# Software Development Lifecycle Object Oriented Design using UML Design Review

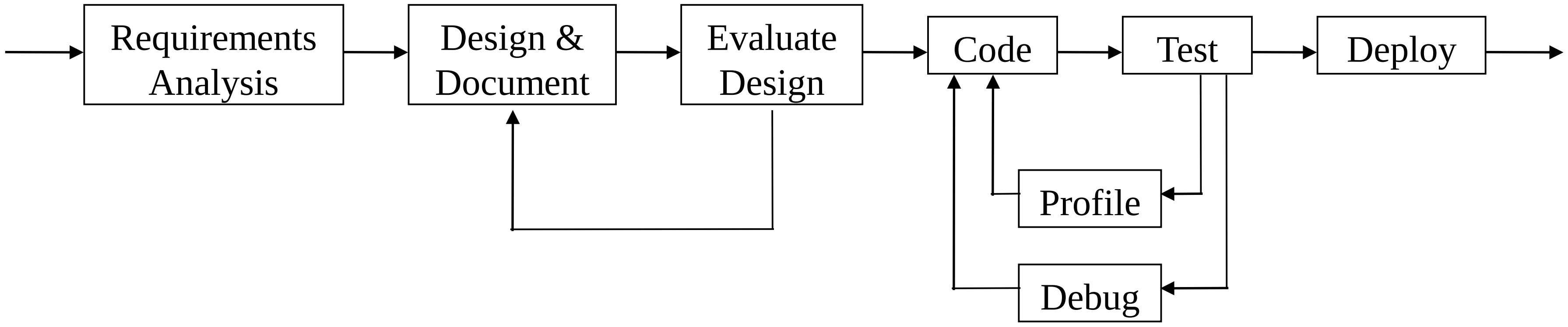
CSE260, Computer Science B: Honors

Stony Brook University

http://www.cs.stonybrook.edu/~cse260

# Software Development Life Cycle?

- Using well proven, established processes



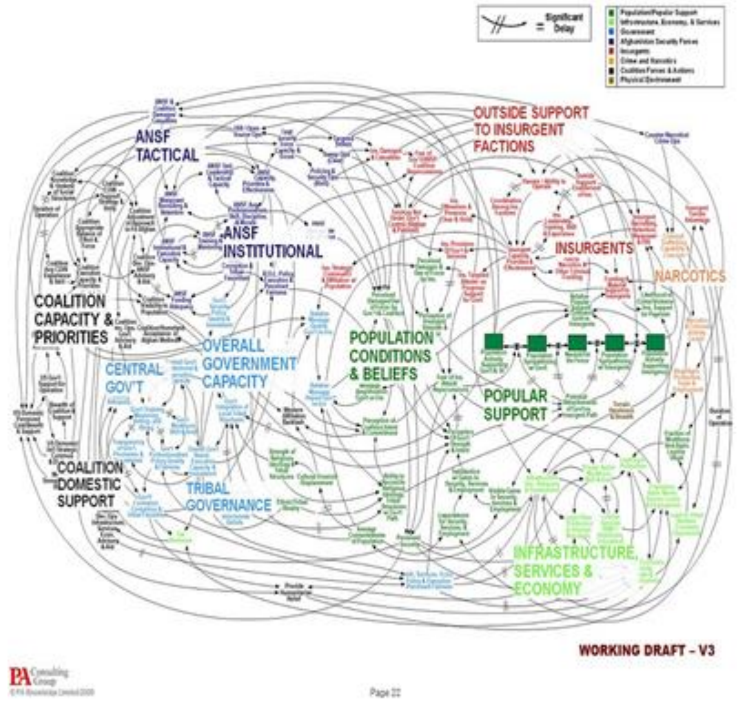
  - preferably while taking advantage of good tools

# Software Development Lifecycle The LONG… answer

- The *methodology* for **constructing** and **measuring** software systems of high quality.
- What properties make a software system high quality?
  - correctness
  - efficiency
  - ease of use (by other programmers in the case of frameworks)
  - reliability/robustness
  - maintainability
  - modifiability
  - extensibility
  - scalability

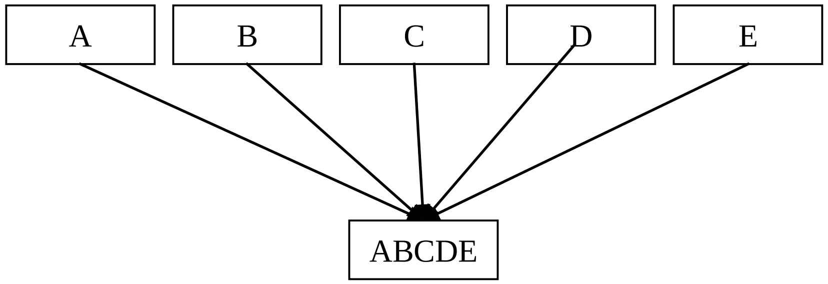
# Klocs (1,000s Source lines of code)

