Topic 1

Introduction and OO Design Principles (SOLID)

Learning Resources

- **Main References:**

  - Design-Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995), Addison Wesley.
  - Applying UML and Patterns, Craig Larman
  - The Unified Modelling Language: User's Guide

    Booch, G, Jacobson, I, Rumbaugh, J. Addison-Wesley.
  - Design Patterns Java Workbook, Steven John Metsker.
  - Succeeding with Agile, by Mike Cohn
  - “Refactoring “ by Martin Fowler

- **Other references as provided in the slides**

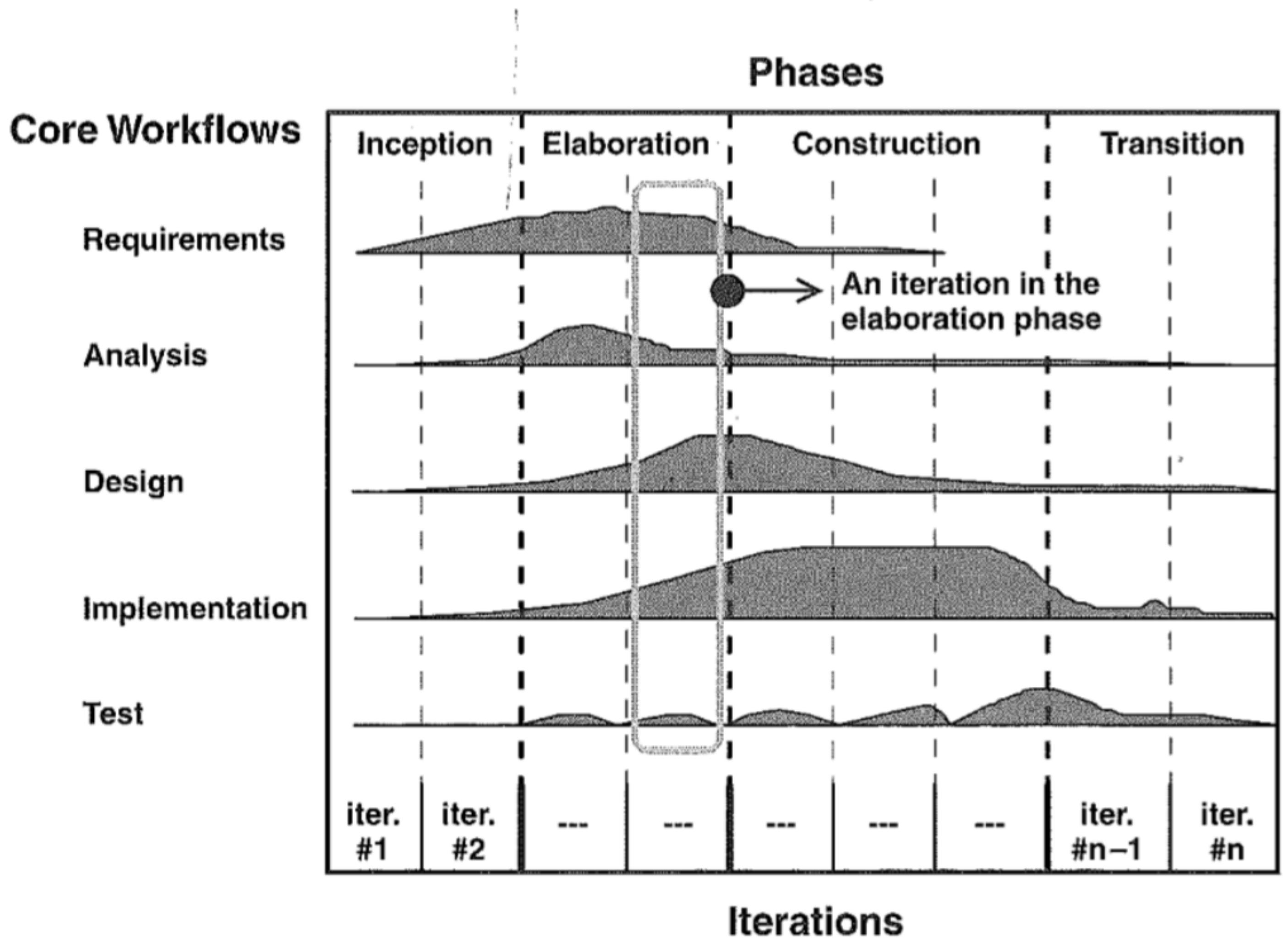
Course Outline:

- Create UML diagrams (a notation for describing systems)
- Apply OO Principles (with a focus on **Design**)
- Apply GRASP and **SOLID** principles/patterns
- Apply the concept of Design by Contract
- Apply the concept of Refactoring
- Apply the classic design patterns as described by Gamma et al.

The Course will not:

- Go into depth concerning project management
- Go into depth with team management
- Go into depth into user interface design
- Go into depth concerning the formal testing or deployment of software
- Go into depth with specific frameworks (since understanding core principles should enable the application of ANY OO framework)

Design as Part of The Software Development Process



From: Jacobsen, Booch, Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 2003.

INCEPTION

- Define scope of the system
  - Requirements
  - Quality
- Outline **architecture**
- Identify critical risks and how to address them
- Make business case
- *Many potential projects may not make it past the inception phase!*

ELABORATION

- Further clarify requirements (getting it right early is best!)
- Expand candidate architecture
- Begin Prototyping
- Continue analyzing/addressing risks
- Finalizing project planning for construction phase

CONSTRUCTION

- This is the major stage where the actual software is developed
- Iterative and incremental process
- Aim towards a functional system (alpha/beta) for the following transition phase

TRANSITION

- The system is rolled out/deployed to customers
- Correct any reported bugs
- Depending upon the complexity of the system transition may be relatively straightforward since the product is already correct at the end of construction
- For entirely new or more complex systems the transition phase may result in significant further development (e.g. Myki!)

