

Systems Analysis & Design Class and Method Design

CIS641

Erik Fredericks // frederer@gvsu.edu

*Adapted from materials provided by Gregory Schymik and the textbook
(Systems Analysis and Design 5th/6th Ed.)*



Las Vegas Startups and Small Businesses

Contact

Leadership: Building Teams, Growing

Outline!

Become familiar with coupling, cohesion, and connascence

Be able to specify, restructure, and optimize object designs

Be able to identify the reuse of predefined classes, libraries, frameworks, and components

Be able to specify constraints and contracts

Be able to create a method specification

More outline!

Review the characteristics of object orientation

Present useful criteria for evaluating a design

Present design activities for classes and methods

Present the concept of constraints & contracts to define object collaboration

Discuss how to specify methods to augment method design

Caution:

- Class/method design must precede coding
- While classes are specified in some detail, jumping into coding without first designing them may be disastrous

Class/Method design

What are some **critical factors (gotchas)** that appear at this stage in a project?

- Or any project that is adding features

What do you need to start writing code?

Is there anything missing?

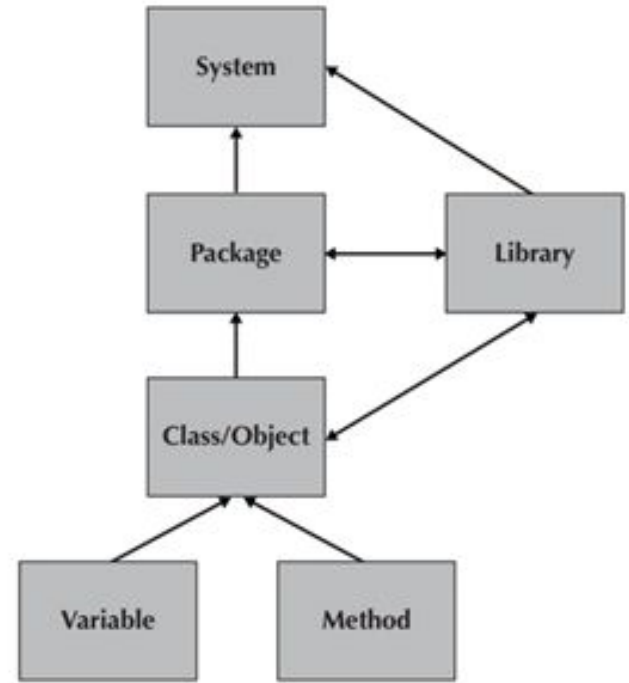


FIGURE 8-1
Levels of Abstraction
in Object-Oriented
Systems

Source: Based on material from David P. Tegarden, Steven L. Tegarden, "A Complexity Model of Object-Oriented Systems," *Decision Support Systems*, 1991, pp. 1-15.

Class/Method design

What classes should own what behavior plus variables?

- Does a CRUDE analysis help here?

WHAT ABOUT ALL THOSE NEAT OO THINGS WE LEARN ABOUT?

- Polymorphism
- Inheritance
- Reflection?!?!?
 - Let's leave this to the programming courses and focus on basic diagrams
 - We are going to talk about them though!

Characteristics of OOSAD

Classes

- Instantiated classes are objects
- Classes are defined with attributes, states and methods
- Classes communicate through messages

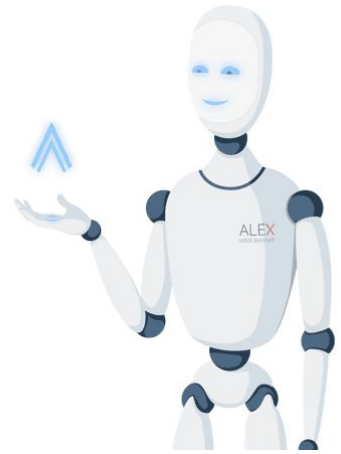
Encapsulation and information hiding

- Combine data and operations into a single object
- Reveal only how to make use of an object to other objects
- Key to reusability

Polymorphism and dynamic binding

Inheritance

OOSAD
is an acronym for
Object Oriented System
Analysis and Design
by allacronyms.com



Polymorphism and dynamic binding

Polymorphism

- The ability to take on several different forms
- Same message triggers different methods in different objects

