COMP 2511
Object Oriented Design & Programming

- Earlier,
  - we looked at different types of relationships between objects
    - Inheritance
    - Association (and types of association)
- This week,

  we will learn about
  - How to code for different types of association
  - Abstract classes, interfaces and polymorphism
  - How to model inheritance in the right way
  - And we start exploring a series of design principles for building good software

Coding for association, aggregation, composition

Association

- A semantically weak relationship between otherwise unrelated objects
- Association represents a "`uses`" relationship between two or more objects in which the objects have their own life-times and there is no "`ownership`"

Aggregation, Composition

Aggregation

- A specialised form of association between two or more objects in which objects have their own life-cycle but there is ownership as well

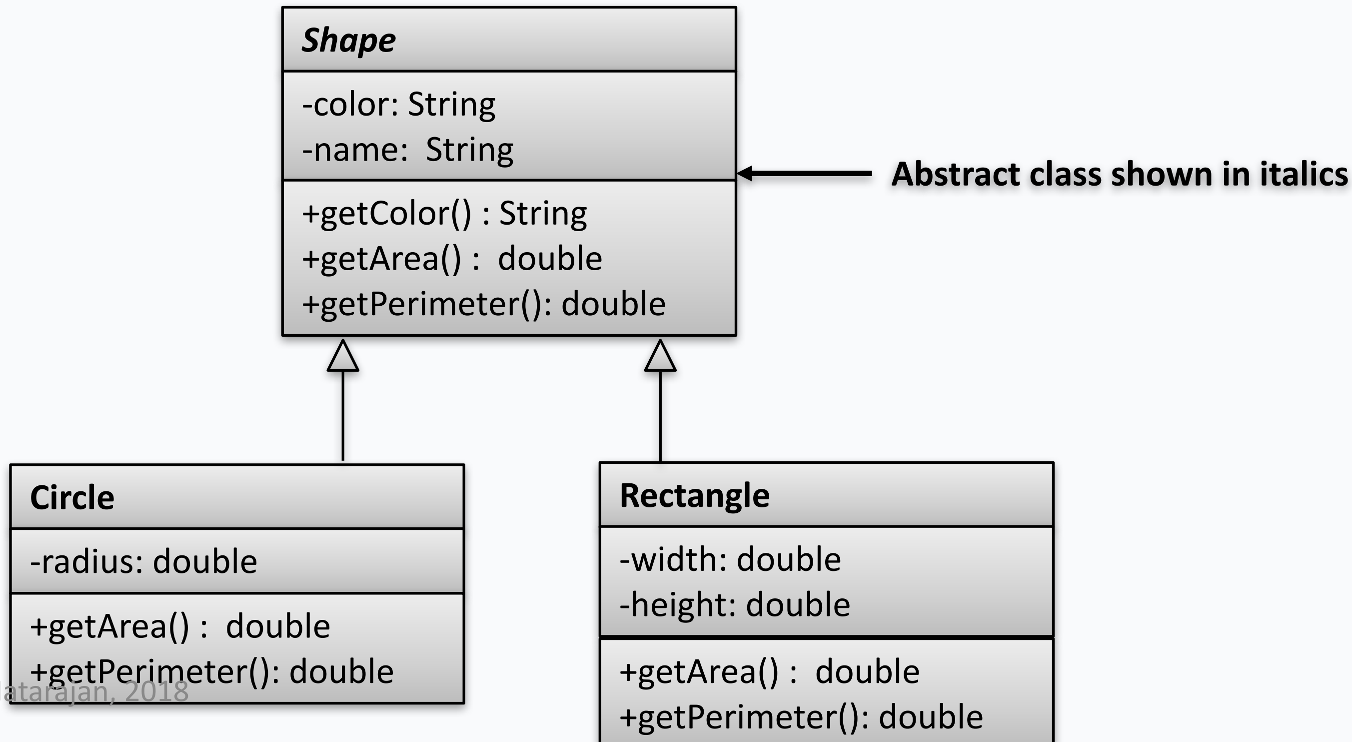
Composition

- A specialised form of aggregation in which if the parent object is destroyed, the child objects ceases to exist
- Often referred to as a "Death Relationship"