Agile Development (from https://agilemanifesto.org/principles.html)

1. Satisfy the customer through early and continuous delivery
2. Welcome changing requirements, even late in development
3. Deliver working software frequently (weeks/months)
4. Business people and developers work together daily
5. Build projects around motivated individuals (support/trust/empower)
6. Favour face-to-face conversation
7. Working software is the primary measure of progress
8. Agile processes promote sustainable development (maintain pace)
9. ***Continuous attention to technical excellence and good design enhances agility (!)***
10. Simplicity - maximize the amount of work **not done**
11. The best architectures, requirements, and designs emerge from self-organizing teams
12. Ongoing team-based reflection and improvement

Agile Development Summary

- Agile development emphasises an *iterative*, *incremental* and *evolutionary* approach
- Favours loosely organised motivated teams
- Favours writing and evolving the smallest subset of functionality that can reasonably work and be tested
- It **DOES NOT** suggest developing the quickest and dirtiest implementation possible as is sometimes misinterpreted! See for example https://www.javaworld.com/article/2073877/joshua-bloch--a-conversation-about-design.html
- *The design techniques promoting software quality covered in this course can and should be applied to an Agile Development Project (as per **point 9** on previous slide)*

Elements of Software Design

- PROCESS
  - The activities leading to the orderly construction of the system’s models
  - Ranging from highly procedural (e.g. Waterfall) to highly iterative/incremental e.g. Agile
- NOTATION
  - The language/rules for expressing each model
  - Compared to a programming language design notation is not intended to produce an executable artefact
- TOOLS
  - The artifacts that eliminate the tedium of model building and enforce rules about the models themselves, so that errors and inconsistencies can be exposed

Goals of Software Design

The main challenge facing software designers is the tendency for requirements to change — customers often do not know what is required until a preliminary version of software is produced.

Goals of Software Design include:

- Functional Requirements – Correctness
- Non-functional requirements/Quality (see next slide)

Non Functional Requirements

- Reliability
- Maintainability
- Understandability
- Adaptability
- Reusability
- Efficiency
- Portability
- Fault Tolerance
- Cost-Effectiveness

Highlighted indicates attributes most readily affected by Design/Code Quality

Stages of Design

- Problem understanding
  - View problem from different angles to discover the design requirements (especially customer perspective).
- Identify one or more solutions
  - Evaluate possible solutions and choose the most appropriate.
- Describe solution abstractions
  - Use graphical, formal or other descriptive notations to describe the components of the design.
- Repeat process for each identified abstraction
  - until the design is expressed in primitive terms.
- Iterative and incremental Process

Design Direction

- **Top-Down Design**
  - **In principle**, top-down design involves starting at the most abstract level (e.g. interfaces, abstract classes and from the top of a class hierarchy).
  - **In practice**, large systems design is rarely top-down. Some branches are designed before others. Designers reuse experience (and sometimes components) during the design process.
- **Bottom Up Design** is useful for problems that are not well understood. Working on small detailed subsystems can elaborate or identify better high-level designs.
- **Meet In The Middle** (use aspects of Bottom up and Top Down)

Core OO Principles (Revision)

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy
5. Coupling
6. Cohesion

Encapsulation/Information Hiding

- Process that binds together data and the methods.
- It separates implementation from the interface.
- It prevents data from being arbitrarily manipulated

```
Class Date
{
      private int day;
      private int month;

      public void setDay(int num)
      { … }
      public void advance(int days)
      { … }

}
```

Abstraction

Abstraction is the process of taking away or removing characteristics from something to reduce it to some set of core characteristics.

Which of these are more abstract?

- Person/Student
- Vehicle/Car

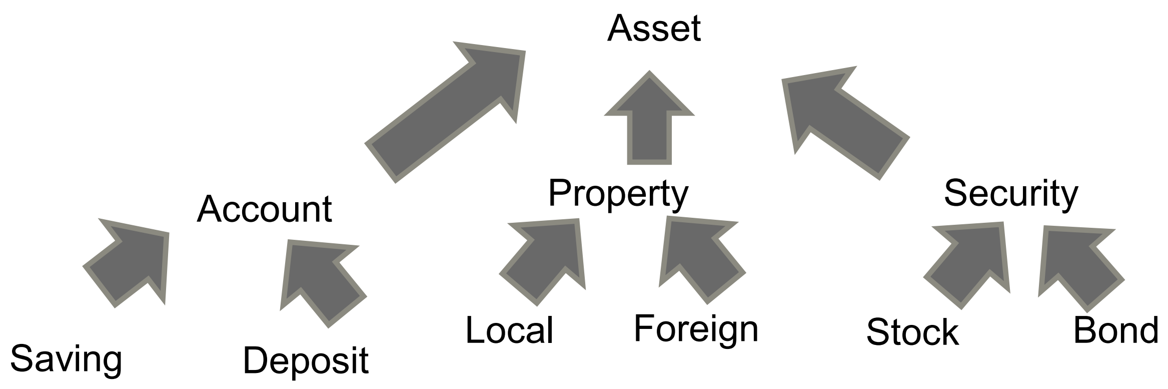
Give one operation that is only applicable to the less abstract class?

Why abstract?

- Allows us to manage complexity by concentrating on the essential aspects
- Allows abstractions to be organized in a hierarchy.

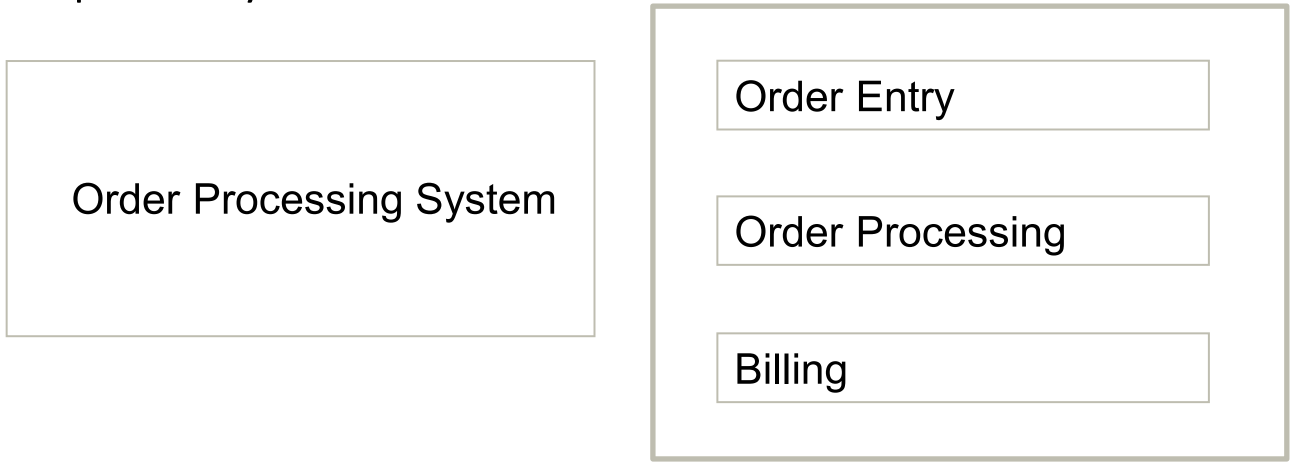
Hierarchy

Allows abstractions to be organized in tree like structure.



