EECE 310 – Software Engineering

Software Design:
Modular Design

Project Proposal Due

- 18 Oct @ 22:00
- One page
  - Description of the selected open source project
  - Link to forked GitHub repository
  - Contribution plans (eg. finding/resolving bugs, adding feature, cleaning up code, writing documentation, creating a model, etc)

Forking GitHub Repos

- One member of the group needs to fork the open source project under their github name/account.
- That same person needs to add the other members of the group as collaborators to this repository.
- The forked project has to remain public.
- For more information regarding GitHub Forking https://help.github.com/articles/fork-a-repo/

Think about the following

- Need to develop a *native* mobile app

  Requirement: Has to work on

  iOS, Android, Windows Phone
- What is your approach?

Ideal World

Would it not be nice to

- Design the app once,
- Generate code for different platforms?

Model-driven engineering

- Model-driven engineering (MDE) is an approach to software development where **models** rather than programs are the principal outputs of the development process.
- The programs that execute on a hardware/software platform are then **generated** automatically from the models.

Types of model

A computation independent model (CIM)

- These model the important **domain** abstractions used in a system. CIMs are sometimes called domain models.

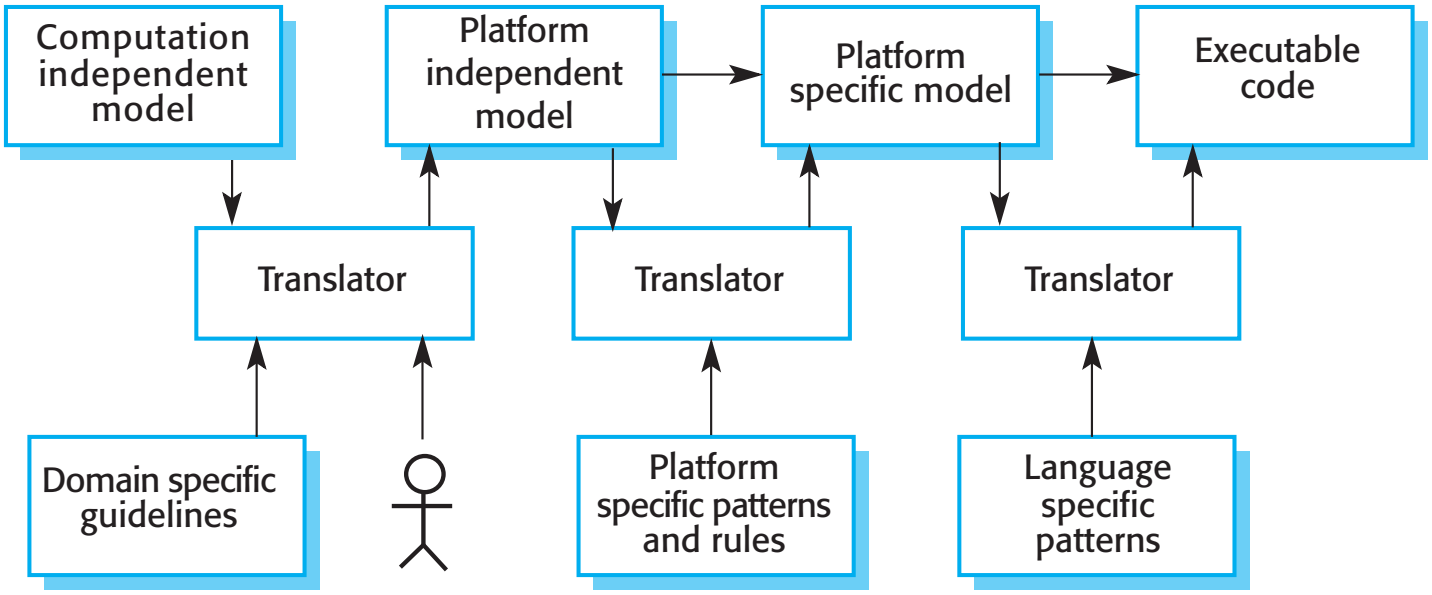
A platform independent model (PIM)

- These model the **operation** of the system **without reference** to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.