- Software Development Lifecycle helps us **measure** code
- As programs get larger, the high quality software goals become much more difficult to achieve.
  - Why?
    - program complexity
    - team complexity


ANSF TACTICAL
OUTSIDE SUPPORT TO INSURGENT FACTIONS
ANSF INSTITUTIONAL
INSURGENTS
NARCOTICS
COALITION CAPACITY & PRIORITIES
OVERALL GOVERNMENT CAPACITY
CENTRAL GOV'T
POPULATION CONDITIONS & BELIEFS
POPULAR SUPPORT
COALITION DOMESTIC SUPPORT
TRIBAL GOVERNANCE
INFRASTRUCTURE, SERVICES & ECONOMY
WORKING DRAFT - V3

# Software Development Lifecycle

- Other Steps to Consider:
  - Software Integration:
    - Done in large projects
    - Combine developed software into a cohesive unit



  - Software Maintenance:
    - Follows Deployment
    - Monitoring and Updating deployed software

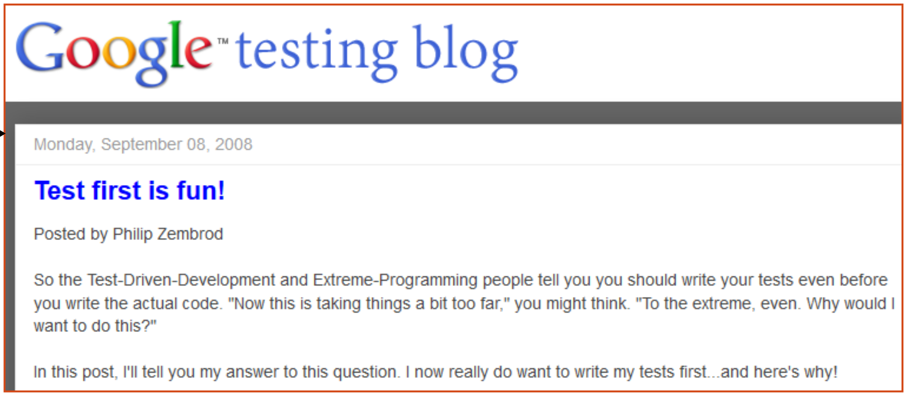
# Software maintenance

- Follows Deployment
- Monitoring and Updating deployed software

# Software Development Lifecycle

- Waterfall Model:
  - Many variations:

1. Requirements Analysis
2. Design
3. Evaluate Design
4. Code
5. Test, Debug, & Profile Components
6. Integrate
7. Test, Debug, & Profile Whole Program
8. Deploy
9. Maintain


Google™ testing blog

Monday, September 08, 2008

**Test first is fun!**

Posted by Philip Zembrod

So the Test-Driven-Development and Extreme-Programming people tell you you should write your tests even before you write the actual code. "Now this is taking things a bit too far," you might think. "To the extreme, even. Why would I want to do this?"

In this post, I'll tell you my answer to this question. I now really do want to write my tests first...and here's why!

# Software Development Lifecycle

- There are other models:
  - Agile Programming
  - Extreme Programming
  - Pair Programming
  - Etc.

# Software Development Lifecycle

- Software Jobs:
  - Programmers = the most time consuming job in software development
    - Additionally, you should know ***how to design, program, test, debug software***
  - Other types of jobs beside programmers:
    - Designer
    - Database, Network, Security Administrator
    - Tester
    - Project Leader
    - Manager
    - Documentation developer / Instructor
    - **Founder/CEO**
  - NOTE: designers & programmers on a project may not be the same people!

## Software Engineering Basics

- Important Principles for creating a Software Solution:
  - First, define the problem
  - Design, then code
  - Always Provide Feedback
- Learn a methodology for constructing software systems of high quality.

## What properties make a software system of *high quality*?

- Correctness
- Efficiency
- Ease of use
  - for the user
  - for other programmers using your framework
- Reliability/robustness
- Reusability (i.e., code reuse with slight or no modification)
- Extensibility
- Scalability (i.e., to handle a growing amount of work in a capable manner)
- Maintainability, Readability, Modifiability, Testability, etc.

