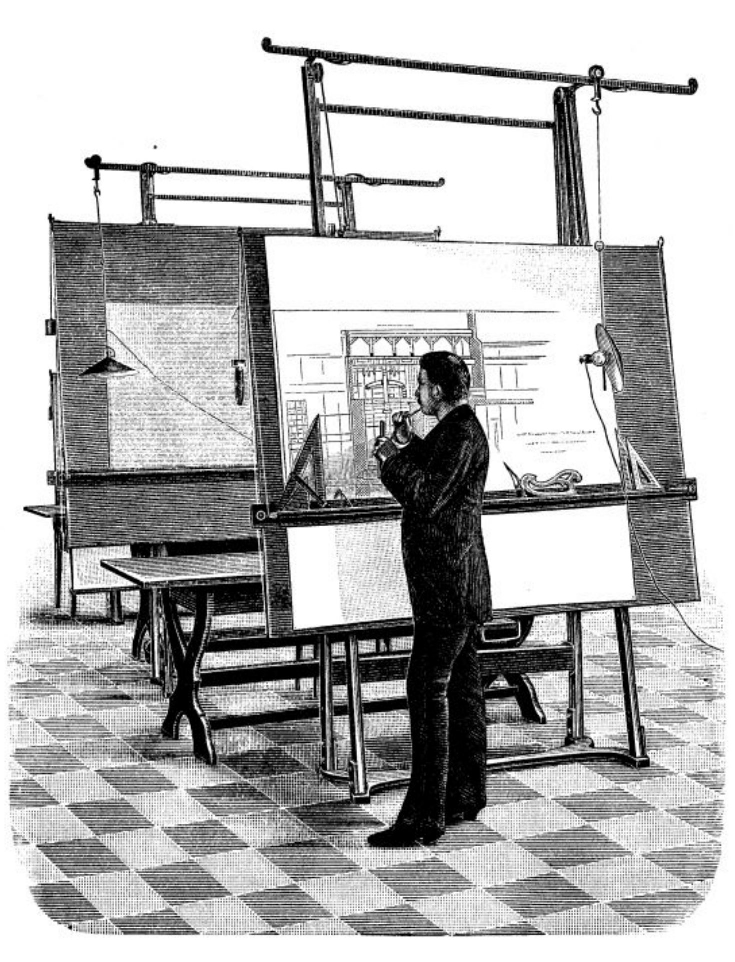
# Software Design



*"You can use an eraser on the drafting table or a sledgehammer on the construction site."*

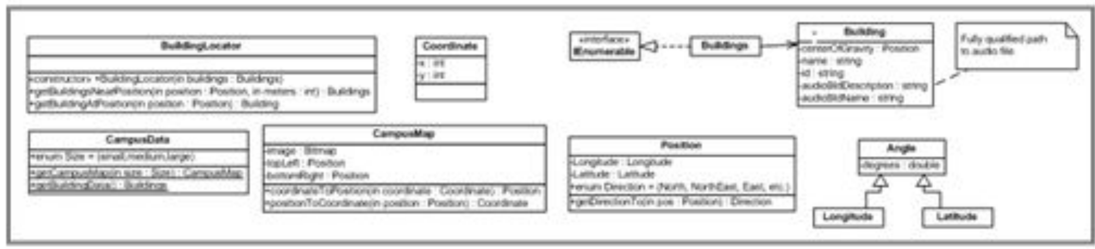
--Frank Lloyd Wright

# Organization of Topics

- Introduction
  - Opportunities for Design
  - What is Software Design?
  - The Importance of Managing Complexity
- What makes design challenging
  - Designs are Abstractions of the *Anticipated* Implementation
  - Design is a Wicked Problem
- Design Concepts
  - Design is a Universal Activity
  - Design Occurs at Different Levels
  - Characteristics of Software Design
  - The Benefits of Good Design
- A Generic Design Process
- Design Methods
- Design Techniques and Tactics
  - Step-Wise Refinement
  - Look for Real-World Objects
  - Noun-Verb Analysis
  - CRC Cards
  - Test-Driven Development
- Core Design Principles and Heuristics
  - Modularity
  - Information hiding
  - Encapsulation
  - Abstraction
  - Coupling and Cohesion
- Supporting Design Principles and Heuristics
  - Don't Repeat Yourself (DRY)
  - Principle of Least Astonishment/Surprise (POLA)
  - Single Responsibility Principle (SRP)
  - Open-Closed Principle (OCP)
  - Interface segregation principle (ISP)
  - Dependency Inversion Principle (DIP)
  - Separation of Concerns
  - Consider Brute Force
- Object-Oriented Design Principles and Heuristics
  - Generalization and Specialization
  - Principle of Substitution (aka Liskov substitution principle)
  - Favor Composition Over Inheritance
  - Law of Demeter

# Opportunities for Design

Product Design
(User Interface)



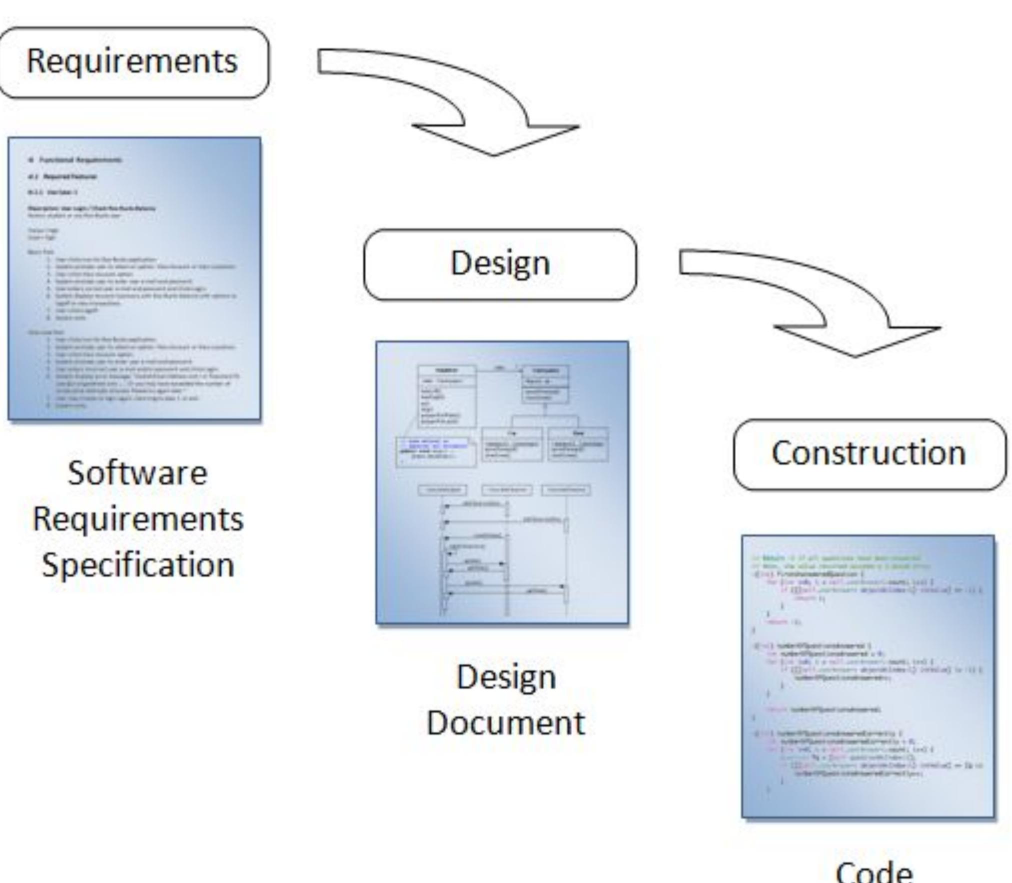

Software Design
(Internal System
Structure)

```
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    // replace the tool bar
    [self.navigationController setToolbarHidden:NO animated:YES];
    modalViewForInstructionsInProcess = NO;
}
```

Code

# What is Software Design?

- Design bridges that gap between knowing what is needed (software requirements specification) to entering the code that makes it work (the construction phase).
- Design is both a verb and a noun.

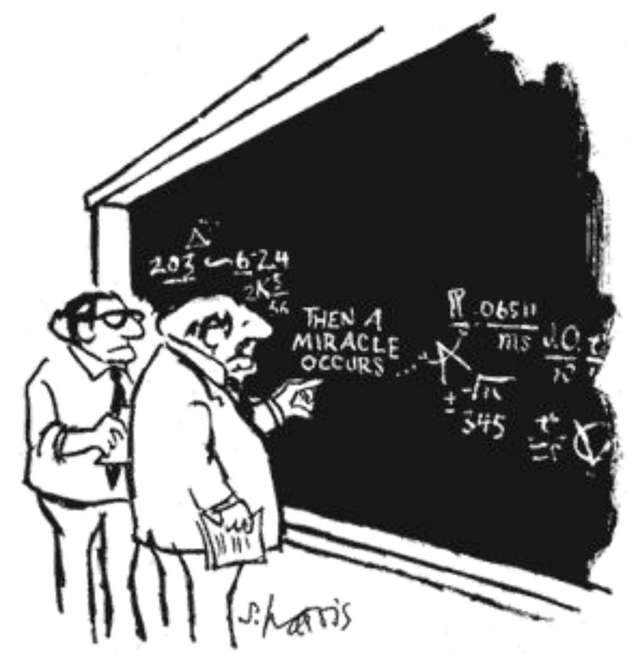


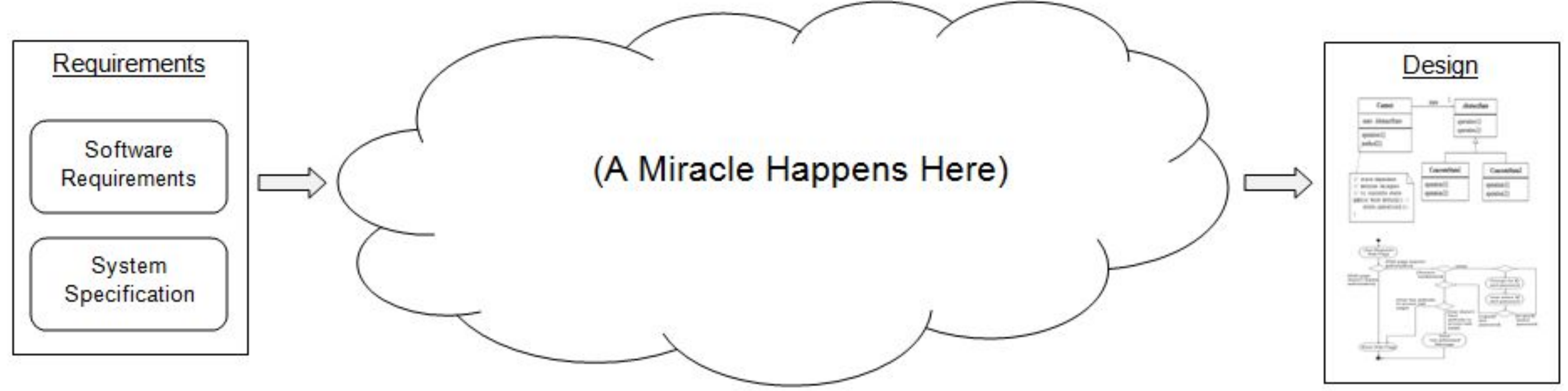
# What is Software Design? [cont]

- During the design phase, software engineers apply their knowledge of the problem domain and implementation technologies in order to translate system specifications into plans for the technical implementation of the software.
- The resulting design expresses the overall structure and organization of the planned implementation. It captures the essence of the solution independent of any implementation language.