Abstract classes, Interfaces and Polymorphism

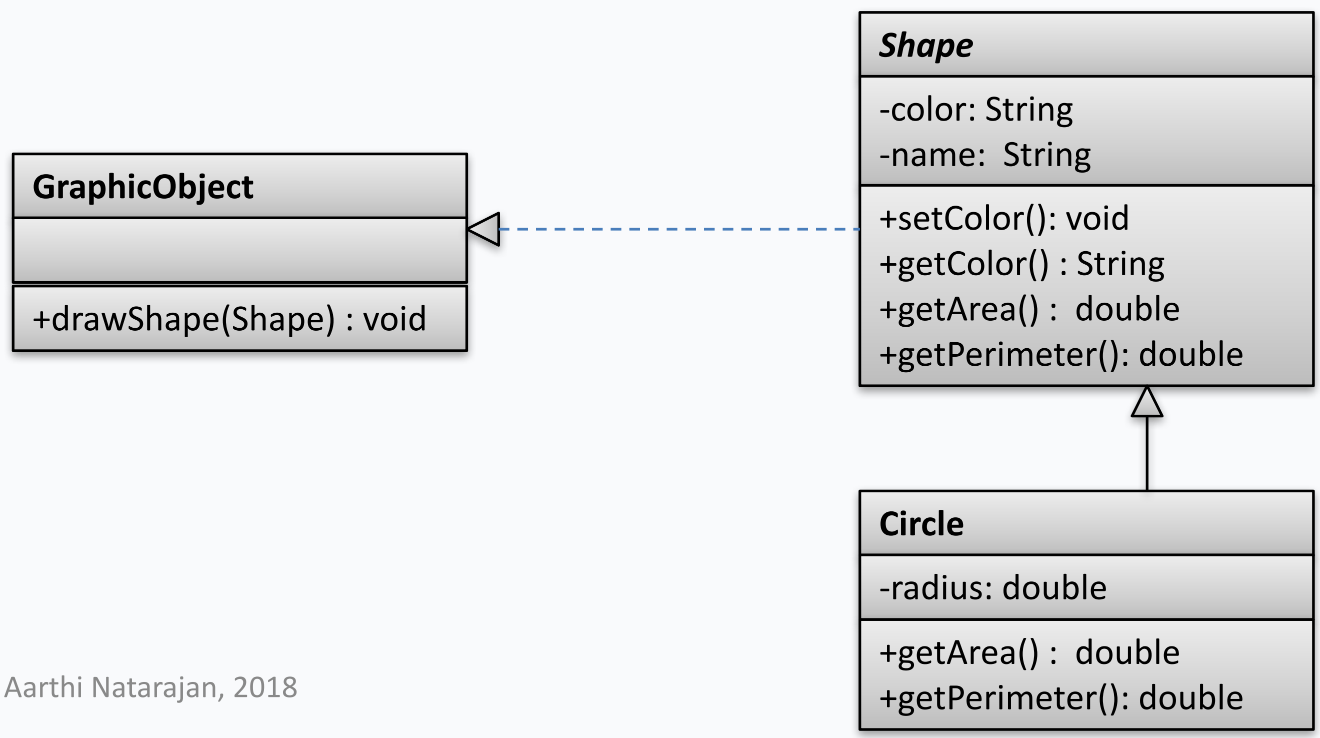
Abstract Classes

- A class which cannot be instantiated, i.e. cannot create object instances of an abstract class
- Serves as a convenient place-holder for factoring out common behaviour in sub-classes
- An abstract class **may** define *abstract* methods (undefined in the abstract class and defined in the sub-class
- A class with one or more abstract methods must be declared as abstract



Interfaces

- An **interface** type defines collection of abstract methods, i.e., defining methods without a body (Java 8 can have default methods)
  - Specifies "*what*" needs to be done, not "*how*" to do it
- A class **implements** an interface, by providing an implementation for **all** the methods in the interface
- Classes can implement multiple interfaces, but extend only one class



Revising Polymorphism

- We have seen polymorphism in Java means:
  - A variable of a particular reference type can change behaviour depending upon the object instance it is pointing to
  - Actual method invoked depends upon object at run-time and done by dynamic-binding
  - Polymorphism works with interfaces as well…

```
GraphicObject g1 = new Rectangle();
GraphicObject g2 = new Circle();
```

    where `Rectangle` and `Circle` implement the interface `GraphicObject`

    polymorphism guarantees that the right `drawshape()` method is applied to ensure correct results

Designing Good Software

The One Constant in software analysis and design

- What is the one thing you can always count on in writing software? - **Change**

Building Good Software

Is all about:

- Making sure your software does what the customer wants it to do – use-case diagram, feature list, prioritise them
- Applying OO design principles to:
  - To ensure the system is flexible and extensible to accommodate changes in requirements
  - To strive for a maintainable, reusable, extensible design

- A change in requirements some-times reveals problems with your system that you did not even know that they existed

- Remember, change is constant and your system should **continually improve** when you add these changes…..else software rots

Why does Software Rot?

We write bad code

Why do write bad code ?

- Is it because **do not know** how to write better code?
- Requirements change in ways that original design did not anticipate
- But changes are not the issue –
  - changes requires refactoring and refactoring requires time and we say we do not have the time
  - Business pressure - changes need to be made quickly – “quick and dirty solutions”
  - changes may be made by developers not familiar with the original design philosophy

Bad code, in fact slows us down

Design Smells

When software rots

it smells…

A design smell

- is a symptom of poor design
- often caused by violation of key design principles
- has structures in software that suggest refactoring

Design Smells (1)

Rigidity

- Tendency of the software being too difficult to change even in simple ways
- A single change causes a cascade of changes to other dependent modules

Fragility

- Tendency of the software to break in many places when a single change is made

Rigidity and fragility complement each other – aim towards minimal impact, when a new feature or change is needed

Design Smells (2)

Immobility

- Design is hard to reuse
- Design has parts that could be useful to other systems, but the effort needed and risk in disentangling the system is too high

Viscosity

- Software viscosity – changes are easier to implement through 'hacks' over 'design preserving methods'
- Environment viscosity – development environment is slow and in-efficient

