Universität Zürich[UZH]

Institut für Informatik

Software Engineering HS 2015

Lecture: Modular Design

Thomas Fritz & Martin Glinz

Many thanks to Philippe Beaudoin, Gail Murphy, David Shepherd, Neil Ernst and Meghan Allen

Reading!

For next lecture: (all required)

Composite Design Pattern
http://sourcemaking.com/design_patterns/composite

Mediator Design Pattern
http://sourcemaking.com/design_patterns/mediator

Façade Design Pattern
http://sourcemaking.com/design_patterns/facade

Modular Design Overview

- Introduction to Modularity

- Principles and Heuristics for good Modularity
  - High Cohesion
  - Loose Coupling
  - **Information Hiding**
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Law of Demeter

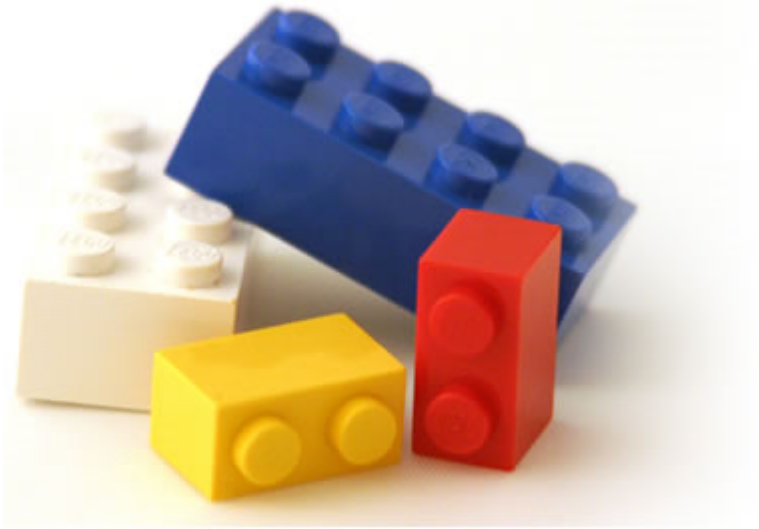
Learning Goals

By the end of this unit you will be able to:

- Critique a UML diagram and provide concrete suggestions of how to improve the design
- Explain the goal of a good modular design and why it is important
- Apply design-principles (the ones presented) appropriately

Recap: Bad Design

Software Design – Modularity



The goal of all software design techniques is to break a complicated problem into simple pieces.

Modular Design

Why Modularity?

Why Modularity?

- Minimize Complexity
- Reusability
- Extensibility
- Portability
- Maintainability
- …

What is a good modular Design?

- There is no “right answer” with design
- Applying heuristics/principles can provide insights and lead to a good design

What is a good modular Design?

Pragmatic Programmer:

Eliminate Effects Between Unrelated Things – design components that are:

self-contained,

independent,

and have a single, well-defined purpose.

Andrew Hunt and David Thomas

Design Principles

Source: [*Gamma et all, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995*]

- Program an Interface not an Implementation
- Favor Composition Versus Inheritance
- Find what varies and encapsulate it

Source: [*R. Martin, "Agile Software Development, Principles, Patterns, and Practices", Prentice-Hall, 2002*]

- Dependency-Inversion Principle
- Liskov Substitution Principle
- Open-Closed Principle
- Interface-Segregation Principle
- Reuse/Release Equivalency Principle
- Common Closure Principle
- Common Reuse Principle
- Acyclic Dependencies Principle
- Stable Dependencies Principle
- Stable Abstraction Principle

Source: [*Larman, "Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development", Prentice-Hall, 20*

- Design principles are codified in the GRASP Pattern
- GRASP (Pattern of General Principles in Assigning Responsabilities)
- Assign a responsibility to the information expert
- Assign a responsability so that coupling remains low
- Assign a responsability so that cohesion remains high
- Assign responsabilities using polymorphic operations
- Assign a highly cohesive set of responsabilities to an artificial class that does not represent anything in the problem domain (when you want to
- Don't talk to strangers (Law of Demeter)

Source: [*Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", Commununication of ACM, 1972*]

- Information Hiding
- Modularity

Source: [*Hunt, Thomas, "The Pragmatic Programmer: From Journeyman to Master", Addison-Wesley, 1999*]