There is a famous cartoon showing two professors at a chalkboard examining a proof that includes the step "then a miracle occurs".

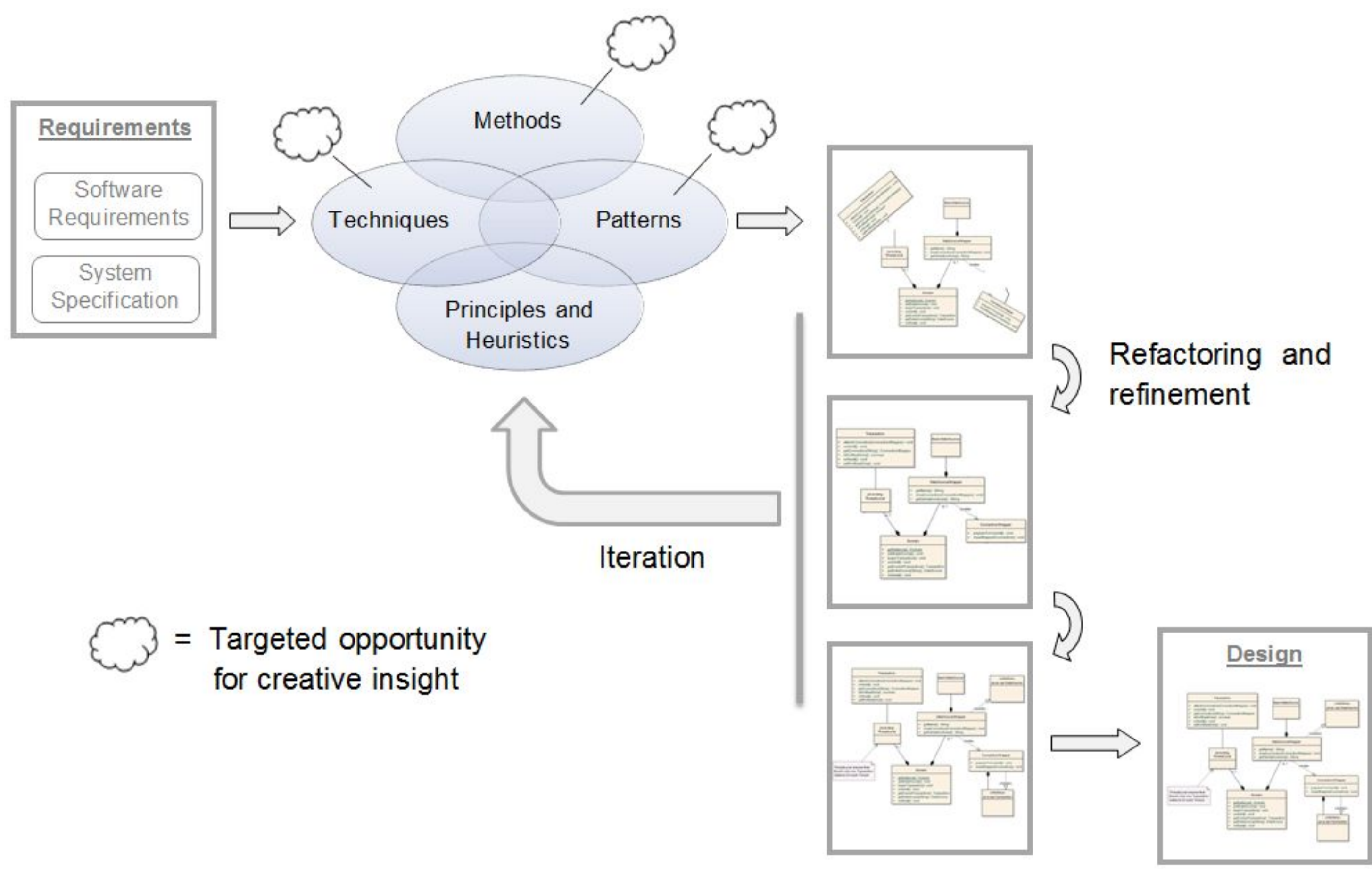
At times it seems like this is the most that can be hoped for during software design.



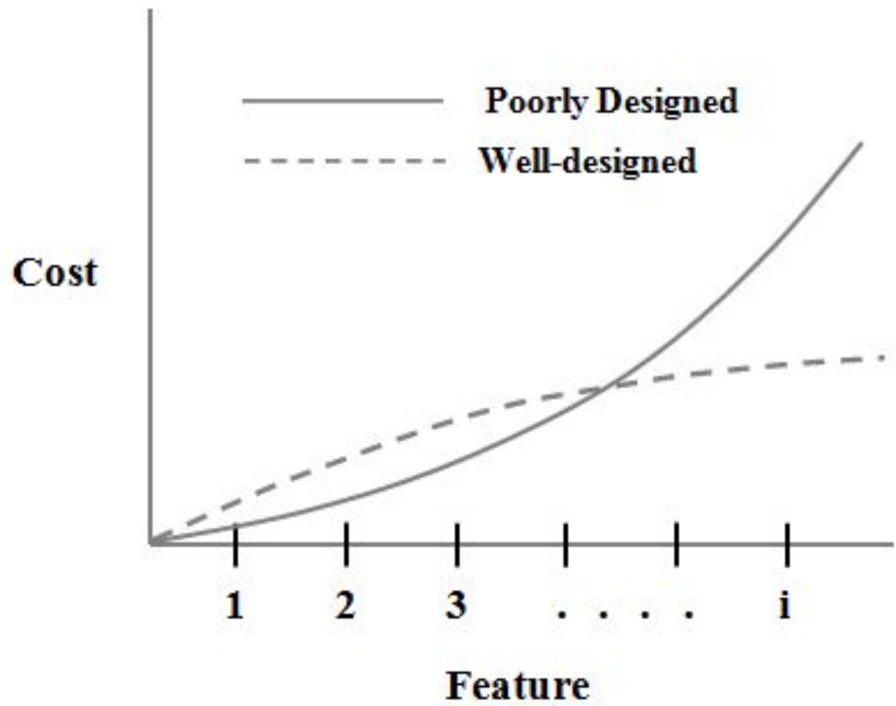
Relying on miracles or exceptional ingenuity isn't a reliable (predictable or repeatable) way of arriving at a design.

The design process can be made more systematic and predictable through the application of methods, techniques and patterns, all applied according to principles and heuristics.



# Importance of Managing Complexity

- Poorly designed programs are difficult to understand and modify.
- The larger the program, the more pronounced are the consequences of poor design.




Cost of adding the $i^{th}$ feature to a well-designed and poorly designed program

# Two Types of Complexity in Software

- To better understand how good design can minimize technical complexity, it's helpful to distinguish between two major types of complexity in software:
  - Essential complexities – complexities that are inherent in the problem.
  - Accidental/incidental complexities – complexities that are artifacts of the solution.
- The total amount of complexity in a software solution is:

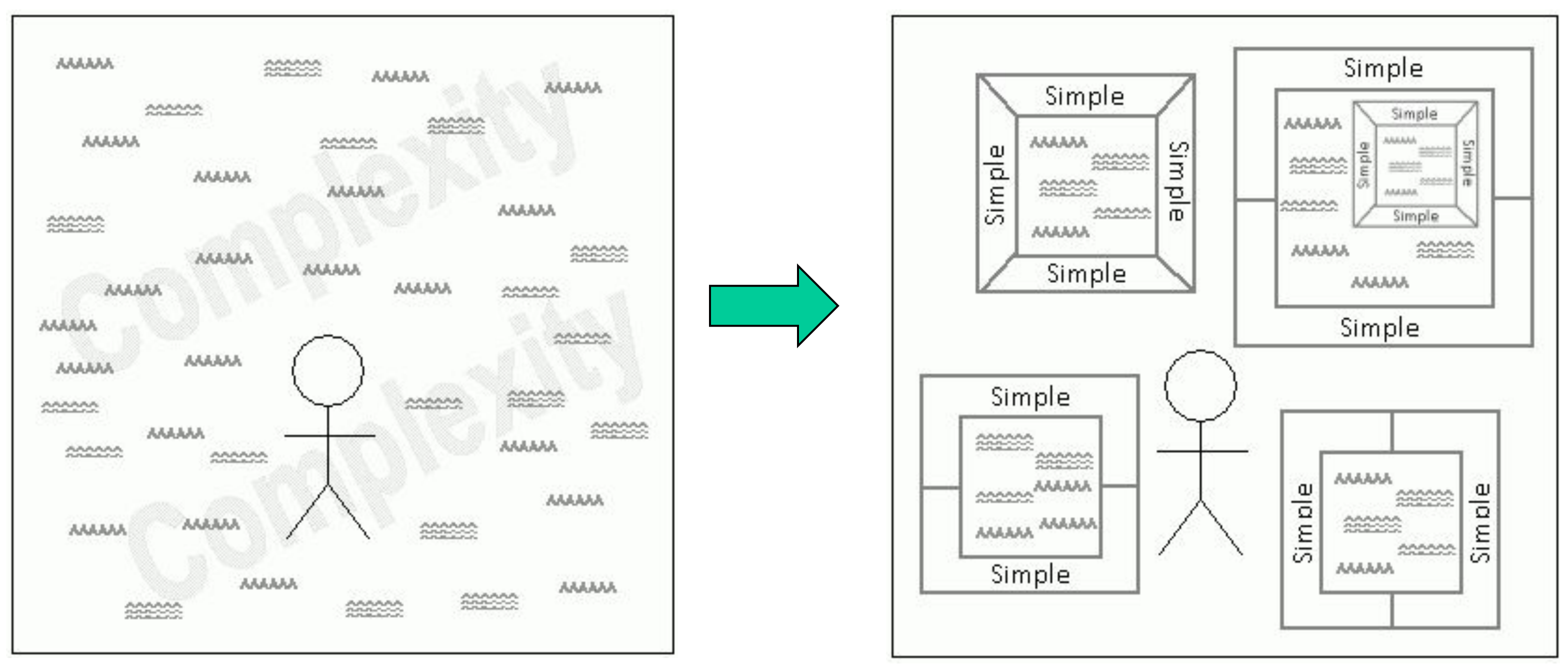
Essential Complexities + Accidental complexities

# Design: An Antidote to Complexity

- Design is the primary tool for managing essential and accidental complexities in software.
- Good design doesn’t reduce the total amount of essential complexity in a solution but it will reduce the amount of complexity that a programmer has to deal with at any one time.
- A good design will manage essential complexities inherent in the problem without adding to accidental complexities consequential to the solution.