Modularity

Allows complex subsystems to be subdivided so that they can be independently created.



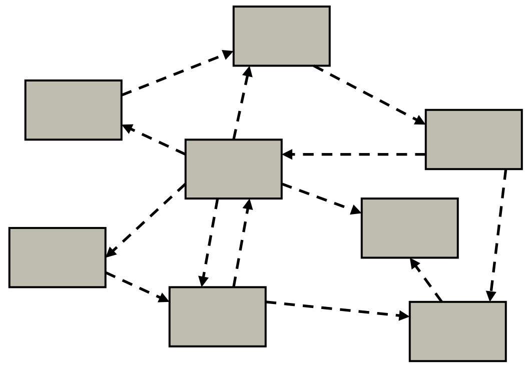
Packages and Classes, and more recently **named modules** are the main modules in a standard Java application.
Different platforms may have different modules, e.g. EJB in Java EE, Activities in Android etc.
Modules may be in the same process or distributed across processes or even physical machines (enterprise components, web services etc.)
Classes are the primary implementation module in OO systems

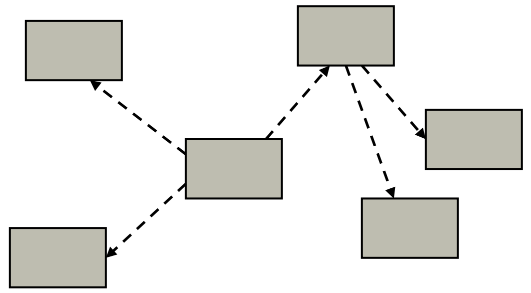
Coupling

The dependence of some classes on others in this context is called **coupling**. This reliance should be minimised in your class designs wherever possible.

Excessive coupling between classes **can create problems** when items are modified – with so many dependencies, changing one will imply that many others may also need to be changed.

Coupling can be objectively measured with automated code analysis tools.


excessively high coupling


low coupling

Cohesion

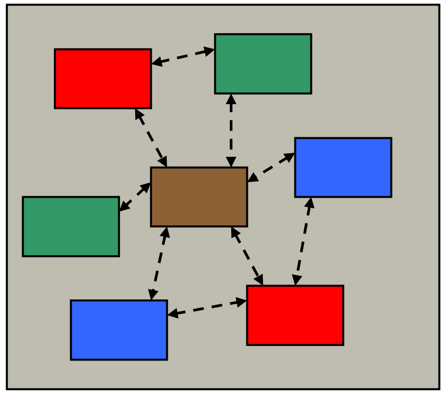
When a class is designed, each of its elements should have a **logical and well-defined role** in the class being able to fulfil its purpose. This **tight relation** of internal components is known as **cohesion**.

A **highly cohesive object** will **not** have items that are **only loosely related** to its role – the object will define **one** concept or perform one task well.

Cohesion is a **good** thing – it reflects a design that is **very clear and pure** to its intended purpose.

Cohesion is harder to measure automatically since it requires semantic understanding.

Coupling and Cohesion is a tradeoff – increasing cohesion (good) can increase coupling (bad).



A highly cohesive object will have elements that are closely inter-related.

Cohesion and Coupling Trade-off

- Cohesion and coupling are a trade-off
  - improving one may have a negative impact on the other
- e.g. Consider a program with only one class
  - it has low coupling (good) since there are no other classes to couple to
  - however cohesion will be low (bad) because the class will be responsible for everything
  - e.g. user interface, database management, unrelated domain logic (accounts, customers etc.)
  - => lots of unrelated methods = not cohesive
- TIP: Write cohesive classes first then try to minimise coupling

Cohesion and Coupling Trade-off (continued)

- Conversely imagine a system with a large number of very small and highly cohesive classes that only have one method
  - One method implies only a single functionality (although the method itself may not be cohesive) so cohesion is high (good).
  - However this system will be highly coupled (bad) because all the small classes have to interact to get work done

SOLID Principles (Software Design Principles)

From *Robert C. Martin, Agile software development : principles, patterns, and practices, Pearson*, *2002*

The Single-responsibility principle

- The Open-closed principle
- The Liskov substitution principle
- The Interface segregation principle
- The Dependency Inversion Principle

The Don't Repeat Yourself Principle (bonus principle!)

The Single-Responsibility Principle (Cohesion!)

- A class should have only one reason to change. -- Robert Martin
- A responsibility = a reason to change
- Separate coupled responsibilities into separate modules