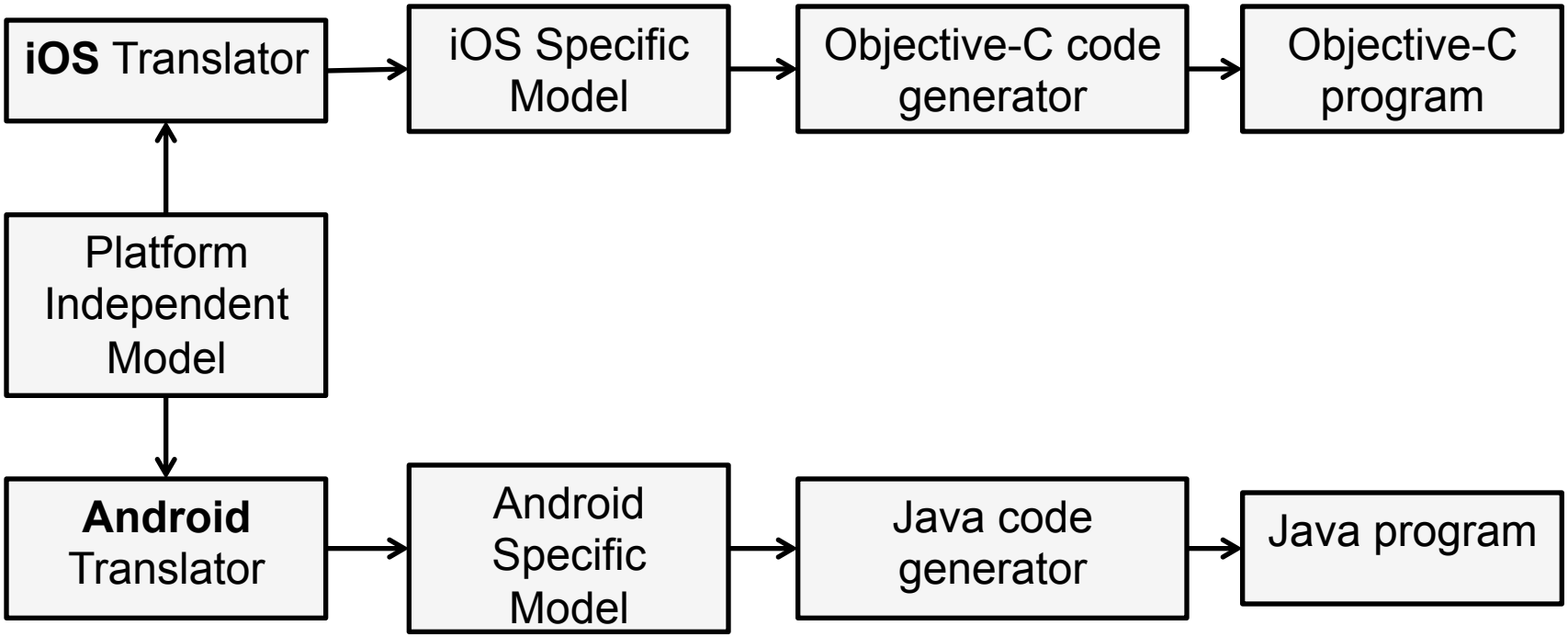
Platform specific models (PSM)

- These are **transformations** of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

MDA transformations



Multiple platform-specific models



Executable UML

- The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible.
- This is possible using a subset of UML 2, called Executable UML or xUML.

Usage of model-driven engineering

Model-driven engineering is still at an early stage of development.

Pros

- Allows systems to be considered at higher levels of abstraction
- Generating code automatically means that it is cheaper to adapt systems to new platforms (Example, iPhone, Android, Windows Phone)

Cons

- Models for abstraction are not necessarily right for implementation.
- Savings from generating code may be outweighed by the costs of developing translators for new platforms.

"It's not the strongest who survive nor the most intelligent, but the ones most **adaptable** to change"

-- Charles Darwin

Modular Design

Learning Goals

- Recognize and critique a bad design and provide concrete suggestions of how to improve it
- Explain the goal of a good modular design and why it is important
- Apply the following design-principles appropriately: high cohesion, loose coupling, Liskov substitution principle, information hiding, open/closed principle, dependency inversion

Goals of Software Design

Any ideas?

Recap: Goals of Software Design

- Specify component interfaces
  - Facilitates component testing and team communication
- Describe component functionality
- Identify opportunities for systematic reuse