# Design Techniques for Dealing with Software Complexity

- Modularity – subdivide the solution into smaller easier to manage components. (divide and conquer)
- Information Hiding – hide details and complexity behind simple interfaces

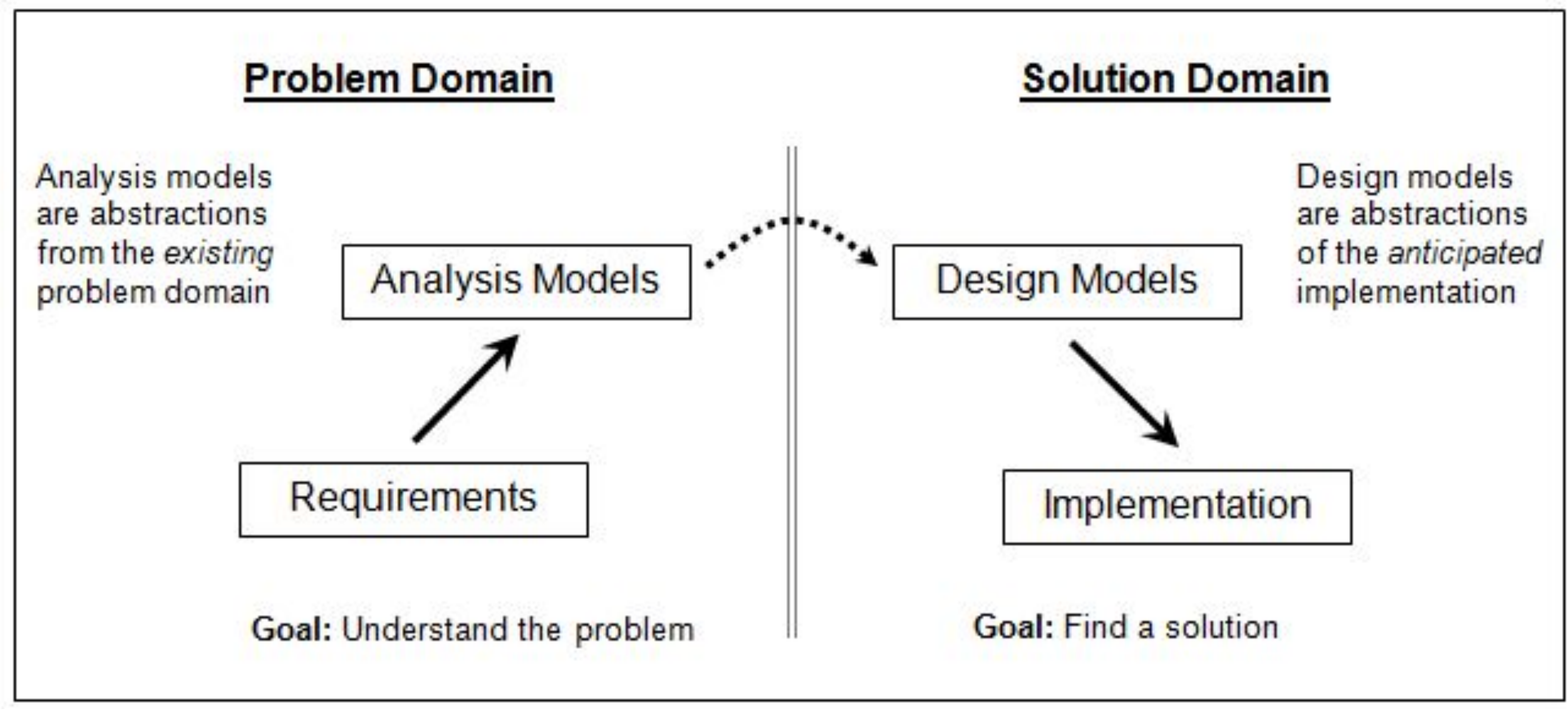


# Additional Techniques for Dealing with Complexity

- Abstraction – use abstractions to suppress details in places where they are unnecessary.
- Hierarchical Organization – larger components may be composed of smaller components. Examples: a complex UI control such as tree control is a hierarchical organization of more primitive UI controls. A book outline represents the hierarchical organization of ideas.

# Why Design is Hard

- Design is difficult because design is an abstraction of the solution which has yet to be created



# Design is a wicked problem

- The term wicked problem was first used to describe problems in social planning but engineers have recognized it aptly describes some of the problems they face as well.
- A wicked problem is one that can only be clearly defined by solving it.
- Need two solutions. The first to define the problem, and the second to solve it in the most efficient way.
- Fred Brooks could have been talking about wicked problems when he advised: “Plan to throw one away; you will anyhow.”

# Design is a Universal Activity

- Any product that is an aggregate of more primitive elements, can benefit from the activity of design.

**Building Design**

Doors, windows, plumbing fixtures, ...

Wood, steel, concrete, glass, ...

**Landscape Design**

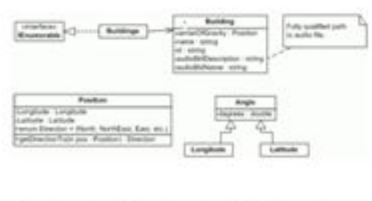
Trees, flowers, grass, rocks, mulch, ...

**User Interface Design**

Tree view, table view, File chooser, ...

Buttons, labels, text boxes, ...

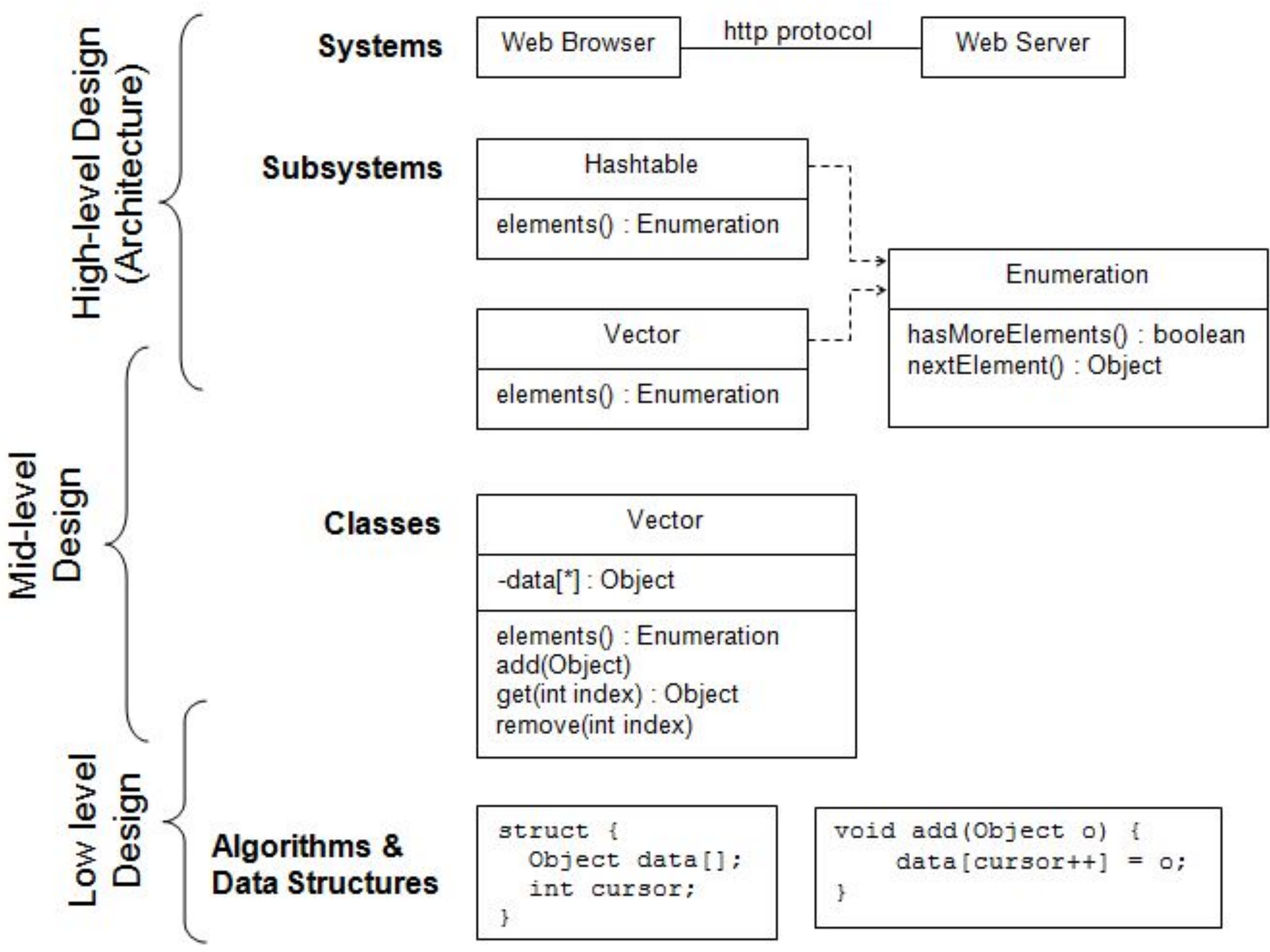
**Software Design**



Classes, procedures, functions, ...

Data declaration, expressions, control flow statements, ...

# Design Occurs at Different Levels




Standard Levels of Design

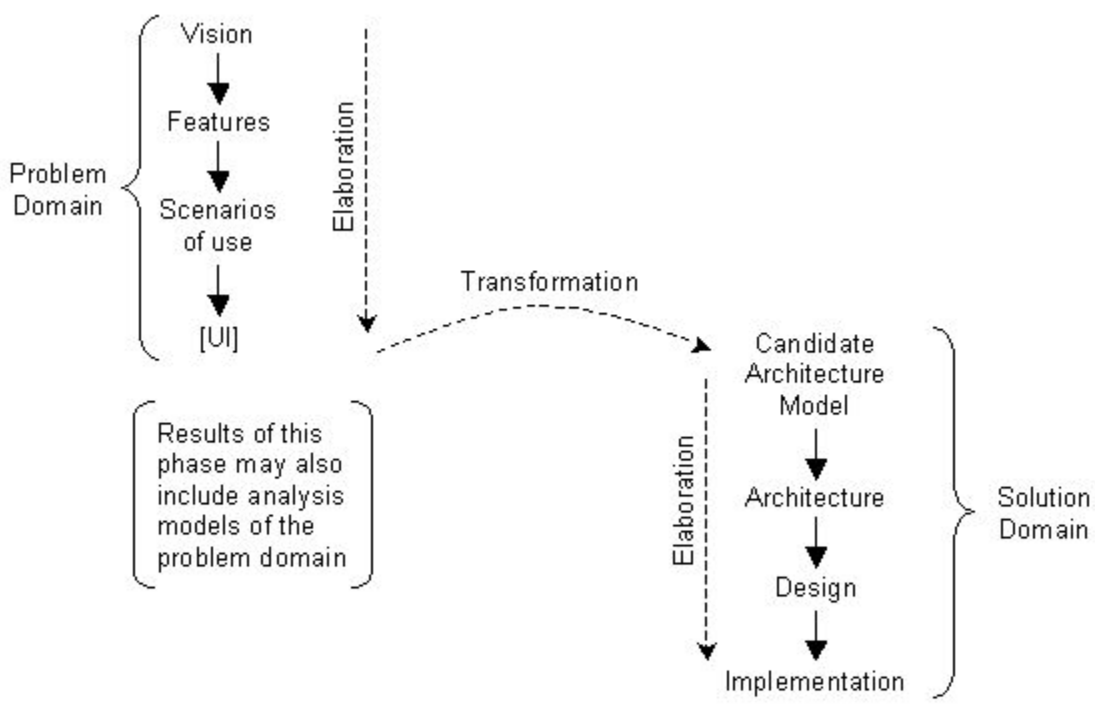
# Characteristics of Software Design

- Non-deterministic – A deterministic process is one that produces the same output given the same inputs. Design is non-deterministic. No two designers or design processes are likely to produce the same output.
- Heuristic – because design is non-deterministic design techniques tend to rely on heuristics and rules-of-thumb rather than repeatable processes.
- Emergent – the final design evolves from experience and feedback. Design is an iterative and incremental process where a complex system arises out of relatively simple interactions.

