Modular (read: better) Design

Pragmatic Programmer:
Eliminate Effects Between Unrelated Things –
design components that are:
self-contained,
independent,
and have a single, well-defined purpose

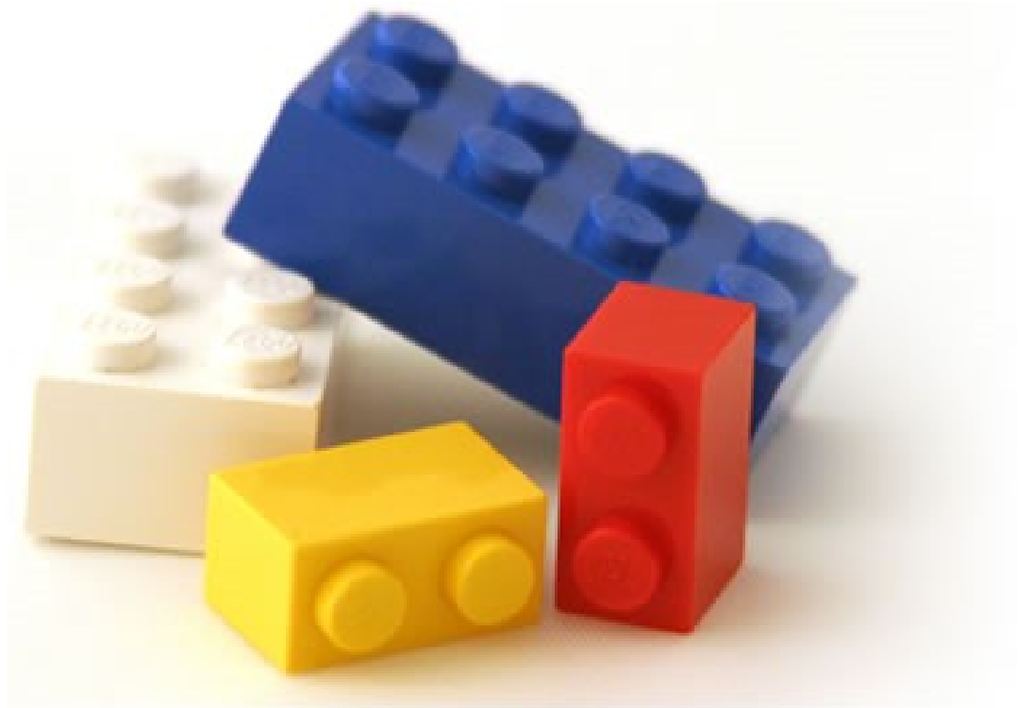
Learning Goals

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 the following design-principles appropriately: high cohesion, loose coupling, principle of least knowledge, Liskov substitution principle, information hiding, open/closed principle.

Bad Design

Software Design – Modularity



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

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

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*]

Pragmatic Programmer:
Eliminate Effects Between Unrelated Things – design components that are:
self-contained,
independent,
and have a single, well-defined purpose

Principles & Heuristics for modular Design

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

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:

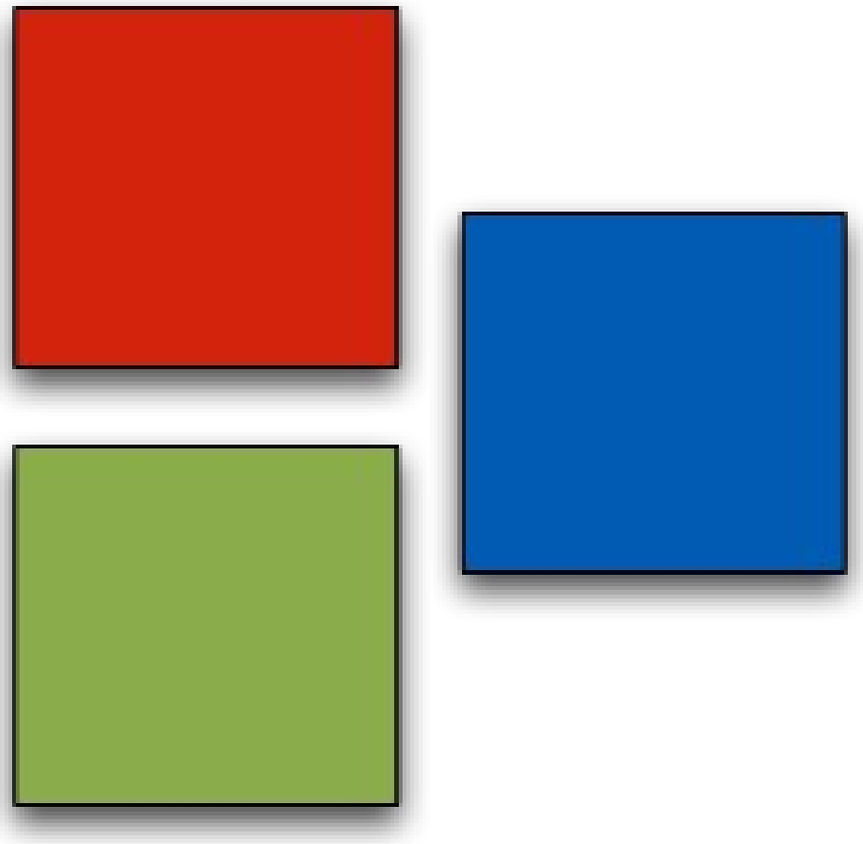
```
public class AddressBook extends LinkedList<Address>
{
    // no need to write an add method, we inherit it
}
```

High Cohesion

http://en.wikipedia.org/wiki/Cohesion_(computer_science)

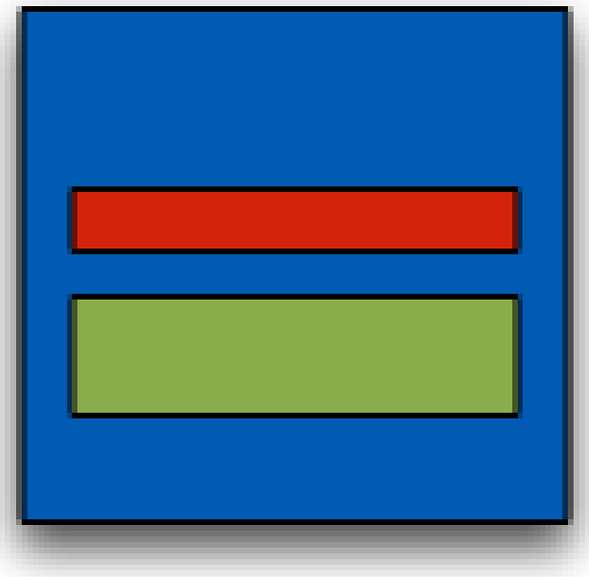
- Cohesion refers to how closely the functions in a module are related
- Modules should contain functions that logically belong together
  - Group functions that work on the same data
- Classes should have a **single responsibility.**

Cohesion (try to increase)



methods that serve the given class tend to be similar in many aspects

versus



- The functionalities embedded in a class, accessed through its methods, have little in common.
- Methods carry out many varied activities, often using coarsely-grained or unrelated sets of data.