Many 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

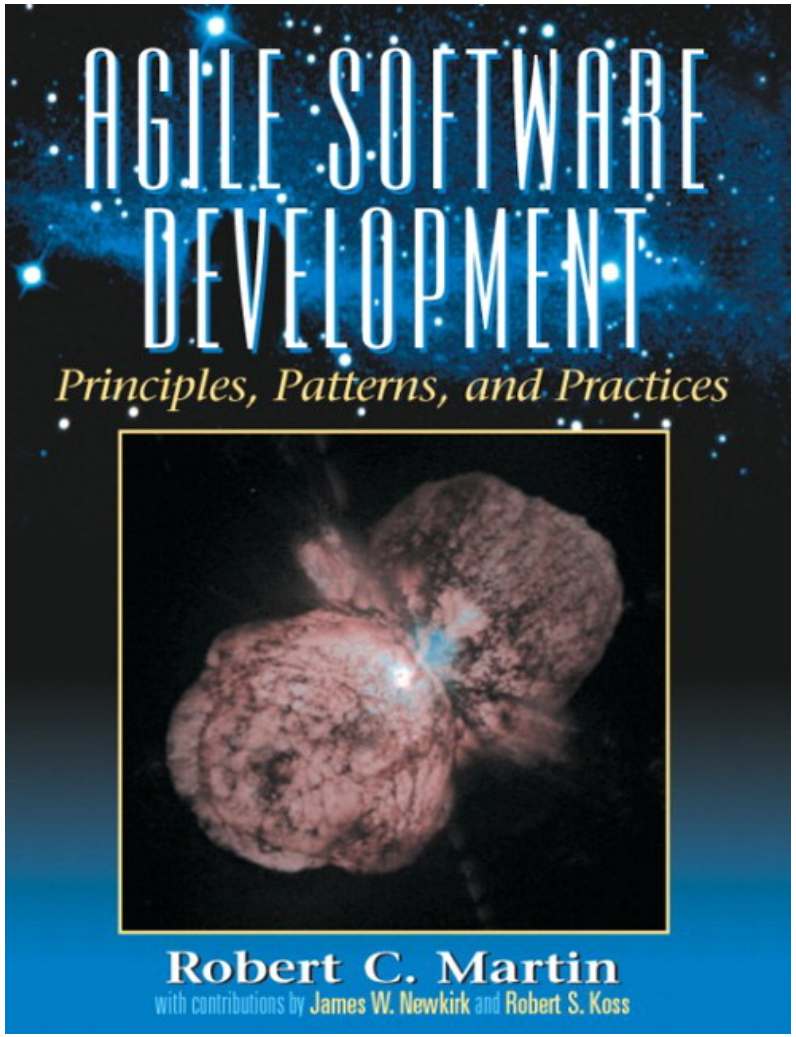
Source: [*Hunt, Thomas, "The Pragmatic Programmer: From Journeyman to Master", Addison-Wesley, 1999*]

- 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

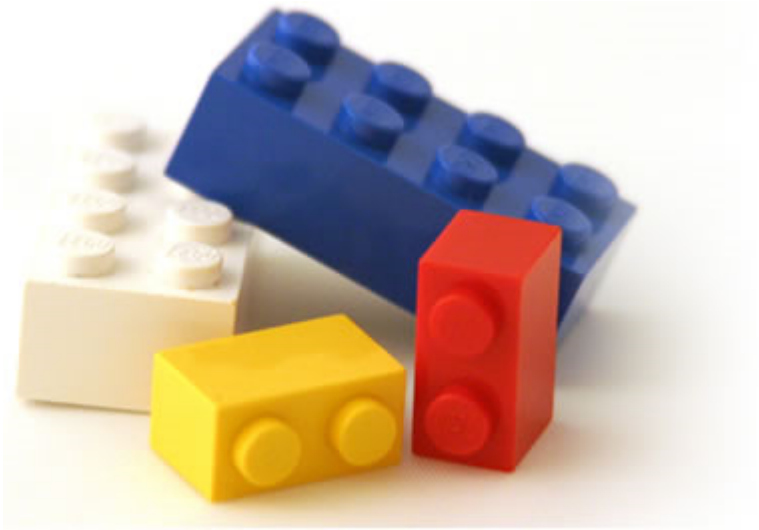
Source: [*Lieberherr,Holland, "Assuring Good Style for Object-Oriented Programs", IEEE Software, September 1989*]

- Law of Demeter

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


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

Software Design – Modularity



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

Why Modularity?

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

Question

Which of these two designs is better?

1)
```
public class AddressBook {

    private LinkedList<Address> theAddresses;
    public void add (Address a){
     theAddresses.add(a);
    }
}
```

2)
```
public class AddressBook extends LinkedList<Address>
{
   // no need for the add method, we inherit it
}
```

Design Smells

What makes a design smell bad?

- ***Rigidity***
  It is *hard to change* because every change affects too many other parts of the system.
- ***Fragility***
  When you make a change, unexpected parts of the system break.
- ***Immobility***
  It is hard to reuse in another application because it cannot be disentangled from the current application.

The design principles discussed in the following are all aimed at preventing “bad” design.