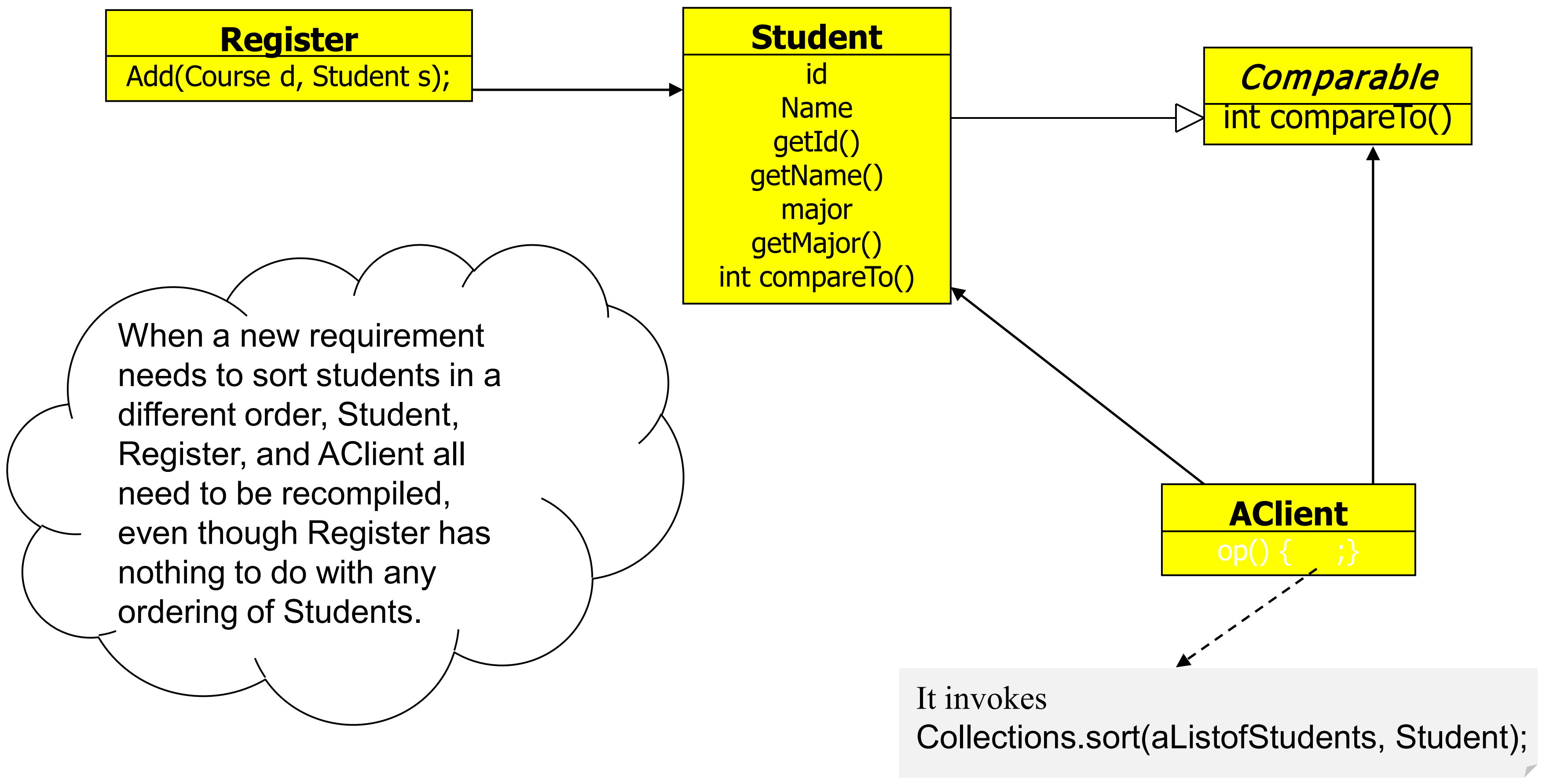
The Single-Responsibility Principle

- Example:
  - Often we need to sort students by their name, or id. So one may make Class Student implement the Java Comparable interface.

```
class Student implements Comparable<Student> {
      …
      int compareTo(Student s) { … }
      …
};
```

  - Student is a business entity; it does not know in what order it should be sorted since the order of sorting is imposed by the client of Student.
  - Worse: every time students need to be ordered differently, we have to recompile Student and all its clients.
  - Cause of the problems: we bundled two separate responsibilities (i.e., student as a business entity and ordering) into one class – a violation of SRP

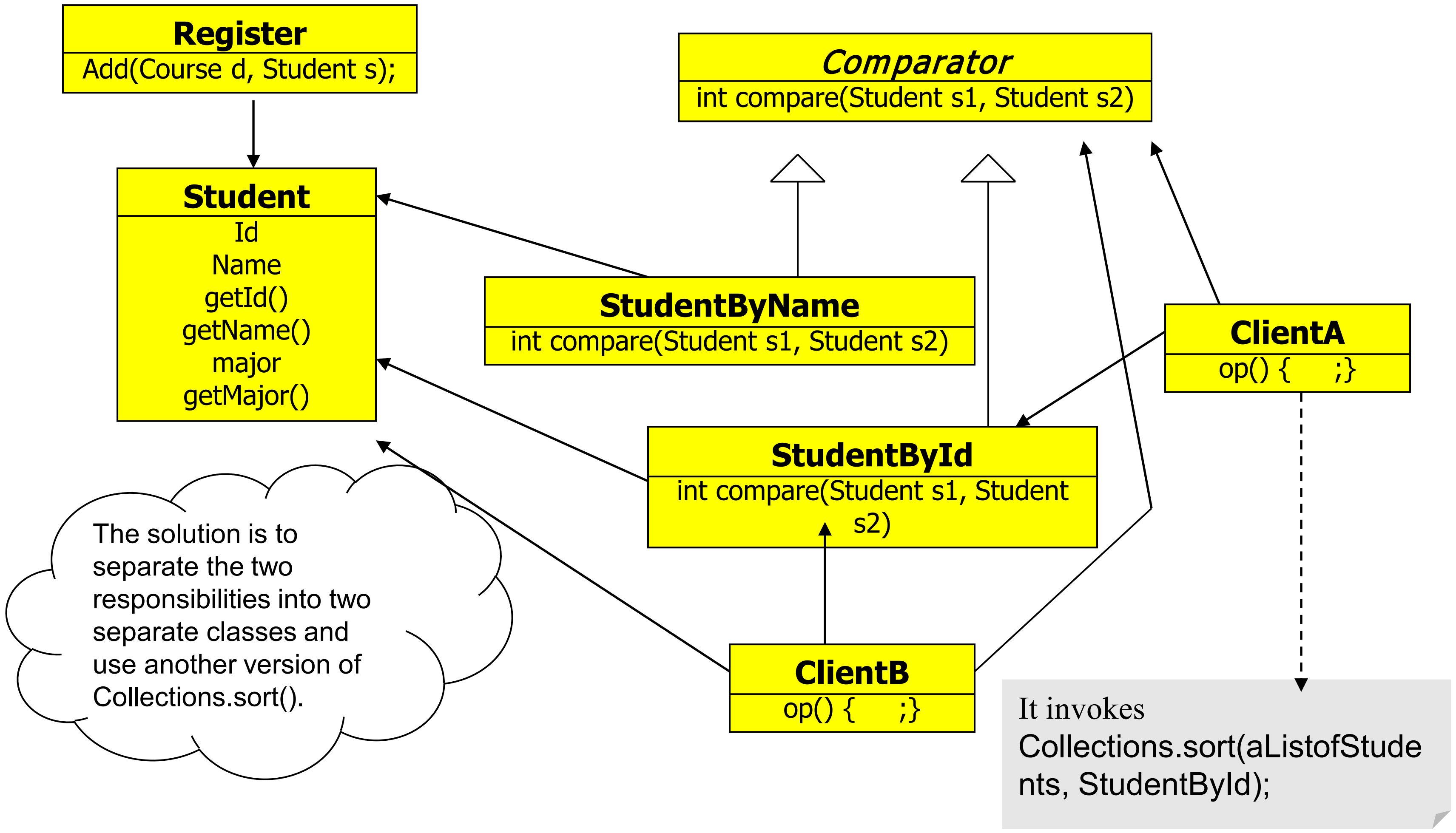
The Single-Responsibility Principle



When a new requirement needs to sort students in a different order, Student, Register, and AClient all need to be recompiled, even though Register has nothing to do with any ordering of Students.