# The Evolution of Designs

- Design as a single step in the software life cycle is somewhat idealized.
- More often the design process is iterative and incremental.
- Designs tend to evolve over time based on experience with their implementation.

# Elaboration and Transformation



# The Benefits of Good Design

- Good design reduces software complexity which makes the software easier to understand and modify. This facilitates rapid development during a project and provides the foundation for future maintenance and continued system evolution.
- It enables reuse. Good design makes it easier to reuse code.
- It improves software quality. Good design exposes defects and makes it easier to test the software.
- Complexity is the root cause of other problems such as security. A program that is difficult to understand is more likely to be vulnerable to exploits than one that is simpler.

# A Generic Design Process

1. Understand the problem (software requirements).
2. Construct a “black-box” model of solution (system specification). System specifications are typically represented with use cases (especially when doing OOD).
3. Look for existing solutions (e.g. architecture and design patterns) that cover some or all of the software design problems identified.
4. Design not complete? Consider using one or more design techniques to discover missing design elements
   - Noun-verb analysis, CRC Cards, step-wise refinement, etc.
   - Take final analysis model and pronounce a first-draft design (solution) model
5. Consider building prototypes
6. Document and review design
7. Iterate over solution (Refactor) (Evolve the design until it meets functional requirements and maximizes non-functional requirements)

# Inputs to the design process

- User requirements and system specification (including any constraints on design and implementation options)
- Domain knowledge (For example, if it’s a healthcare application the designer will need some knowledge of healthcare terms and concepts.)
- Implementation knowledge (capabilities and limitations of eventual execution environment)

# Desirable Internal Design Characteristics

- Minimal complexity – Keep it simple. Maybe you don't need high levels of generality.
- Loose coupling – minimize dependencies between modules
- Ease of maintenance – Your code will be read more often then it is written.
- Extensibility – Design for today but with an eye toward the future. Note, this characteristic can be in conflict with "minimize complexity". <u>Engineering is about balancing conflicting objectives</u>.
- Reusability – reuse is a hallmark of a mature engineering discipline
- Portability – works or can easily be made to work in other environments
- High fan-in on a few utility-type modules and low-to-medium fan-out on all modules. High fan-out is typically associated with high complexity.
- Leanness – when in doubt, leave it out. The cost of adding another line of code is much more than the few minutes it takes to type.
- Stratification – Layered. Even if the whole system doesn't follow the layered architecture style, individual components can.
- Standard techniques – sometimes it's good to be a conformist! Boring is good. Production code is not the place to try out experimental techniques.

# *That's not the way I would have done it* is not a criteria for evaluating a design

- When evaluating a design some designers have a tendency to dismiss a design simply because it's not what they would have done.
- The feeling could be a sign that some underlying design principle was violated or it could simply be a difference in personal preference.
- If a design is not what you would have done, look for principles of good design that have been violated. If you can't find any, that suggests the design is OK (or you have discovered a new principle of good design), just not what you would have done. You can still offer an alternate design for consideration, but any criticism would be inappropriate.

# Attributes of a design

- Static structure of system (components and their relationships)
- Interactions between components
- System data and its structure (database scheme)
- Physical packaging and distribution of components
- Algorithms

# Design Methods

- Design methods provide a procedural description for obtaining a design solution
- Most methods include:
  - A representation part or notation for representing problem and intermediate forms of the design solution (usually from different view points). Examples: UML, pseudocode.
  - Process part or procedures to following in developing the solution
  - Heuristics – guidelines and best practices for making decisions and assessing intermediate and final results.

    Remember, design isn’t deterministic.

# Design – Representational Forms

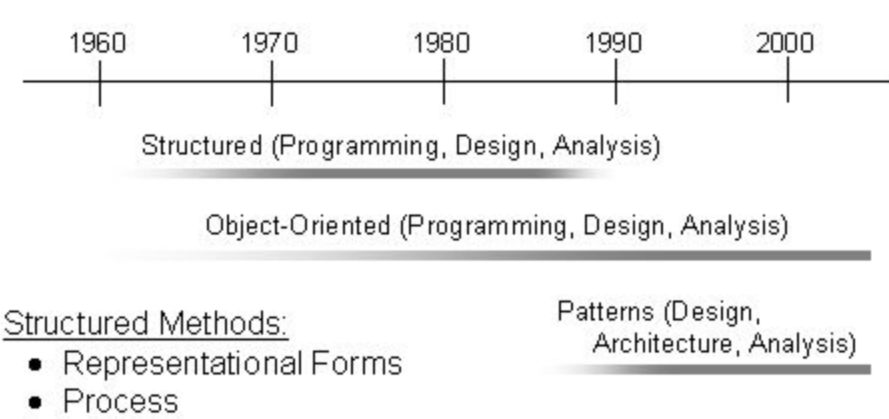
- Class diagrams for static structure
- Sequence diagrams for dynamic behavior
- Textual and visual form of use cases are used to create and validate analysis and design representational forms
- Other UML models are also useful for understanding the problem and conceptualizing a solution (state machine diagram, activity diagram, etc.)

# Design Representational Forms

- Offers particular abstractions of the system from a certain perspective (viewpoint)
- Types of representational forms
  - Visual models
  - Text
  - Pseudocode
- Design representations: Functional, static structural, dynamic behavioral, data modeling (database schema)

# Evolution of Design Methods

- Patterns play an important role in the design methods of today



# Methods and Patterns

- Methods and patterns are the principle techniques for dealing with the challenges of design
- They are useful for:
  - Creating a design
  - Documenting and communicating a design
  - Transferring design knowledge and experience between practitioners

# Patterns

- A design pattern is a reusable solution to a commonly occurring design problem
- Design patterns are adapted for the unique characteristics of the particular problem
- Just as there are levels of design, there are levels of design patterns:
  - Architecture Styles/Patterns
  - Design Patterns
  - Programming Idioms

# Design Methods

- Generic
  - Structured System Analysis and Structured Design
  - Object-Oriented Analysis and Design
- Specific
  - Jackson System Development (JSD)
  - DSDM

# Structured Analysis and Design

- General representational forms: data flow diagram (DFD) and structure chart
- General design process: (1) model the system processes and information flow with a DFD, (2) *transform* the DFD into a hierarchical set of subprograms
- General heuristics: (1) use concepts of coupling and cohesion when deciding how to apportion responsibility among subprograms, (2) try to identify a central transform in the DFD (or create your own) such that all other processes can be subordinate to this transform in the resulting structure chart.

# Design Strategies, Techniques and Tactics

# Design Strategies

- Look for real-world objects
- Top-Down Decomposition
- Bottom-Up Aggregation/Composition - start with what you know the system needs to do. The API you are using might dictate portions of the design. If you aren't completely familiar with the API, bottom-up design might be the best place to start.
- Round-Trip Gestalt
- Organizational Influences on Design (More often limited to architecture, Conway's Law)

# Top-Down vs Bottom-up Design

- Which approach is better (top-down or bottom up) if the implementation technologies are new (i.e. you have minimal experience with the programming language and/or environment)?

# Look for Real World Objects

- Start with an object decomposition based on real-world objects. You can start the design phase by promoting analysis models to design models and evolve them as solution models.
- The objects' role in the real world will suggest certain attributes and behaviors (operations).
- The "real-world" doesn't necessarily imply tangibility. It might be the virtual world of a game. For example, a zombie qualifies as a "real world" object in a game.

# Stepwise Refinement

- A problem is approached in stages. Similar steps are followed during each stage, with the only difference being the level of detail involved.
- Variations on stepwise refinement:
  - Top-down
  - Bottom up
  - Functional decomposition

# Noun/Verb Analysis

- Classes and their associated behavior can be discovered in the narration in the requirements document that describes the requirements of the system.
- A very general guideline is that nouns indicate classes and verbs the operations on classes.

# CRC Cards

- CRC stands for Class-Responsibility-Collaboration.
- CRC cards are a very effective low tech way of identifying classes, responsibilities, and collaborations between classes.

| Class Name | |
|---|---|
| Responsibilities | Collaborators |
| | |

# Core Design Principles and Heuristics

# The Importance of Managing Complexity

- Humans have a very limited capacity for dealing with complexity *directly*.
- Findings from George Miller’s famous paper: “The Magical Number Seven, Plus or Minus Two”
  - Expose the average person to the same stimuli at different levels (pitch, loudness, brightness, etc.) and he or she will be able to discriminate between about 7 different values.
  - The capacity of short-term memory is about 7 items. When given a list of unrelated items, humans are about to recall about 7 of them.
- We can compensate for our limited cognitive abilities by employing techniques such as chunking (modularity), abstraction, information hiding, etc.

# Modularity

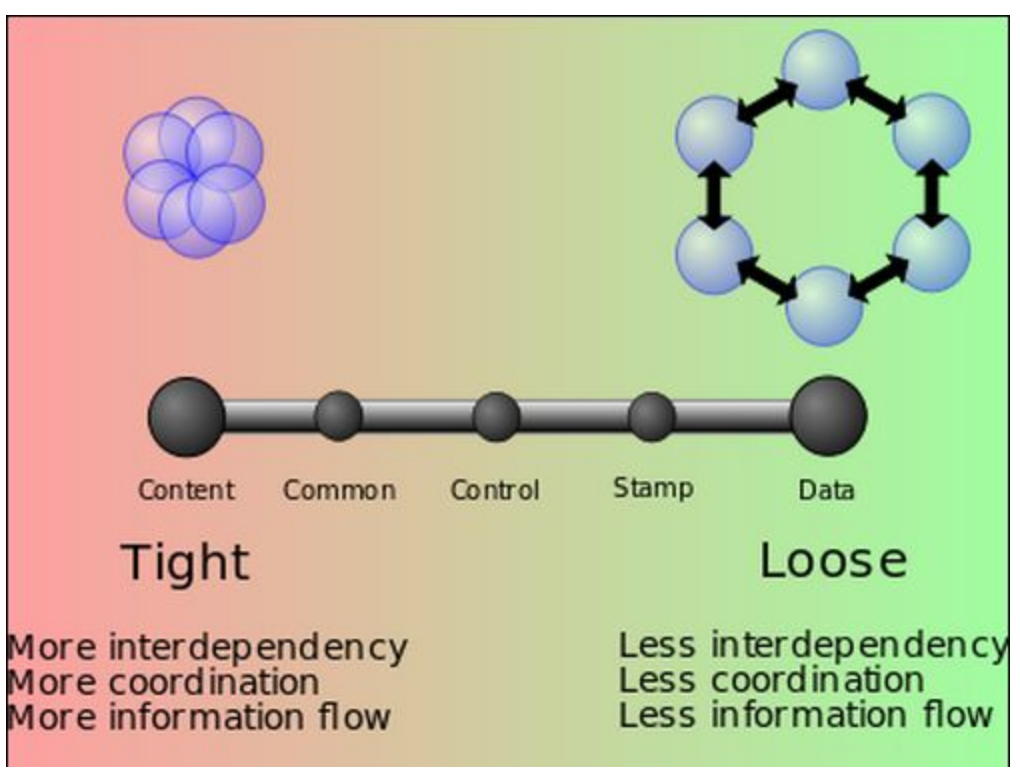
- The goal of design is to partition the system into modules and assign responsibility among the components in a way that:
  - High cohesion within modules, and
  - Loose coupling between modules
- Modularity reduces the total complexity a programmer has to deal with at any one time assuming:
  1. Functions are assigned to modules in away that groups similar functions together (Separation of Concerns), and
  2. There are small, simple, well-defined interfaces between modules (information hiding)
- The principles of cohesion and coupling are probably the most important design principles for evaluating the effectiveness of a design.

