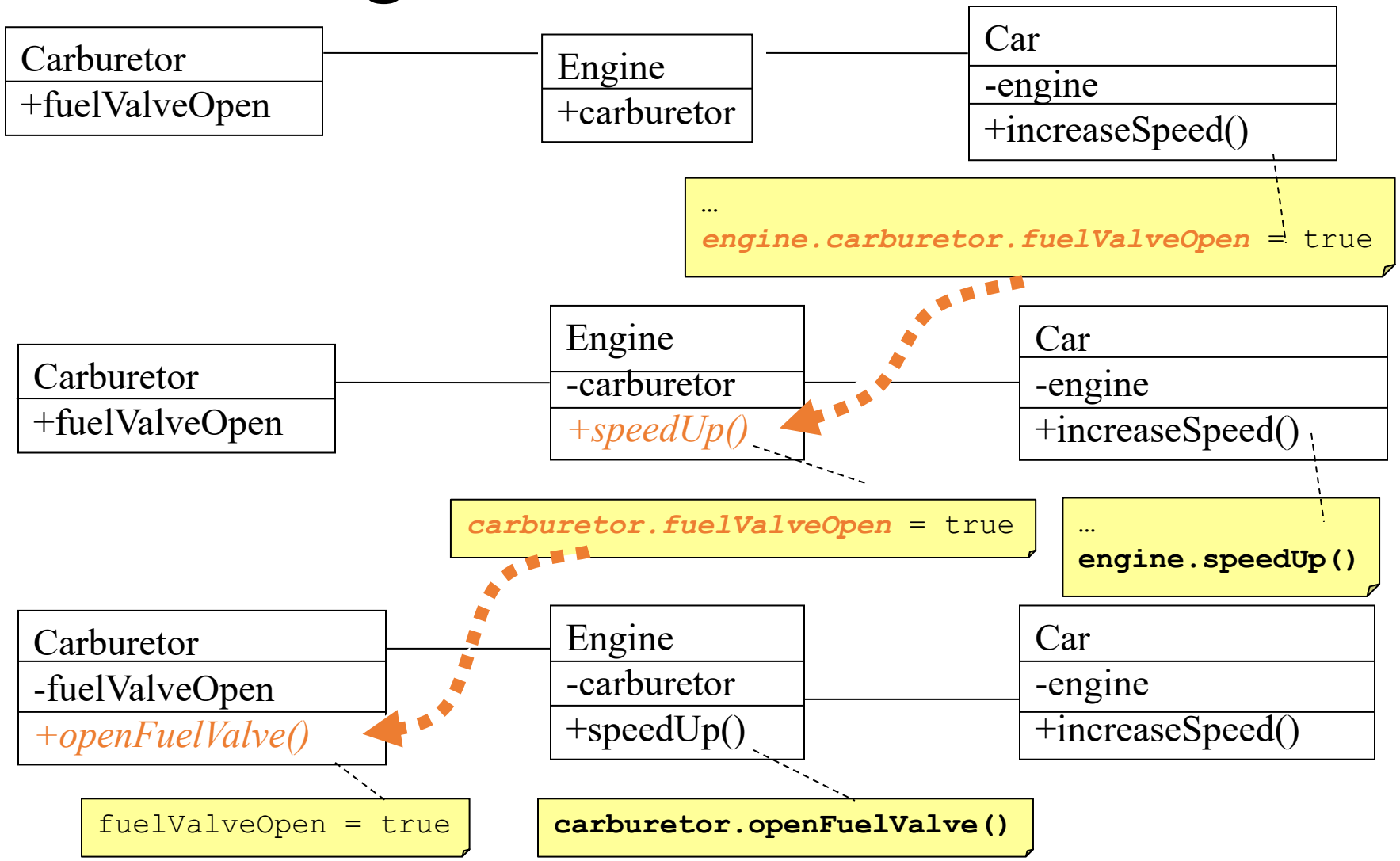
The Law of Demeter, violated



Law of Demeter: A method "M" of an object "O" should invoke only the methods of the following kinds of objects.

