SOFTWARE DESIGN AND ARCHITECTURE

SWE 4601

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, ...

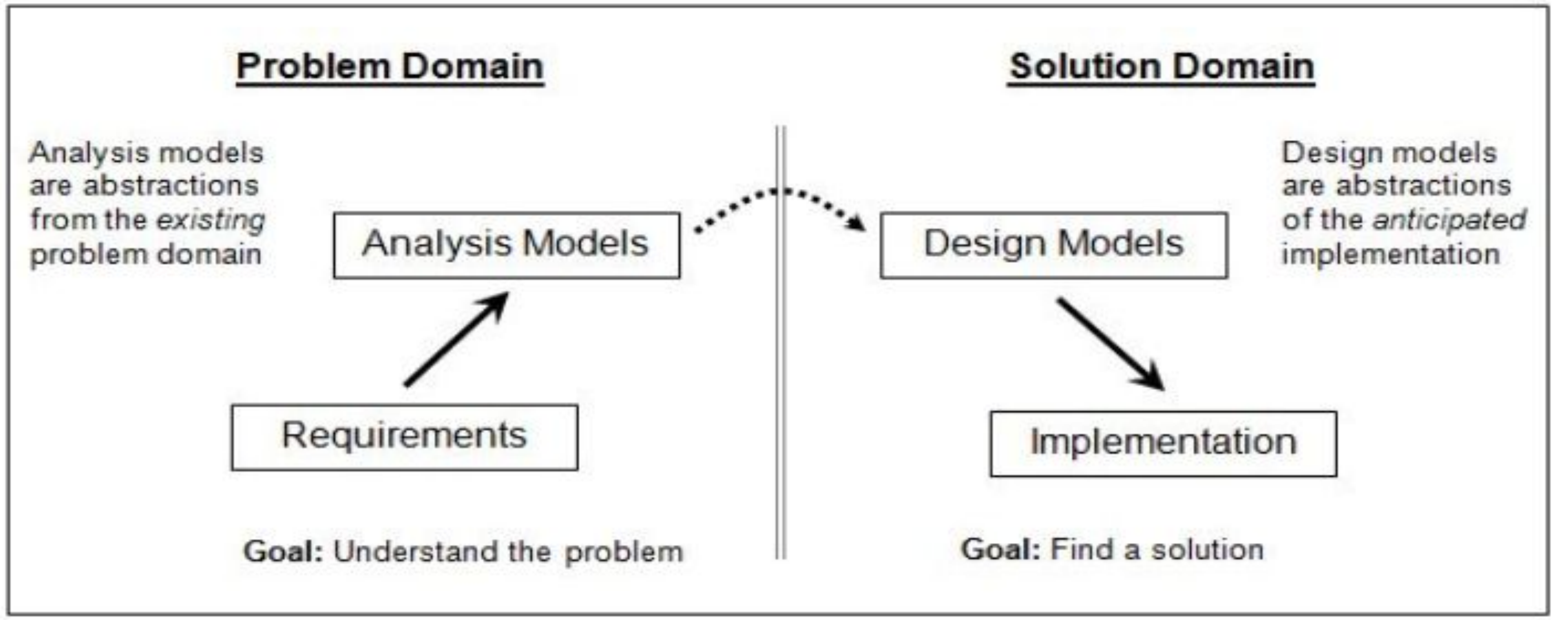
Software Design

Classes, procedures, functions, ...