# Coupling

- Coupling is the measure of dependency between modules. A dependency exists between two modules if a change in one could require a change in the other.
- The degree of coupling between modules is determined by:
  - The number of interfaces between modules (quantity), and
  - Complexity of each interface (determined by the type of communication) (quality)

# Types of Coupling

- Content coupling (also known as Pathological coupling)
- Common coupling
- Control coupling
- Stamp coupling
- Data coupling



# Content Coupling

- One module directly references the contents of another
  1. One module modifies the local data or instructions of another
  2. One module refers to local data in another
  3. One branches to a local label of another

# Common Coupling

- Two or more modules connected via global data.
  1. One module writes/updates global data that another module reads

# Control Coupling

- One module determines the control flow path of another. Example:

```
print(milesTraveled, displayMetricValues)
. . .
public void print(int miles, bool displayMetric) {
    if (displayMetric) {
        System.out.println(. . .);
        . . .
    else { . . .}
}
```

# Stamp Coupling

- Passing a composite data structure to a module that uses only part of it. Example: passing a record with three fields to a module that only needs the first two fields.

# Data Coupling

- Modules that share data through parameters.

# Coupling between CSS and JavaScript

- A well-designed web app modularizes around:
  - HTML files which specify data and semantics
  - CSS rules which specify the look and formatting of HTML data
  - JavaScript which defines behavior/interactivity of page
- Assume you have the following HTML and CSS definitions.

- HTML:

```
<!doctype html>
<html>
<head>
  <script type="text/javascript" src="base.js"></script>
  <link rel="stylesheet" href="default.css">
</head>
<body>
  <button onclick="highlight2()">Highlight</button>
  <button onclick="normal2()">Normal</button>
  <h1 id="title" class="NormalClass">CSS <--> JavaScript Coupling</h1>
</body>
</html>
```

coupling-example.html

- CSS:

```
.NormalClass {
  color:inherit;
  font-style:normal;
}
```

default.css

- Output:

- Suppose you want to change the style of the title in response to user action (clicking on a button).
- This is behavior or action so it must be handled with JavaScript.
- Evaluate the coupling of the following two implementation options. Both have the same behavior.

# Option A

- JavaScript code modifies the style attribute of HTML element.

```
function highlight() {
    document.getElementById("title").style.color="red";
    document.getElementById("title").style.fontStyle="italic";
}

function normal() {
    document.getElementById("title").style.color="inherit";
    document.getElementById("title").style.fontStyle="normal";
}
```

base.js

# Option B

- JavaScript code modifies the class attribute of HTML element.

```
function highlight() {
    document.getElementById("title").className = "HighlightClass";
}

function normal() {
    document.getElementById("title").className = "NormalClass";
}
```

base.js

```
.NormalClass {
  color:inherit;
  font-style:normal;
}

.HighlightClass {
  color:red;
  font-style:italic;
}
```

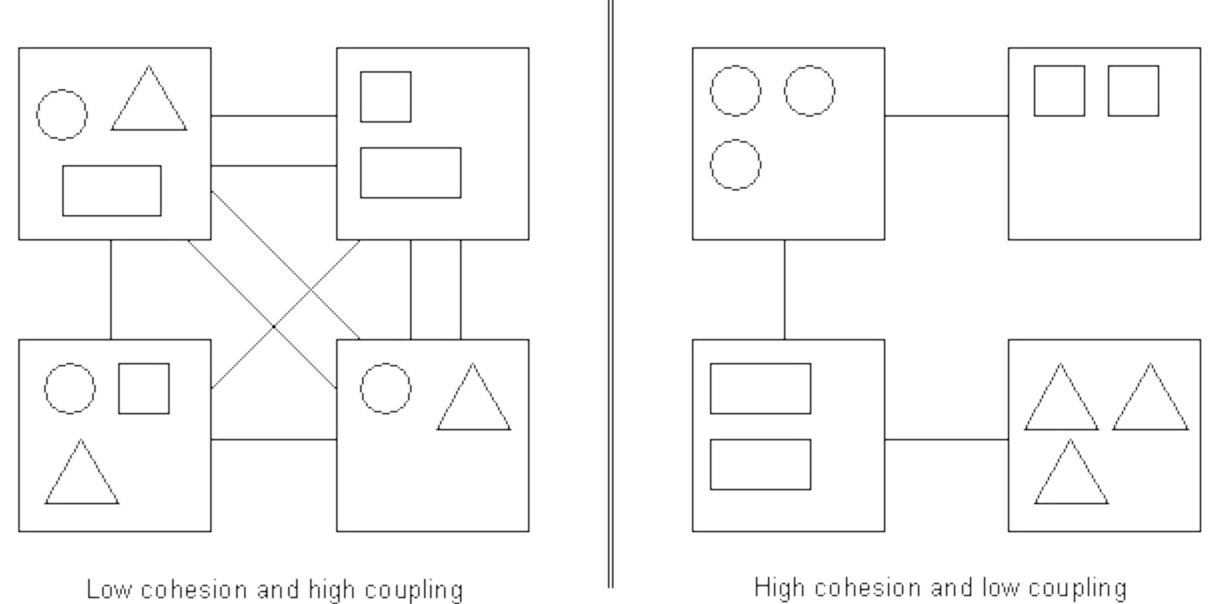
default.css

# Cohesion

- Cohesion is a measure of how strongly related the functions or responsibilities of a module are.
- A module has high cohesion if all of its elements are working towards the same goal.

# Cohesion and Coupling

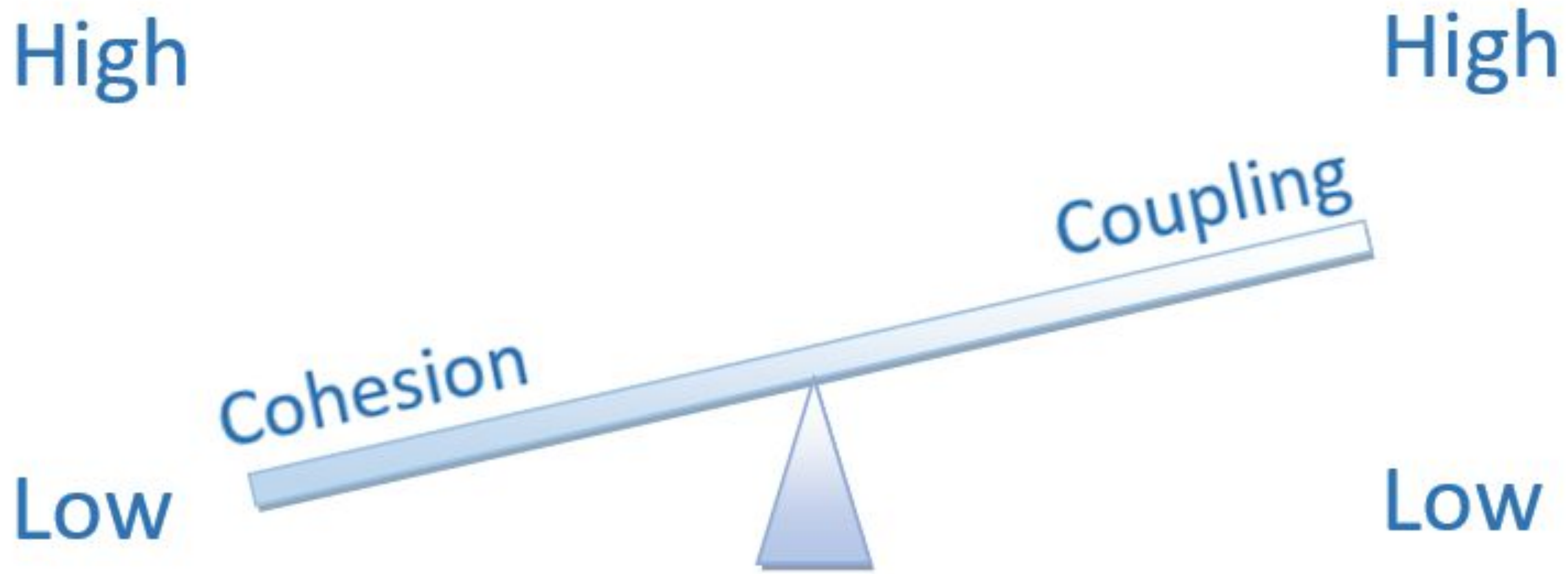
- The best designs have high cohesion (also called strong cohesion) within a module and low coupling (also called weak coupling) between modules.



# Benefits of high cohesion and low coupling

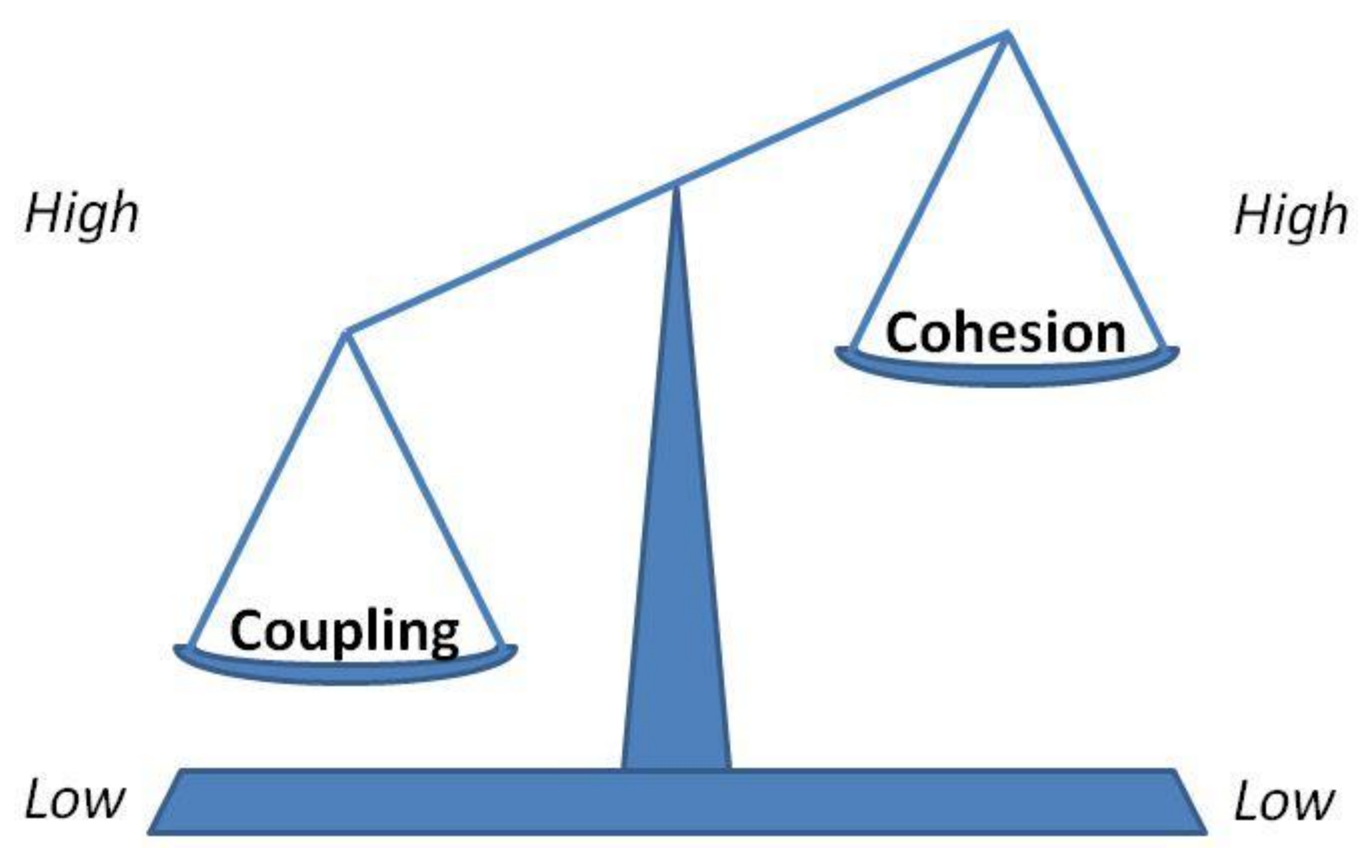
1. Modules are easier to read and understand.
2. Modules are easier to modify.
3. There is an increased potential for reuse
4. Modules are easier to develop and test.