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

Different Types of Cohesion

Coincidental cohesion (bad)

Coincidental cohesion is when parts of a module are grouped arbitrarily; the only relationship between the parts is that they have been grouped together (e.g. a “Utilities” class).

Logical cohesion (bad)

Logical cohesion is when parts of a module are grouped because they logically are categorized to do the same thing, even if they are different by nature (e.g. grouping all mouse and keyboard input handling routines).

Temporal cohesion

Temporal cohesion is when parts of a module are grouped by when they are processed - the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).

Procedural cohesion

Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file).

Communicational cohesion

Communicational cohesion is when parts of a module are grouped because they operate on the same data (e.g. a module which operates on the same record of information).

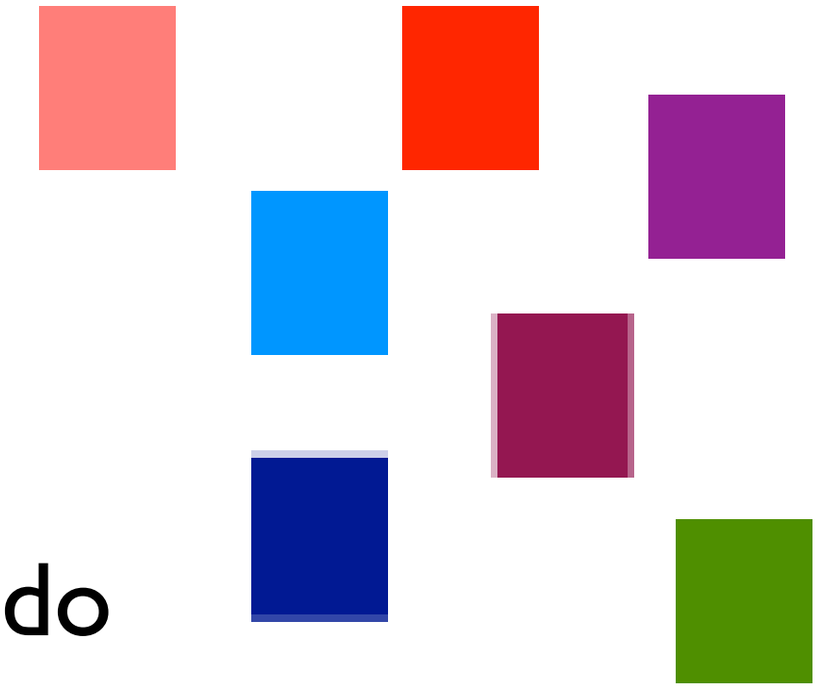
Sequential cohesion (very good)

Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data).

Functional cohesion (best)

Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module (e.g. tokenizing a string of XML).

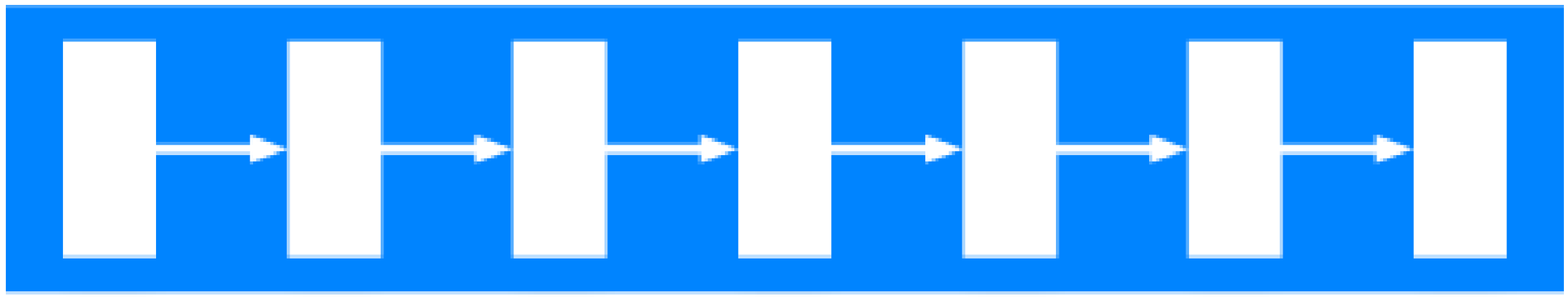
Functional Cohesion (best!)



- Functionally cohesive objects do ONE thing only.
- Good because they're easy to reuse, and understand
- Warning: functionally cohesive can proliferate and get very tiny (overly fine grained, overly numerous)

Find more detail at: http://it.toolbox.com/blogs/enterprise-solutions/design-principles-cohesion-16069