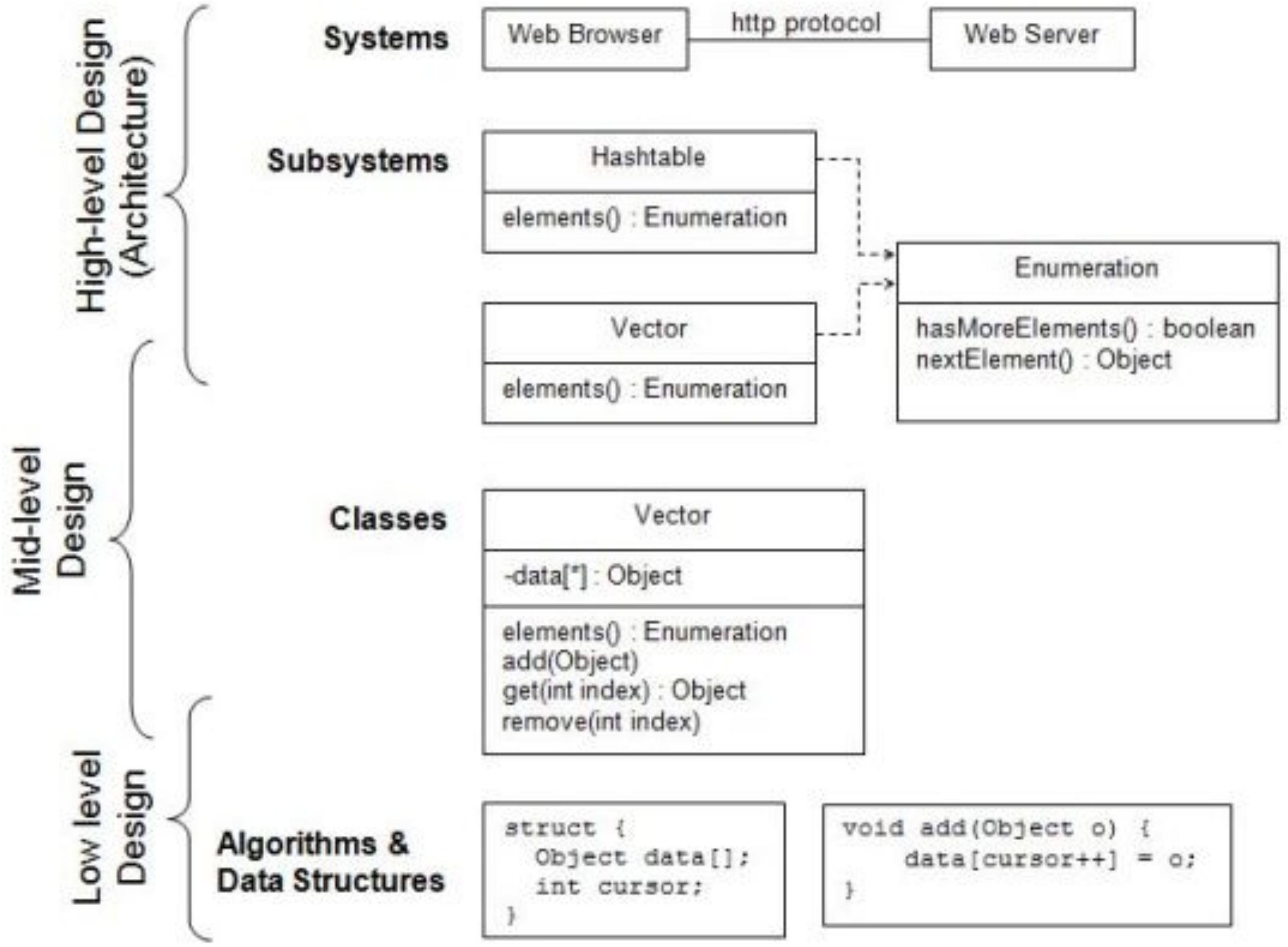
Data declaration, expressions, control flow statements, ...

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 - 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.


Problem Domain
Solution Domain
Analysis models are abstractions from the *existing* problem domain
Analysis Models
Requirements
Design Models
Design models are abstractions of the *anticipated* implementation
Implementation
Goal: Understand the problem
Goal: Find a solution

DESIGN OCCURS AT DIFFERENT LEVELS



```
struct {
  Object data[];
  int cursor;
}
```

```
void add(Object o) {
    data[cursor++] = o;
}
```