1. *itself*
2. *its parameters*
3. *any object it creates /instantiates*
4. *its direct component objects*

Eliminate Navigation Code



Responsibility-driven Design

Responsibility-driven design

- Pensare alla progettazione di un sistema in termini di:
  - Classe
  - Responsabilità
  - Collaborazioni
- Ciascuna classe dovrebbe avere delle responsabilità e dei ruoli ben precisi→ Alta Coesione
- La classe che possiede i dati dovrebbe essere responsabile di processarli → Basso Accoppiamento
  - Ciascuna classe dovrebbe essere responsabile di gestire i propri dati → Incapsulamento
- E' facilitata dall'uso di CRC Cards

Le CRC Cards

- Introdotte Kent Beck e Ward Cunningham per insegnare progettazione O-O
- Una card CRC è una scheda da associare ad ogni classe, per rappresentare i seguenti concetti:
  - Nome della Classe
  - Responsabilità della Classe
  - Collaboratori della Classe

- Responsabilità: conoscenza che la classe mantiene o servizi che la classe fornisce
- Collaboratore: un’altra classe di cui è necessario sfruttare la conoscenza o i servizi, per ottemperare alle rensponsabilità sopra indicate

Una CRC Card

- Associata ad ogni classe del class diagram di design

Class Name	
Superclasses	
Subclasses	
Responsibilities	Collaborators

Conseguenze

- Le CRC cards spingono ad ottenere un design con chiare responsabilità e controllo sul numero di classi associate
  - Se non riesco a specificare chiaramente le responsabilità di una classe, c'è un errore di design
  - Se una classe ha bisogno di troppe classi collaboratrici, c'è un potenziale errore di design

Localizzare le modifiche

- La progettazione RDD, con CRC Cards, porta ad una localizzazione delle modifiche
- Quando è necessario operare una modifica, il minor numero possibile di classi dovrebbero essere coinvolte

- La qualità del proprio codice si osserva proprio quando sorge l'esigenza di fare modifiche

Altre Best Practices di Design

Evitare Classi "God"

god class: a class that hoards much of the data or functionality of a system

- Poor cohesion – little thought about why all the elements are placed together
- Reduces coupling but only by collapsing multiple modules into one (which replaces dependences between modules with dependences within a module)

A god class is an example of an *anti-pattern*: a known bad way of doing things

Definizione di Metodi

- Design method signatures carefully
  - Avoid long parameter lists (max 7±2)
  - Perlis: “If you have a procedure with ten parameters, you probably missed some.”
  - Especially error-prone if parameters are all the same type
  - Avoid methods that take lots of Boolean “flag” parameters

- Use overloading judiciously
  - Can be useful, but avoid overloading with same number of parameters, and think about whether methods really are related

Definizione di Attributi

- A variable should be made into a field if and only if:
  - It is part of the inherent internal state of the object
  - It has a value that retains meaning throughout the object's life
  - Its state must persist past the end of any one public method

- All other variables can and should be local to the methods in which they are used
  - Fields should not be used to avoid parameter passing
  - Not every constructor parameter needs to be a field

Definizione di Costruttori

- Constructors should take all arguments necessary to initialize the object's state – no more, no less

- Object should be completely initialized after constructor is done
  - (i.e., the rep invariant should hold)

- Shouldn't need to call other methods to “finish” initialization

Good names

Adhere to generally accepted naming conventions

- Class names: generally nouns
  - Beware "verb + er" names, e.g. **`Manager`**, **`Scheduler`**, **`ShapeDisplayer`**
- Interface names often –able/-ible adjectives:

  `Iterable`, **`Comparable`**, …
- Method names: noun or verb phrases
  - Nouns for observers: **`size`**, **`totalSales`**
  - Verbs+noun for observers: **`getX`**, **`isX`**, **`hasX`**
  - Verbs for mutators: **`move`**, **`append`**
  - Verbs+noun for mutators: **`setX`**
  - Choose affirmative, positive names over negative ones

    `isSafe` not **`isUnsafe`**

    `isEmpty` not **`hasNoElements`**

Bad names

`count`, **`flag`**, **`status`**, **`compute`**, **`check`**, **`value`**, **`pointer`**, names starting with **`my`**…

- Convey no useful information

Describe what is being counted, what the "flag" indicates, etc.