It invokes
Collections.sort(aListofStudents, Student);

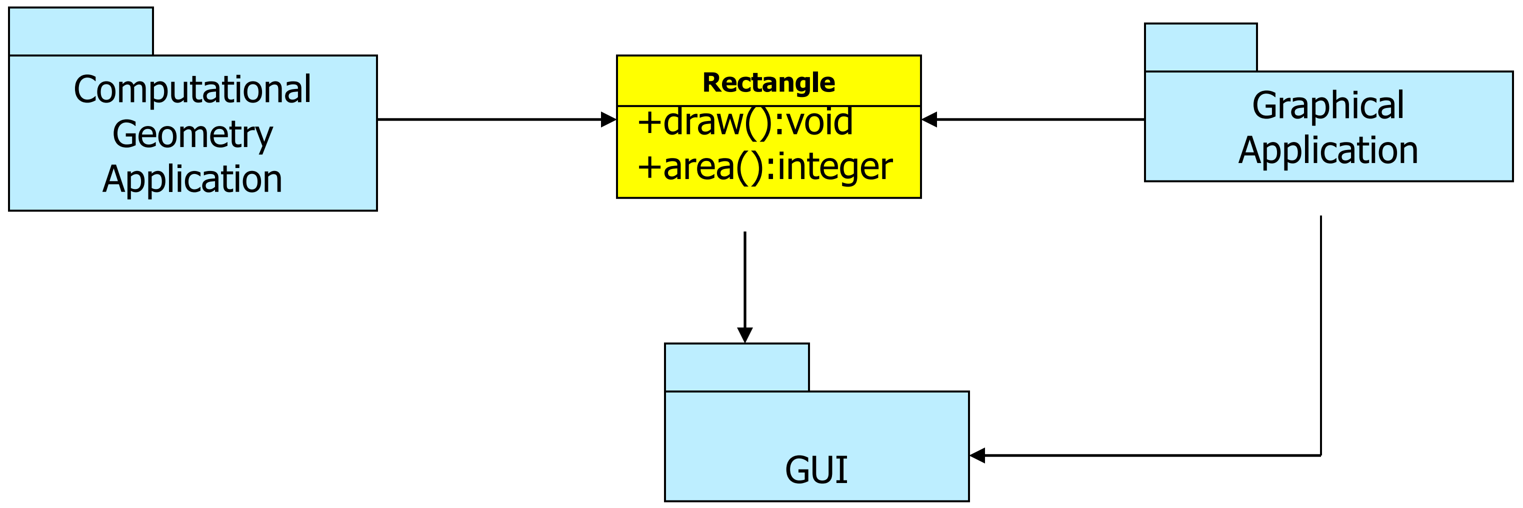
The Single-Responsibility Principle


Register
Add(Course d, Student s);
Student
Id
Name
getId()
getName()
major
getMajor()
Comparator
int compare(Student s1, Student s2)
StudentByName
int compare(Student s1, Student s2)
StudentById
int compare(Student s1, Student s2)
ClientA
op() {   ;}
ClientB
op() {   ;}

The solution is to separate the two responsibilities into two separate classes and use another version of Collections.sort().

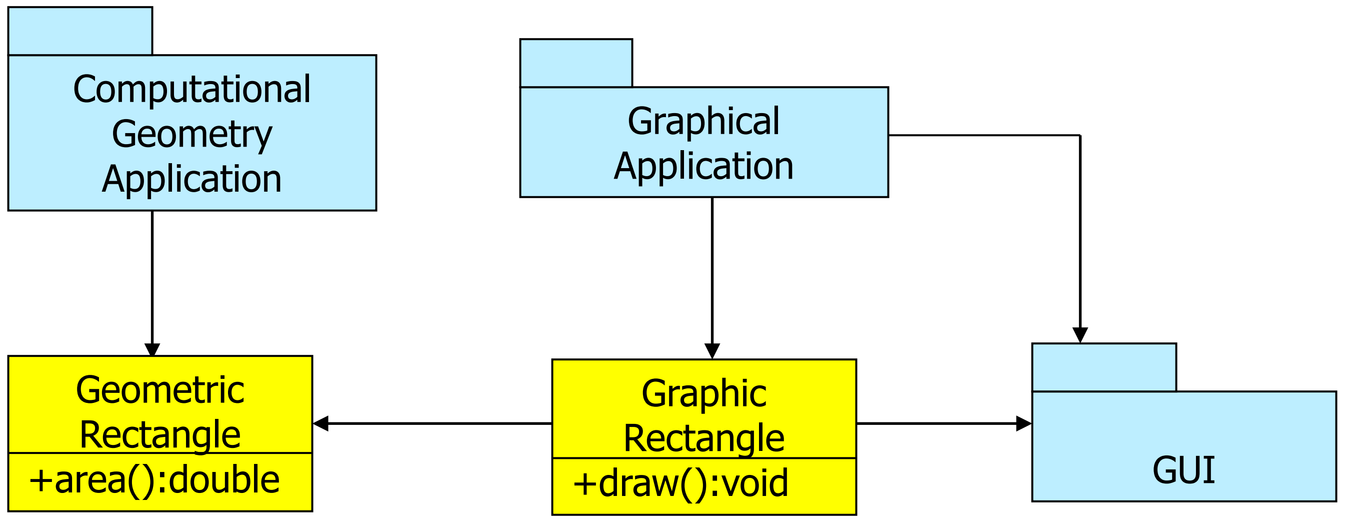
It invokes Collections.sort(aListofStudents, StudentById);