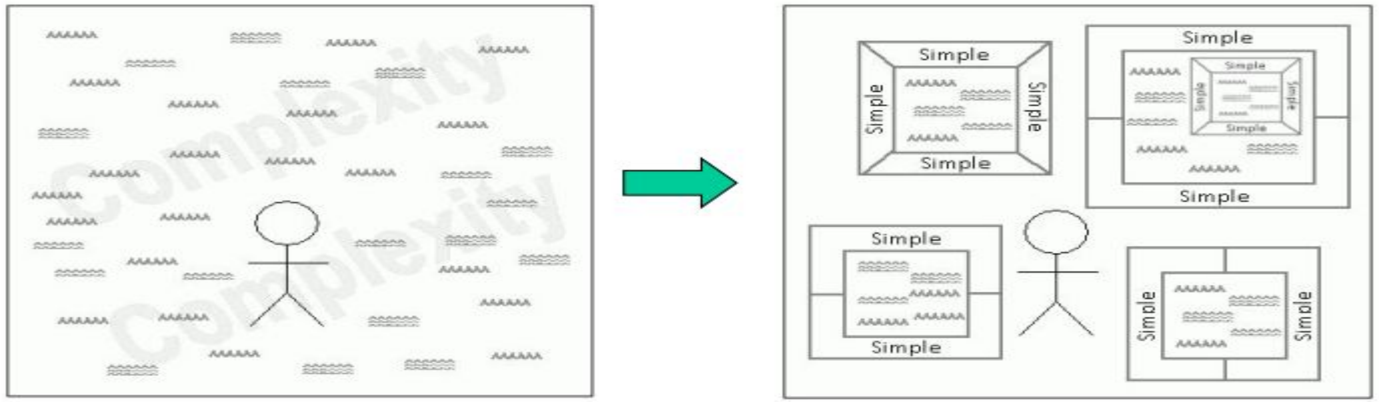
Standard Levels of Design

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.
  - Accidental complexities –challenges that developers unintentionally make for themselves as a result of trying to solve a problem.
  - Essential complexities – complexities that are inherent in the problem
  - Example: Sales forecasting, analyzing the behavior of your funnel are all essential to a CRM.
  - hard-to-read charts, mental math, or reports that don’t instantly update in real time are all examples of accidental complexity.

DESIGN TECHNIQUES FOR DEALING WITH SOFTWARE COMPLEXITY

- Modularity – subdivide the solution into smaller easier to manage components. (divide and conquer)
- Information Hiding – hide information to avoid inadvertent change.
- 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.



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.
  - 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.

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.

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". Engineering is about balancing conflicting objectives.
- Reusability – reuse is a hallmark of a mature engineering discipline
- Portability – works or can easily be made to work in other environment
- 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.
- 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.

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)

OTHERS EVALUATION TO DESIGN

- *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.

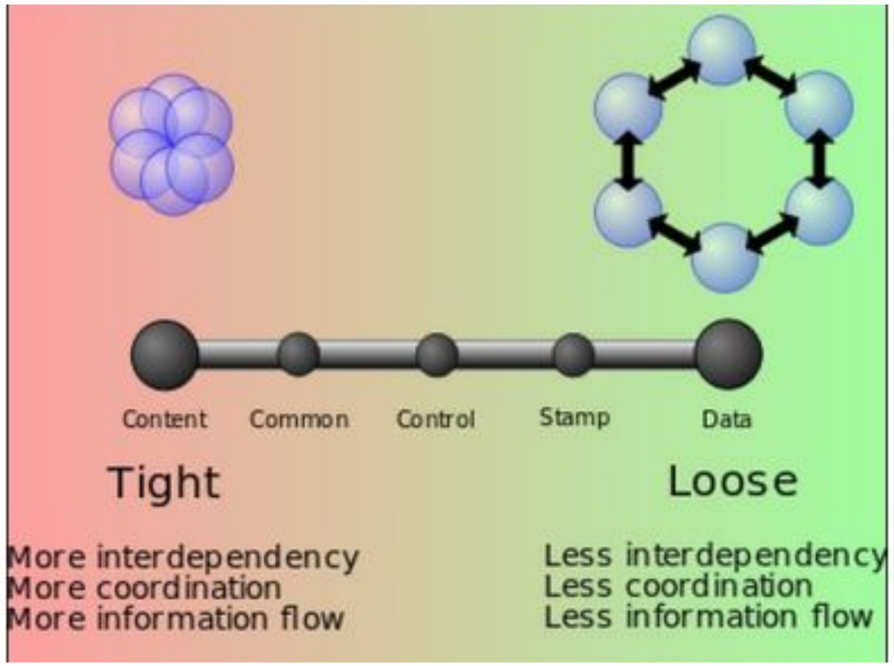
DESIGN METHODS

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:
  - Functions are assigned to modules in away that groups similar functions together (Separation of Concerns), and
  - 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

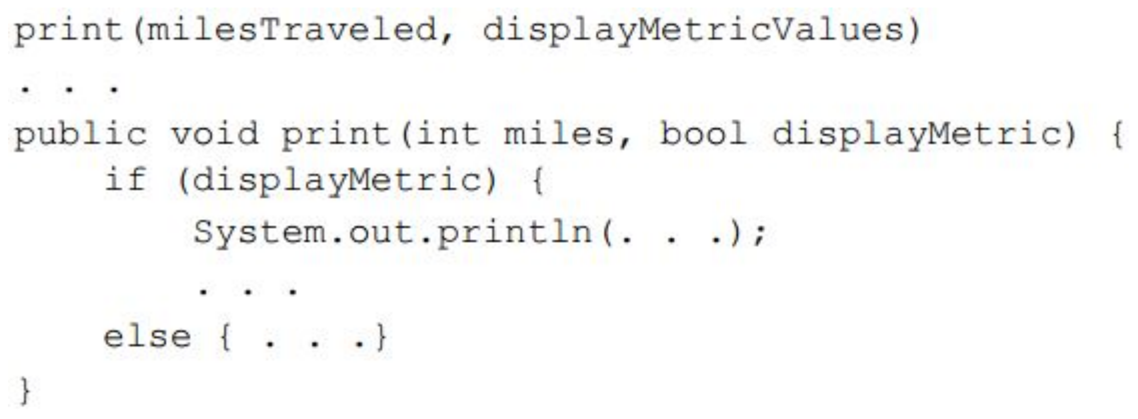
- Coupling is the measure of dependency between modules. A dependency exists between two modules if a change in one could require a change in the other.
- The degree of coupling between modules is determined by:
  – The number of interfaces between modules (quantity), and
  – Complexity of each interface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