Opacity

- Tendency of a module to be difficult to understand
- Code must be written in a clear and expressive manner

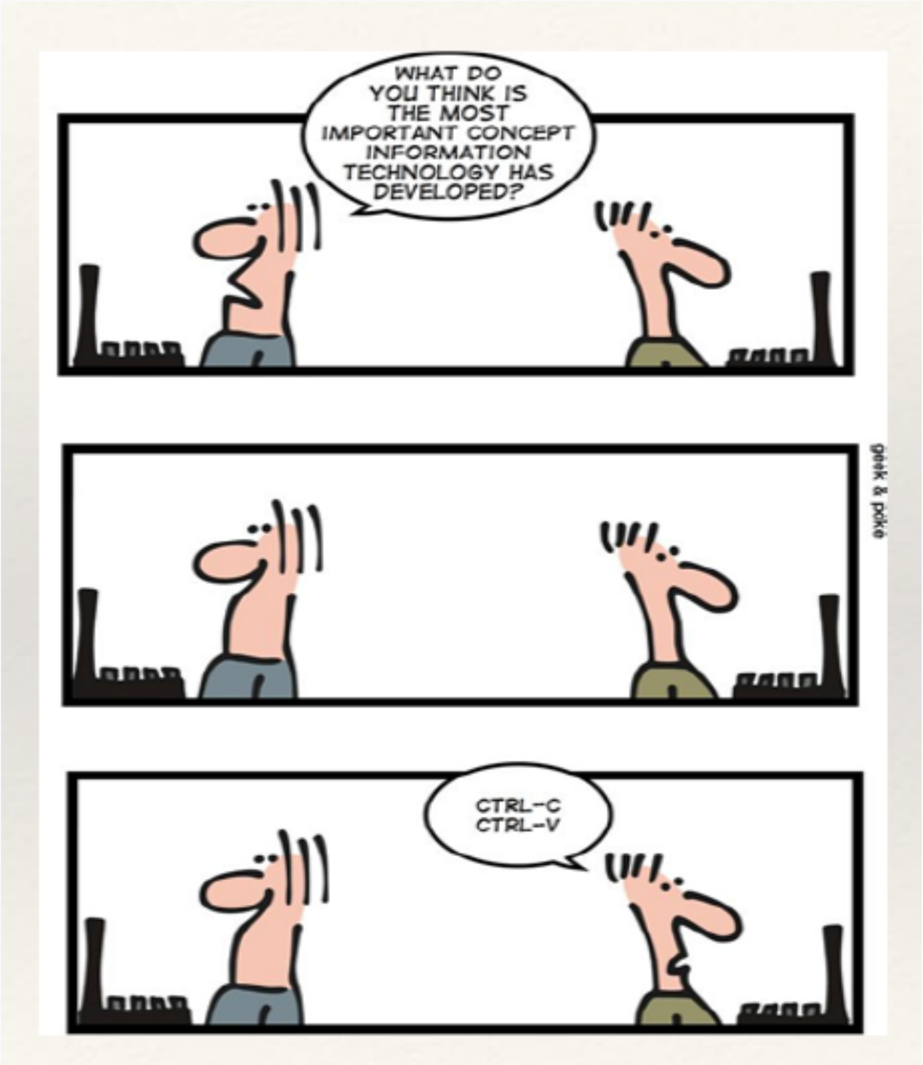
Design Smells (3)

Needless complexity

- Contains constructs that are not currently useful
- Developers ahead of requirements

Needless repetition

- Design contains repeated structures that could potentially be unified under a single abstraction
- Bugs found in repeated units have to be fixed in every repetition



Characteristics of Good Design

So, we know when our design smells…
But how do we measure if a software is well-designed?

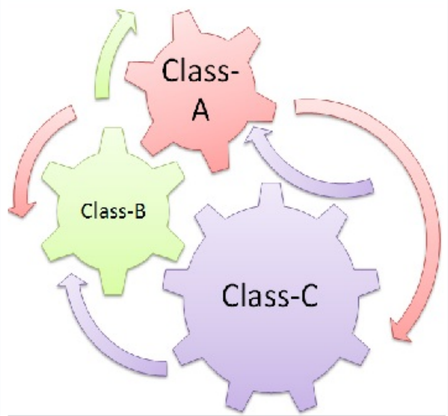
The design quality of software is characterised by

- Coupling
- Cohesion

Good software aims for building a system with loose coupling and high cohesion among its components so that software entities are:

- Extensible
- Reusable
- Maintainable
- Understandable
- Testable

Coupling



- Is defined as the degree of interdependence between components or classes
- High coupling occurs when one component **A** depends on the internal workings of another component **B** and is affected by internal changes to component **B**
- High coupling leads to a complex system, with difficulties in maintenance and extension…eventual software rot
- Aim for loosely coupled classes - allows components to be used and modified independently of each other
- But “zero-coupled” classes are not usable – striking a balance is an art!

Cohesion

- The degree to which all elements of a component or class or module work together as a functional unit
- Highly cohesive modules are:
  - much easier to maintain and less frequently changed and have higher probability of reusability
- Think about
  - How well the lines of code in a method or function work together to create a sense of purpose?
  - How well do the methods and properties of a class work together to define a class and its purpose?
  - How well do the classes fit together to create modules?
- Again, just like zero-coupling, do not put all the responsibility into a single class to avoid low cohesion!