Dynamic binding

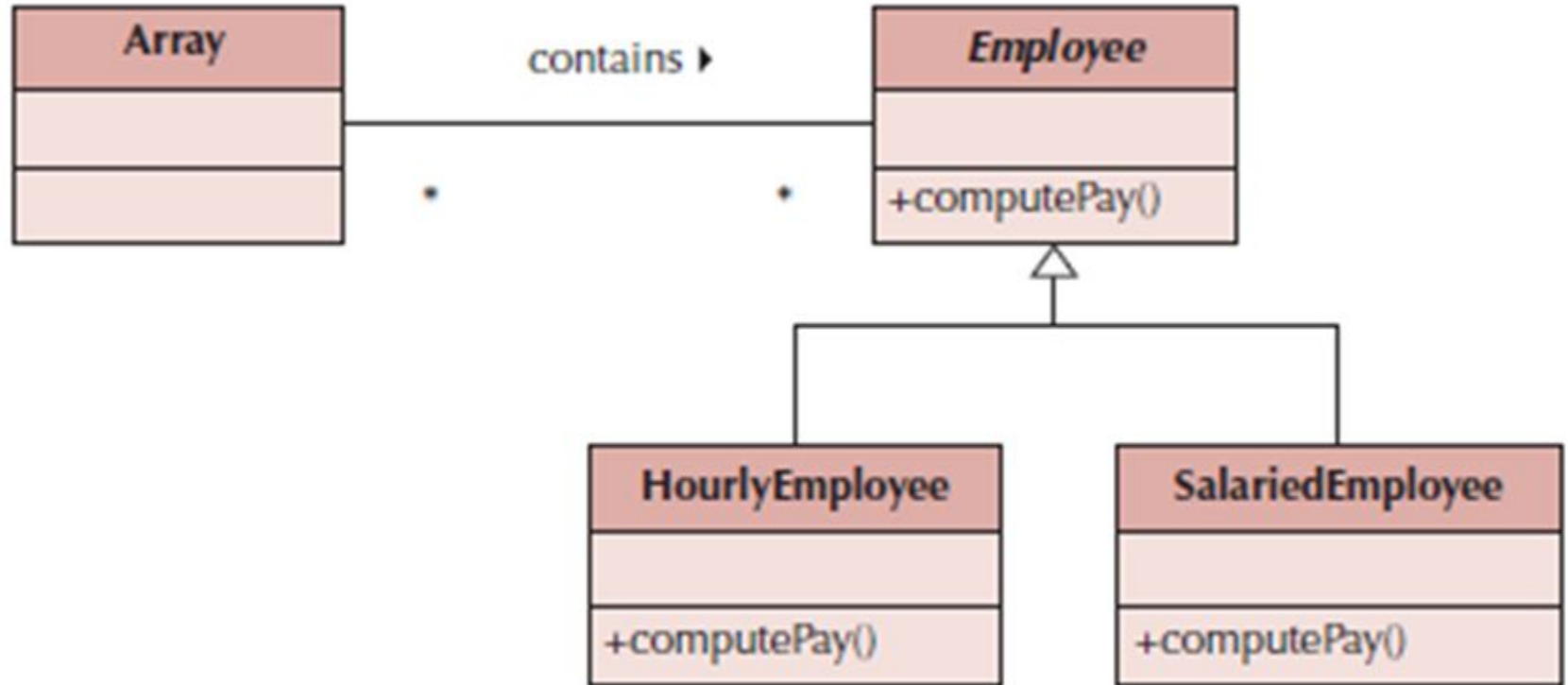
- Methods—the specific method used is selected at run time
- Attributes—data type is chosen at run time
- Implementation of dynamic binding is language specific

Decisions made at run time may induce run-time errors

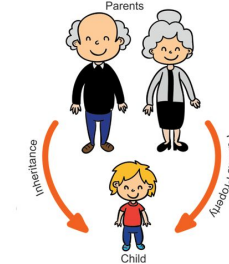
Need to ensure semantic consistency

<https://beginnersbook.com/2013/04/java-static-dynamic-binding/>

Polymorphism



Inheritance



Permits reuse of existing classes with extensions for new attributes or operations

Types

- Single inheritance -- one parent class
- Multiple inheritance -- multiple parent classes (not supported by all programming languages)
- Redefinition of methods and/or attributes
 - Not supported by all programming languages
 - May cause inheritance conflict