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
- 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 { . . .}
}
```

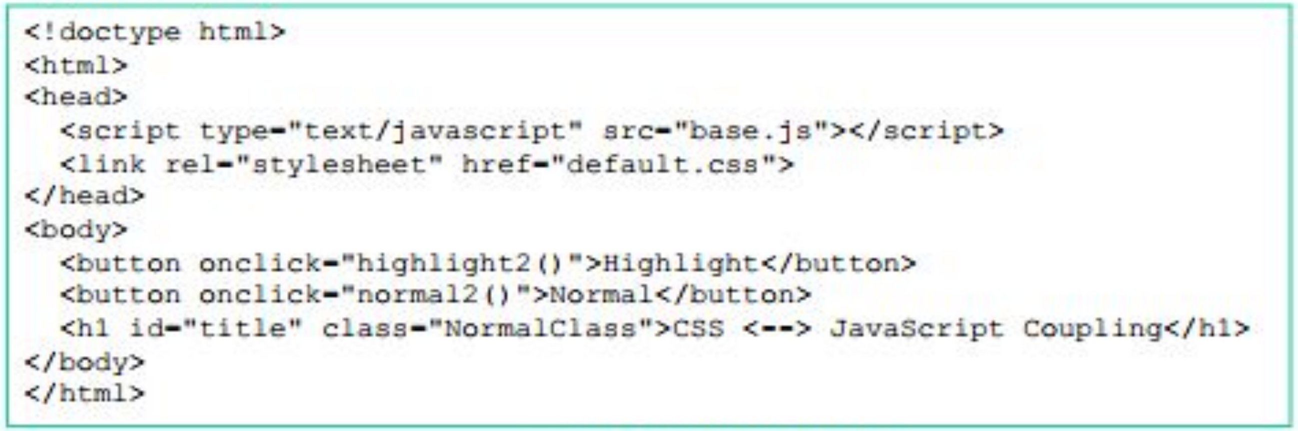
COUPLING

- 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.
- External Coupling: In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.

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.

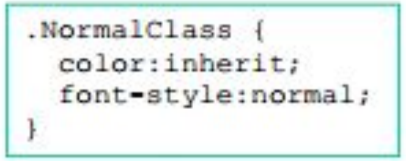
- HTML:



```
<!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

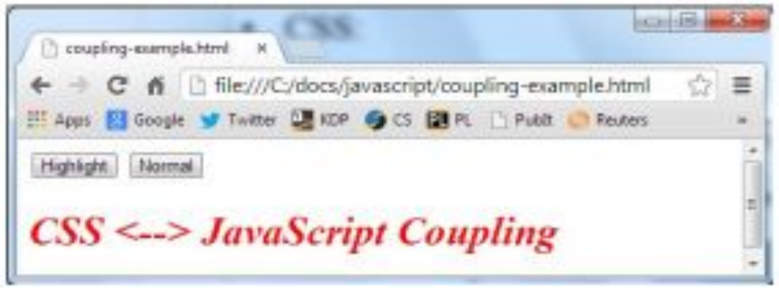
- CSS:



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

default.css

- Output:



coupling-example.html
file:///C:/docs/javascript/coupling-example.html
Apps Google Twitter KDP CS PL Publt Reuters
Highlight
Normal