- DRY – Don't Repeat yourself
- Make it easy to reuse
- Design for Orthogonality
- Eliminate effects between unrelated things
- Program close to the problem domain
- Minimize Coupling between Modules
- Design Using Services
- Always Design for Concurrency
- Abstractions Live Longer than details

Source: [*Lieberherr,Holland, "Assuring Good Style for Object-Oriented Programs", IEEE Software, September 1989*]

- Law of Demeter

Source: [*Raymond, "Art of Unix Programming", Addison-Wesley, 2003*]

Principles & Heuristics for modular Design

- High Cohesion
- Loose Coupling
- Information Hiding
- Open/Closed Principle
- Liskov Substitution Principle
- Law of Demeter
- ….

Discussion question

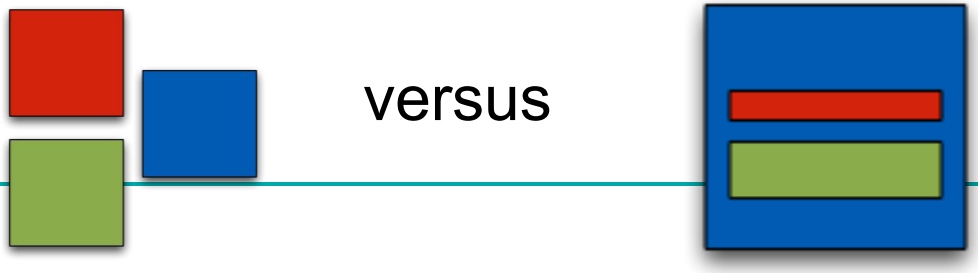
- Which of these two designs is better?

A.
```
public class AddressBook {
    private LinkedList<Address> theAddresses;
    public void add (Address a)
        {theAddresses.add(a);}
        // ... etc. ...
  }
```

B.
```
public class AddressBook
        extends LinkedList<Address> {
  // no need to write an add method, we inherit it
}
```

High Cohesion

- Cohesion refers to how closely the functions in a module are related
- Modules should contain functions that logically belong together
  - Group functions that work on the same data
- Classes should have a **single responsibility** (no schizophrenic classes)



High or low cohesion?

```
public class EmailMessage {
    …
    public void sendMessage() {…}
    public void setSubject(String subj) {…}
    public void setSender(Sender sender) {…}
    public void login(String user, String passw) {…}
    ….
}
```

The Or-Check

- A class description that describes a class in terms of alternatives is probably not a class but a set of classes

  "A ClassRoom is a location where students attend tutorials **OR** labs"

  May need to be modeled as two classes:

  TutorialRoom and ComputerLab

Loose Coupling

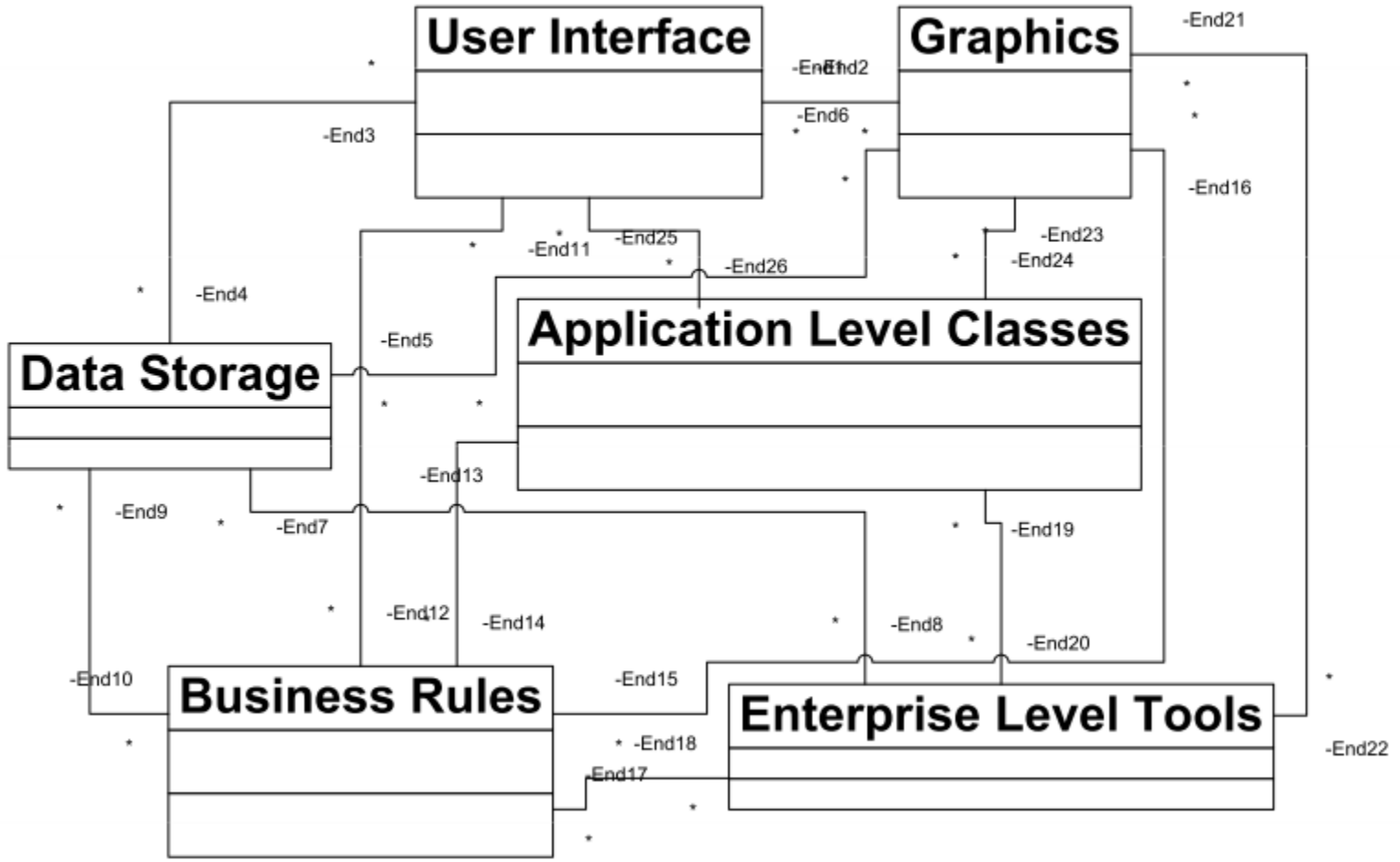
- Coupling assesses how tightly a module is related to other modules

- Goal is loose coupling:

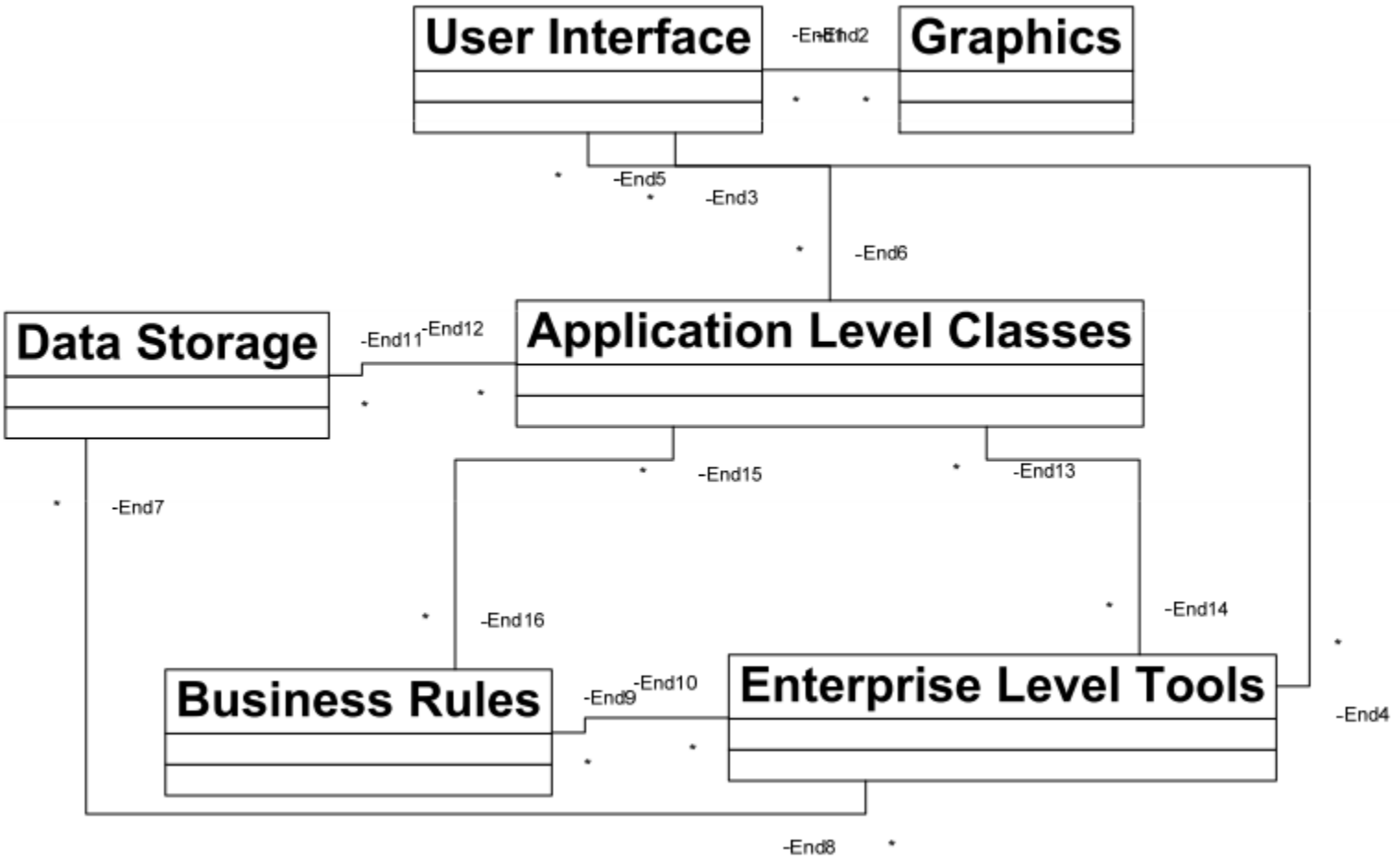
  modules should depend on as few modules as possible