Designers must know what the chosen programming language supports

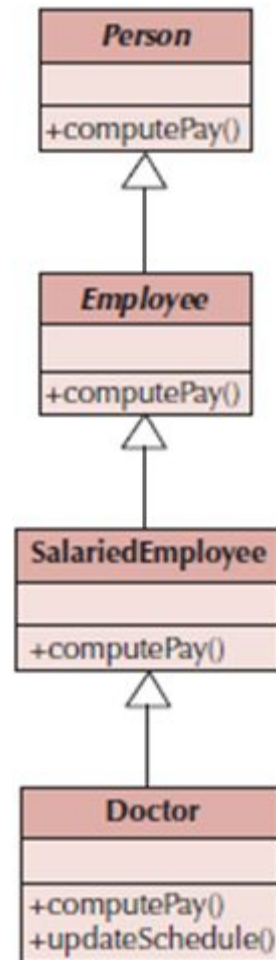
Inheritance conflicts

An attribute or method in a sub-class with the **same name** as an attribute or method in the super class

Cause is poor classification of sub-classes:

- Generalization semantics are violated, or
- Encapsulation and information hiding principle is violated

May also occur in cases of multiple inheritance



Design criteria

A set of metrics to evaluate the design

Coupling

- Degree of the closeness of the relationship between classes

Cohesion

- Degree to which attributes and methods of a class support a single object

Connascence

- Degree of interdependency between objects

Coupling

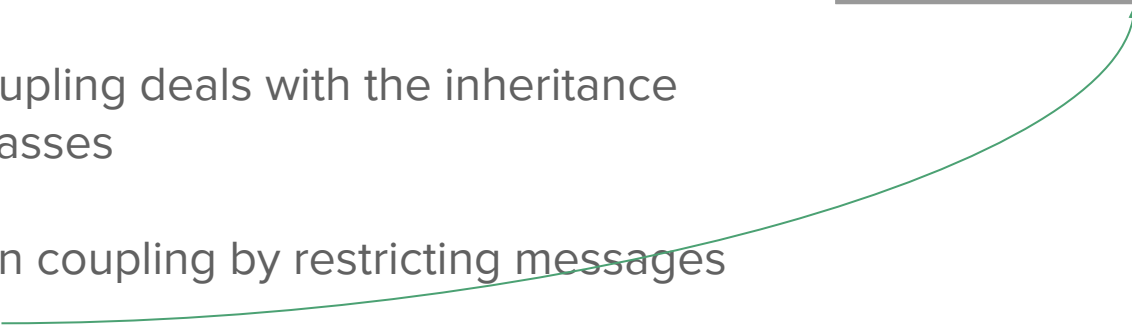
Close coupling means that changes in one part of the design may require changes in another part

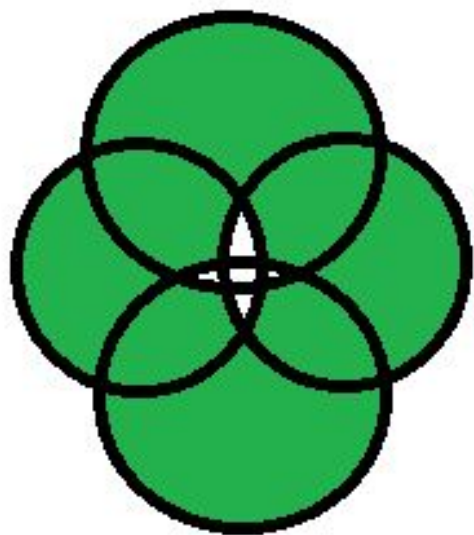
Types

- Interaction coupling measured through message passing
- Inheritance coupling deals with the inheritance hierarchy of classes

Minimize interaction coupling by restricting messages
(Law of Demeter)

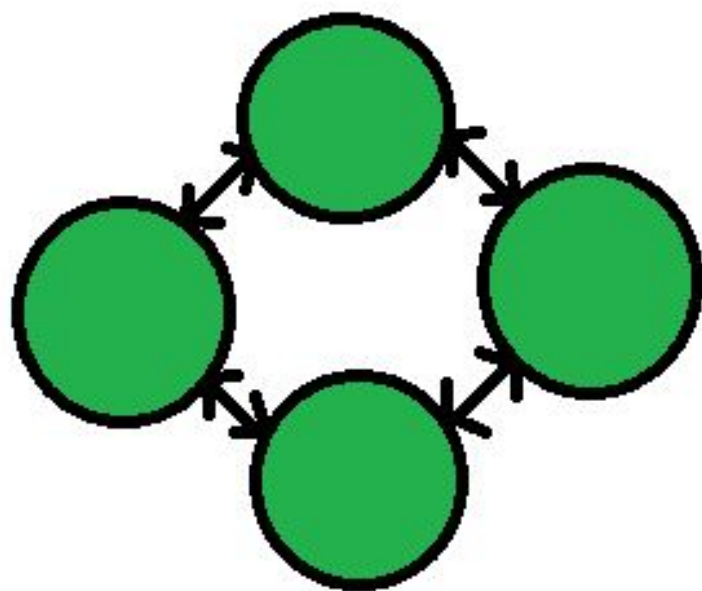
Minimize inheritance coupling by using inheritance to support only generalization/specialization and the principle of substitutability

- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
 - Each unit should only talk to its friends; don't talk to strangers.
 - Only talk to your immediate friends.
- 



Tight coupling:

1. More Interdependency
2. More coordination
3. More information flow



Loose coupling:

1. Less Interdependency
2. Less coordination
3. Less information flow

Law of Demeter

Messages should be sent only by an object:


to itself

to objects contained in attributes of itself or a superclass

to an object that is passed as a parameter to the method

to an object that is created by the method

to an object that is stored in a global variable

Level	Type	Description
Good	No Direct Coupling	The methods do not relate to one another; that is, they do not call one another.
	Data	The calling method passes a variable to the called method. If the variable is composite (i.e., an object), the entire object is used by the called method to perform its function.
	Stamp	The calling method passes a composite variable (i.e., an object) to the called method, but the called method only uses a portion of the object to perform its function.
	Control	The calling method passes a control variable whose value will control the execution of the called method.
	Common or Global	The methods refer to a "global data area" that is outside the individual objects.
Bad	Content or Pathological	A method of one object refers to the inside (hidden parts) of another object. This violates the principles of encapsulation and information hiding. However, C++ allows this to take place through the use of "friends."
<p>Source: These types were adapted from Meilir Page-Jones, <i>The Practical Guide to Structured Systems Design</i>, 2nd ed. (Englewood Cliffs, NJ: Yardon Press, 1988); and Glenford Myers, <i>Composite/Structured Design</i> (New York: Van Nostrand Reinhold, 1978).</p>		

Cohesion

A cohesive class, object or method refers to a single thing

Types


- Method cohesion
 - Does a method perform more than one operation?
 - Performing more than one operation is more difficult to understand and implement
- Class cohesion
 - Do the attributes and methods represent a single object?
 - Classes should not mix class roles, domains or objects
- Generalization/specialization cohesion
 - Classes in a hierarchy should show “a-kind-of” relationship, not associations or aggregations

Method

Level	Type	Description
Good	Functional	A method performs a single problem-related task (e.g., calculate current GPA).
	Sequential	The method combines two functions in which the output from the first one is used as the input to the second one (e.g., format and validate current GPA).
	Communicational	The method combines two functions that use the same attributes to execute (e.g., calculate current and cumulative GPA).
	Procedural	The method supports multiple weakly related functions. For example, the method could calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.
	Temporal or Classical	The method supports multiple related functions in time (e.g., initialize all attributes).
	Logical	The method supports multiple related functions, but the choice of the specific function is chosen based on a control variable that is passed into the method. For example, the called method could open a checking account, open a savings account, or calculate a loan, depending on the message that is send by its calling method.
Bad	Coincidental	The purpose of the method cannot be defined or it performs multiple functions that are unrelated to one another. For example, the method could update customer records, calculate loan payments, print exception reports, and analyze competitor pricing structure.

Source: These types were adapted from Page-Jones, *The Practical Guide to Structured Systems*, and Myers, *Composite/Structured Design*.

Class

Level	Type	Description
Good	Ideal	The class has none of the mixed cohesions.
	Mixed-Role	The class has one or more attributes that relate objects of the class to other objects on the same layer (e.g., the problem domain layer), but the attribute(s) have nothing to do with the underlying semantics of the class.
	Mixed-Domain	The class has one or more attributes that relate objects of the class to other objects on a different layer. As such, they have nothing to do with the underlying semantics of the thing that the class represents. In these cases, the offending attribute(s) belongs in another class located on one of the other layers. For example, a port attribute located in a problem domain class should be in a system architecture class that is related to the problem domain class.
Worse	Mixed-Instance	The class represents two different types of objects. The class should be decomposed into two separate classes. Typically, different instances only use a portion of the full definition of the class.
Source: Page-Jones, <i>Fundamentals of Object-Oriented Design in UML</i> .		