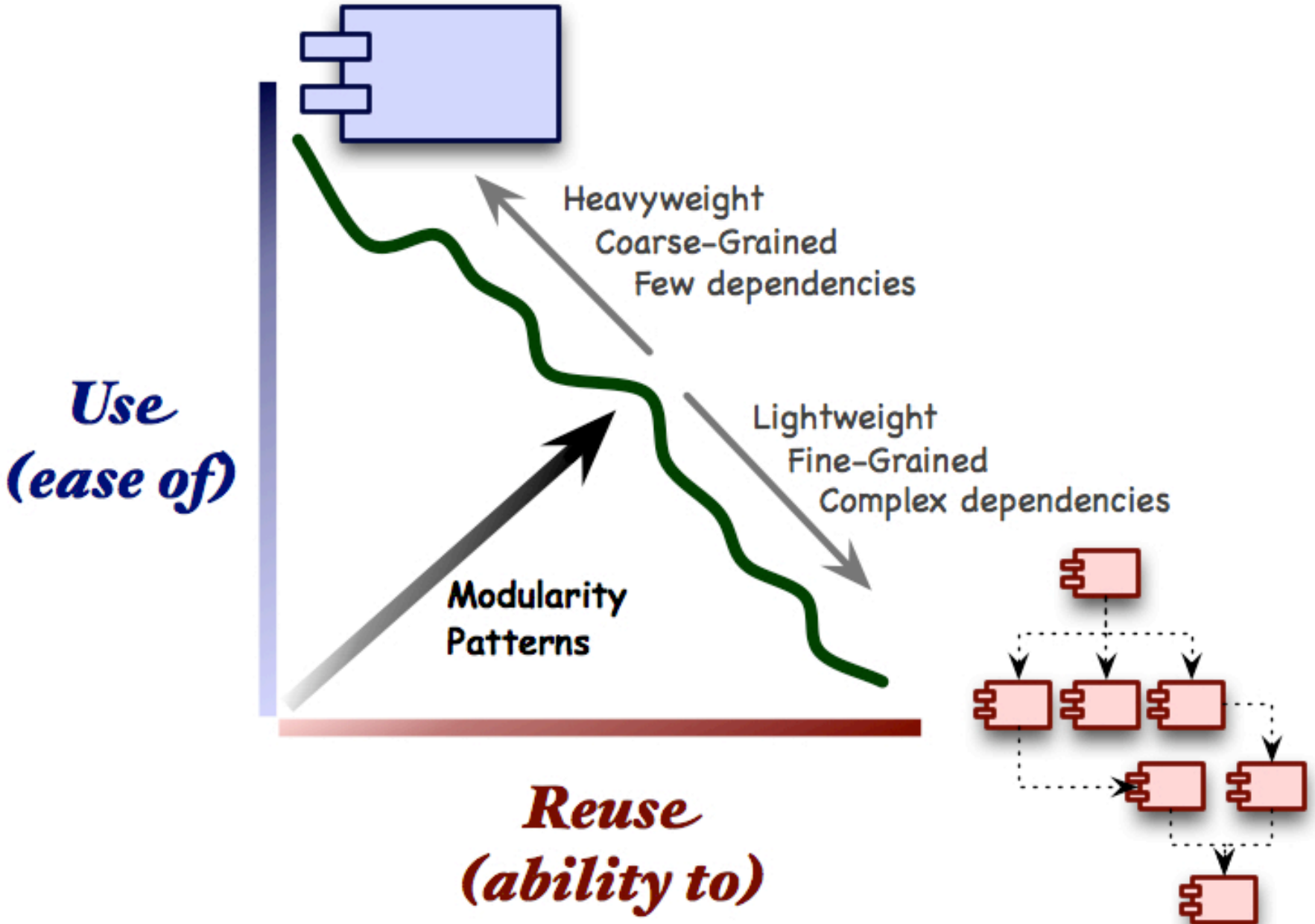
Principles & Heuristics for modular Design

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

Abstraction and Decomposition

- Decomposition handles complexity by splitting large problems into smaller problems
- This "divide and conquer" concept is common to all life-cycle processes and design techniques (across different engineering disciplines)
- Example: A large and complex production web service

  - Divide in smaller *modules: client, server, database, communication, authentication, etc*


Heavyweight
Coarse-Grained
Few dependencies
Lightweight
Fine-Grained
Complex dependencies
Modularity
Patterns
Use
(ease of)
Reuse
(ability to)

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.**

High or low cohesion?

```
public class EmailMessage {

    …

    public void login(String user, String passw)
{…}

    public void sendMessage() {…}

    public void setSubject(String subj) {…}

    public void setSender(Sender sender) {…}

   …

}
```