And, applying design principles is the key to creating high-quality software

"Design principles are key notions considered fundamental to many different software design approaches and concepts."

- SWEBOK v3 (2014)

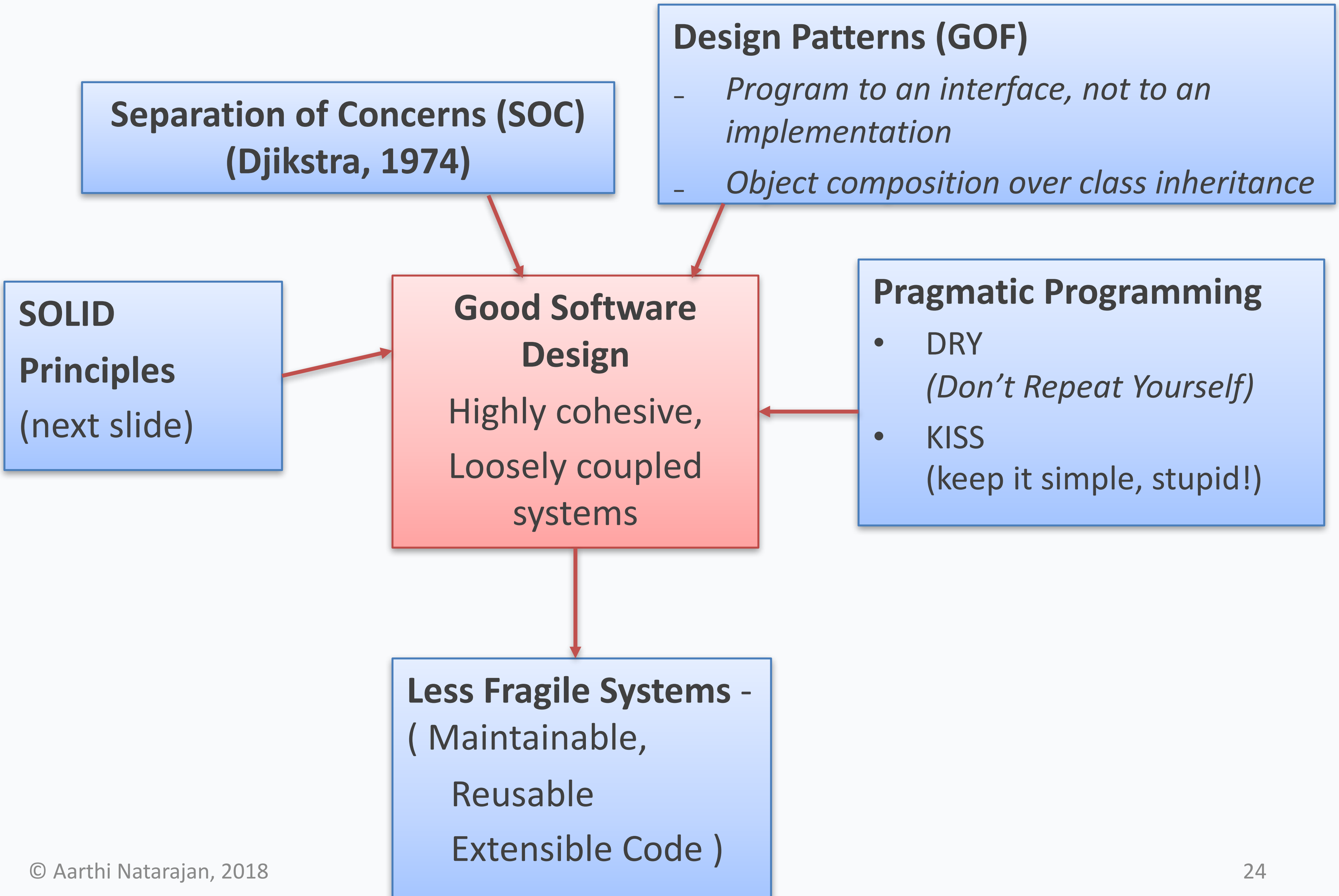
"The critical design tool for software development is a mind well educated in design principles"

- Craig Larman

What is a “design principle”?

A basic tool or technique that can be applied to designing or writing code to make software more maintainable, flexible and extensible

Several Design Principles…One Goal



SOLID

- Single responsibility principle: A class should only have a single responsibility.
- Open–closed principle: Software entities should be open for extension, but closed for modification.
- Liskov substitution principle: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- Interface segregation principle: Many client-specific interfaces are better than one general-purpose interface.
- Dependency inversion principle: One should "depend upon abstractions, [not] concretions.”