The Single-Responsibility Principle



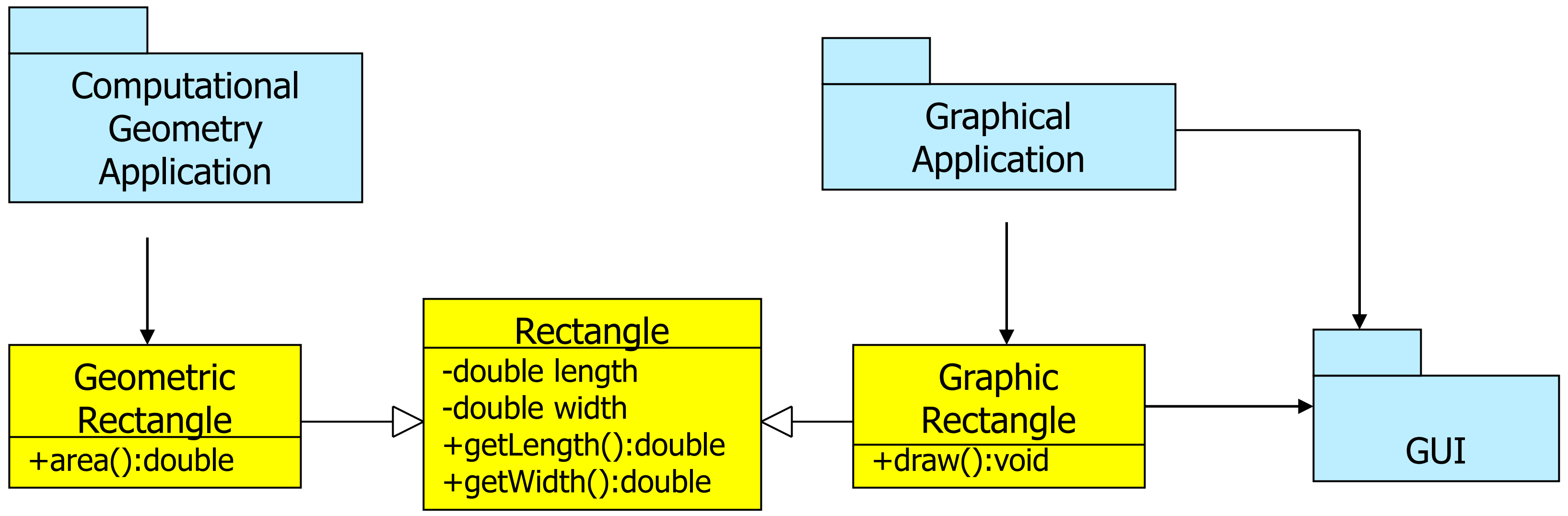
Class Rectangle may be forced to make changes from two different unrelated sources. One is from the Computational Geometry Application (CGA). E.g., adding area function for length and width of type double. The other is from Graphical Application (GA). E.g., add draw() in Microsoft Windows to the existing Unix draw method. A change from either of the two source would cause the other application to recompile.

The Single-Responsibility Principle



- Package CGA is no longer dependent on graphical side of Rectangle and thus it becomes independent of package GUI. Any change caused by graphical application no longer requires CGA to be recompiled.
- However, any changes from the CGA side may cause GA to be recompiled.

The Single-Responsibility Principle



Class Rectangle contains the most primitive attributes and operations of rectangles. Classes GeometricRectangle and GraphicRectangle are independent of each other. A change from either side of CGA or GA, it would not cause the other side to be recompiled.

The Open-Closed Principle

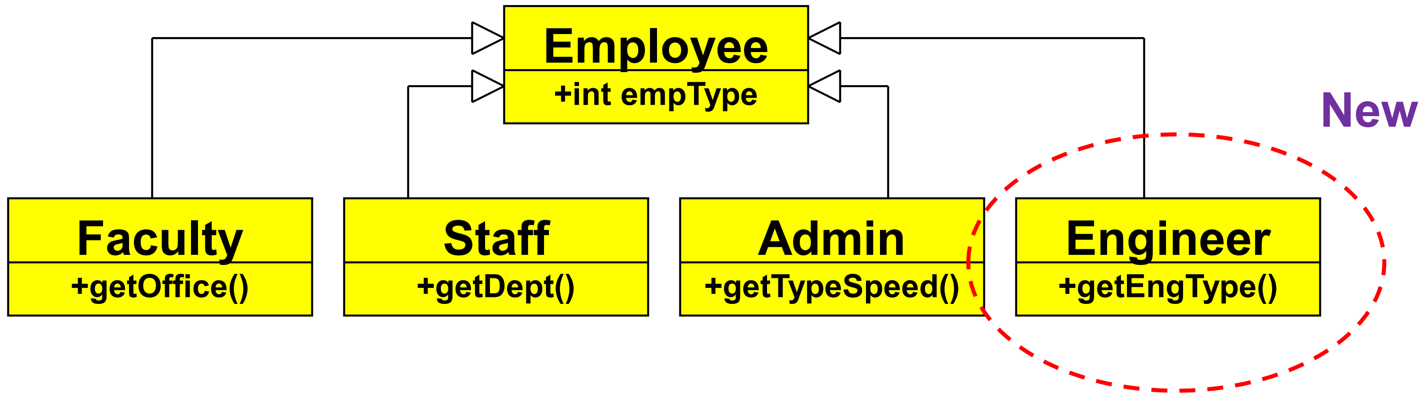
- “Software entities (classes, modules, functions, etc,) should be open for extension, but closed for modification*” (– R. Martin) i.e., allow behaviour to be extended without modifying its source
- * that effects client code
- OCP aims to allow **change** to code without requiring you to modify client code (other code that uses your modified code)
- OCP aims to allow **extension** without modifying the code being extended
- To make a class open for extension, closed for modification:
  - program to interfaces (or abstract classes), not implementation (concrete classes).
  - provide strong information hiding and encapsulation (private, final etc.) so that clients are isolated from implementation details

The Open-Closed Principle (Java)

- When designing for inheritance in Java it is important to clearly **document** any *self-use* in overridable methods (i.e. non-final, public and protected)
  - This is important since otherwise a subclass may override a method and change expected behavior by not adhering to the required self-use, i.e. not calling other methods that the superclass expected to be called as part of its implementation
- *Judiciously* expose 'hooks' for extension by providing protected methods which are called at expected points of future difference
  - Too many and you unnecessarily commit to implementation decisions
  - Too few and the class lacks utility and cannot be meaningfully extended
  - In rare cases you may also provide protected attributes although this is riskier and may lead to the subclass accidentally breaking superclass functionality
- **Superclass constructors should not invoke overridable methods since the method will run before the subclass constructor and may not have been properly initialized**