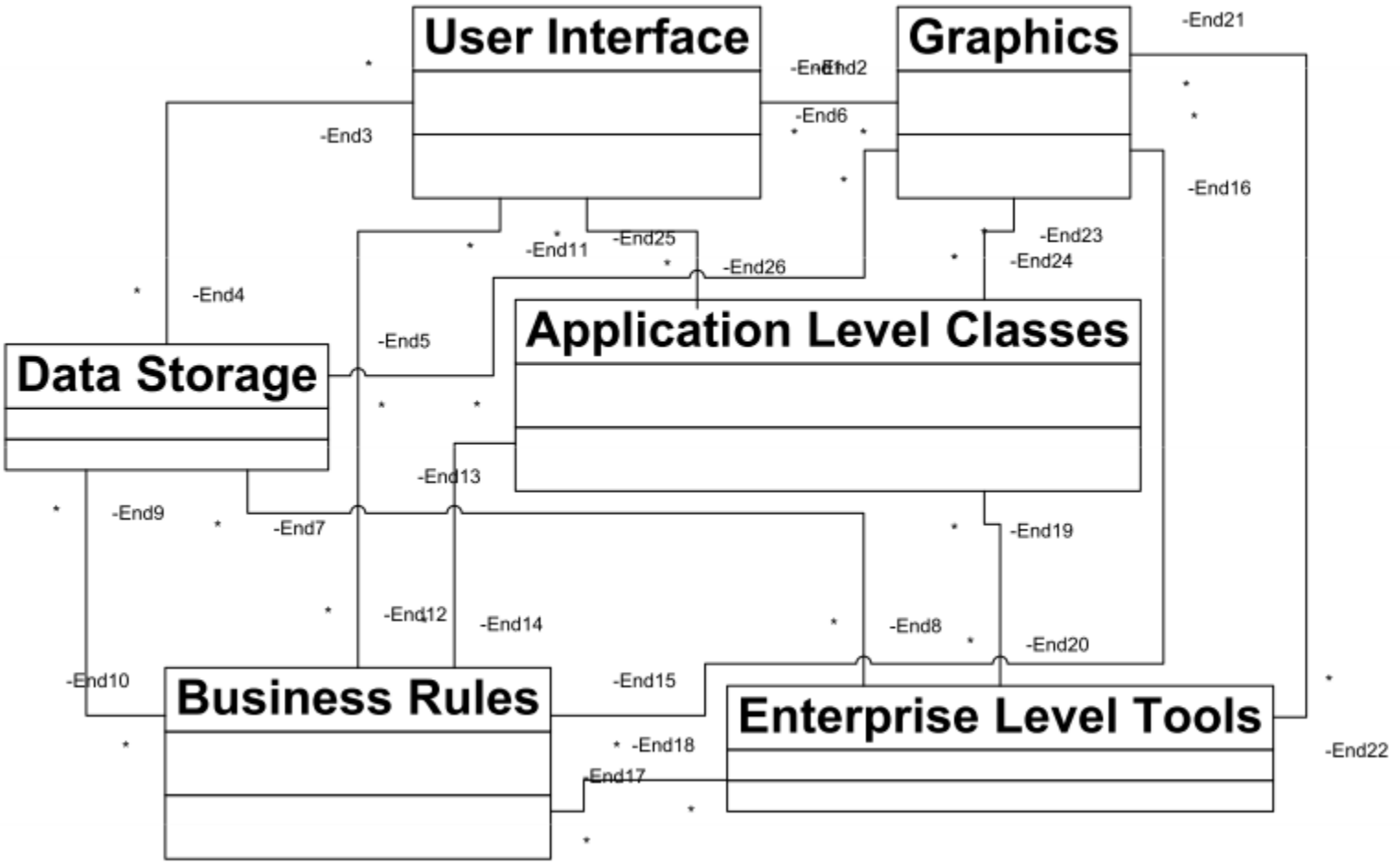
Low cohesion: login should not be there

```
public class EmailMessage {
    …
    public void sendMessage() {…}
    public void setSubject(String subj) {…}
    public void setSender(Sender sender) {…}
    public void saveDraft() {…}
    ….
}
```

Loose Coupling

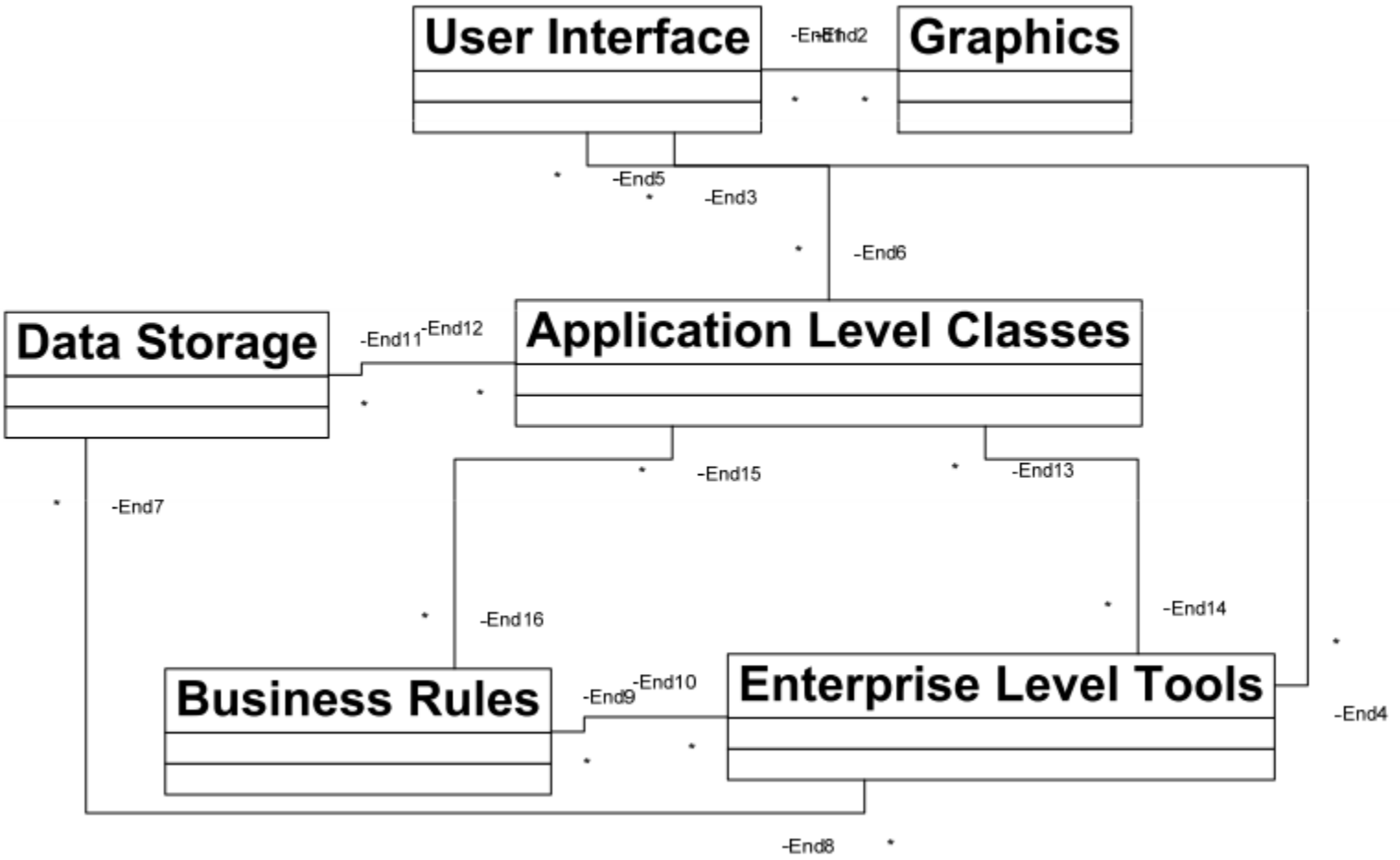
- Coupling assesses how tightly a module is related to other modules
- Goal is loose coupling:
  - modules should depend on as few other modules as possible
- Changes in modules should not impact other modules; easier to work with them separately