- Changes in modules should not impact other modules; easier to work with them separately

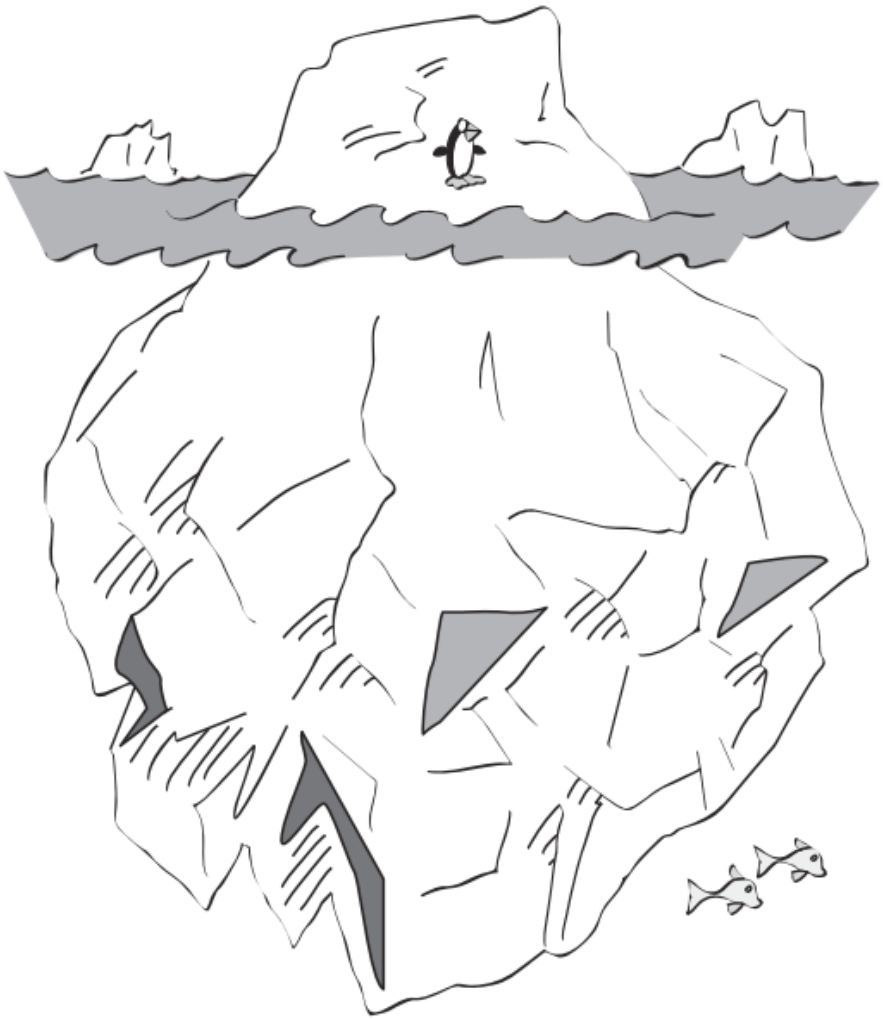
Tightly or loosely coupled?