- EXAMPLE: Look at Javadoc for `java.util.AbstractCollection.remove()`

Example: NonAdherence to OCP – : Programming to Concrete Classes

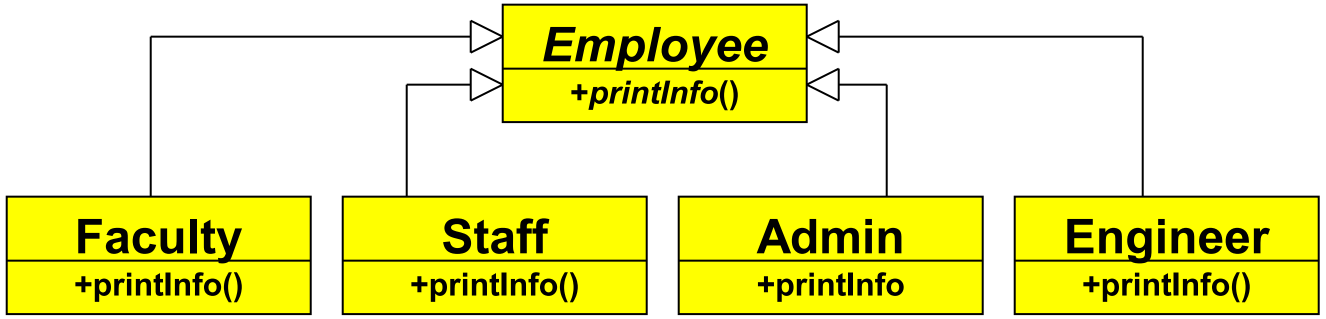


```
void printEmpRoster(Employee[] emps) {
     for (int i; i<emps.size(); i++) {
          if (emps[i].empType == FACULTY)
               printfFaculty((Faculty)emps[i]);
          else if (emps[i].empType ==STAFF)
               printStaff((Staff)emps[i]);
          else if (emps[i].empType == ADMIN)
               printSecretary((Secretary)emps[i]);
     }
}
```

What if we need to add `Engineer`?

Derived classes implement different print methods.
Changes are required for any new class.

Adherence to OCP – programming to Abstract Classes



```
void printEmpRoster(Employee[] emps)
{
      for (int i; i<emps.size(); i++)
      {
            emps[i].printInfo();
      }
}
```

When `Engineer` is added, `printEmpRoster()` does not even need to recompile.

`PrintEmpRoster()` is open to extension, closed for modification.

Liskov Substitution Principle (LSP)

- LSP states that if subclass S is a subclass of class C then anywhere C is valid, the subtype S can be substituted and still work correctly (passing as a parameter etc.)
- This facilitates the use of generic code that does not know the specifics of the subclass S
- However LSP can be difficult to achieve in practice:
  - Consider the case where class SQUARE extends and specialises RECTANGLE
  - What are the implications of LSP here?
    - e.g. extendOneSide(RECTANGLE) method
    - consider pre/post conditions and invariants
    - related to invariant/covariant arguments in superclass/subclass (generics can help here!)

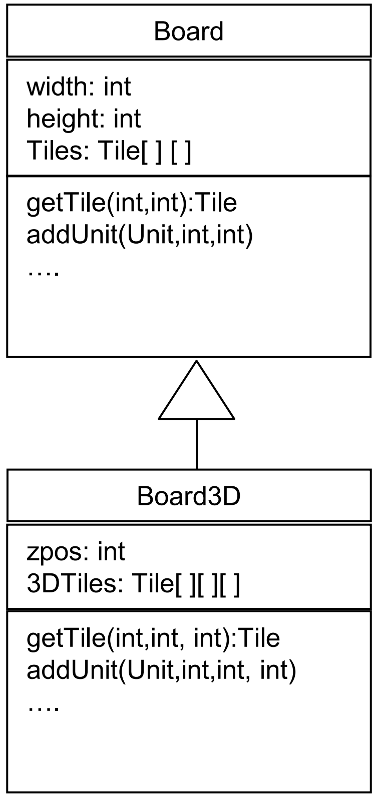
Syntax Conventions of LSP

- *Contravariance* of method arguments in the subtype
  - subtype CANNOT have more specialised parameters in the subtype (on method overrides)
  - since the class could not be used to replace the superclass that was passed more general params
  - but can have more general parameters
  - Java does not allow this, overrides are *invariant*
- *Covariance* of return types in the subtype
  - subtype CAN have more specialised returns in the subtype (on method overrides)
  - as long as the return type supports LSP since it can be used anywhere the superclass was
  - Java allows this
- No new exceptions in subtype, except where such exceptions are themselves subtypes of supertype exceptions
- See `topic1.oosd.lsp.LSP.java`

Behavioural Conventions of LSP

- *Preconditions* cannot be strengthened in a subtype
- *Postconditions* cannot be weakened in a subtype
- *Invariants* of the supertype must be preserved in a subtype
  - Design by contract covered in more detail later this semester
- History constraint (the "history rule").
  - Subtypes should not introduce state changes that are not permissible in the supertype
  - This can be enforced through strong use of private attributes and providing protected methods for state change

Misusing Subclassing

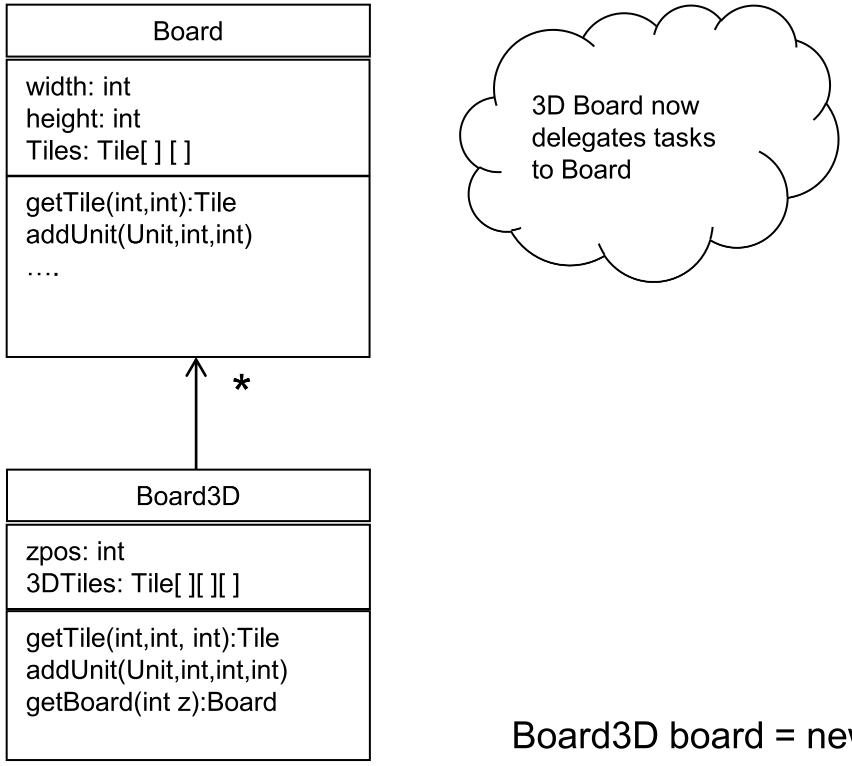


The designer subclassed Board to create 3DBoard

The methods getTile(int,int):Tile and addUnit(Unit,int,int) have no meaning in 3DBoard

```
Board board = new Board3D();
…
Tile tile = board.getTile(3,4);
```

Use Delegation Instead (Handing over responsibility)



```
Board3D board = new Board3D();
…
Tile tile = board.getBoard(2).getTile(3,4);
```

The Interface Segregation Principle

- "Clients should not be forced to depend on methods that they do not use." – R. Martin
- When we bundle functions for different clients into one interface, we create unnecessary coupling among the clients
  - When one client causes the interface to change, all other clients are forced to recompile
- Solution:
  - Keep interfaces small and COHESIVE
- **NOTE**: ISP is similar to SRP however you could have a cohesive class but split out operations to multiple interfaces e.g. read and write interfaces for an I/O class
- e.g. many Java interfaces have only one method!