# Coupling and Cohesion Tend to be Inversely Correlated



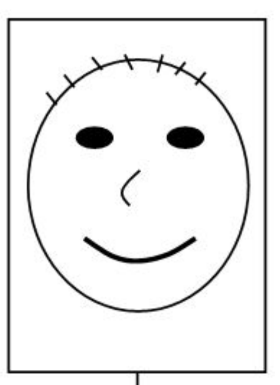

When Coupling is high, cohesion tends to be low and vise versa.

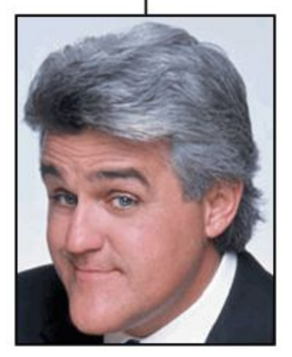
# Relationship between Coupling and Cohesion



# Abstraction

- Abstraction is a concept used to manage complexity
- An abstraction is a generalization of something too complex to be dealt with in its entirety
- Abstraction is for humans not computers
- Abstraction is a technique we use to compensate for the relatively puny capacity of our brains (when compared to the enormous complexity in the world around us)
- There aren't enough neurons (or connections) in our brain to process the rich detail around us during a single moment in time
- Successful designers develop abstractions and hierarchies of abstractions  for complex entities and move up and down this hierarchy with splendid ease





# New York City Street

# New York City Street

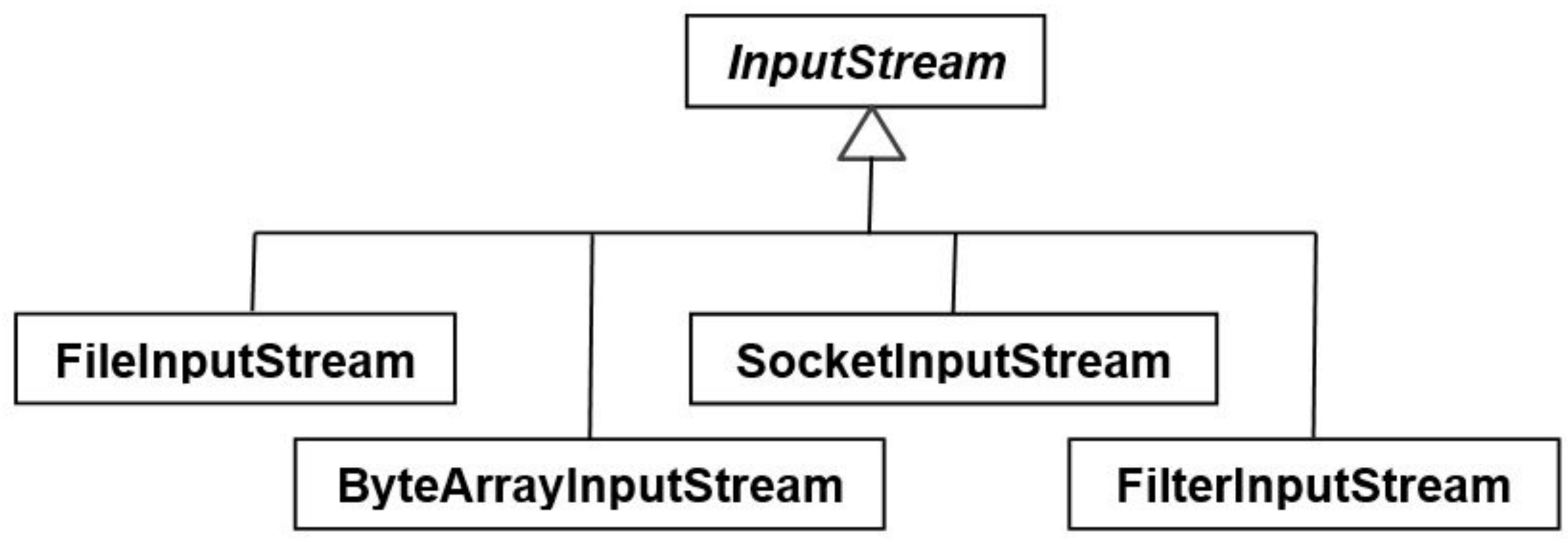
# Form Consistent Abstractions

- "Abstraction is the ability to engage with a concept while safely ignoring some of its details."
- Base classes and interfaces are abstractions. i.e UIComponent (any GUI toolkit), Mammal (classic superclass when discussing OO design)
- The interface defined by a class is an abstraction of what the class represents
- A procedure defines an abstraction of some operation.
- "The principle benefit of abstraction is that it allow you to ignore irrelevant details."

```java
public class Driver {

    public static void main(String[] args) throws IOException {

        // concretePrint() only works for FileInputStream's
        String fileName = "input.txt";
        FileInputStream fileInput = new FileInputStream(fileName);
        concretePrint(fileInput);

        // abstractPrint() will work for any input
        //   stream.
        abstractPrint(fileInput);

        String message = "Something I want to print.";
        InputStream baInputStream = new ByteArrayInputStream(message.getBytes());
        abstractPrint(baInputStream);

        // Even works with input streams from remote URL's
        URL url = new URL("");
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000);
        conn.setConnectTimeout(15000);
        conn.setRequestMethod("GET");
        conn.setDoInput(true);
        conn.connect();

        InputStream socketInputStream = conn.getInputStream();
        abstractPrint(socketInputStream);
    }
```

```java
private static void concretePrint(FileInputStream fileInput) throws IOException {
    BufferedReader reader = new BufferedReader(new InputStreamReader(fileInput));
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}


// This method is more abstract than the one above.
// It can be used in more situations.
private static void abstractPrint(InputStream is)  throws IOException {
    BufferedReader reader = new BufferedReader(new InputStreamReader(is));
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
```

**This method is more abstract**




InputStream is an abstract class with several concrete subclasses.

# Information Hiding

- Information hiding is a design principle
- The information hidden can be data, data formats, behavior, and more generally, design decisions
- When information is hidden there is an implied separation between interface and implementation. The information is hidden behind the interface
- Parnas encourages programmers to hide "difficult design decisions or design decisions which are likely to change"
- The clients of a module only need to be aware of its interface. Implementation details should be hidden

# Information Hiding [Cont]

- Want to hide design and implementation decisions—especially those likely/subject to change.
- Information hiding implies encapsulation and abstraction. You are hiding details which creates an abstraction.
- When skillfully applied, information hiding has the effect of hiding complexity.

# Example 1: Evaluate the following design in terms of information hiding

```
class PersistentData {
    public ResultSet read(string sql);
    // write returns the number of rows effected
    public int write(string sql);
}
```

**Sample Client Code**

```
PersistentData db = new PersistentData();
db.write("UPDATE Employees SET Dependents = 2
          WHERE EmployeeID = 47");
```

# Example 1: Evaluate the following design in terms of information hiding

```
class EmployeeGateway {
    public static EmployeeGateway find(int ID);
    public void setName(string name);
    public string getName();
    public void setDependents(int dependents);
    public int getDependents();
    // insert() returns ID of employee
    public int insert();
    public void update();
    public void delete();
}
```

**Sample Client Code**

```
EmployeeGateway e = EmployeeGateway.find(47);
e.setDependents(2);
e.update();
```

# Example 2: Evaluate the following class design in terms of information hiding

```
class Course {
    private Set students;

    public Set getStudents() {
        return students;
    }
    public void setStudents(Set s) {
        students = s;
    }
}
```

# Example 2: Improved Information Hiding

```
class Course {
    private Set students;

    public Set getStudents() {
        return Collections.
                unmodifiableSet(students);
    }
    public void addStudent(Student student) {
        students.add(student);
    }
    public void removeStudent(Student student) {
        students.remove(student);
    }
}
```

# Why Practice Information Hiding?

- Hiding complexity – limiting the amount of information you have to deal with at any one time
- Reducing dependencies on design and implementation decisions to minimize the impact of changes. (Avoid the ripple effect of changes.)
- "Large programs that use information hiding were found …to be easier to modify—by a factor of 4—than programs that don't.[Korson and Vaishnavi 1986]

# Encapsulation

- Encapsulation is an implementation mechanism for enforcing information hiding and abstractions.
- There is no clear widely accepted definition of encapsulation. It can mean:
  - A grouping together of related things (records, arrays)
  - A protected enclosure (object with private data and/or methods)

# Supporting Design Principles and Heuristics

# Don’t Repeat Yourself (DRY)

- In general, every piece of knowledge should have a single, unambiguous, authoritative representation within a system.
- Most programmers will recognize this principle as it applies to coding: you shouldn’t have duplicate code (e.g. cutting and pasting code or its close cousin: copy-paste-modify).
- More generally, it applies to documentation, test cases, test plans, etc.
- Repeating yourself invites maintenance problems. You have two or more locations that have to be kept synchronized/consistent.

# Principle of Least Astonishment/Surprise (POLA)

- The POLA is probably more applicable during UI design, but is also relevant during software design.
- In short, don’t surprise the user (UI design) or programmer (software design) with unexpected behavior. Users and developers should be able to rely on their intuition.
- Most web users expect clicking on the icon in the upper left-hand corner of a web page will take them to the home page of the web site. It would be a surprise if it did anything else.
- During software design use descriptive names for variables, methods and classes. The name of a method should reflect what it does. The name of a variable should reflect the use or meaning of the data it holds.
- Putting business logic in a class called Settings would be a violation of the POLA. Most programmers would expect a class called Settings to contain constants only.