Connascence

Classes are so interdependent that a **change in one necessitates a change in the other**

Good programming practice should:

- Minimize overall connascence; however, when combined with encapsulation boundaries, you should:
 - Minimize across encapsulation boundaries (less interdependence between or among classes)
 - Maximize within encapsulation boundary (greater interdependence within a class)
- A sub-class should never directly access any hidden attribute or method of a super class

```
function print(){  
  console.log(`Movie titles are ${titles()}`);  
}
```

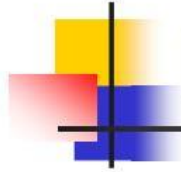
Connascence of Name

```
function titles(){  
  var sql = "SELECT * FROM Movies";  
  const result = db.query(sql);  
  // ...  
}
```

Connascence of Name

Connascence of Name

```
CREATE TABLE Movies  
-- ...
```



More Examples

- Connascence of name: subclass inherits a variable, it must use the same name.
- Connascence of convention: in C++, a non-zero int is considered TRUE. Also, enumeration types.
- Connascence of algorithm: Binary search expects the members of a Tree to be sorted ascending.

Type	Description
Name	If a method refers to an attribute, it is tied to the name of the attribute. If the attribute's name changes, the content of the method will have to change.
Type or Class	If a class has an attribute of type A, it is tied to the type of the attribute. If the type of the attribute changes, the attribute declaration will have to change.
Convention	A class has an attribute in which a range of values has a semantic meaning (e.g., account numbers whose values range from 1000 to 1999 are assets). If the range would change, then every method that used the attribute would have to be modified.
Algorithm	Two different methods of a class are dependent on the same algorithm to execute correctly (e.g., insert an element into an array and find an element in the same array). If the underlying algorithm would change, then the insert and find methods would also have to change.
Position	The order of the code in a method or the order of the arguments to a method is critical for the method to execute correctly. If either is wrong, then the method will, at least, not function correctly.
Source: Meilir Page-Jones, "Comparing Techniques by Means of Encapsulation and Connascence" and Meilir Page-Jones, <i>Fundamentals of Object-Oriented Design in UML</i> .	

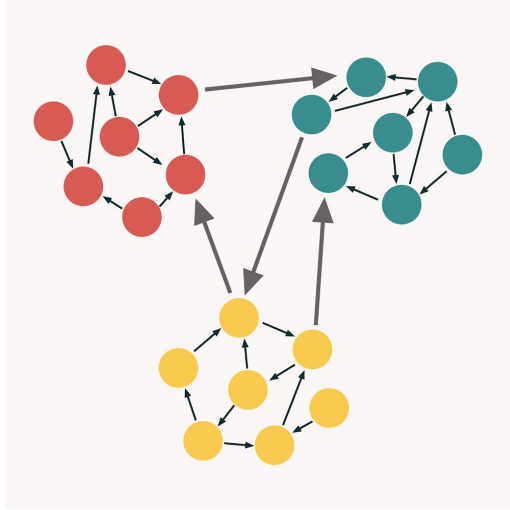
Cohesion vs. coupling

""""

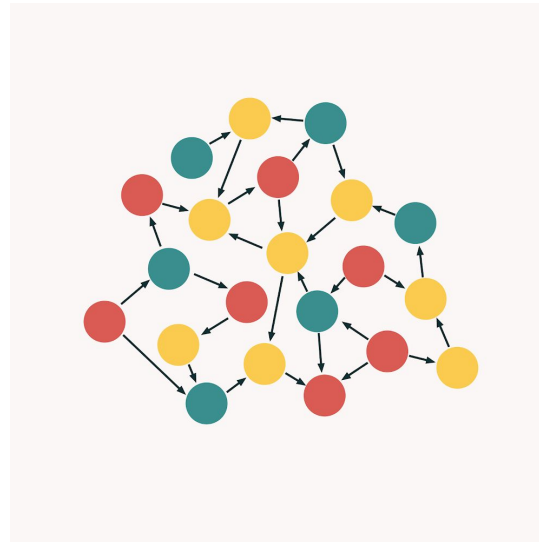
- Cohesion represents the degree to which a part of a code base forms a logically single, atomic unit.
- Coupling represents the degree to which a single unit is independent from others.