Sequential Cohesion



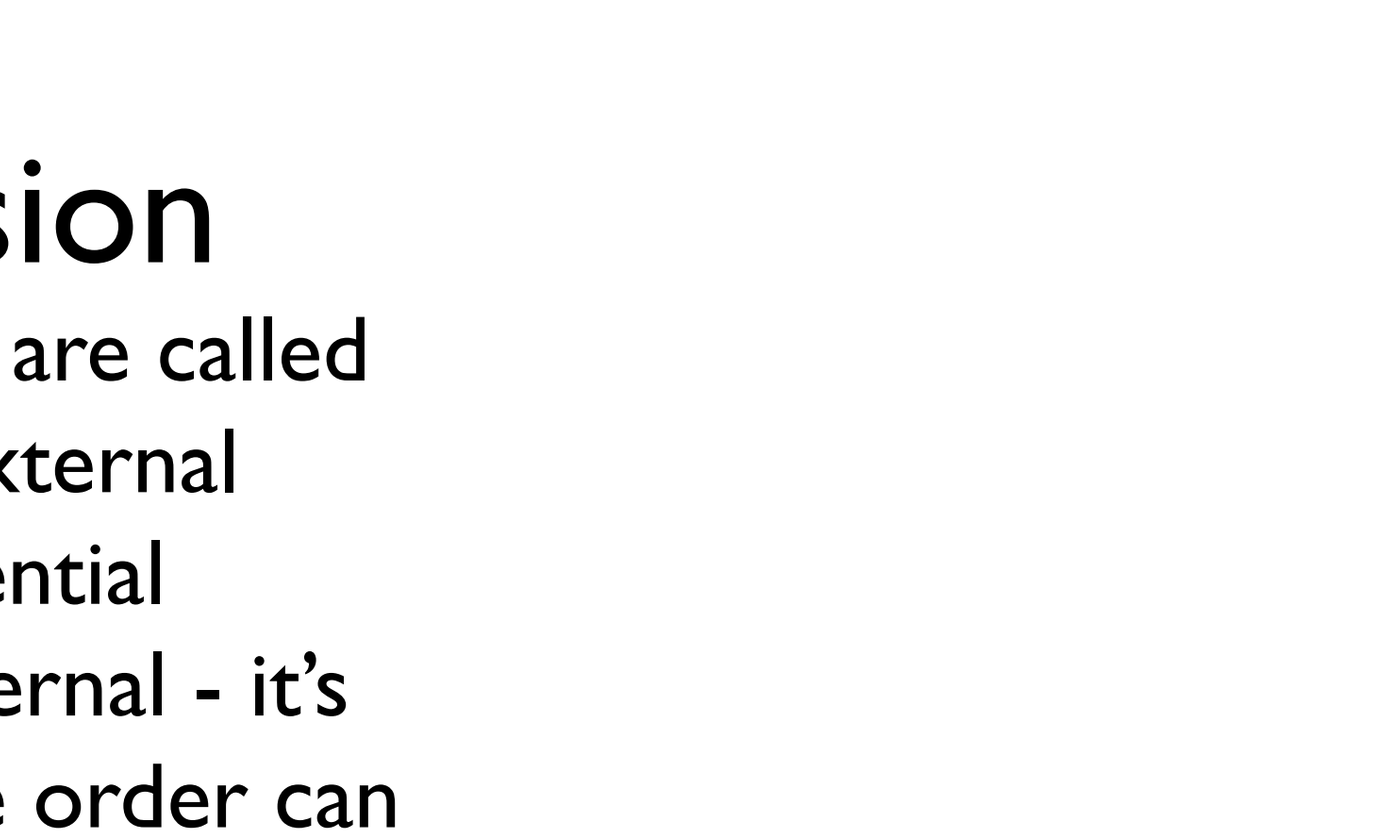
- Methods in a class chain together (pipe and filter style)
- Good because it has good coupling (class is basically independent) and is easy to maintain
- Warning: more difficult to reuse because they usually only make sense in their original implementation context.

Communicational Cohesion

- All methods perform some filtration on the same input data.
- Can usually be straightforwardly separated into functionally cohesive modules
- But still, these are easy to maintain
- May be segmented in terms of external uses (client modules only need one of the services of the communicationally cohesive module)

Procedural Cohesion

- A cluster of methods that are called one after another by an external class (different from sequential because the chain isn't internal - it's externally invoked and the order can change)
- Not as easily maintained
- Not as easily translated to other implementation contexts (low reusability)



Temporal Cohesion

- Performs activities related in time (all of initialization for instance)
- Maintenance is difficult because developers are sometimes tempted to share code between these methods, causing tangling and internal dependencies.
- Client objects might want to invoke part of the behavior of the class, but can't isolate it.

Logical Cohesion

- Methods only related because they seem to "logically" go together (grouping all I/O or device handling routines)
- These modules are usually hard to reuse in a different context
- They are hard to maintain because they often are highly internally tangled.

Coincidental Cohesion

- The worst!!
- Module is just a bucket of methods, with no higher abstraction, and no generalizable concept.
- Impossible to maintain, because of internal tangling and confusion
- Impossible to reuse out of context, because it is entirely context specific