Tightly or loosely coupled?



Information Hiding

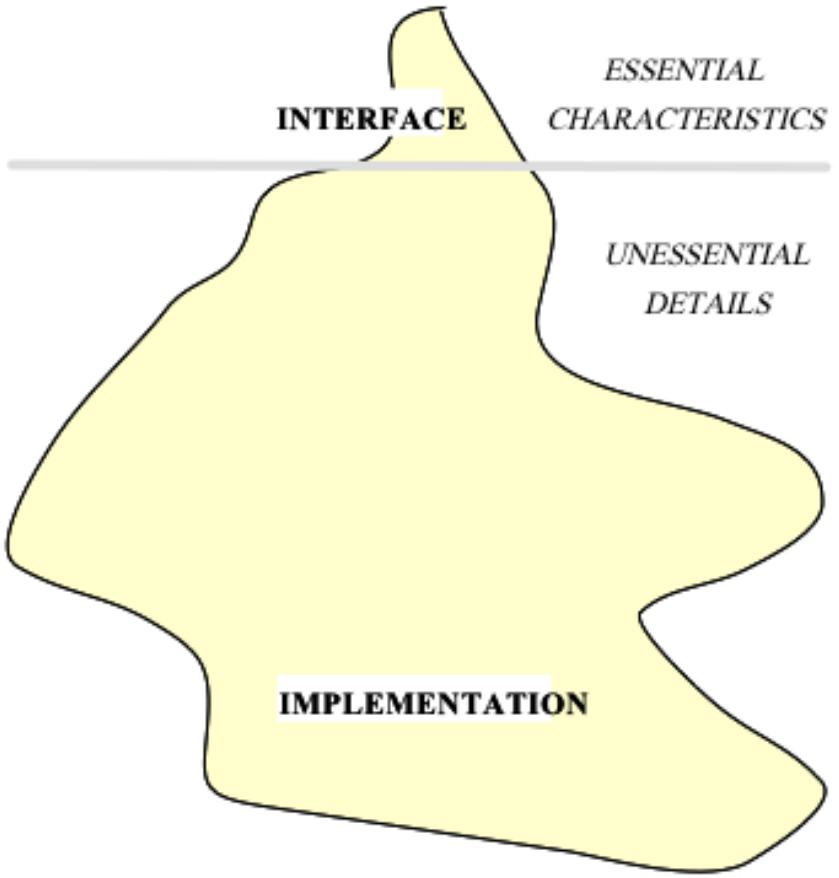


A good class is a lot like an iceberg: seven-eights is under water, and you can see only the one-eight that's above the surface.

from CodeComplete by Steve McConnell

Information Hiding

- Only expose **necessary** functions
- Abstraction hides complexity by emphasizing on essential characteristics and suppressing detail
- Caller should not assume anything about *how* the interface is implemented
- Effects of internal changes are localized



Information Hiding: Example 1

The chief scientist of the elementary particle research lab asks the new intern about his latest results: “So what is the average momentum of these neutral particles?”

a) 42

b) Hmmm. Take this pile of sheet with my observations, here is the textbook that explains how to calculate momentum, also you will need to search online for the latest reference tables. Oh, and don’t forget to correct for multiplicity!

Which answer is the most likely to get the intern fired?

Information Hiding: Example 2

- Class DentistScheduler has
  - A public method *automaticallySchedule()*
  - Private methods:
    - *whoToScheduleNext()*
    - *whoToGiveBadHour()*
    - *isHourBad()*

- To use DentistScheduler, just call *automaticallySchedule()*
  - Don't have to know how it's done internally
  - Could use a different scheduling technique: no problem!