""""

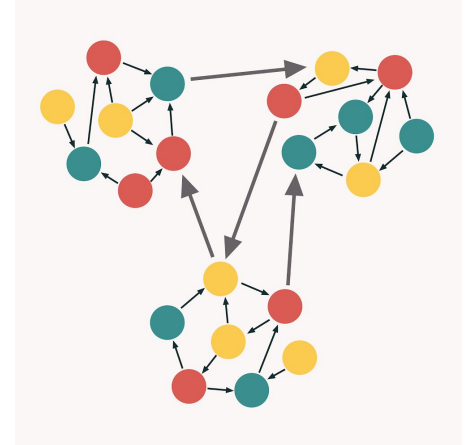
Examples



High cohesion, low coupling



"God object" → high cohesion, high coupling

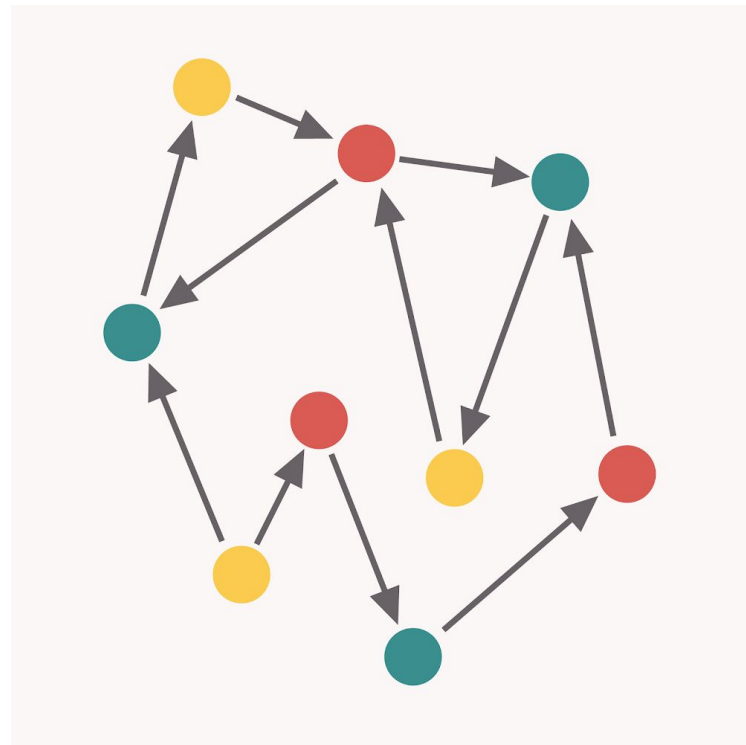


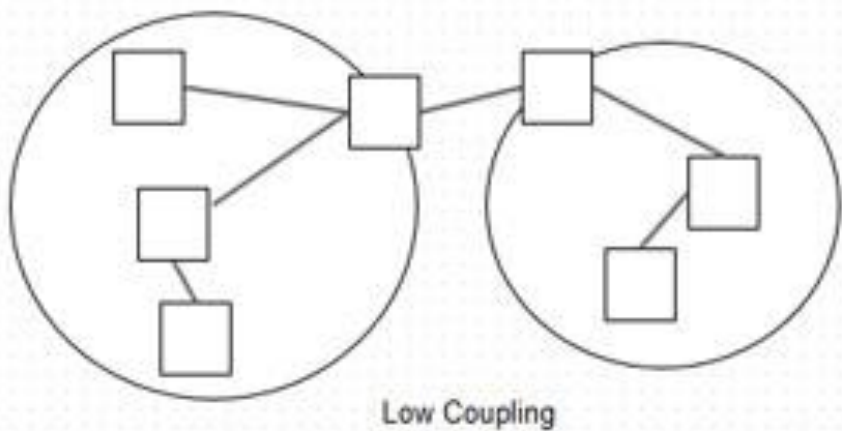
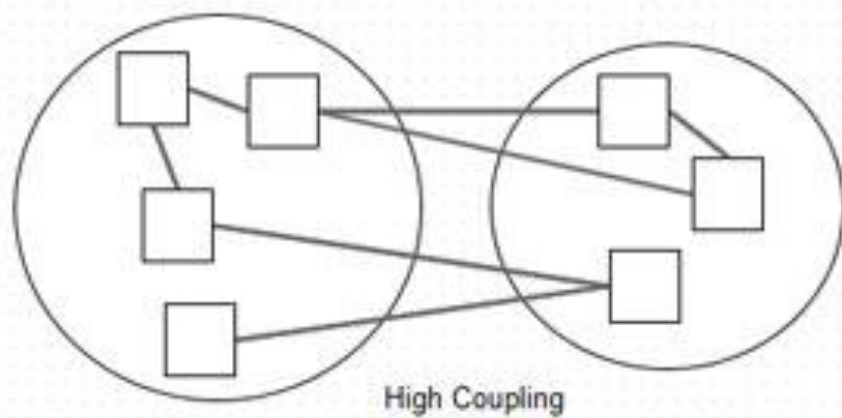
Poor boundaries