## What properties make a software system of *high quality*?

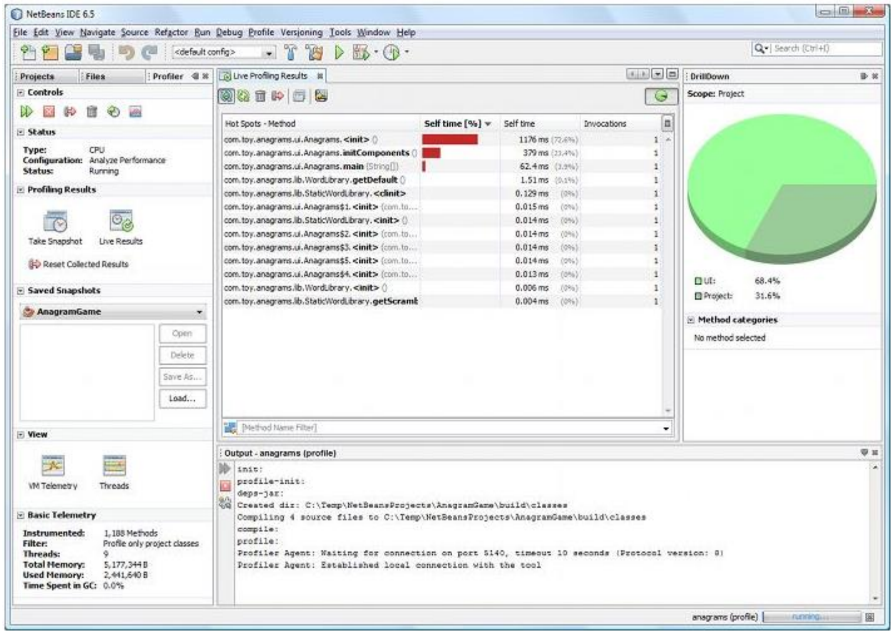
- Correctness (think of GE or IBM large engineering systems)
- Efficiency (think of Google Search)
- Ease of use
  - for the user (think of Apple UI and products)
  - for other programmers using your framework (see MS Visual…)
- Reliability/robustness (think of NASA software)
- Reusability (see Apache Software Foundation software, e.g. HTTP server)
- Extensibility (see Android OS growth to most popular mobile platform)
- Scalability (think of Oracle DBs)
- Maintainability, Readability, Modifiability, Testability, etc.

# Correctness

- Does the program perform its intended function?
  - And does it produce the correct results?
- This is not just an implementation (coding) issue
  - Correctness is a function of the problem definition
- A flawed Requirements Analysis results in a flawed Design
- A flawed Design results in a flawed program
  - Garbage In – Garbage Out

## Efficiency

- Plan for efficiency
  - wisely choose your data structures & algorthms (including their complexity, e.g., O(N)) in the design phase.
  - tools & technologies too.
- Does the program meet user performance expectations?
- If not, find the bottlenecks
  - done after implementation
  - called *profiling*



Status

| | |
|---|---|
| Type: | CPU |
| Configuration: | Analyze Performance |
| Status: | Running |

Basic Telemetry

| | |
|---|---|
| Instrumented: | 1,188 Methods |
| Filter: | Profile only project classes |
| Threads: | 9 |
| Total Memory: | 5,177,344 B |
| Used Memory: | 2,441,640 B |
| Time Spent in GC: | 0.0% |

| Hot Spots - Method | Self time [%] | Self time | Invocations |
|---|---|---|---|
| com.toy.anagrams.ui.Anagrams.<init> () | | 1176 ms (72.6%) | 1 |
| com.toy.anagrams.ui.Anagrams.initComponents () | | 379 ms (23.4%) | 1 |
| com.toy.anagrams.ui.Anagrams.main (String[]) | | 62.4 ms (3.9%) | 1 |
| com.toy.anagrams.lib.WordLibrary.getDefault () | | 1.51 ms (0.1%) | 1 |
| com.toy.anagrams.lib.StaticWordLibrary.<clinit> | | 0.129 ms (0%) | 1 |
| com.toy.anagrams.ui.Anagrams$1.<init> (com.to... | | 0.015 ms (0%) | 1 |
| com.toy.anagrams.lib.StaticWordLibrary.<init> () | | 0.014 ms (0%) | 1 |
| com.toy.anagrams.ui.Anagrams$2.<init> (com.to... | | 0.014 ms (0%) | 1 |
| com.toy.anagrams.ui.Anagrams$3.<init> (com.to... | | 0.014 ms (0%) | 1 |
| com.toy.anagrams.ui.Anagrams$5.<init> (com.to... | | 0.014 ms (0%) | 1 |
| com.toy.anagrams.ui.Anagrams$4.<init> (com.to... | | 0.013 ms (0%) | 1 |
| com.toy.anagrams.lib.WordLibrary.<init> () | | 0.006 ms (0%) | 1 |
| com.toy.anagrams.lib.StaticWordLibrary.getScramb | | 0.004 ms (0%) | 1 |