Tightly or loosely coupled?



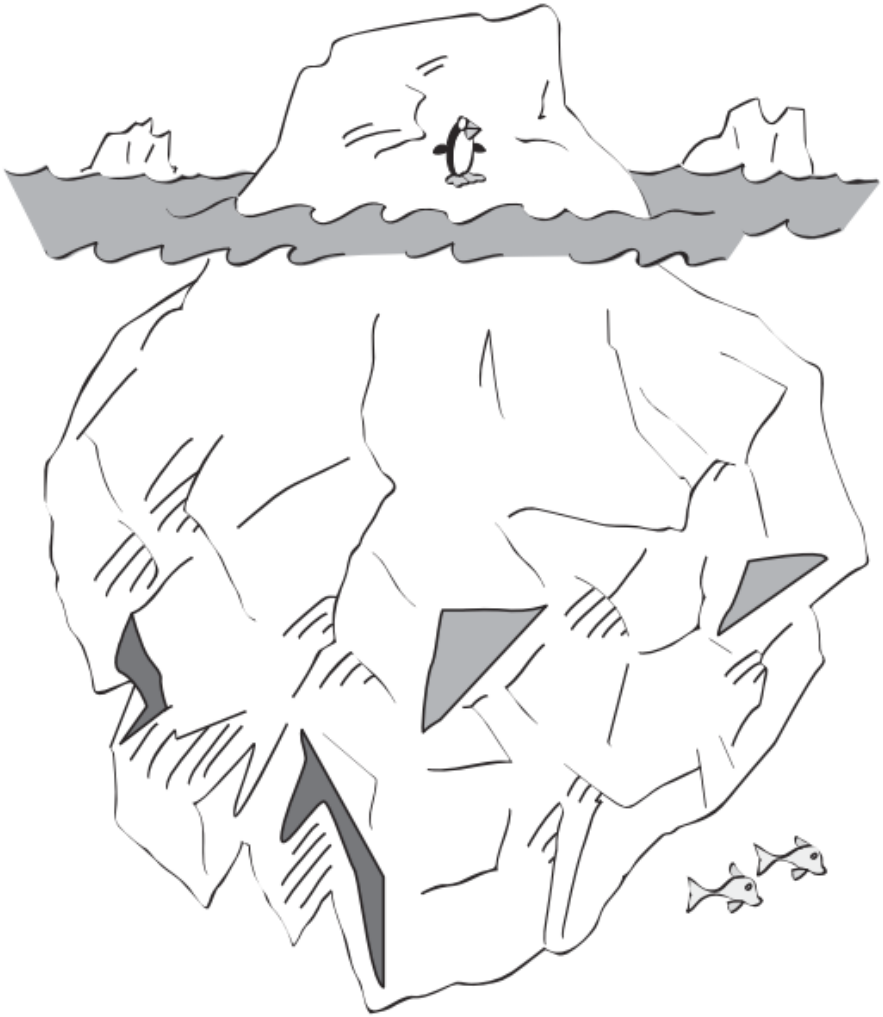
from Alverson (UW)

Tightly or loosely coupled?



from Alverson (UW)