Open/Closed Principle

- Classes should be open for extensions
  - It is often desirable to modify the behavior of a class while reusing most of it
- Classes should be closed for change
  - Modifying the source code of a class risks breaking every other class that relies on it
- Achieved through inheritance and dynamic binding

Open/Closed and Information Hiding

- Modifying the source code of a class risks breaking every other class that relies on it
- However, information hiding says that we should not assume anything about implementation
- So is there a need to keep classes *closed for change*?
  - Yes because the implied behavior should never change!
  - Inherit to reuse an interface while changing the behavior

Open/Closed Example

```
class Drawing {
  public void drawAllShapes(List<IShape> shapes) {
      for (IShape shape : shapes) {
          if (shape instanceof Square()) {
              drawSquare((Square) shape);
          } else if (shape instanceof Circle) {
              drawCircle((Circle) shape));
  } } }

  private void drawSquare(Square square) {  …// draw the square…  }
  private void drawCircle(Circle square) {  …// draw the circle…  }
}
```

Open/Closed Example

```
class Drawing {
    public void drawAllShapes(List<IShape> shapes) {
        for (IShape shape : shapes) {
            shape.draw();
} } }

interface IShape {
    public void draw();
}

class Square implements IShape {
    public void draw() {  // draw the square  }
}
```

Open/Closed Caveat

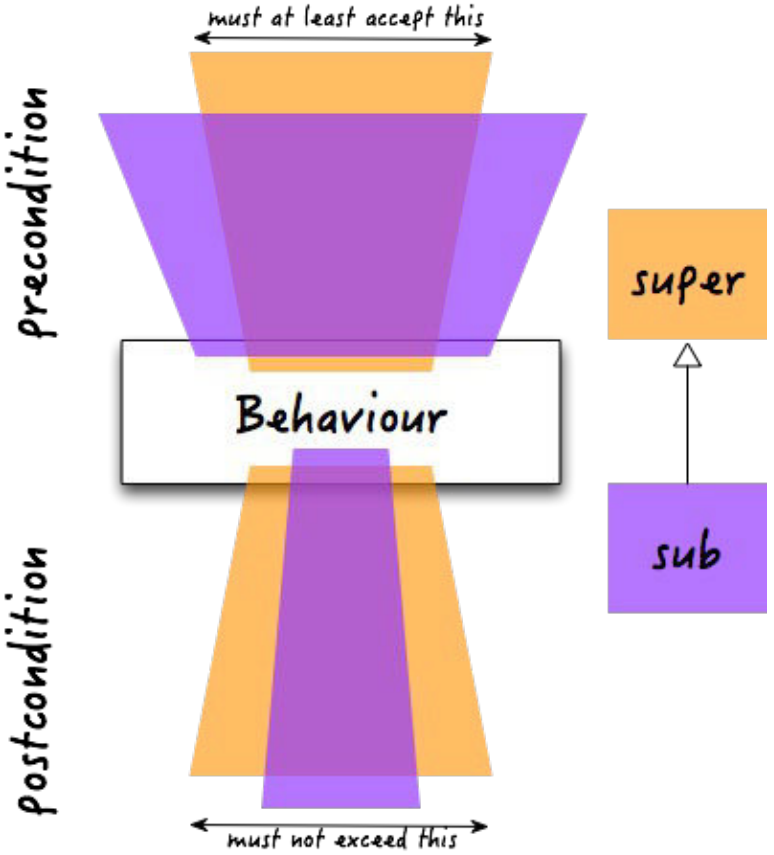
- nice in theory, but in practice deriving a class to modify its behavior is not always best thing to do
- however, it becomes increasingly important to adhere to the open/closed principle as classes mature and are more and more relied upon
- some classes are not meant to be reusable, so the Open/Closed principle doesn't apply

Liskov Substitution Principle

An object of a superclass should always be substitutable by an object of a subclass

- Subclass has same or weaker preconditions
- Subclass has same or stronger postconditions