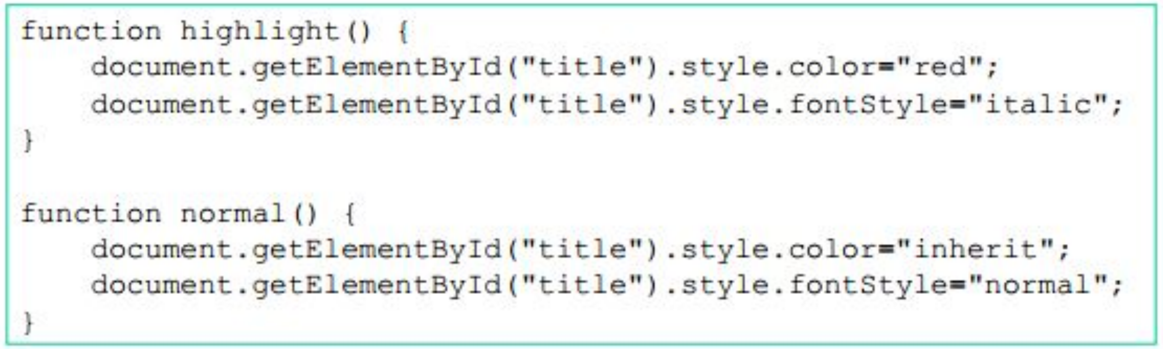
CSS <--> JavaScript Coupling

COUPLING BETWEEN CSS AND JAVASCRIPT

- 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.

COUPLING BETWEEN CSS AND JAVASCRIPT

- **Option A**
  - JavaScript code modifies the <u>style</u> attribute of HTML element.

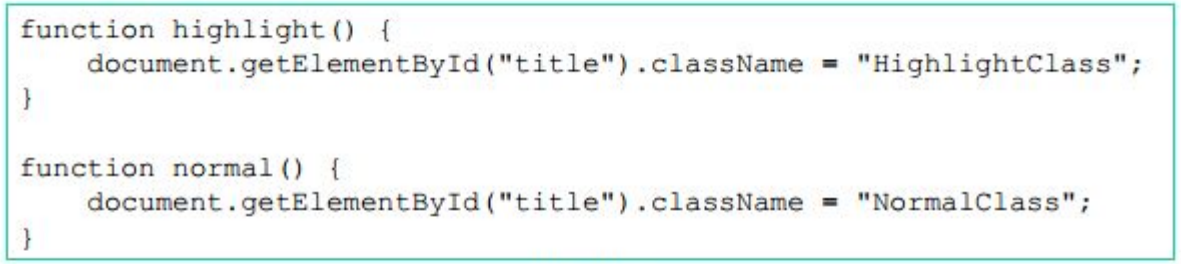


```
function highlight() {
    document.getElementById("title").style.color="red";
    document.getElementById("title").style.fontStyle="italic";
}

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

base.js

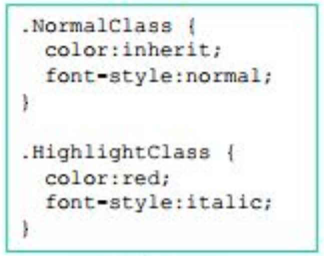
- **Option B**
  - JavaScript code modifies the <u>class</u> attribute of HTML element.