When to use design principles

- Design principles help eliminate design smells
- But, don't apply principles when there no design smells
- Unconditionally conforming to a principle (just because it is a principle is a mistake)
- Over-conformance leads to the design smell – needless complexity

Design Principle #1

The Principle of Least Knowledge or Law of Demeter

Design Principle #1

The Principle of Least Knowledge (Law of Demeter) – Talk only to your friends

- Classes should know about and interact with as few classes as possible
- Reduce the interaction between objects to just a few close “friends”
- These friends are “immediate friends” or “local objects”
- Helps us to design “loosely coupled” systems so that changes to one part of the system does not cascade to other parts of the system
- The principle limits interaction through a set of rules

The Principle of Least Knowledge (Law of Demeter)

A method in an object should only invoke methods of:

- The object itself
- The object passed in as a parameter to the method
- Objects instantiated within the method
- Any component objects
- And not those of objects returned by a method

Don't dig deep inside your friends for friends of friends of friends and get in deep conversations with them -- don't do

- e.g. o.get(name).get(thing).remove(node)

Principle of Least Knowledge, Rule 1:

A method **M** in an object **O** can call on any other method within **O** itself

- This rule makes logical sense, a method encapsulated within a class can call any other method that is also encapsulated within the same class

```
public class M {
 public void methodM() {
        this.methodN();
 }
 public void methodN() {
        // do something
 }
}
```

- Here `methodM()` calls `methodN()` as both are methods of the same class

Principle of Least Knowledge, Rule 2:

A method **M** in an object **O** can call on any methods of parameters passed to the method **M**

- The parameter is local to the method, hence it can be called as a friend

```
public class O {

public void M(Friend f) {
// Invoking a method on a parameter passed to the method is
// legal
f.N();
}

public class Friend {

 public void N() {
 // do something
 }
}
```

Principle of Least Knowledge, Rule 3:

A method **M** can call a method **N** of another object, if that object is instantiated within the method **M**

- The object instantiated is considered "local" just as the object passed in as a parameter

```
public class O {

   public void M() {
        Friend f = new Friend();
        // Invoking a method on an object created within the
        // method is legal
        f.N();
   }

public class Friend {
        public void N() {
         // do something
        }
}
```

Principle of Least Knowledge, Rule 4:

Any method **M** in an object **O** can call on any methods of any type of object that is a direct component of **O**

- This means a method of a class can call methods of classes of its instance variables

```
public class O {

   public Friend instanceVar = new Friend();

   public void M4() {
   // Any method can access the methods of the friend class
   F through the instance variable "instanceVar"
   instanceVar.N();
   }
public class Friend {
       public void N() {
        // do something
       }
}
```

Well-designed Inheritance

Design Principle #2
LSP (Liskov Substitution Principle)

LSP is about well-designed inheritance

Barbara Liskov (1988) wrote:

> *If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.*

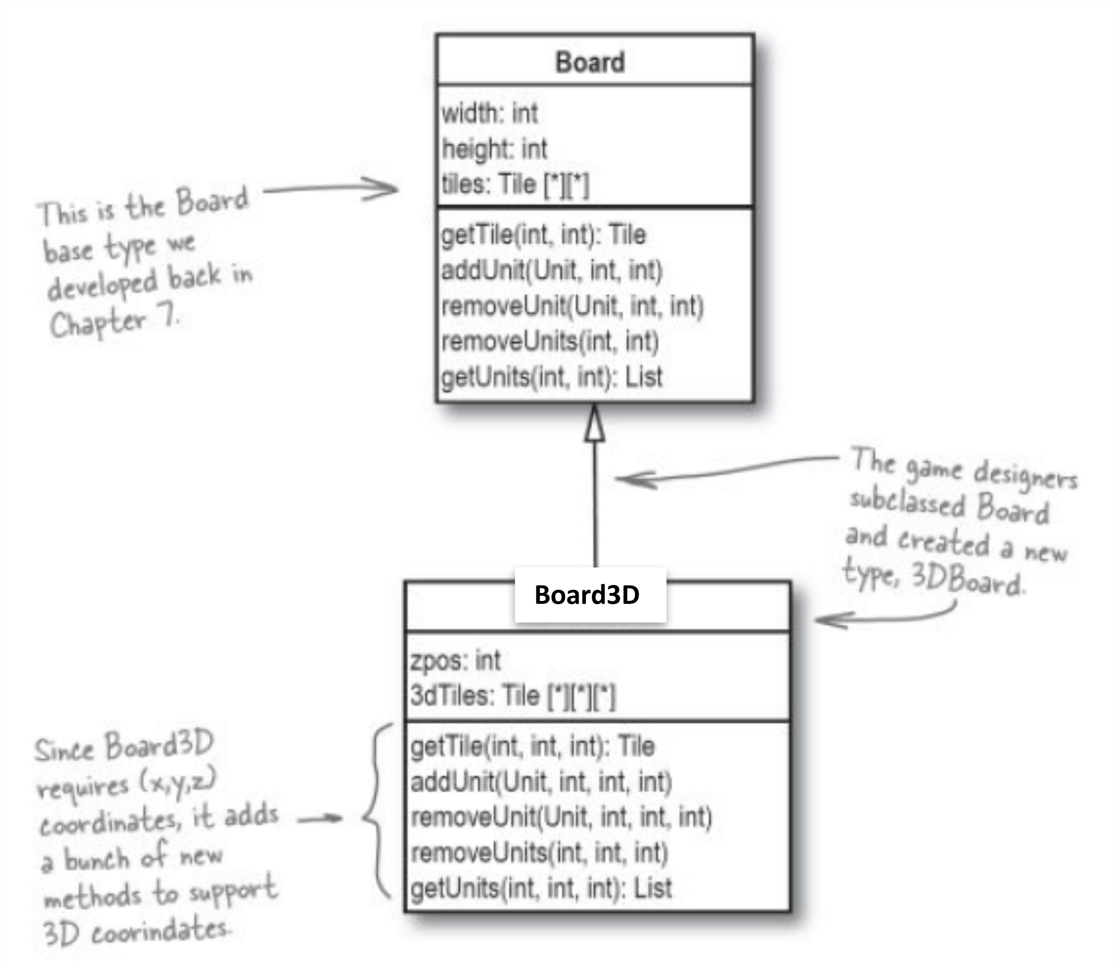
Bob wrote:

> *subtypes must be substitutable for their base types*

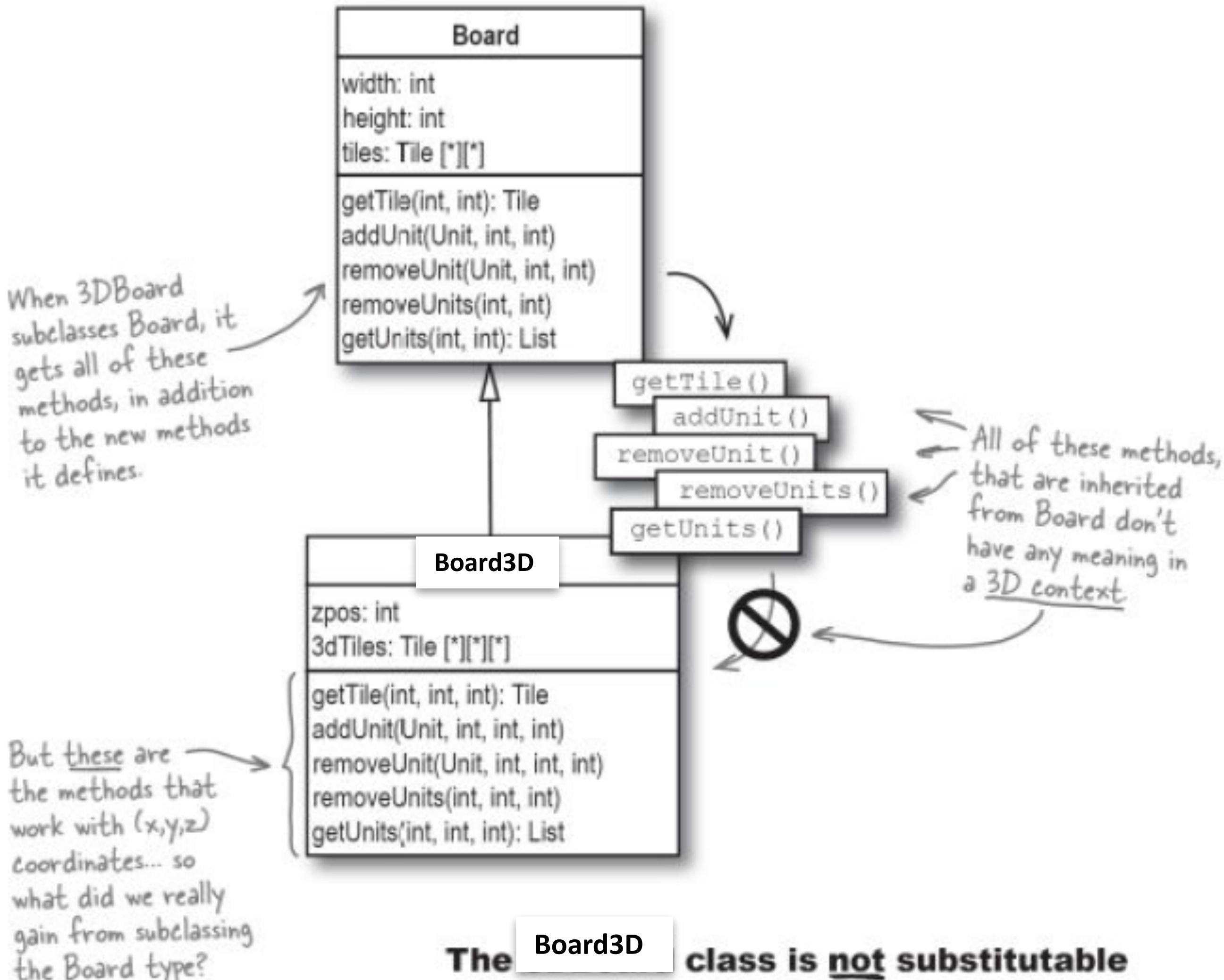
Lecture demo: Square vs Rectangle

What is the problem with Square-Rectangle IS A relationship?

Another LSP Example: A board game



LSP reveals hidden problems with the above inheritance structure



The Board3D class is <u>not</u> substitutable for Board, because none of the methods on Board work correctly in a 3D environment. Calling a method like getUnits(2, 5) <u>doesn't</u> <u>make</u> <u>sense</u> for 3DBoard. So this design violates the LSP.

Even worse, we don't know what passing a coordinate like (2,5) even <u>means</u> to 3DBoard. This is <u>not</u> a good use of inheritance.