Output - anagrams (profile)

```
init:
profile-init:
deps-jar:
Created dir: C:\Temp\NetBeansProjects\AnagramGame\build\classes
Compiling 4 source files to C:\Temp\NetBeansProjects\AnagramGame\build\classes
compile:
profile:
Profiler Agent: Waiting for connection on port 5140, timeout 10 seconds (Protocol version: 8)
Profiler Agent: Established local connection with the tool
```

## Ease of Use for End User

- Is the GUI easy to learn to use?
  - a gently sloped learning curve
- What makes a GUI easy to use?
  - familiar GUI structures
  - familiar icons when possible instead of text
  - components logically organized & grouped
  - appealing to look at
    - colors, alignment, balance, etc.
  - forgiving of user mistakes
  - help, tooltips, and other cues available
  - etc.

## Ease of Use for other Programmers

- In particular for frameworks and tools
  - the Java API is developed to be easy to use
- Should you even build a framework?
  - Yes, you will be a software developer.
- What makes a framework easy to use?
  - logical structure
  - naming choices (classes, methods, etc.)
  - flexibility (usable for many purposes)
  - feedback (exceptions for improper use)
  - documentation (APIs & tutorials)
  - etc.

## Reliability/Robustness

- Does your program:
  - anticipate erroneous input?
  - anticipate all potential program conditions?
  - handle erroneous input intelligently?
    - think about this in the design stage
  - provide graceful degradation?
    - *Graceful degradation* (or *Fault-tolerance*) is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components.
      - If an error condition occurs in your program, should your program:
        - crash?, exit?, notify the user and exit?, provide an approximated service? Not always possible to save it.
      - For example: What should Web Browsers do with poorly formatted HTML?

## Feedback

- Provide feedback to End users due to: bad input, equipment failure, missing files, etc.
  - How?
    - popup dialogs, highlighting (red text in Web form), etc.
- Provide feedback to other programmers using your framework due to: passing bad data, incorrect initialization, etc.
  - How?
    - exception throwing, error value returning, etc.

# Flexibility in a Framework

- Programmers need to know:
  - when and why things in a framework might go wrong

AND

  - when and why things in a framework do go wrong
- How?
  - customized response:
    - **`System.out.println`** notifications
    - **`GUI`** notifications
    - Web page generated and sent via Servlet notification
    - etc.

## Reusability

- Code serving multiple purposes.
- Who cares?
  - management does
    - avoid duplication of work (save $)
  - software engineering does
    - avoid duplication of work (save time & avoid mistakes)
- How can we achieve this?
  - careful program decomposition (from methods to classes and packages)
  - separate technology-dependent components

## Extensibility

- Can the software easily be extended?
  - can it be used for other purposes
    - plug-ins
    - exporters
    - add-ons
- Extensibility Example:
  - In NetBeans, Tools → Plugins
    - Anyone can make a plugin
    - Download, install, and use
  - In Eclipse IDE, Help → Install New Software plugin

## Scalability

- How will the program perform when we increase:
  - # of users/connections
  - amount of data processed
  - # of geographic locations users are from
- A function of design as well as technology

## More high quality software properties

- Maintainability
- Readability
- Modifiability
- Testability
- All of these, as with the others, must be considered early in design

## Design, then develop

- We will design all classes before coding
  - not easy to do
  - UML is used for software design
- You cannot design a system unless you really understand the necessary technology
  - designs cannot be created without testing

## Design Approaches

- Have other "*similar*" problems been solved?
  - Do *design patterns* exist to help?
- Employ:
  - data-driven design
    - data-driven programming is a programming paradigm in which the program statements describe the data to be matched and the processing required rather than defining a sequence of steps to be taken.
  - top-down design
    - a top-down approach is the breaking down of a system to gain insight into its compositional sub-systems

## Data-driven Design

- From the problem specification, extract:
    - nouns (they are objects, attributes of objects)
    - verbs (they are methods)
- Divide data into separate logical, manageable groupings
    - these will form your objects
- Note needs for data structures or algorithms
    - design your data management classes early on