```
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;
}
```

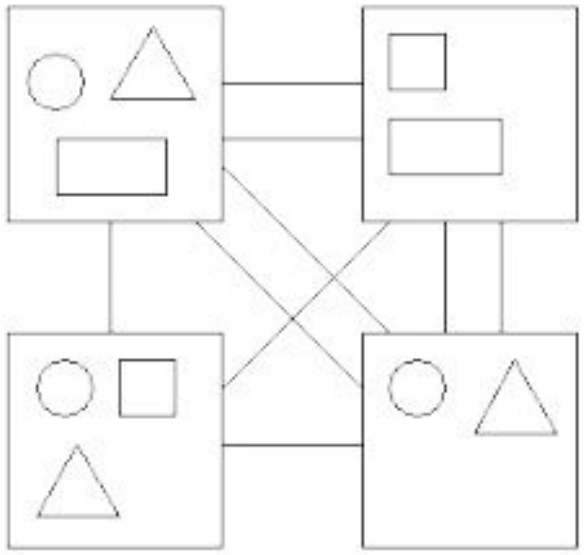
default.css

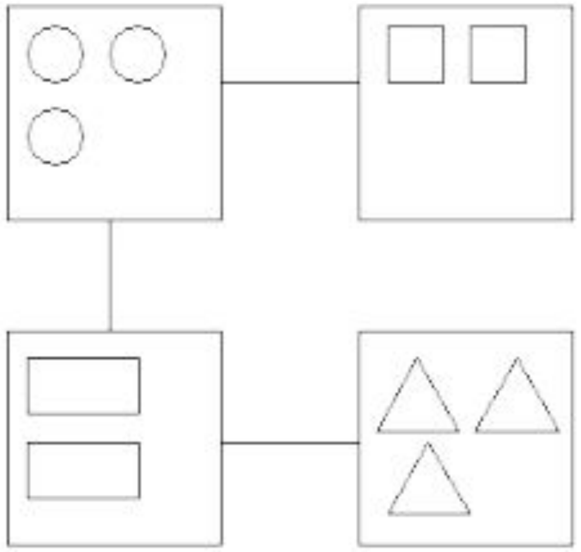
COHESION

- 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

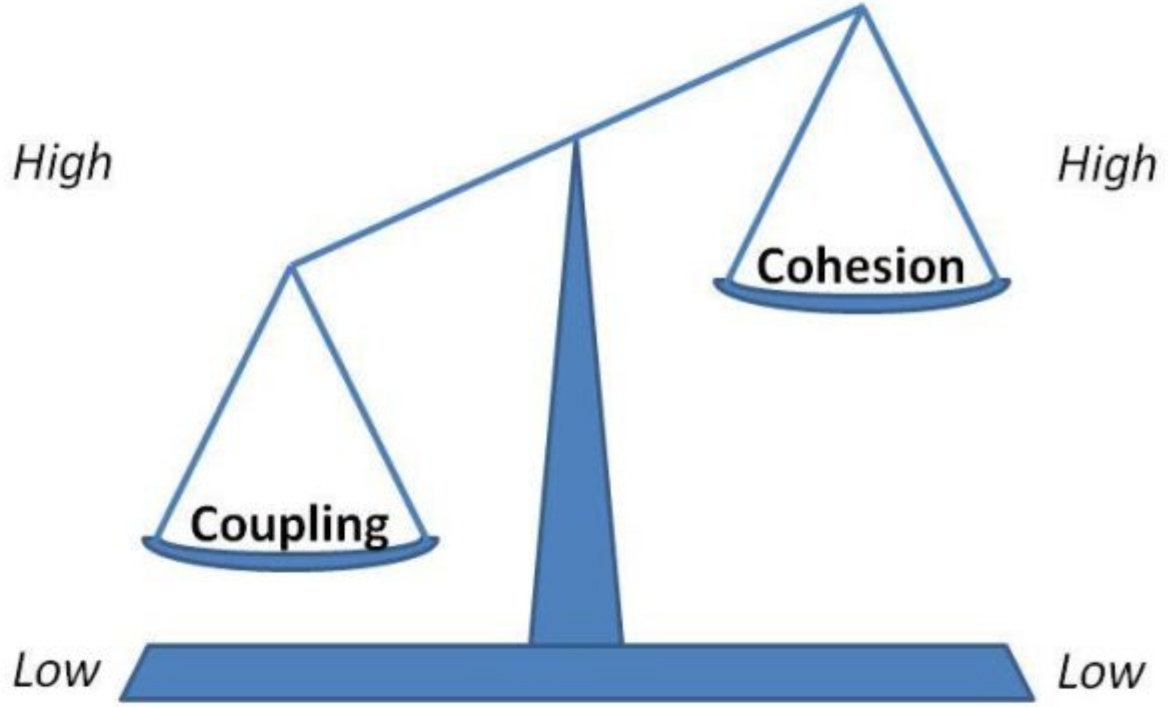
- The best designs have high cohesion (also called strong cohesion) within a module and low coupling (also called weak coupling) between modules.


Low cohesion and High coupling


High cohesion and low coupling

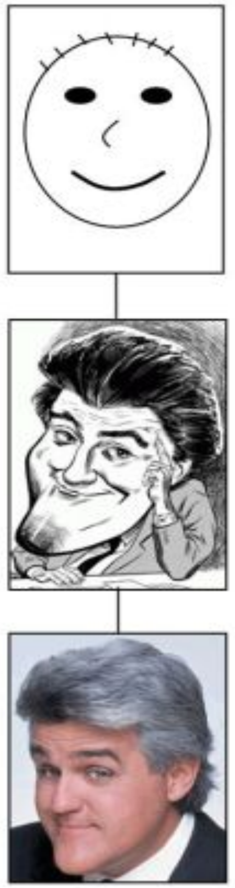
BENEFITS OF HIGH COHESION AND LOW COUPLING

- Modules are easier to read and understand.
- Modules are easier to modify.
- There is an increased potential for reuse
- Modules are easier to develop and test.



ABSTRACTION

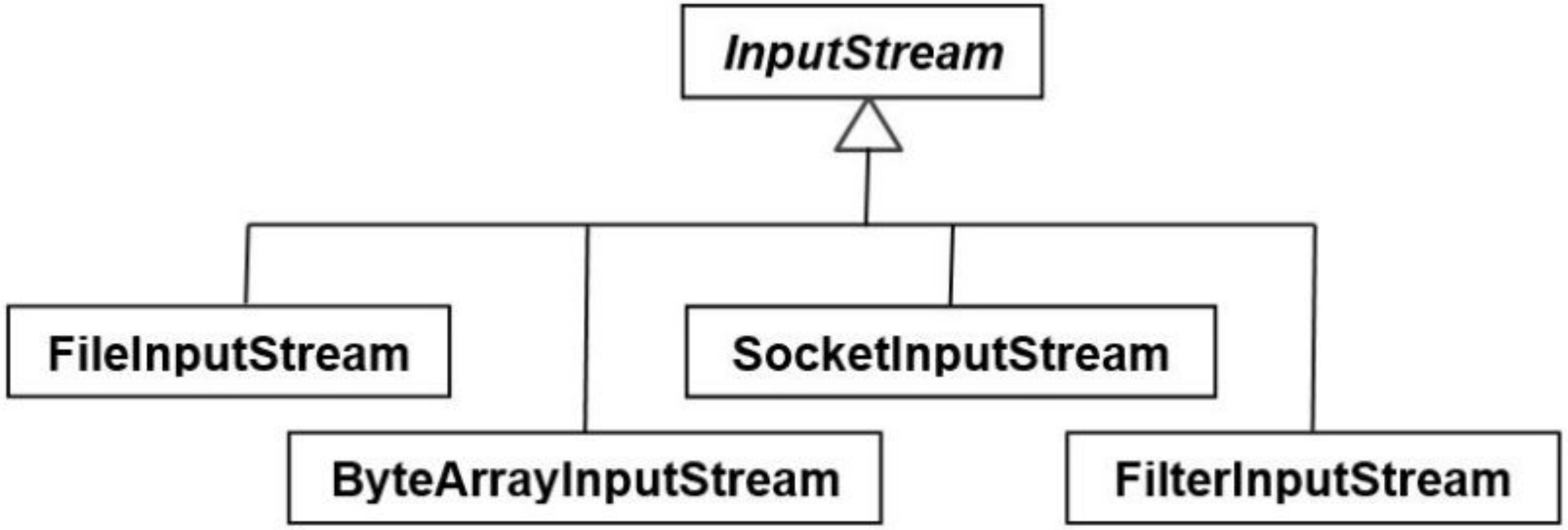
- Abstraction is a concept used to manage complexity
- An abstraction is a generalization of something too complex to be dealt with in its entirety