(`java.lang.Comparable, Iterable, Runnable` etc.)

- Many java classes implement more than one interface

Dependency Inversion Principle

- High level (abstract) modules should not depend upon low level (concrete) modules, both should depend upon abstractions
- Abstractions should not depend upon details, details should depend upon abstractions
- High level modules should not be forced to change because of a change in low level / technology layers

SOLUTION:

- Don't depend on anything concrete, depend only upon abstraction
- Drives you towards low coupling
- A simple way to achieve dependency inversion is for clients to specify the concrete classes that will be used (see following slides)
- Concrete classes implement known interfaces
- Object creation through Factories (covered later in the semester)
- Directory/naming services and component lookup is another mechanism
- Related to Inversion of Control which is covered later this semester

Without Dependency Injection

```
// An example without dependency injection
public class User {
   // Internal reference to the service used by this client
   private Service service;
   User() {    // Constructor
      // a specific implementation in constructor instead of DI
       this.service = new ServiceExample();
   }
   // Method within this client that uses the service
   public String welcome() {
       return “Welcome" + service.getName();
   }
}
```

Constructor and Setter Injection

// Constructor injection

```
User(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

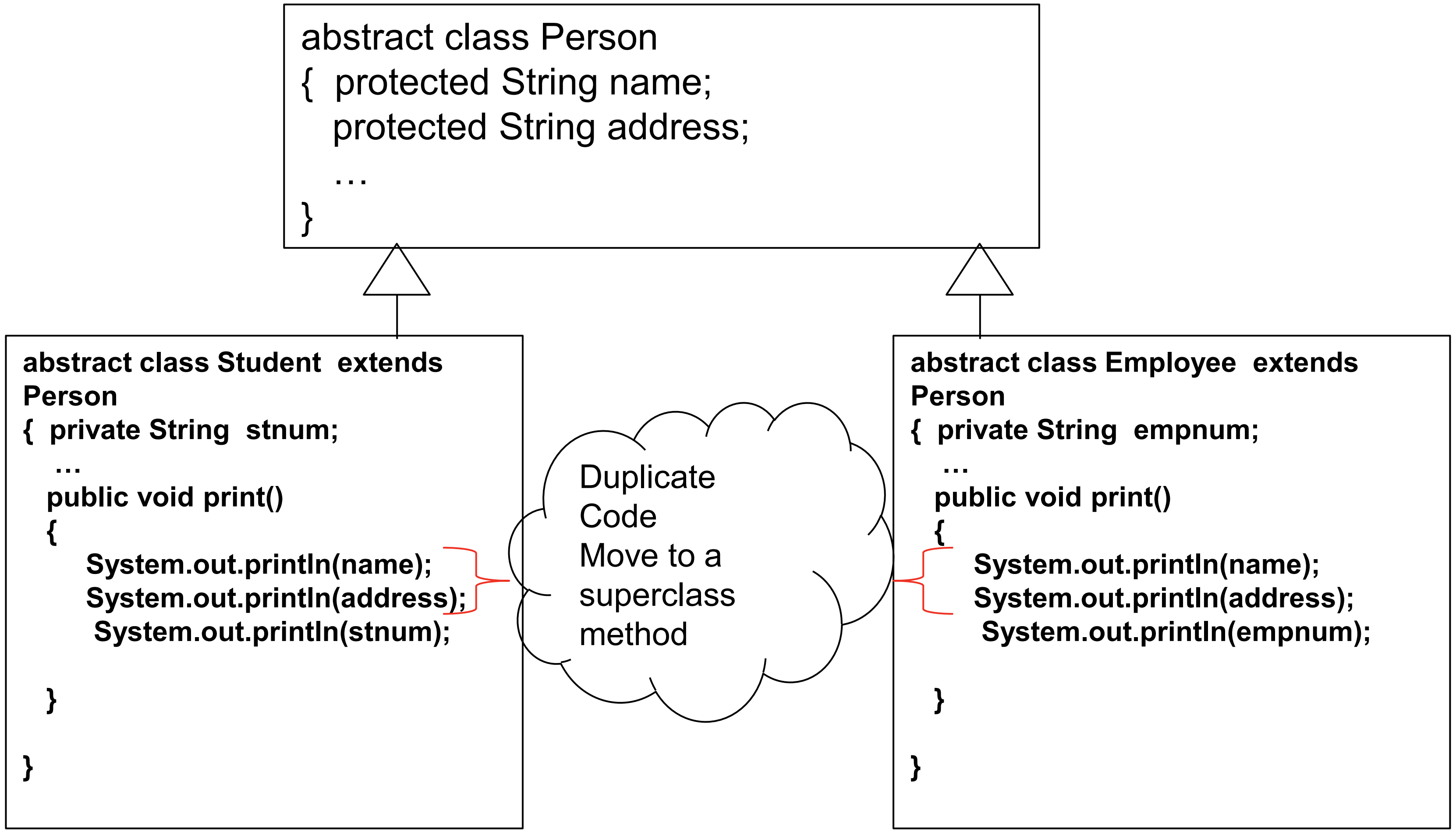
// Setter injection

```
public void setService(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

Frameworks such as the Java Spring Framework support automated dependency injection

The <u>D</u>on’t Repeat Yourself Principle (an extra D for SOLID!)

- This principle requires us to avoid duplicate code by abstracting out things that are common and placing those in a single location



```
abstract class Person
{  protected String name;
   protected String address;
   …
}
```

```
abstract class Student  extends
Person
{  private String  stnum;
    …
   public void print()
   {
        System.out.println(name);
        System.out.println(address);
         System.out.println(stnum);

   }

}
```

```
abstract class Employee  extends
Person
{  private String  empnum;
    …
   public void print()
   {
        System.out.println(name);
        System.out.println(address);
         System.out.println(empnum);

   }

}
```

Summary

This is a practical course focussing on:

- OO Principles/Modelling
- Design by Contract
- Refactoring
- Common Design Patterns (SOLID, GRASP, GoF)