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) {…}
    ….
}
```

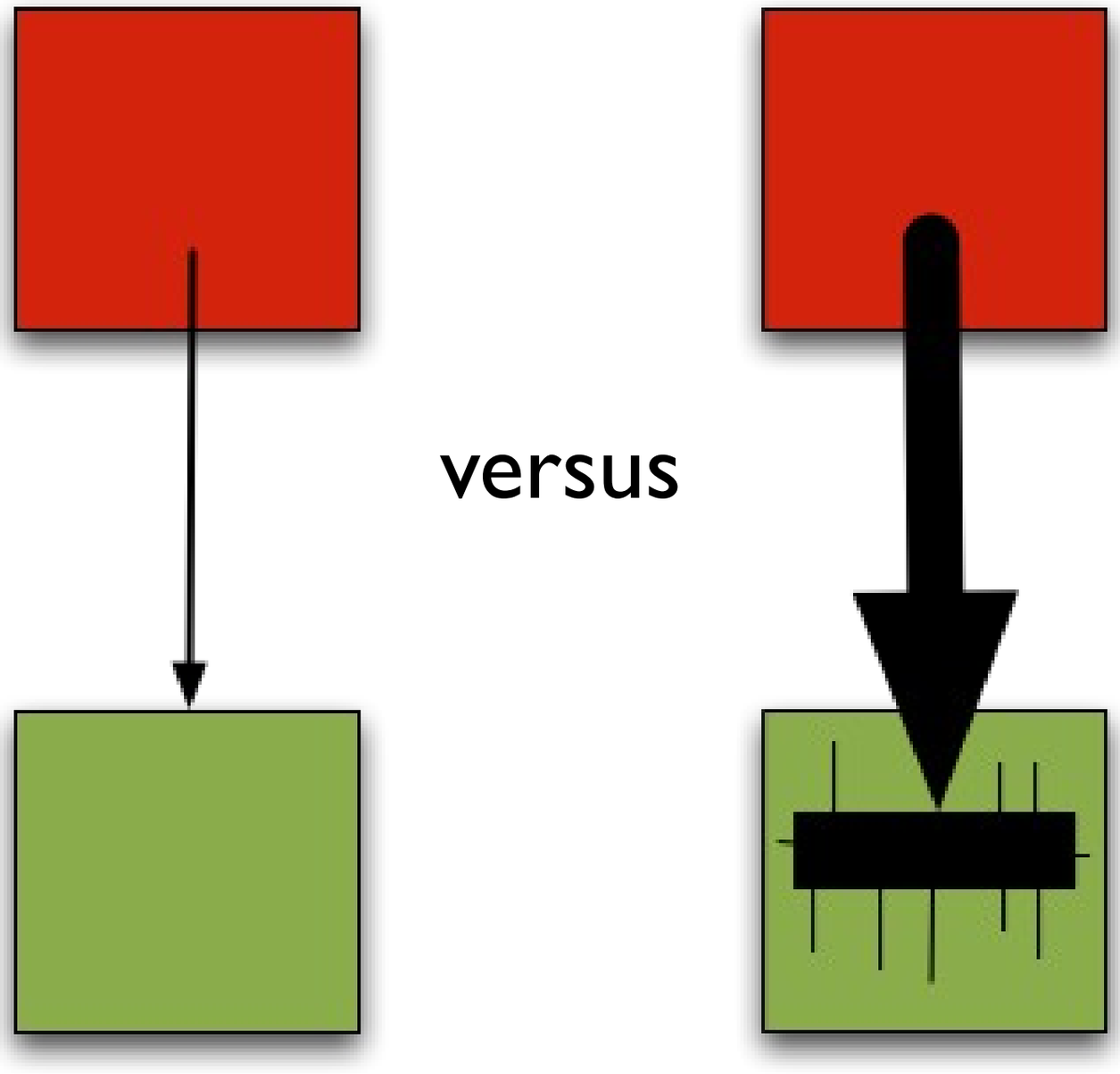
remember:
classes should be
“about” one thing

Loose Coupling

http://en.wikipedia.org/wiki/Coupling_(computer_science)

- Coupling assesses how tightly a module is related to other modules
- Goal is loose coupling:
  - modules should depend on as few other modules as possible
- Changes in modules should not impact other modules; easier to work with them separately

Coupling (try to decrease)



versus

A change in one module usually forces a ripple effect of changes in other modules.

Assembly of modules might require more effort and/or time due to the increased inter-module dependency.

A particular module might be harder to reuse and/or test because dependent modules must be included.

Different Types of Coupling

Content coupling (high)

Content coupling is when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module). Therefore changing the way the second module produces data (location, type, timing) will lead to changing the dependent module.

Common coupling

Common coupling is when two modules share the same global data (e.g., a global variable). Changing the shared resource implies changing all the modules using it.

External coupling

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.This is basically related to the communication to external tools and devices.

Control coupling

Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

Stamp coupling (Data-structured coupling)

Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g., passing a whole record to a function that only needs one field of it).

This may lead to changing the way a module reads a record because a field that the module doesn't need has been modified.

Data coupling

Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

Message coupling (low)