- Abstraction is for humans not computers
- Abstraction is a technique we use to compensate for the relatively puny capacity of our brains (when compared to the enormous complexity in the world around us)
- Successful designers develop abstractions and hierarchies of abstractions for complex entities and move up and down this hierarchy with splendid ease



ABSTRACTION

- “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
- “The principle benefit of abstraction is that it allow you to ignore irrelevant details.“

ABSTRACTION

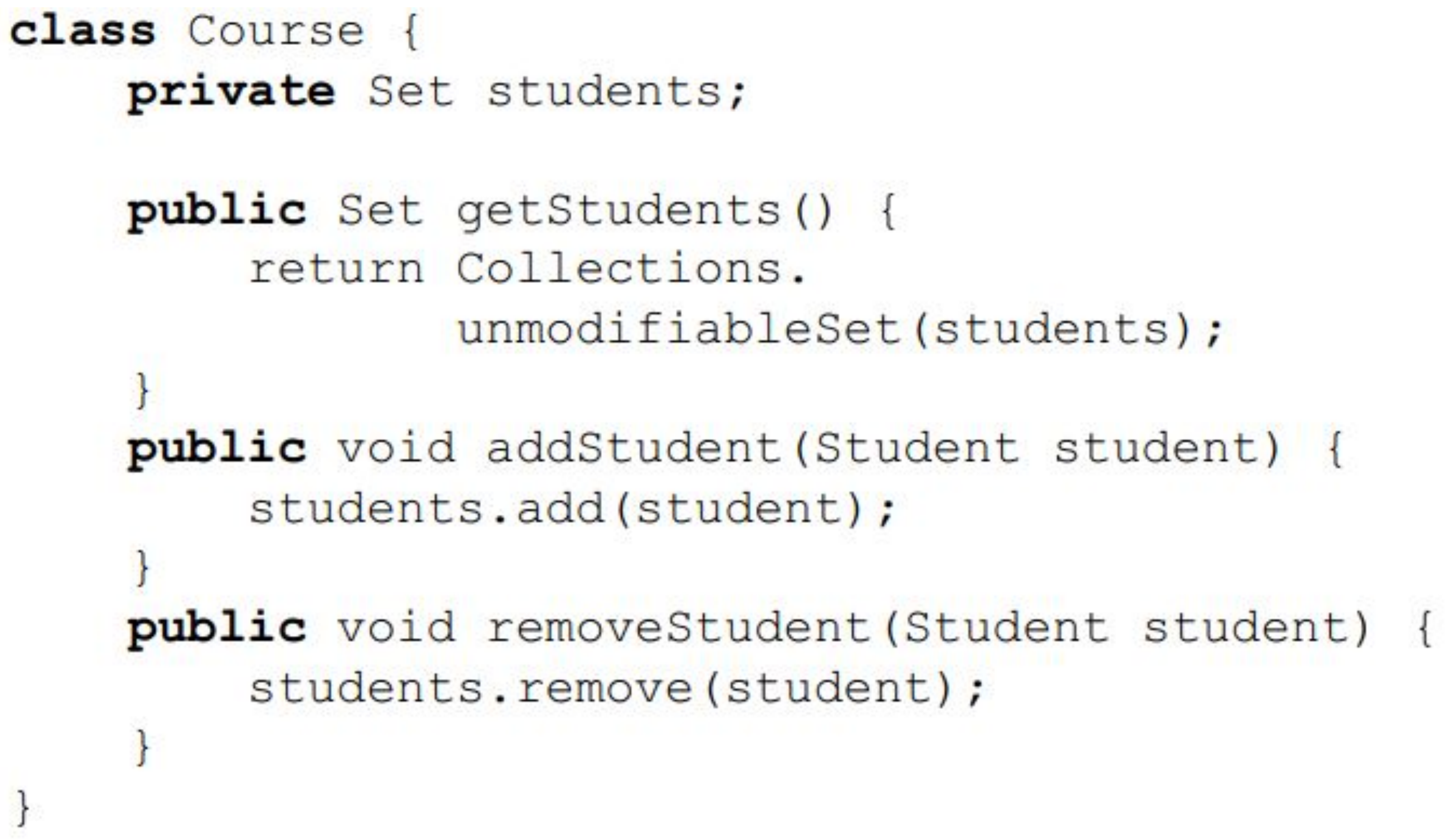


InputStream is an abstract class with several concrete subclasses.

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
- Information hiding implies encapsulation and abstraction. You are hiding details which creates an abstraction.
- 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.

EXAMPLE OF 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);
    }
}
```