# Separation of Concerns

- The functions, or more generally concerns, of a program should be separate and distinct such that they may be dealt with on an individual basis.
- Separation of concerns helps guide module formation. Functions should be distributed among modules in a way that minimizes interdependencies with other modules.
- Example: many web applications are structured around the 4-tier web architecture:
  1. Presentation or UI
  2. Business Logic
  3. Data Access
  4. Database (typically relational)
- Each layer encapsulates a related set of related functions.

# Single Responsibility Principle (SRP)

- SRP is a subtle variation on the concept of cohesion.
- A cohesive module is one where all the elements of the module are functionally related (separation of concerns).
- A module that conforms to the SRP is one that has a single reason to change. The SRP defines a responsibility as a reason to change.
- Example: let's say you had a custom UI control that also included code needed to save it's state when the app loses focus. If the UI control doesn't have any event handling code you could argue it is a good example of separation of concerns. (It has one concern: display view.) However, there are two reasons it might change: (1) view changes, (2) change in storage method used to persist state.

# Open-Closed Principle (OCP)

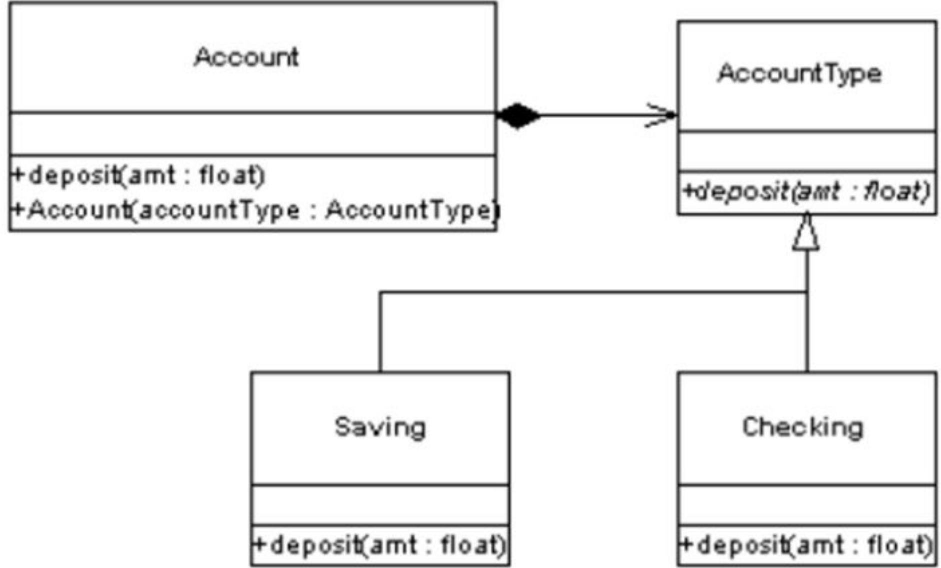
- Simply stated, the OCP says that modules (functions, classes, etc) should be open for extension but closed for modification.
- Translation: it should be possible to extend the function/behavior of a module without having to modify its code.
- Example: Web browser plug-in architecture. You can add support for a new media type without making major changes to existing code.

# OCP [Cont]

- Categories in Objective-C allow you to extend the behavior of classes while remaining compliant with the OCP. (You can actually add methods to a class at runtime.)
- Delegation is another pattern for designing a class that is open for extension but closed for modification.

# OCP Using Delegation

```
public class Account {
  private AccountType _accountType;

  public  Account(AccountType accountType) {
    this._accountType = accountType;
  }

  public void deposit(float amt) {
    this._accountType.deposit(amt);
  }
}
```




```
public abstract class AccountType {
  public abstract void deposit(float amt);
}
```

```
public class Saving extends AccountType {
  public void deposit(float amt) {
    //deposit saving amt.
  }
}
```

```
public class Checking extends AccountType {
  public void deposit(float amt) {
    //deposit checking amt.
  }
}
```

# Final Classes can abide by O/C

In open-closed principle (open for extension, closed for modification) you can still use the final modifier. Here is one example:

```
public final class ClosedClass {

    private IMyExtension myExtension;

    public ClosedClass(IMyExtension myExtension)
    {
        this.myExtension = myExtension;
    }

    // methods that use the IMyExtension object

}

public interface IMyExtension {
    public void doStuff();
}
```

The `ClosedClass` is closed for modification inside the class, but open for extension through another one. In this case it can be of anything that implements the `IMyExtension` interface. This trick is a variation of dependency injection since we're feeding the closed class with another, in this case through the constructor. Since the extension is an `interface` it can't be `final` but its implementing class can be.

# Dependency Inversion Principle (DIP)

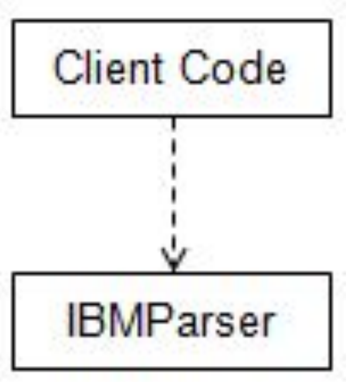
- The DIP formalizes the general design concept of Inversion of Control (IoC).
- Both are sometimes called the Hollywood Principle because they describe a phenomena where the reused component embraces the Hollywood cliché, “Don't call us, we'll call you.”

# Dependency Inversion Principle (DIP)

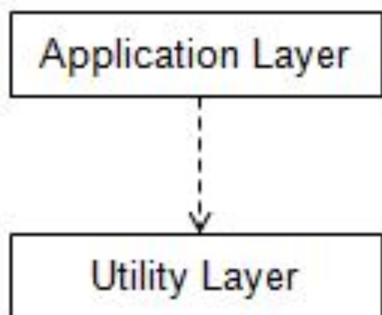
- There are two parts to the DIP:
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend upon details. Details should depend upon abstractions.
- Example: many class libraries have a factory method for creating an XML parser. At runtime a specific implementation of a parser is provided to application code requesting a parser. High level modules depend on the abstract interface to the parser. Detailed implementation of the parser depends only on the abstract interface it implements.

# Example

Imagine a programming language, let's call it Java, that didn't want to implement an XML parser but instead wanted to make community-written parsers available to programmers. One option is to ask the community to submit parsing classes and then make these classes available to programmers. Programmers would write code that looked like:




```
IBMParser p = new IBMParser();
p.parse(inputStream);
```

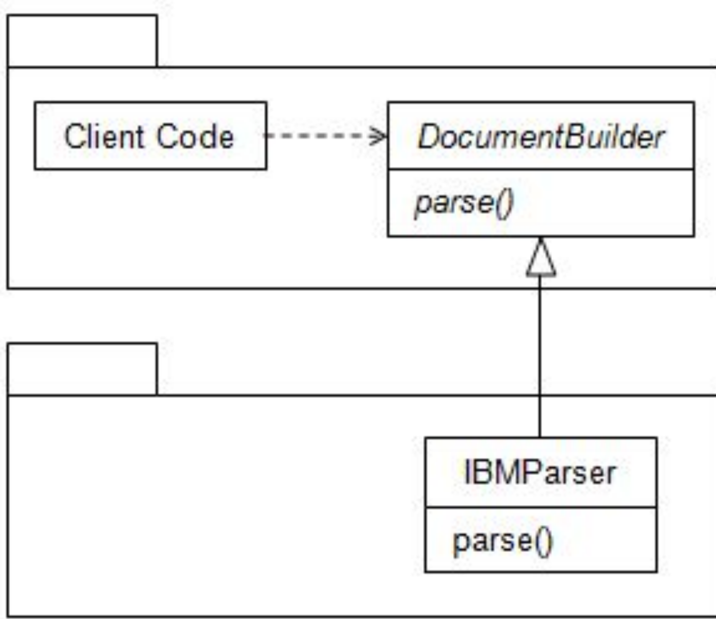



The obvious problem with this is the dependency between client code and utility code. The consequences are a little more severe since it is a class from a outside source, but this type of dependency is common between client code and utility classes.

# Example [Cont]

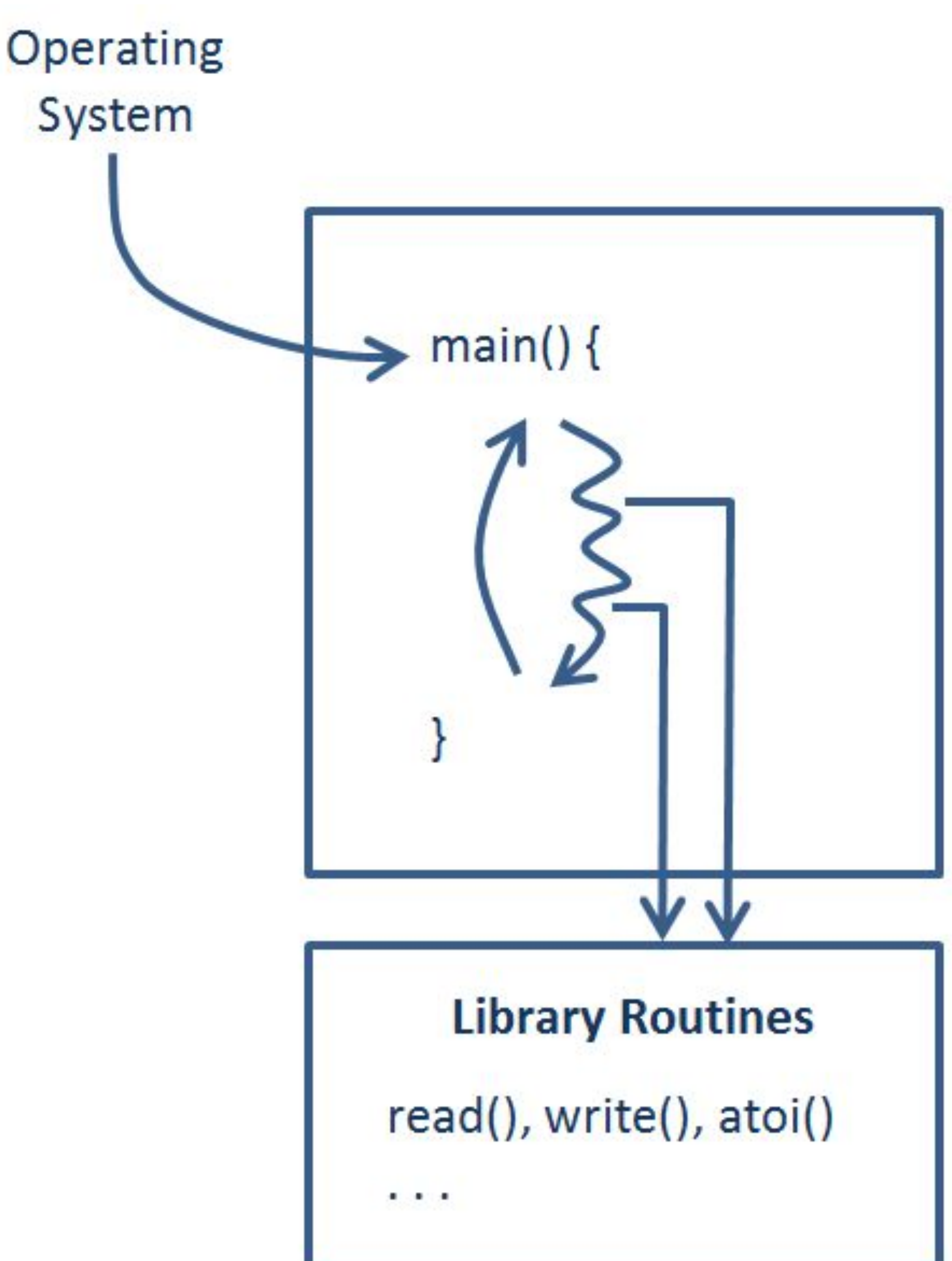
Another option is to define an abstract base class for XML parsers, let's call it DocumentBuilder, and have vendors implement their parsers as subclasses of this abstract base class. The important point is this abstract class is owned by the language designers. The code supplied by XML parser vendors is dependent on this abstract class.