This is the loosest type of coupling. It can be achieved by state decentralization (as in objects) and component communication is done via parameters or message passing

No coupling

Modules do not communicate at all with one another.

Data Coupling (really really good!)

- Data is passed by parameters, and all parameters are used.
- Warning: don't pass too many data elements -- if you have a really long list of parameters, then you may want to rethink partitioning.

Stamp Coupling

- A record is passed, but only some fields are used.
- It's a loose form of coupling
- Promotes odd bundles of data

Control Coupling

- Objects influence other's internal behavior through calls
- A calls B, with the parameter "fast". This means that B chooses a different strategy.
- This requires A to know what B might do internally which might make changing B hard later.

External Coupling

- A module has an integral dependency on an externally imposed format or relies on the a 3rd party device/library/etc.
- Problematic if that 3rd party element changes!
- Solution: Use a wrapper pattern, where all reliance on the 3rd party is encapsulated. That way, if the 3rd party s/w changes, you only need to change the wrapper.

Common Coupling

- Objects rely on the same global data
- This causes tight coupling (the use of global data by one object is seen by the other)
- May be tough to debug (who changed the data?!)
- May be tough to upgrade (if I introduce a new change, will that break everyone else?)

Content Coupling

- Worst kind of coupling!!
- Object refers to another's internals (changing internal fields of another object without going through the getter/setter interface)

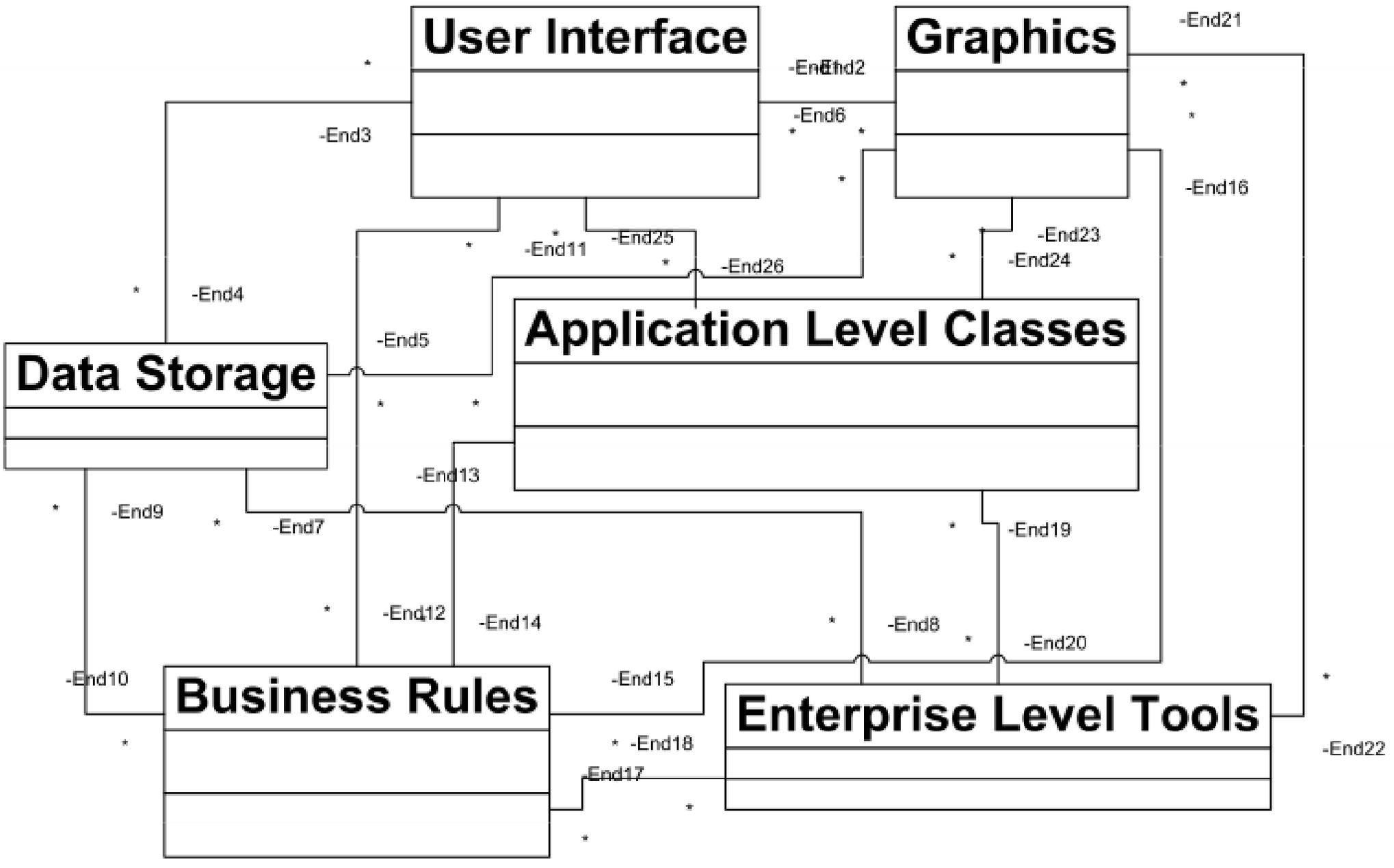
Semantic coupling: *The most insidious kind of coupling occurs when one module makes use not of some syntactic element of another module but of some semantic knowledge of another module's inner workings.*

Code Complete 2, Chapter 5, page 102 (pdf on the course webpage)