`numberOfStudents`, **`isCourseFull`**, **`calculatePayroll`**, **`validateWebForm`**, …

But short names in local contexts are good:

Good: `for(i = 0; i < size; i++) items[i]=0;`

Bad:

```
for(theLoopCounter =  0;
        theLoopCounter <
theCollectionSize;
        theLoopCounter++)

   theCollectionItems[theLoopCounter]=0;
```

Class design ideals

Cohesion and coupling, already discussed

Completeness: Every class should present a complete interface

Consistency: In names, param/returns, ordering, and behavior

Completeness

Include *important* methods to make a class easy to use

Counterexamples:

- A mutable collection with **`add`** but no **`remove`**
- A tool object with a **`setHighlighted`** method to select it, but no **`setUnhighlighted`** method to deselect it
- **`Date`** class with no date-arithmetic operations

Also:

- Objects that have a natural ordering should implement **`Comparable`**
- Objects that might have duplicates should implement **`equals`** (and therefore **`hashCode`**)
- Most objects should implement **`toString`**

But…

Don't include everything you can possibly think of

- If you include it, you're stuck with it forever (even if almost nobody ever uses it)

Tricky balancing act: include what's useful, but don't make things overly complicated

- You can always add it later if you really need it

Consistency

A class or interface should have consistent names, parameters/returns, ordering, and behavior

Use similar naming; accept parameters in the same order

Counterexamples:

`setFirst(int index, String value)`
`setLast(String value, int index)`

`Date/GregorianCalendar` use 0-based months

`String` methods: `equalsIgnoreCase`, `compareToIgnoreCase`;
but `regionMatches(boolean ignoreCase)`

`String.length()`, `collection.size()`

Open-Closed Principle

Software entities should be *open for extension*, but *closed for modification*

- When features are added to your system, do so by adding new classes or reusing existing ones in new ways
- If possible, don't make changes by modifying existing ones – existing code works and changing it can introduce bugs and errors.

Related: Code to interfaces, not to classes

Example: accept a **`List`** parameter, not **`ArrayList`** or **`LinkedList`**

Refer to objects by their interfaces

Documenting a class

- Keep internal and external documentation separate

- External: /** ... */ Javadoc for classes, interfaces, methods
  - Describes things that clients need to know about the class
  - Should be specific enough to exclude unacceptable implementations, but general enough to allow for all correct implementations
  - Includes all pre/postconditons, etc.

- Internal: // comments inside method bodies
  - Describes details of how the code is implemented
  - Information that clients wouldn't and shouldn't need, but a fellow developer working on this class would want – invariants and internal pre/post conditions especially

Documentazione del codice

- If a program is incorrect, it matters little what the docs say
- If documentation does not agree with the code, it is not worth much
- Consequently, code must largely document itself. If not, rewrite the code rather than increasing the documentation of the existing complex code. Good code needs fewer comments than bad code.

- Comments should provide additional information from the code itself. They should not echo the code.
- Mnemonic variable names and labels, and a layout that emphasizes logical structure, help make a program “self-documenting”

Enums help document

Consider use of **`enums`**, even with only two values – which of the following is better?

```
oven.setTemp(97, true);
oven.setTemp(97, Temperature.CELSIUS);
```

Independence of views

- Confine user interaction to a core set of “view” classes and isolate these from the classes that maintain the key system data

- Do not put print statements in your core classes
  - This locks your code into a text representation
  - Makes it less useful if the client wants a GUI, a web app, etc.

- Instead, have your core classes return data that can be displayed by the view classes
  - Which of the following is better?

```
public void printMyself()
public String toString()
```

Last thoughts (for now)

- Always remember your reader
  - Who are they?
    - Clients of your code
    - Other programmers working with the code
      - (including yourself in 3 weeks/months/years)
  - What do they need to know?
    - How to use it (clients)
    - How it works, but more important, why it was done this way (implementers)
- Read/reread style and design advice regularly
- Keep practicing – mastery takes time and experience
- You'll always be learning. Keep looking for better ways to do things!