Example

Destructive decoupling

- Decouple codebase so much that it loses focus





Object design activities

An extension of analysis and evolution activities

Expand the descriptions of **partitions**, **layers** & **classes** by:

- Adding specifications to the current model
- Identifying opportunities to reuse classes that already exist
- Restructuring the design
- Optimize the design
- Map the problem domain classes into a programming language

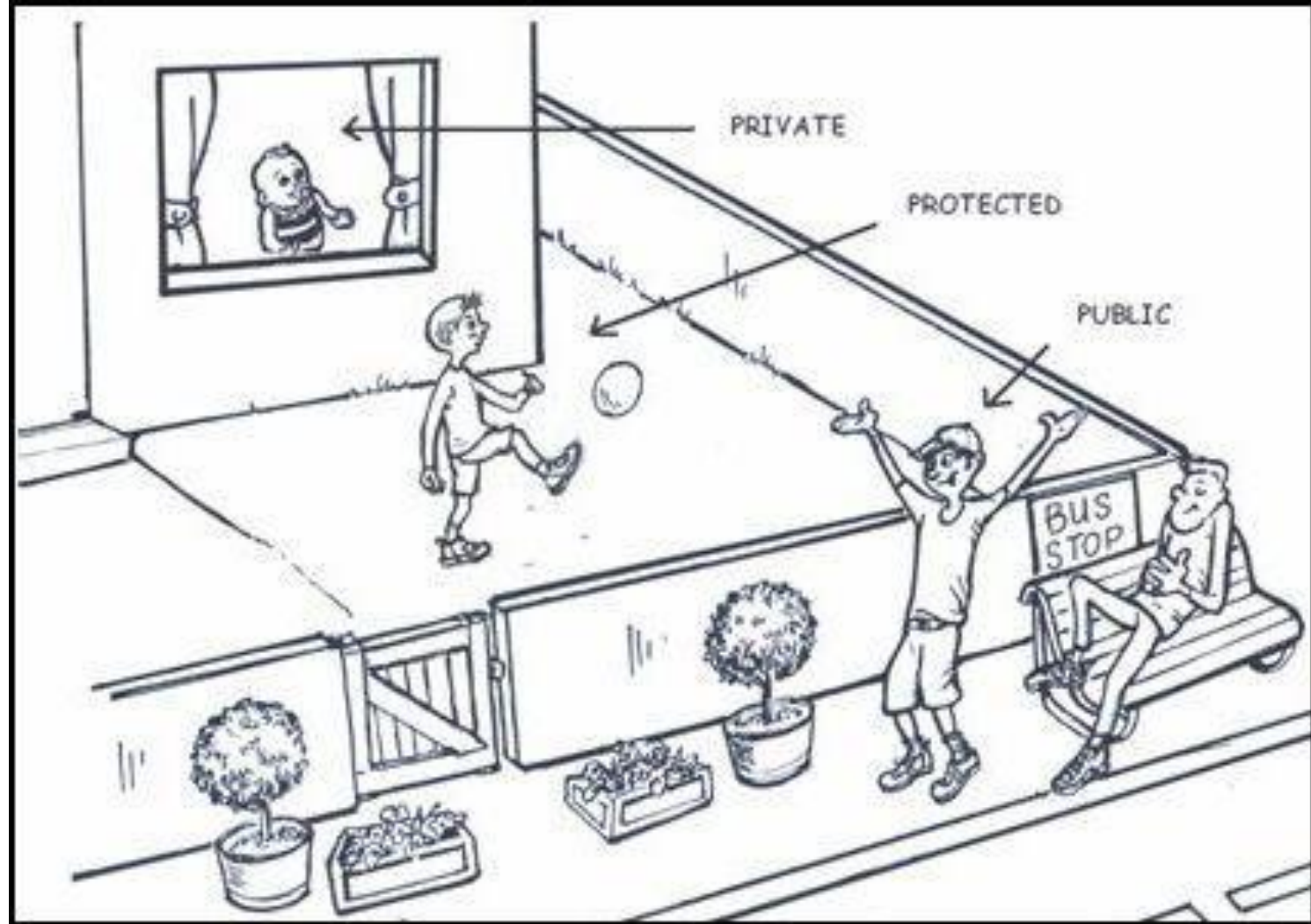
Adding specifications

Review the current set of analysis models

- All classes included are both sufficient and necessary to solve the problem
- No missing attributes or methods
- No extra or unused attributes or methods
- No missing or extra classes

Examine the visibility of classes

- **Private** → **not visible**
- **Public** → **visible to other classes**
- **Protected** → **visible only to members of the same super class**