Semantic coupling is dangerous *because changing code in the used module can break code in the using module in ways that are completely* ***undetectable by the compiler****. When code like this breaks, it breaks in subtle ways that seem unrelated to the change made in the used module, which turns debugging into a Sisyphean task.*

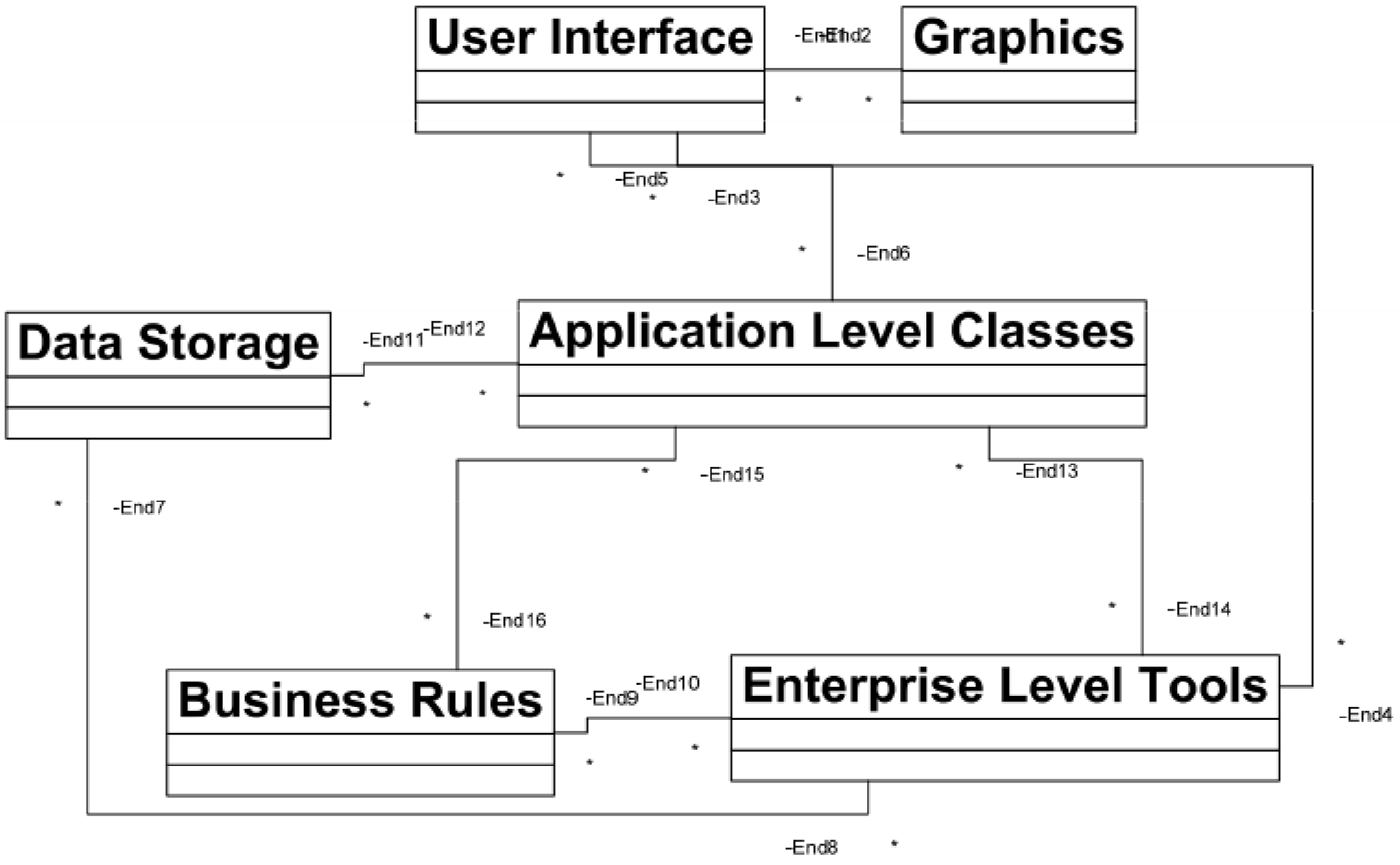
The point of loose coupling *is that an effective module provides an additional level of abstraction—****once you write it, you can take it for granted****. It reduces overall program complexity and allows you to focus on one thing at a time. If using a module requires you to focus on more than one thing at once—knowledge of its internal workings, modification to global data, uncertain functionality—the abstractive power is lost and the module's ability to help manage complexity is reduced or eliminated.*

Tightly or loosely coupled?



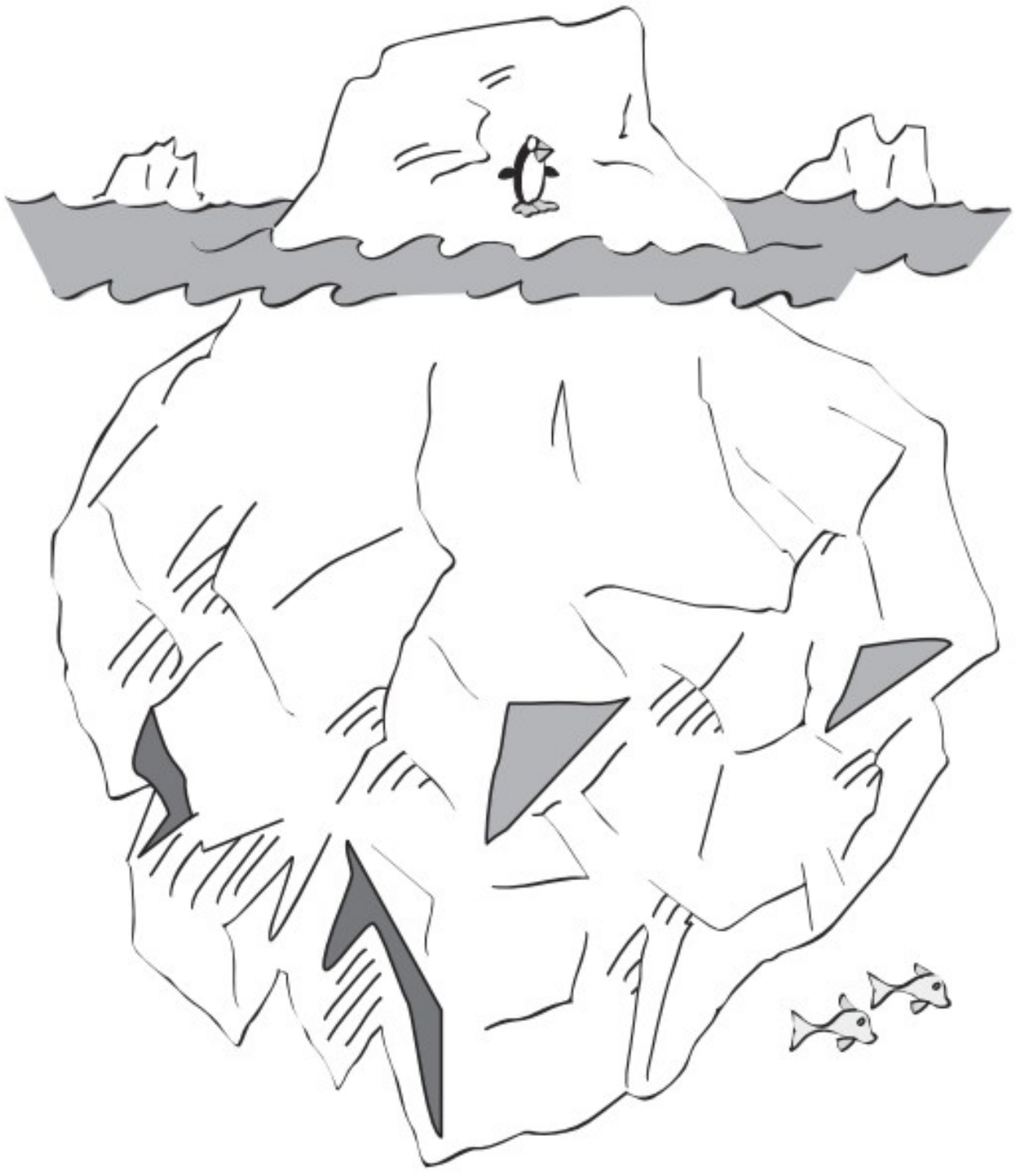
from Alverson (UW)

Tightly or loosely coupled?



from Alverson (UW)

Information Hiding

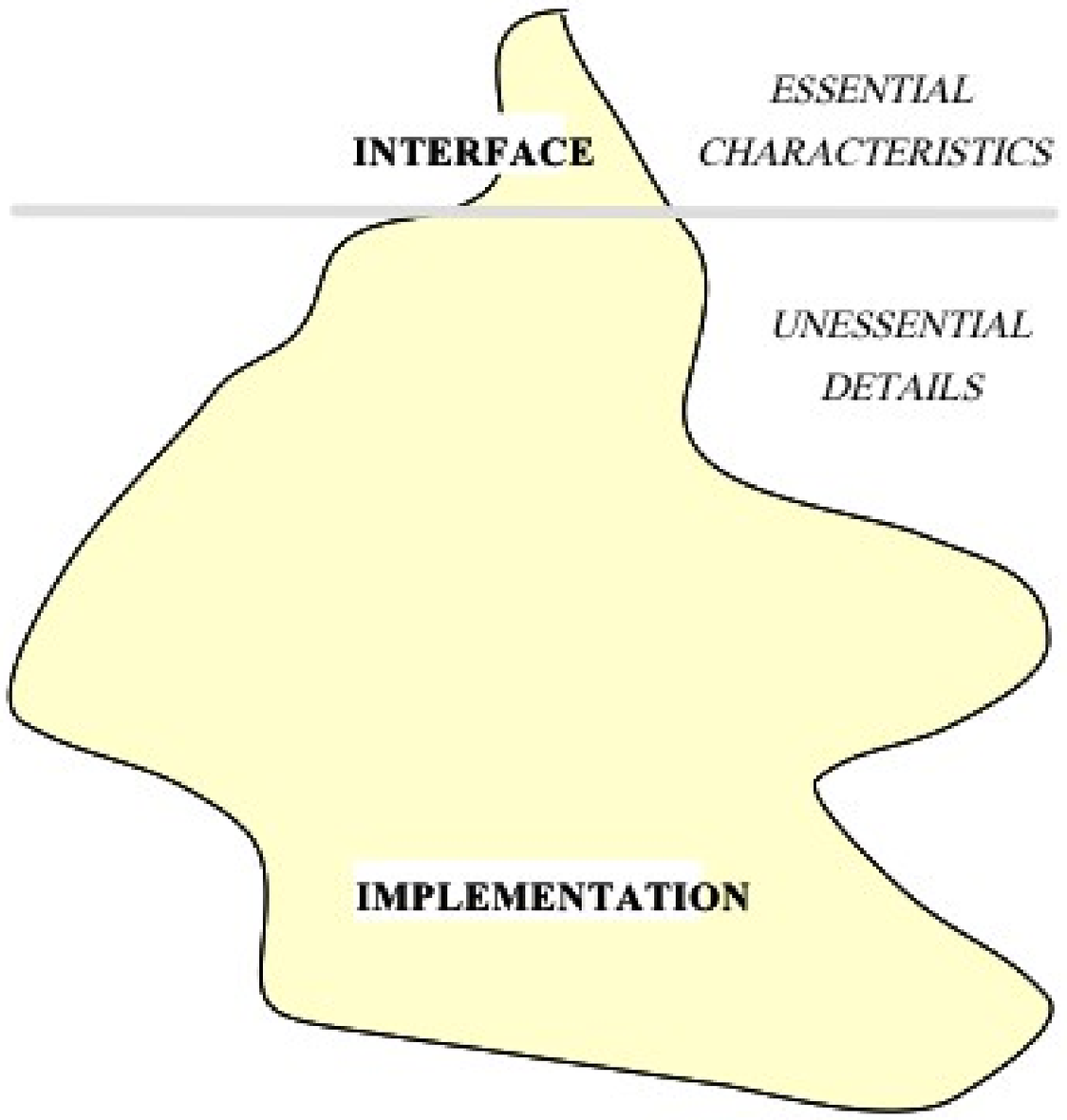


A good class is a lot like an iceberg: seven-eights is under water, and you can see only the one-eight that's above the surface.

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



http://www.fatagnus.com/program-to-an-interface-not-an-implementation/

Information Hiding: Example

- 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!

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

Open/Closed Principle

A class must be **closed** for internal change

But must be **open** for *extensions*

When designing classes, do not plan for brand new functionality to be added by modifying the core of the class.
Instead, design your class so that extensions can be made in a modular way, to provide new functionality by leveraging the power of the inheritance facilities of the language, or through pre-accommodated addition of methods.

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…
}
```