## Data-driven Design gives the Class relationships

- Think *data flow*:
  - What *HAS* what?
  - What *IS* what?
  - What *USES* what?
  - Where should data go?
  - Static or non-static?
- Design patterns will help us make these decisions
- Bottom line: think modular
  - no 1000 line classes or 100 line methods

## Modularity

- How reusable are your classes?
  - can they be used in a future project?
- Think of programmers, not just users
- Can individual classes be easily separated and re-used?
  - Separate Functionality from Presentation
  - Separate Data from Mechanics

## Functionality vs. Presentation

The *state manager*:

- manages the state of one or more user interface controls such as text fields, OK buttons, radio buttons, etc. in a graphical user interface.
  - In this user interface programming technique, the state of one UI control depends on the state of other UI controls.
- classes that do the work of managing data & enforcing rules on that data

- Why separate the state management and the UI?
  - so we can design several different UIs for a state manager
  - so we can change the state management without changing the UI
  - so we can change the UI without changing the state manager
  - reuse code that is proven to work
  - This is a common principle throughout GUI design
    - even for Web sites (separate content from presentation)
    - **different programmers for each task**

## Choosing Data Structures

- Internal data structures
  - What is the natural representation of the given data?
  - Trade-offs: Setup vs. access speeds
  - Keep data ordered?
    - Ordered by what?
    - Which access algorithms?

# UML Diagrams

- UML - Unified Modeling Language diagrams are used to design object-oriented software systems
  - represent systems visually = Client-friendly!
  - provides a system architecture
  - makes coding more efficient and system more reliable
  - diagrams show relationships among classes and objects
- Can software engineering be automated?
  - Visual programming
  - Patterns & frameworks
  - Computer-Aided Software Engineering (CASE) tools

## Types of UML Diagrams

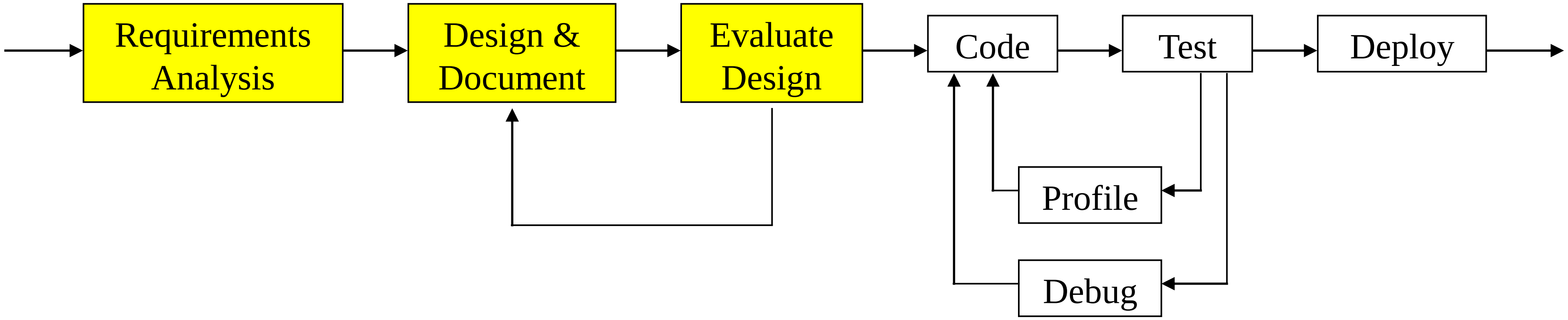
- Types of UML diagrams that we will make in CSE260:
  - Use Case Diagram
  - Class Diagram
  - Sequence Diagram
- Other types of UML diagrams (you will make in our CSE308):
  - State, Activity, Collaboration, Communication, Component, & Deployment Diagrams

## What will we use UML Diagrams for?

- Use Case Diagrams
  - describe all the ways users will interact with the program
- Class Diagrams
  - describe all of our classes for our app
- Sequence Diagrams
  - describe all event handling

# Software Development Life Cycle

- Requirements Analysis & design stages:



- Correctness, Efficiency, Ease of use, Reliability/robustness, Reusability, Maintainability, Modifiability, Testability, Extensibility, Scalability
  - do we consider these properties in the implementation stages?
    - Little because it is too late to make a big impact.