Information Hiding

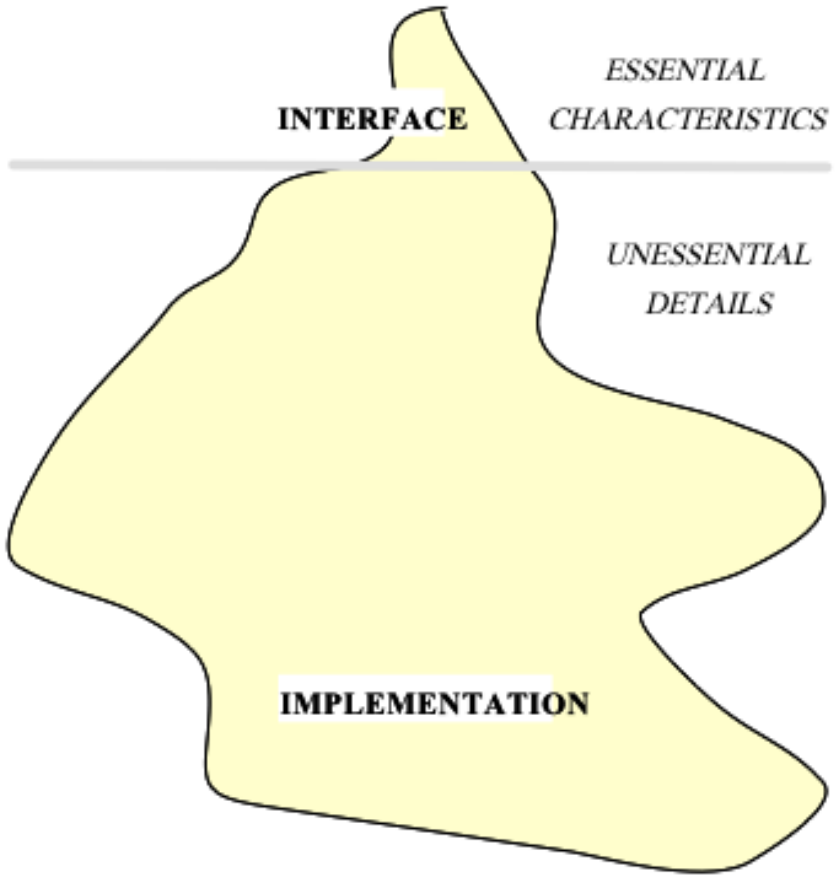


A good class is a lot like an iceberg: 7/8th is under water, and you can see only the 1/8th that's above the surface.

from "Code Complete" 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 (JPacman Game)
- 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 Exercise

```
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) {..}
  private void drawCircle(Circle square) {..}
}
```

Open/Closed Solution

```
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

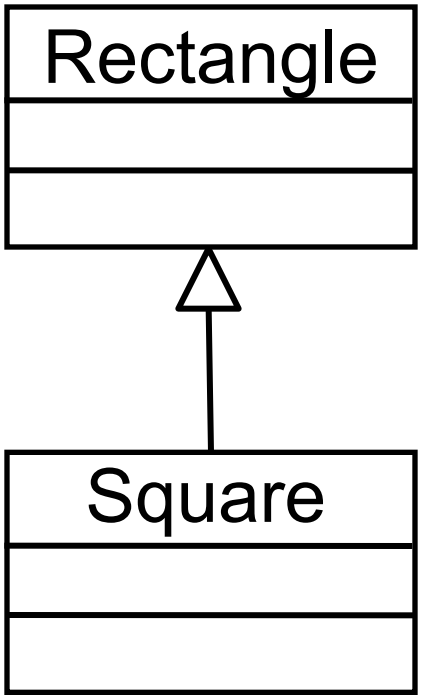
- When code is still immature, classes may need change frequently
- However, it becomes increasingly important to adhere to the open/closed principle as classes mature and are more and more relied upon **by others**

Liskov Substitution Principle (LSP)

- An object of a superclass should always be substitutable by an object of a subclass
  - Without affecting the behavior of the system
  - Correctness is preserved
    - Preconditions cannot be strengthened in a subtype.
    - Postconditions cannot be weakened in a subtype.
    - Invariants of the supertype must be preserved in a subtype.

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 in the program
   Rectangle r = new Square();
   r.setWidth(5);
   r.setHeight(6);
```

Does this violate the LSP principle?

LSP Example – Rectangle & Square

```
// somewhere else in the program
   Rectangle r = new Square();
   r.setWidth(5);
   r.setHeight(6);
   assert(r.getWidth() * r.getHeight() == 30);
```

FAIL because (6 * 6 = 36)

Fixing violations of LSP

- LSP shows that a design can be structurally consistent
  - A Square IS A Rectangle
- But behaviorally **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."

LSP Example – Rectangle & Square

- Postcondition for Rectangle setWidth(w) method

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

- Square setWidth(w) 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.