With this new arrangement, the dependency is reversed or inverted.



# Inversion of Control and Frameworks

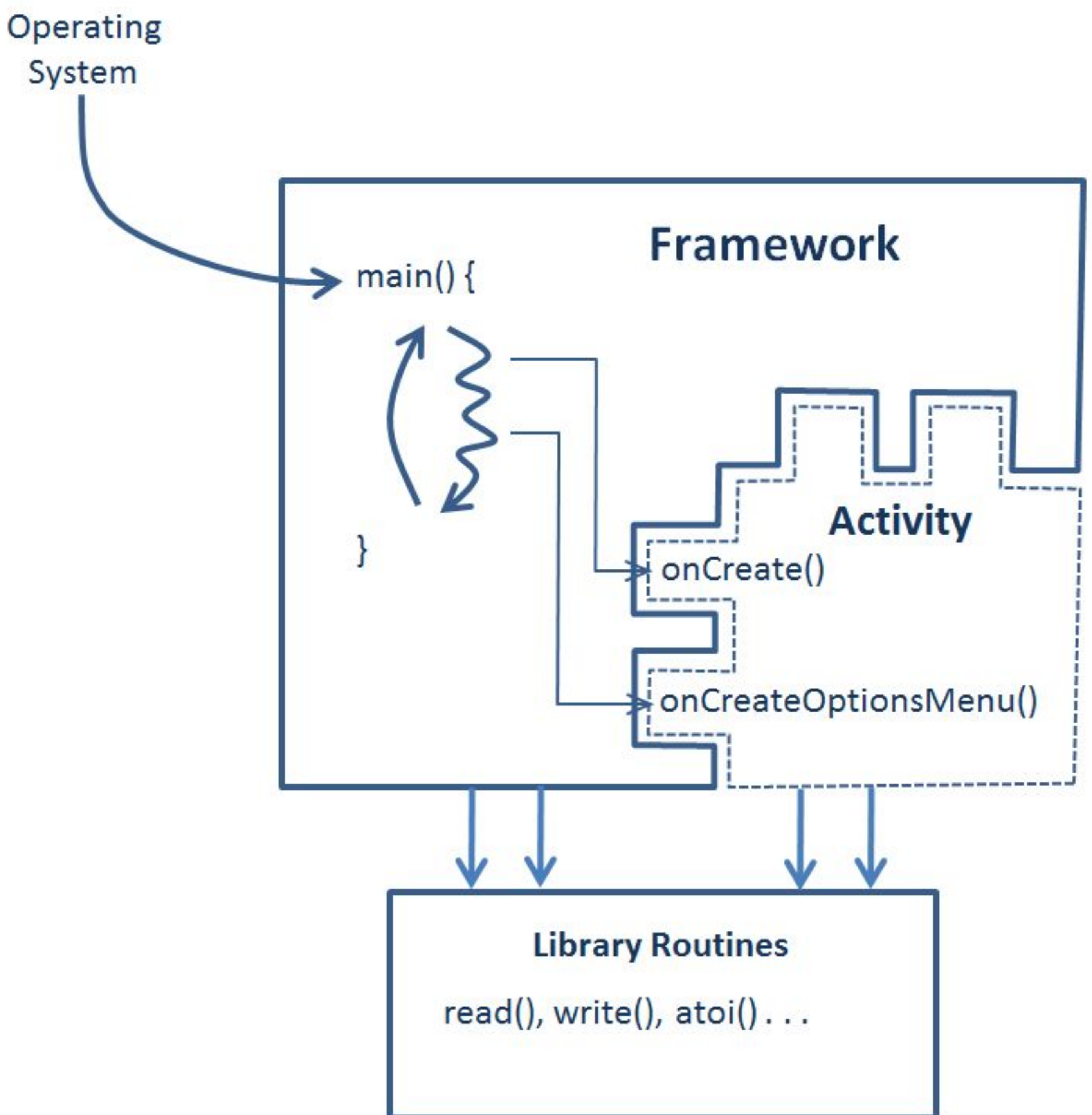
- Much of programming today consists of extending existing frameworks rather than writing traditional procedural programs that retain responsibility for the control flow of an application.
- With traditional procedural programming, control is passed from the operating system to the main entry point of the program. Control may pass temporarily to library subroutines, but the main program has complete control and sole responsibility for the sequence of activities that comprise the application.



Operating System
main() {
}
Library Routines
read(), write(), atoi()
. . .

# Frameworks

- Programming with frameworks inverts the locus of control. The main thread of control resides in the framework rather than your code.
- Your code is hooked into the framework and called by the framework as needed.
- With procedural programming, every routine you write (except the main entry point) is called from your code. When extending a framework, many of the routines you write are only called by the framework. At first it might seem strange to have routines in your code with no apparent caller.


Operating System
Framework
main() {
}
Activity
onCreate()
onCreateOptionsMenu()
Library Routines
read(), write(), atoi() . . .

# Program to an interface, not an implementation

- There are considerations for the client and service.

Service    Client

```
interface ILogger {                 f1(ILogger l);
   void log(String);
}

abstract class ALogger {            f2(ALogger l);
   abstract void log(String);
}

class Logger {                      f3(Logger l);
   void log(String);
}
```

# Interface segregation principle (ISP)

- In short, the message of the ISP is: avoid “fat” interfaces. An interface is considered “fat” if it attracts clients interested in only a portion of the methods offered by the interface and different clients are interested in different portions of the interface.

OLD TIMER

MOTIVA
www.oneinhundred.com

# Interface segregation principle (Cont.)

- The suggested alternative to “fat” interfaces, is to define multiple “skinny” interfaces each with a small group of methods that appeal to different classes of clients.
- Interfaces should be cohesive, that is, focused on one thing.
- Clients should not be forced to implement or depend on portions of an interface they don’t use

# Example

- Java has two interfaces for mouse events, one for common mouse events (MouseListener) and one for motion events (MouseMotionListener). Grouping all events into one interface would have violated the ISP.

```
interface MouseListener {
  void mouseClicked(MouseEvent e);
  void mousePressed(MouseEvent e);
  void mouseReleased(MouseEvent e);
  void mouseEntered(MouseEvent e);
  void mouseExited(MouseEvent e);
}
```

```
interface MouseMotionListener {
  void mouseDragged(MouseEvent e);
  void mouseMoved(MouseEvent e);
}
```

# SOLID Principles of Object-Oriented Design

- S – Single responsibility principle
- O – Open/closed principle
- L – Liskov substitution principle
- I – Interface segregation principle
- D – Dependency inversion principle

# Consider Design by Contract

- Formalize class contracts.
- You an define the services of a routine in terms of pre- and post-conditions. This makes it very clear what to expect.

# Try Design for Testing

- Create a test-friendly design
- A test-friendly module is likely to exhibit other important design characteristics.
- Example: you would avoid circular dependencies. Business logic will be better isolated from UI code if you have to test it separately from the UI

# Don’t overlook brute force as an option

- Sometimes its better to use an inelegant design if the cost of a better design is prohibitive.
- You can also encapsulate it behind a well-designed interface.

# Consider Experimental Prototyping

- "Sometimes you can't really know whether a design will work until you better understand some implementation detail."
- SM recommends starting with a specific question that you then answer with by writing the "absolute minimum amount of throwaway code."
- Need to be disciplined about how you go about experimental prototyping and how you use the results. There is a strong temptation to start writing production code.

# API Design and Use

Stop here

# Design Knowledge and Skills

- Ability to understand and use design methods, patterns, principles, heuristics and techniques. This knowledge is useful for:
  - Creating designs
  - Evaluating  or assessing designs
- Ability to communicate designs and design ideas
  - Architect to programmer (communicate implementation specs)
  - Among colleagues (propose a design)
  - Among practitioners (share experience and expertise)

# Design Qualities

- Fitness for purpose. Satisfies product requirements
- Reliability
- Robustness
- Efficiency
- Usability
- Maintainability
- Evolvability
- Reusability

# References

- How to Design a Good API and Why it Matters, ACM.
- http://www.possibility.com/Cpp/CppCodingStandard.html [read later]

# Common Subsystems

- Business rules
- UI
- Database access – having a data access component allows the business logic and other components to deal with data in the form or at the level of abstraction as it exists in the problem domain. The data access layer isolates the rest of the program from the details of how the data is actually stored. Very few business people view their data in terms of tables and relationships.
- System dependencies – isolate hardware and other environmental dependencies.

# Anticipate Change

- Start during requirements by documenting potential changes and their likelihood.
- Areas of code likely to change should be isolated. (e.g. hidden in a class.)
- Architect around stable ideas. Avoid putting volatile ideas in interfaces.
- Anticipating changes is not the same as designing ahead. Anticipating change guides design decisions. It doesn’t justify new code that’s not needed at the moment.

# Areas Most Likely To Change

- Business rules
- Hardware dependencies – keyboard, controller, file system.
- Input / Output
- Difficult areas of code – sections of code done poorly are likely to change in the future.
- Data-size Constraints

# Divide and Conquer

- Even the best minds can’t hope to fully understand any but the most trivial software programs. They shouldn’t have to.
- A good design is one that allows an individual with average capabilities to fully understand the program by looking at it in pieces.
- “The goal of all software design techniques is to break a complicated problem into simple pieces.”
- You should be able to focus on one module nearly independently of others. (Loose Coupling)

# Designers often have to deal with competing priorities

- Performance versus Readability/Maintainability
- Reusability/Extensibility versus understandability
- <All quality attributers> versus Cost & Schedule
- Game programmers often trade understandability, maintainability, <just about everything else> for performance and time-to-market.
- Priorities drive tradeoff decisions
- The best solutions balance competing priorities

# References

# Kinds of Coupling

- Simple data-parameter coupling – passing primitive data types: f(int,float). No global data.
- Simple object coupling – a module instantiates another object.
- Object-parameter coupling – one module passes another module an object rather than a primitive type: f(SomeClass). Higher coupling than simple-data-parameter
- Semantic coupling – one module makes use of semantic information about another module.

# Semantic Coupling

- Semantic coupling is depending on knowledge of how something is observed to work. Things are a little better if how it works is a documented part of the interface or behavior.
- Examples:
  - Module1 passes a control flag to module2. The control flag affects processing in module2. Module1 makes assumptions about internal workings of module2.
  - Module2 uses global data after it has been modified by Module1.
  - You know you are suppose to call initialize() before calling f() but you use special knowledge of the class to conclude that in some cases you can skip calling initialize() before calling f().