## Where to begin?

- Understand and Define the problem
  - the point of a requirements analysis
  - What are system input & output?
  - How will users interact with the system?
  - What data must the system maintain?
- Generate a problem specification document
  - defines the problem
  - defines what needs to be done to solve the problem

# Requirements Analysis

- i.e. Software Specification (or spec.)
  - A textual document
  - It serves two roles. It:
    - defines the problem to be solved
    - explains how to solve it
  - This is the input into the software design stage
- **What goes in a requirements analysis (RA)?**
  - The why, where, when, what, how, and who:
    - Why are we making this software?
    - Where and when will it be created?
    - What, exactly, are we going to make?
    - How are we going to make it?
    - Who will be performing each role?

## Requirements Analysis

- **What really goes in a RA?**
  - Detailed descriptions of all:
    - necessary data (including how to query it, views, forms, inserts)
    - program input and output
    - GUI screens & controls
    - user actions and program reactions
- **Where do you start?**
  - Interviews with the end users
    - What do they need?
    - What do they want?

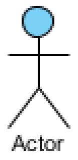
## UML *Use Case Diagrams*

- A set of scenarios that describe an interaction between a user and a system
- Done first in a project design
  - helps you to better understand the system requirements
- To draw a Use Case Diagram:
  - List a sequence of steps a user might take in order to complete an action.
  - Example actor: a user placing an order with a sales company

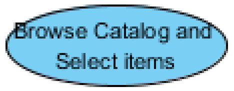
# UML Use Case Diagrams

- Human Actor: Stick figure with name underneath.

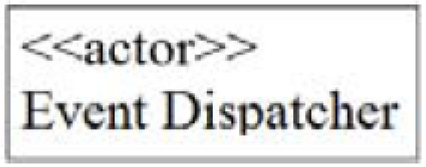
Name usually identifies type of actor.



- Use Case: Oval enclosing name of use case.

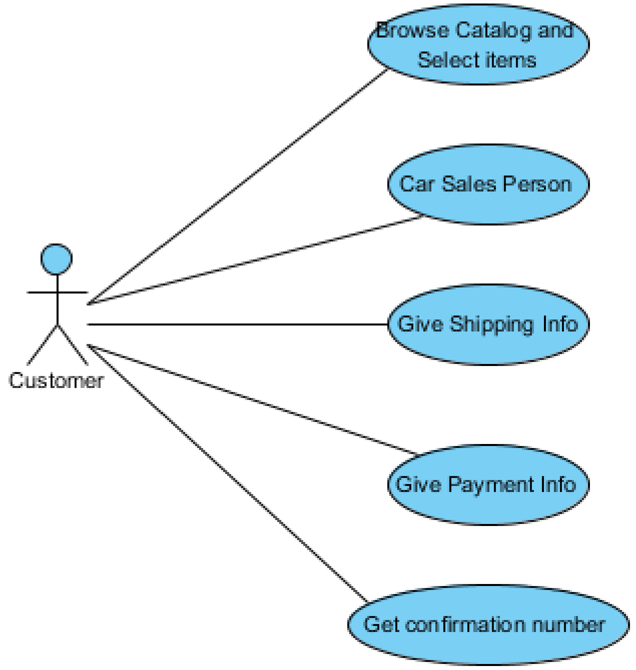


- Non-Human Actor: Stick figure, or a rectangle enclosing the stereotype <<actor>> and the name of the actor. A stereotype indicates the type of UML element (when it isn't evident from the shape).