Adding specifications

Decide on **method signatures**:

- Name of the method
- Parameters or arguments to pass
- Type of value(s) to be returned

Define **constraints** that must be **preserved** by the objects

- Preconditions, post-conditions, and invariants
- Decide how to handle constraint violations

Identify opportunities for reuse

Design patterns

- Groupings of classes that help solve a commonly occurring problem

Framework

- Set of implemented classes that form the basis of an application

Class libraries

- Set of implemented classes, but more general in nature than a framework

Components

- Self-contained classes used as plug-ins to provide specific functionality

Choice of approaches depends on the layer

Restructuring (the design)

Factoring

- Separating aspects from a class to simplify the design

Normalization

- Aids in identifying missing classes

Assure all inheritance relationships support only generalization/specialization semantics

Optimizing (the design)

Balance understandability with efficiency

Methods:

- Review access paths between objects
- Review all attributes of each class
- Review direct (number of messages sent by a method) and indirect fan-out (number of messages by methods that are induced by other methods)
- Consider execution order of statements in often-used methods
- Avoid re-computation by creating derived attributes and triggers
- Consider combining classes that form a one-to-one association

Mapping problem-domain classes

Factor out multiple inheritance if using a language that supports only single inheritance

Factor out all inheritance if the language does not support inheritance

Avoid implementing an object-oriented design in non-object languages

Constraints and contracts

A contract is a set of constraints and guarantees

- If the requestor (client) meets the constraints, the responder (server) will guarantee certain behavior
 - Constraints must therefore be unambiguous
- Contracts document message passing between objects
- A contract is created for each visible method in a class
- Should contain enough information for the programmer to understand what the method is supposed to do

Constraint types

- Precondition → must be true before the method executes
- Post-condition → must be true after the method finishes
- Invariant → must always be true for all instances of a class

<https://riptutorial.com/c/example/1810/precondition-and-postcondition>

https://www.cs.cornell.edu/courses/cs3110/2019sp/textbook/basics/pre_post_conditions.html

Basically:

- Use OCL (check the book for the formal syntax) or
- Consider them to be effectively if-statements

Use whichever format is necessary for your environment!

Method Name:	Class Name:	ID:
Clients (Consumers):		
Associated Use Cases:		
Description of Responsibilities:		
Arguments Received:		
Type of Value Returned:		
Pre-Conditions:		
Post-Conditions:		

Method specification

Documentation details for each method

- Allows programmers to code each method

Must be explicit and clear

No formal standards exist, but information should include:

- General information (e.g., method name, class name, etc.)
- Events—anything that triggers a method (e.g., mouse click)
- Message passing including values passed into a method and those returned from the method
- Algorithm specifications
- Other applicable information (e.g., calculations, procedure calls)

Method Name: insertOrder	Class Name: OrderList	ID: 100
Contract ID: 123	Programmer: J. Doe	Date Due: 1/1/12
Programming Language: <input type="checkbox"/> Visual Basic <input type="checkbox"/> Smalltalk <input type="checkbox"/> C++ <input type="checkbox"/> Java		
Triggers/Events: Customer places an order		
Arguments Received: Data Type:	Notes:	
Order	The new customer's new order.	
Messages Sent & Arguments Passed: ClassName.MethodName:	Data Type:	Notes:
OrderNode.new()	Order	
OrderNode.getOrder()		
Order.getOrderNumber()		
OrderNode.setNextNode()	OrderNode	
self.middleListInsert()	OrderNode	
Arguments Returned: Data Type:	Notes:	
void		
Algorithm Specification: See Figures 8-30 and 8-31.		
Misc. Notes: None.		

GROUPWORK!

(You thought you were getting out of it this time
MUAHAHHAAHAHAHAHAHAHAHAHAHAHAHAHAHAHAHA)

(ahahaha)

(ha)

ಠ_ಠ

GROUPWORK

Select **one** of your classes that you've created

- If you didn't, come up with a class

Create:

- 1) A method signature
- 2) At least one precondition
- 3) At least one postcondition
- 4) At least one invariant

Adding specifications

Decide on **method signatures**:

- Name of the method
- Parameters or arguments to pass
- Type of value(s) to be returned

Define **constraints** that must be **preserved** by the objects

- Preconditions, post-conditions, and invariants
- Decide how to handle constraint violations

Singleton example

Facade example

Observer example

Let's check out MVC too for fun! (and profit)

Let's check out some 350 slides!

(I added them to BB if you want the full deal)