What are the issues?

LSP states that subtypes must be substitutable for their base types

```
Board board = new Board3D()
```

But, when you start to *use* the instance of `Board3D` like a `Board`, things go wrong

```
Artillery unit = board.getUnits(8,4)
```

Board here is actually an instance of the sub-type Board3D

But, what does this method for a 3D board?

Inheritance and LSP indicate that any method on `Board` should be able to use on a `Board3D`, and that `Board3D` can stand in for `Board` without any problems, so the above example clearly violates LSP

Solve the problem without inheritance

So what options are there besides inheritance?

- Delegation – delegate the functionality to another class
- Composition – reuse behaviour using one or more classes with composition

Design Principle: *Favour composition over inheritance*

If you favour delegation, composition over inheritance, your software will be more flexible, easier to maintain, extend

Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method
- The access level cannot be more restrictive than the overridden method's access level.

  E.g., if the super class method is declared `public` then the overriding method in the sub class cannot be either `private` or `protected`.
- A method declared `final` cannot be overridden.
- Constructors cannot be overridden.

Can static methods be over-ridden?

Static methods can be defined in the sub-class with the same signature

– This is not overriding, as there is no run-time polymorphism
– The method in the derived class hides the method in the base class

```
Lecture demo...
```

Rules for Method Overriding

Covariance of return types in the overridden method

- The return type in the overridden method should be the same or a sub-type of the return type defined in the super-class
- This means that return types in the overridden method may be narrower than the parent return types

```java
public class AnimalShelter {

    public Animal getAnimalForAdoption() {

        return null;
    }

    public void putAnimal(Animal someAnimal){

    }

}
```

```java
public class CatShelter extends AnimalShelter {

    /*
     * @see AnimalShelter#getAnimalForAdoption()
     */
    @Override
    public Cat getAnimalForAdoption() {

        //Returning a narrower type than parent
        return new Cat();
    }

}
```

Rules for Method Overriding

What about Contra-variance of method arguments in the overridden method

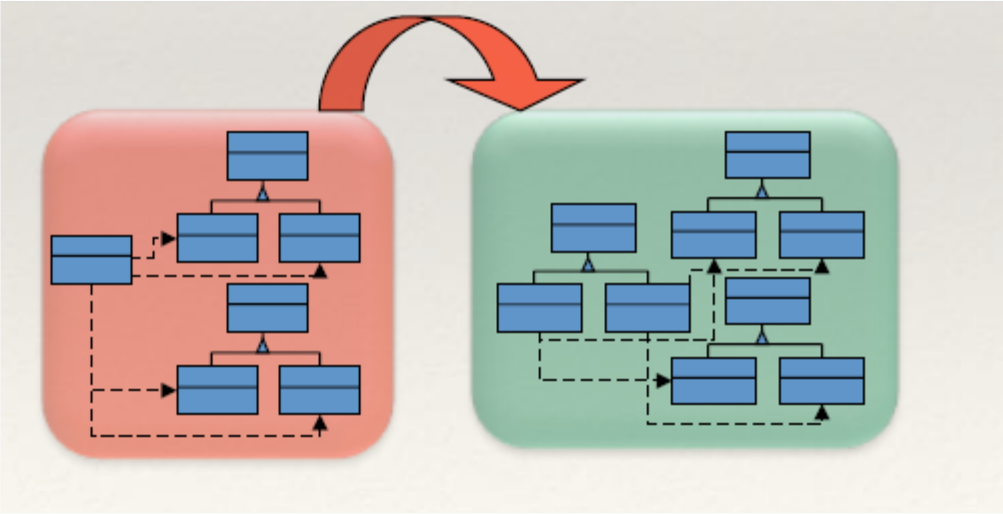
Can arguments to methods in sub-class be wider than the arguments passed in the parent's method ?

```
public class CatShelter extends AnimalShelter {


    /*
     * @see AnimalShelter#putAnimal(Animal)
     */
    // Java sees this as an unrelated method.
    // This is not actually overriding parent method
    public void putAnimal(Object someAnimal) {
        // do something
    }
```

Refactoring

The process of restructuring (changing the internal structure of software) software to make it *easier to understand* and *cheaper to modify* <u>without</u> changing its *external, observable behaviour*



Why should you refactor?

- Refactoring improves design of software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster
- Refactoring helps you to conform to design principles and avoid design smells

When should you refactor?

Tip: *When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature*

Refactor when:

- You add a function (swap hats between adding a function and refactoring)
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review

Common Bad Code Smells

- **Duplicated Code**
  - Same code structure in more than one place or
  - Same expression in two sibling classes
- **Long Method**
- **Large Class** (when a class is trying to do too much, it often shows up as too many instance variables)
- **Long Parameter List**
- **Divergent Change** (when one class is commonly changed in different ways for different reasons)
- **Shotgun Surgery** (The opposite of divergent change, when you have to make a lot of little changes to a lot of different classes

The Video Rental Example

What is wrong with the design?