This class assumes developers will modify the drawSquare and drawCircle methods directly to change their behaviour. This results in what looks like unplanned change!

```
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  }}
```

this class has made specialising the shape draw method much more straightforward (also indicating that developers see this potential change coming!)

Liskov Substitution Principle

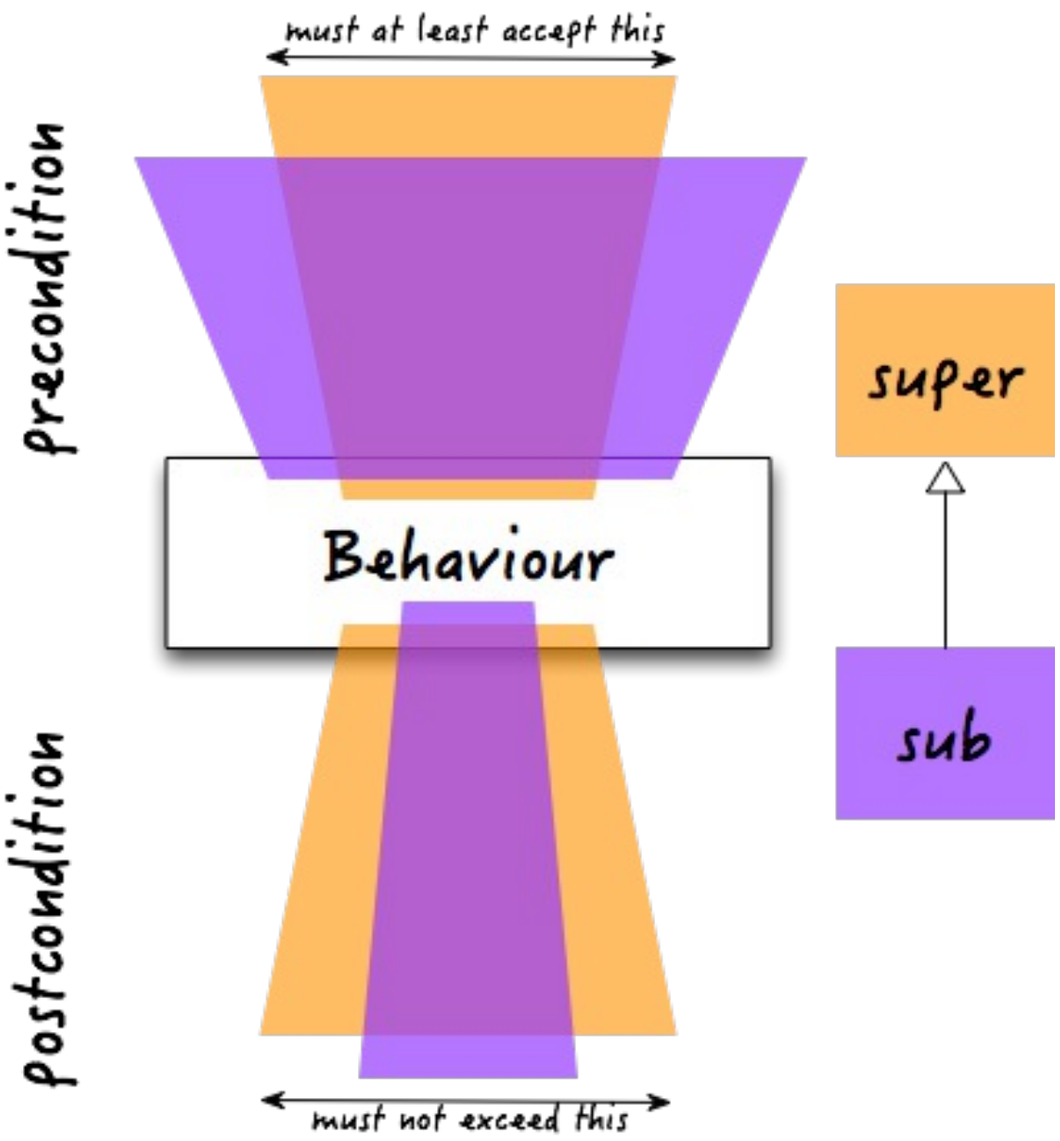
Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

[Barbara Liskov and Jeanette Wing, A Behavioral Notion of Subtyping, ACM Transactions on Programming Languages and Systems, Vol 16, No 6. November 1994, Pages 1811-1841.]

if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program

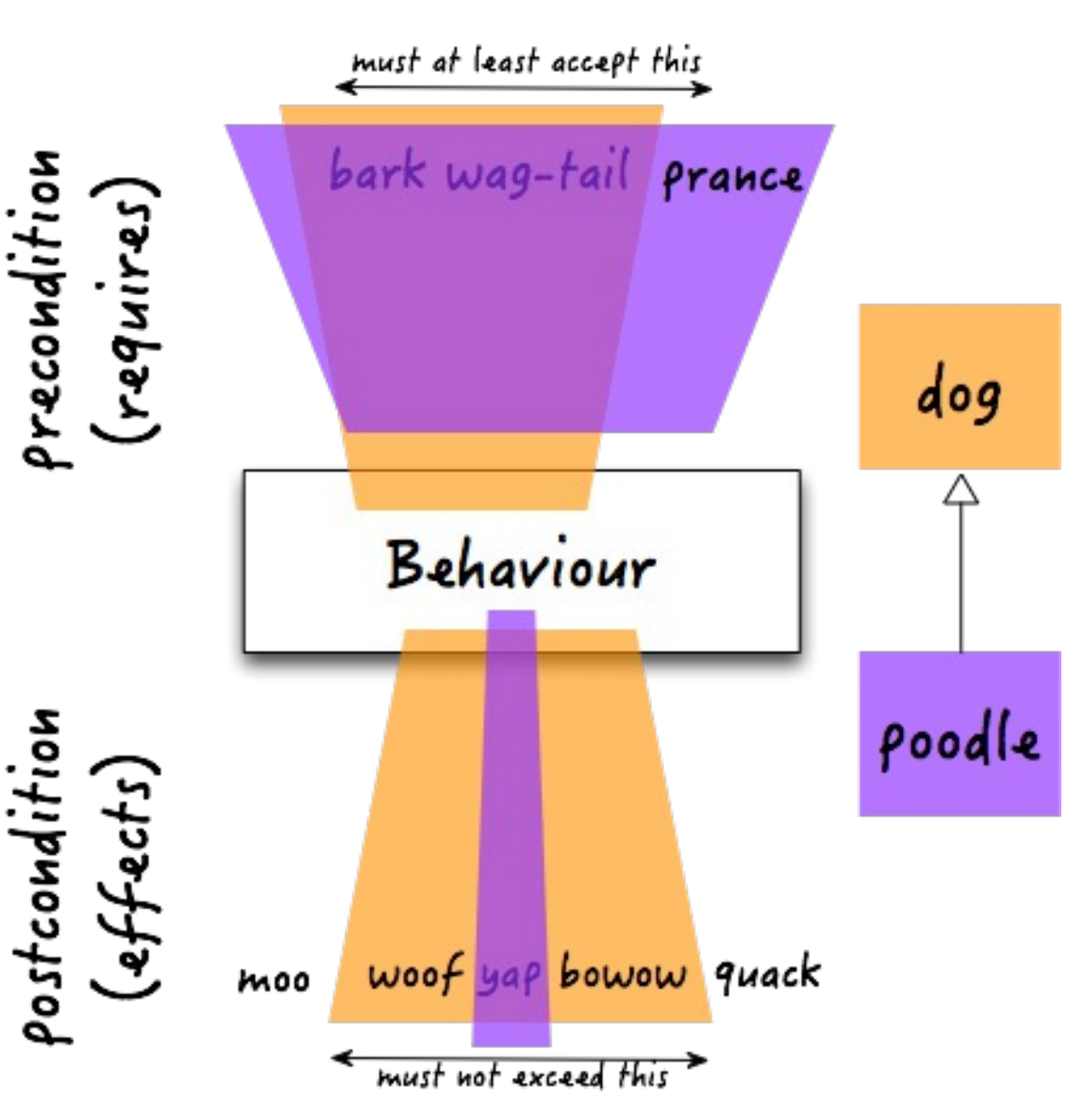
http://en.wikipedia.org/wiki/Liskov_substitution_principle