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.)

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.

DEPENDENCY INVERSION PRINCIPLE (DIP)

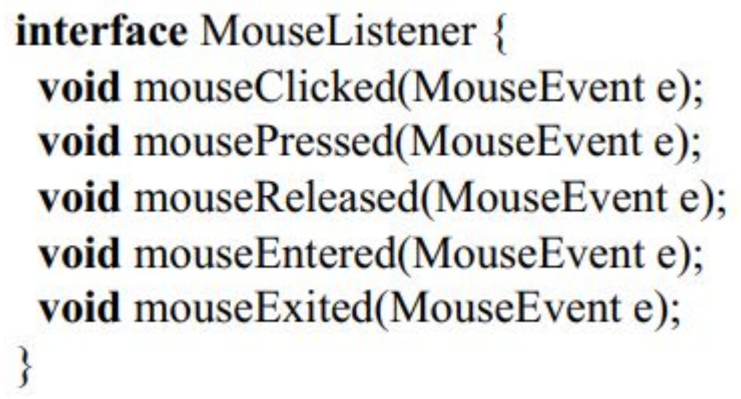
- The DIP formalizes the general design concept of Inversion of Control (IoC).
- Both are sometimes called the Hollywood Principle because they describe a phenomena where the reused component embraces the Hollywood cliché, “Don't call us, we'll call you.”
- High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

INTERFACE SEGREGATION PRINCIPLE (ISP)

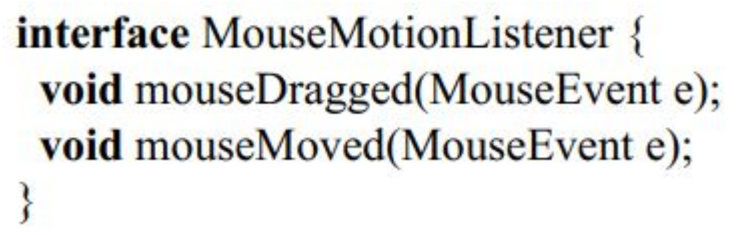
- In short, the message of the ISP is: avoid "fat" interfaces. An interface is considered "fat" if it attracts clients interested in only a portion of the methods offered by the interface and different clients are interested in different portions of the interface.
- The suggested alternative to "fat" 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 (Program to an interface, not an implementation)

INTERFACE SEGREGATION PRINCIPLE (ISP)

- 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

- 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