Dependency Inversion

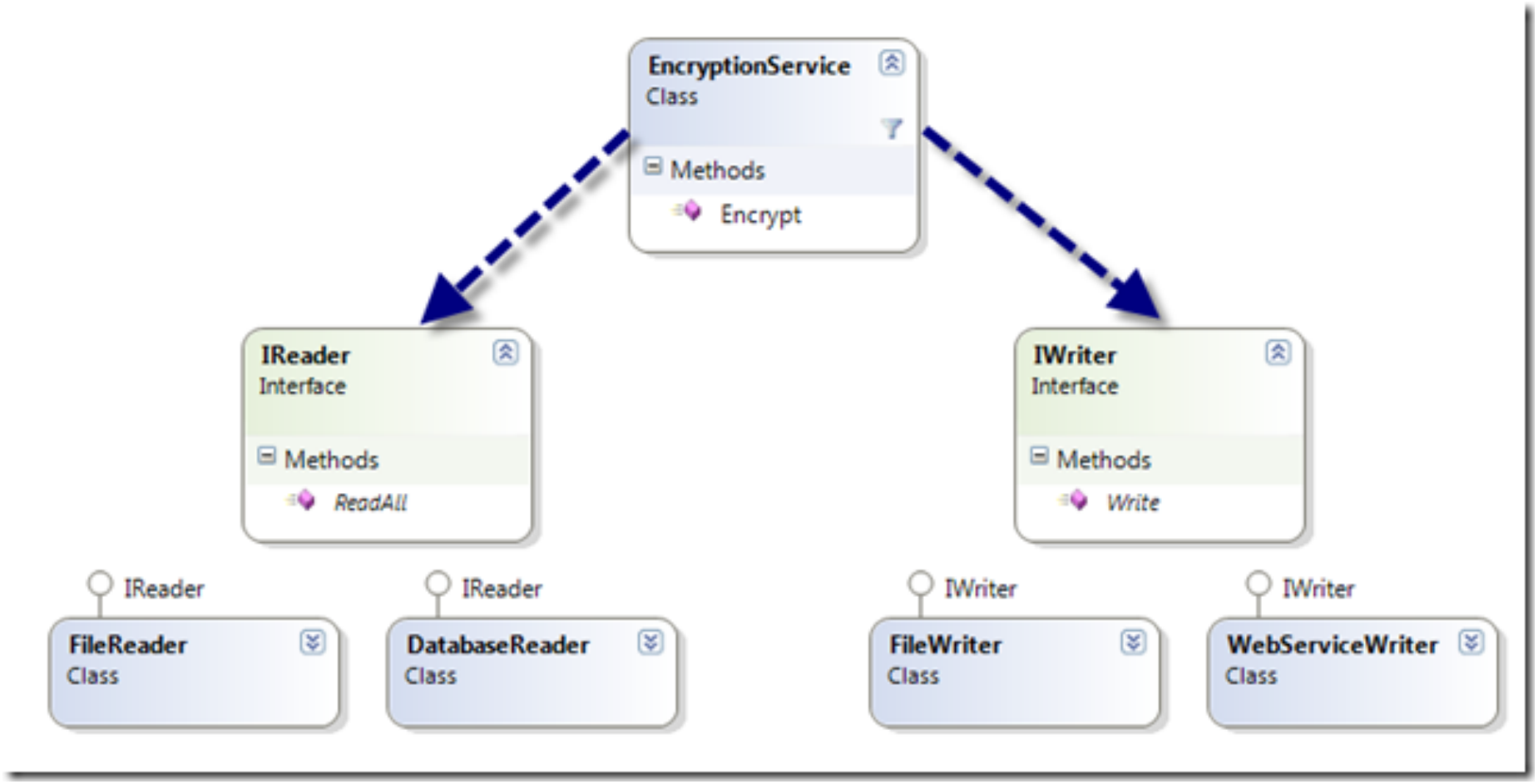
Depend upon abstractions

- Use **interfaces** not implementations
- You could always easily replace the implementation

1. High level components should not depend upon low level components. Instead, both should depend on abstractions.
2. Abstractions should not depend upon details. Details should depend upon the abstractions.

```
//in class EncriptionService
public void Encrypt(string src, string tar)  {
  // Read content
  FileReader r = new ReaderFile(src);
  byte[] encryptedContent = DoEncryption(r.getContent());
  // write encrypted content
  FileWriter w = new FileWriter(tar);
  w.setContent(encryptedContent);
}
```

Dependency Inversed



```
//in class EncriptionService
public void Encrypt(string src, string tar)  {
  // Read content
  IReader r = new ReaderFile(src);
  byte[] encryptedContent = DoEncryption(r.getContent());
  // write encrypted content
  IWriter w = new FileWriter(tar);
  w.setContent(encryptedContent);
}

Class FileReader implements IReader {
  …
}
```

Which principle is violated in these?

a) 52 different "import …" statements at the top of a Java file that are used

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?

a) 52 different "import …" statements at the top of a Java file that are used **(loose coupling)**

b) public final class Bird { … } **(Open/Closed principle)**

c) Point x = body.getCenterOfMassPos(); **(Information Hiding)**

```
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);
```

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

Remember…

Always code as if the guy maintaining your code would be a violent psychopath and he knows where you live.