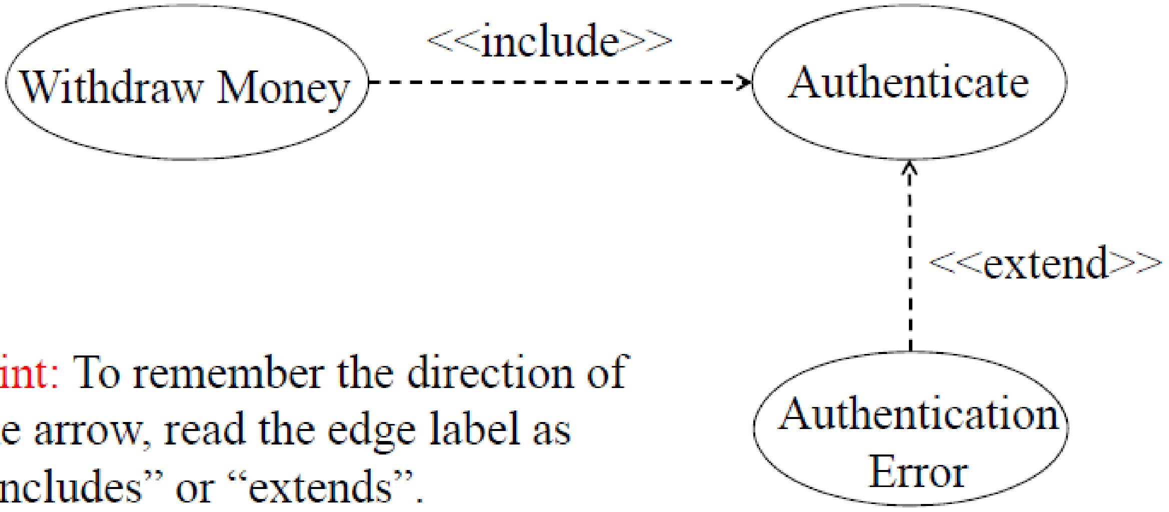
# UML Use Case Diagrams

- Relationships Between Actors and Use Cases:
  - Solid edge between an actor A and a use case U means that actor A participates in use case U.



# UML Use Case Diagrams

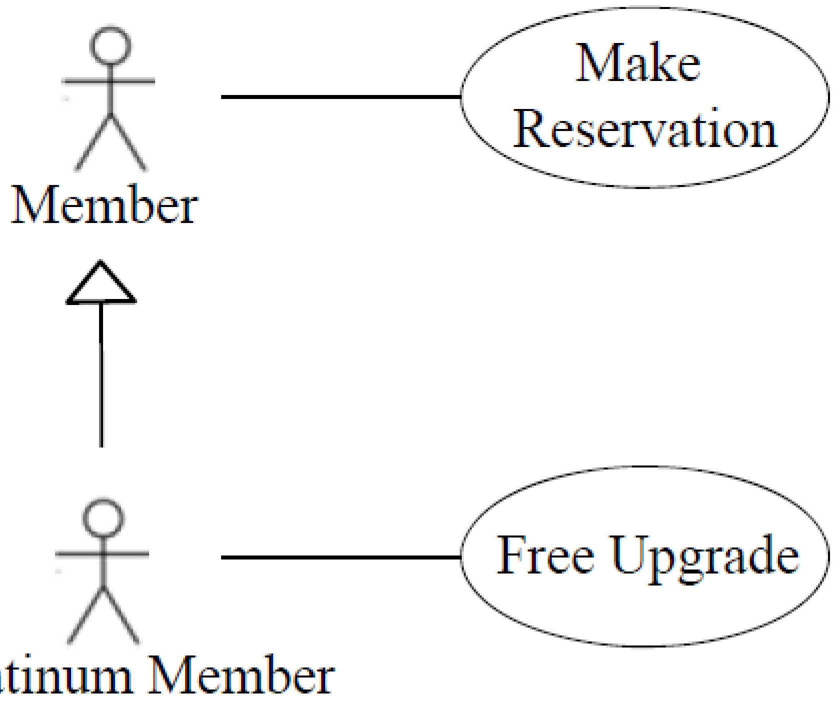
- Relationships Between Use Cases:
  - Include: dashed arrow labeled <<include>> from use case U1 to use case U2 means U2 is part of the primary flow of events of U1.
  - Extend: dashed arrow labeled <<extend>> from use case U2 to use case U1 means U2 is part of a secondary flow of events of U1.



Hint: To remember the direction of the arrow, read the edge label as "includes" or "extends".

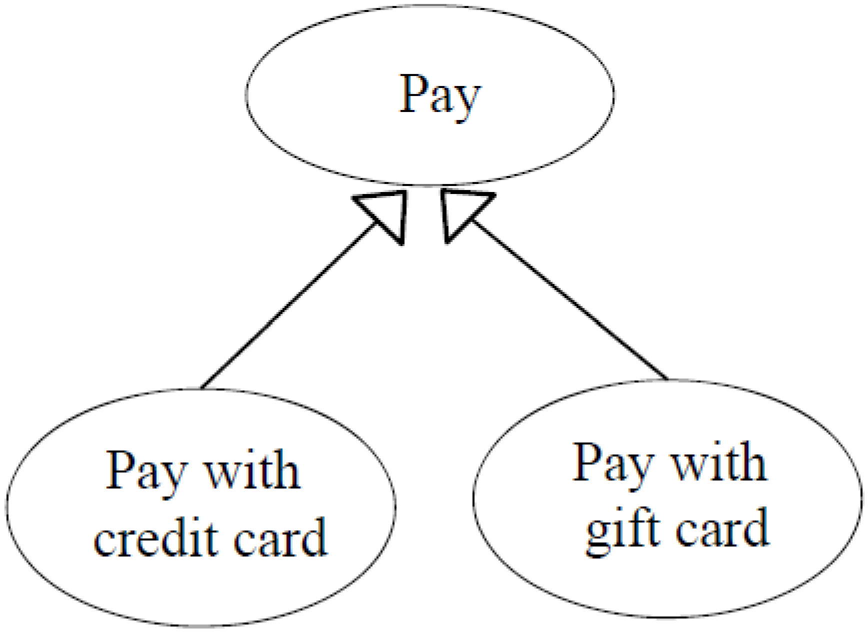
# Relationships Between Actors

- Generalization: Solid line with triangular arrowhead from actor A1 to actor A2 means that A2 is a generalization of A1. This implies that A1 participates in all use cases that A2 participates in. Generalization is similar to inheritance.