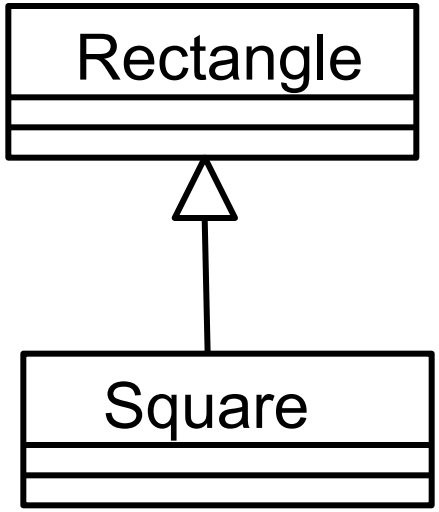
Derived methods should not *assume more* or *deliver less*



Picture from Marc Palyart's slides

Liskov Substitution Principle

Example



Reasonable to derive a square from a rectangle?

see http://www.objectmentor.com/resources/articles/lsp.pdf

LSP Example – Rectangle & Square

```
class Rectangle {

  private double fWidth, fHeight;

  public void setWidth(double w) { fWidth = w; }
  public void setHeight(double h) { fHeight = h; }
  public double getWidth() { return fWidth; }
  public double getHeight() { return fHeight; }
}
```

LSP Example – Rectangle & Square

```
class Square extends Rectangle {

  public void setWidth(double w) {
      super.setWidth(w);
      super.setHeight(w);
   }

  public void setHeight(double h) {
      super.setHeight(h);
      super.setWidth(h);
   }
}
```

LSP Example – Rectangle & Square

```
// somewhere else
public void calculate(Rectangle r) {
   r.setWidth(5);
   r.setHeight(6);
   assert(r.getWidth() * r.getHeight() == 30);
}

// somewhere else
Rectangle r = new Square(…);
calculate(r);
```

LSP Example – Rectangle & Square

- Postcondition for Rectangle setWidth(…) method

  assert((fWidth == w) && (fHeight == old.fHeight));

- Square setWidth(…) has weaker postcondition
  - does not conform to (fHeight == old.fHeight)
- Square has stronger preconditions
  - Square assumes fWidth == fHeight
- In other words
  - Derived methods assume more and deliver less.

LSP Continued

LSP shows that a design can be structurally consistent (A Square ISA Rectangle)

But behaviourally **inconsistent**

So, we must verify whether the pre and postconditions in properties will hold when a subclass is used.

“It is only when derived types are completely substitutable for their base types that functions which use those base types can be reused with impunity, and the derived types can be changed with impunity.”

Law of Demeter

(a.k.a. Principle of Least Knowledge)

- Assume as little as possible about other modules
- Restrict method calls to your immediate *friends*

  "Only talk to your friends"

Law of Demeter for classes

- Method M of object O should only call methods of:
  - O itself
  - M's parameters
  - Any object created in M
  - O's direct component objects

- "Single dot rule"
  - "a.b.method(…)" breaks LoD
  - "a.method(…)" does not

Class Activity

Which principle is violated?

a) 52 different “import …” statements at the top of a Java file

b) public final class Bird { … }

c) Point x = body.getCenterOfMassPos();
   Vec s = body.getCenterOfMassSpeed();
   Vec a = body.getCenterOfMassAcceleration();
   a = a + force * body.getMass();
   s = s + a * dt;
   x = x + s * dt;
   body.setCenterOfMassPos(x);
   body.setCenterOfMassSpeed(s);
   body.setCenterOfMassAcceleration(a);

Which principle is violated?

d) public class Road extends Highway { … }

e) rect.setHeight(52);

// Following line is not needed because setHeight updates maximum height

// rect.setMaxHeight(52);

f) public class System {

```
        public void changeCarSpeed();
        public void changeCarColor();
        public void changeHighwayMaxSpeed();
        public void updatePoliceCarPosition();
};
```

g) public class Kid extends Person {

```
        // Inherited from parent class. Does nothing because kids
        // do not know how to “Reason”
        public void Reason() {} }
```

Class Activity – Revisiting Your Design

- Examine your class diagrams and check for the design principles in your design
  - Did you violate any
  - Did you use any, which ones and why
- Be able to articulate which principles you used and why!

Modular Design Summary

- Goal of design is to manage complexity by decomposing problem into simple pieces

- Many principles/heuristics for modular design
  - Strong cohesion, loose coupling
  - Call only your friends
  - Information Hiding
    - Hide details, do not assume implementation
  - Open/Closed Principle
    - Open for extension, closed for modification
  - Liskov Substitution Principle
    - Subclass should be able to replace superclass