Is it wrong to write a quick and dirty solution OR is it an aesthetic judgment (dislike of ugly code) …

- Overly long `statement()` method , poorly designed that does far too much, tasks that should be done by other classes (Code Smell: Long Method)
- What if customer wanted to generate a statement in HTML? - Impossible to reuse any of the behaviour of the current statement method for an HTML statement. (Code Smell: Duplicated code)
- What about changes?
  - What happens when “charging rules” change?
  - what if the user wanted to change the way the movie was classified
- The code is a maintenance night-mare (Design smell: Rigidity)

Improving the design

Apply a series of fundamental refactoring techniques:

Technique #1: Extract Method

- Find a logical clump of code and use Extract Method.

  Which is the obvious place? the `switch` statement

- Scan the fragment for any variables that are local in scope to the method we are looking at

  (`Rental r` and `thisAmount`)

- Identify the changing and non-changing local variables
- Non-changing variable can be passed as a parameter
- Any variable that is modified needs more care, if there is only one, you could simply do a return

Improving the design

Technique #2: Rename variable

- Is renaming worth the effort? Absolutely
- Good code should communicate what it is doing clearly, and variable names are a key to clear code. Never be afraid to change the names of things to improve clarity.

Tip

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Improving the design

#3: Move method

- Re-examine method `calculateRental()` in class `Customer`
- Method uses the `Rental` object and not the `Customer` object
- Method is on the wrong object

Tip

Generally, a method should be on the object whose data it uses

Improving the design

What OO principles do Extract Method and Move Method use?

They make code reusable through Encapsulation and Delegation

But, isn't encapsulation about keeping your data private?

The basic idea about encapsulation is to protect information in one part of your application from other parts of the application, so

- You can protect data
- You can protect behaviour – when you break the behaviour out from a class, you can change the behaviour without the class having to change

And what is delegation?

- The act of one object forwarding an operation to another object to be performed on behalf of the first object

Improving the design

#4: Replace Temp With Query

- A technique to remove unnecessary local and temporary variables
- Temporary variables are particularly insidious in long methods and you can loose track of what they are needed for
- Sometimes, there is a performance price to pay

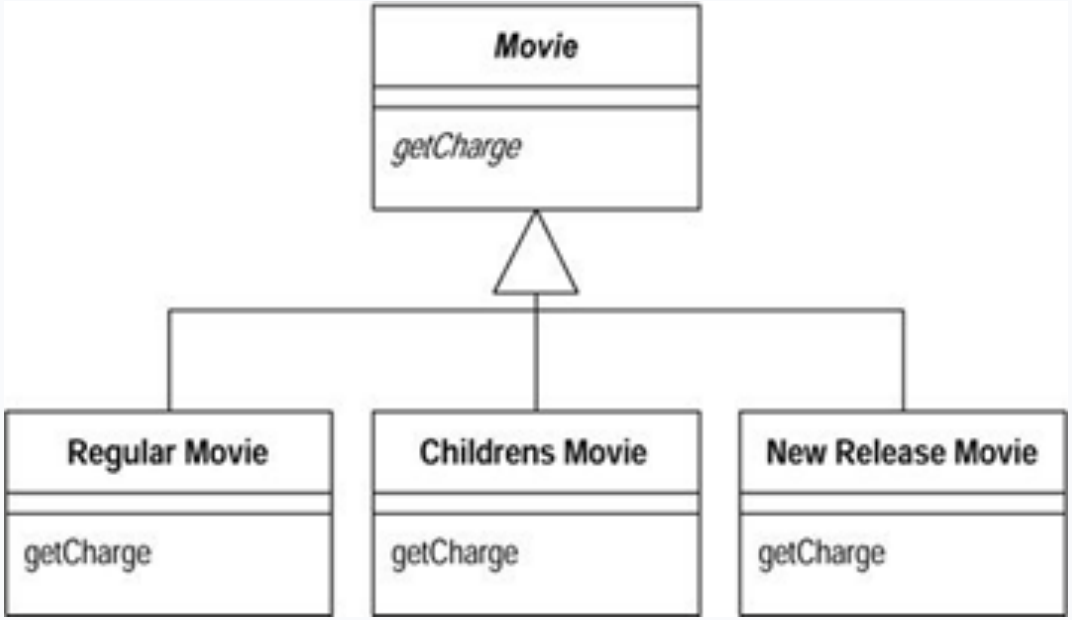
Improving the design

#5: Replacing conditional logic with Polymorphism

- The switch statement – an obvious problem, with two issues
- class `Rental` Is tightly coupled with class `Movie` - a switch statement based on the data of another object – not a good design
- There are several types of movies with its own type of charge, hmm… sounds like inheritance

Improving the design

- `A base` class `Movie` class with method `getPrice()` and sub-classes `NewRelease`, `ChildrenMovie` and `Regular`
- This allows us to replace `switch` statement with polymorphism



- Sadly, it has one flaw...a movie can change its classification during its life-time

So, what options are there besides inheritance ?

- Composition – reuse behaviour using one or more classes with composition
- Delegation: delegate the functionality to another class

…this is the second time, this week we have said, we need something more than inheritance

So, next ...

- **Design Principle:** Favour composition over inheritance
- **More refactoring techniques to solve our "switch" problem**
  - Replace type code with Strategy/State Pattern
  - Move Method
  - Replace conditional code with polymorphism