Liskov Substitution Principle

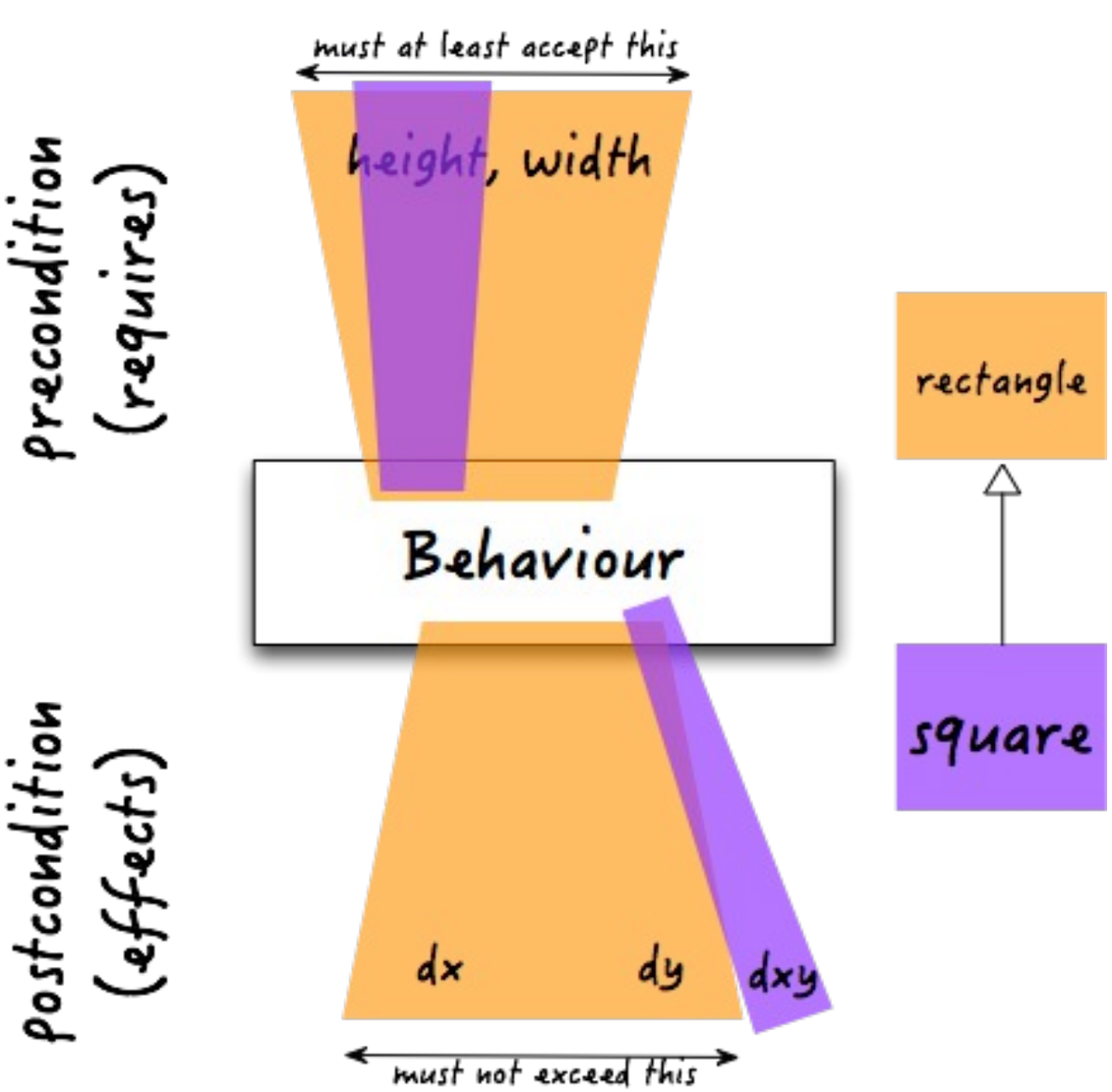


- An object of a superclass should always be substitutable by an object of a subclass
  - Subclass has same or weaker preconditions
  - Subclass has same or stronger postconditions
- Derived methods should not *assume more* or *deliver less*

Liskov Substitution Principle



GOOD



BAD

LSP Example

```
class Rectangle {
  protected int m_width;
  protected int m_height;

  public void setWidth(int width){
    m_width = width;
  }
  public void setHeight(int height){
    m_height = height;
  }
  public int getWidth(){
    return m_width;
  }
  public int getHeight(){
    return m_height;
  }
  public int getArea(){
    return m_width * m_height;
  }
}
```

```
class Square extends Rectangle {
  public void setWidth(int width){
    m_width = width;
    m_height = width;
  }
  public void setHeight(int height){
    m_width = height;
    m_height = height;
  }
}
```

LETS TEST IT

```
public static void main (String args[])
{
  // Can come from a factory ...
  Rectangle r = new Square();
  r.setWidth(5);
  r.setHeight(10);
  System.out.println(r.getArea());
}
```

What's the result of the output here ?

Fixing violations of LSP

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."

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