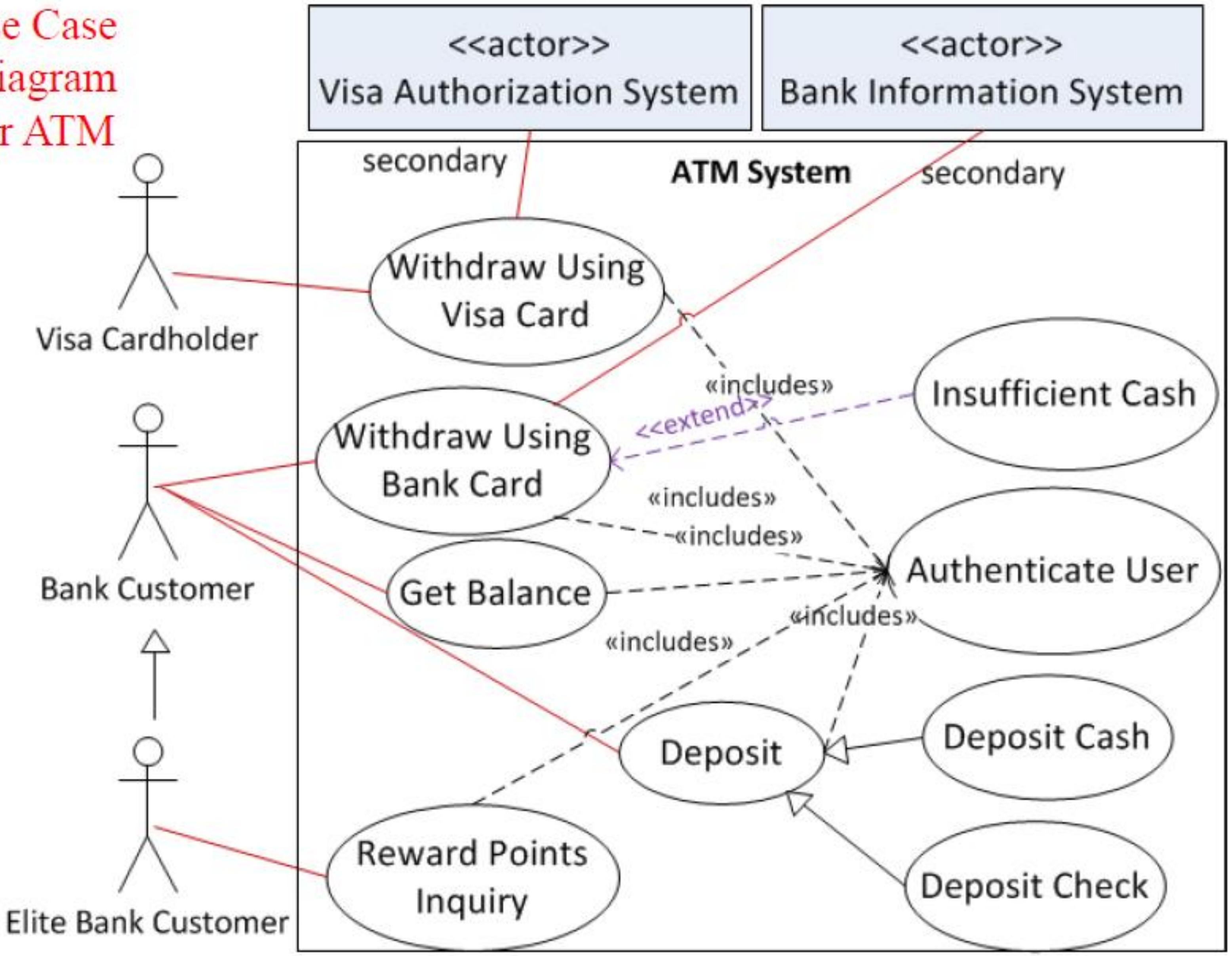


## Relationships Between Use Cases

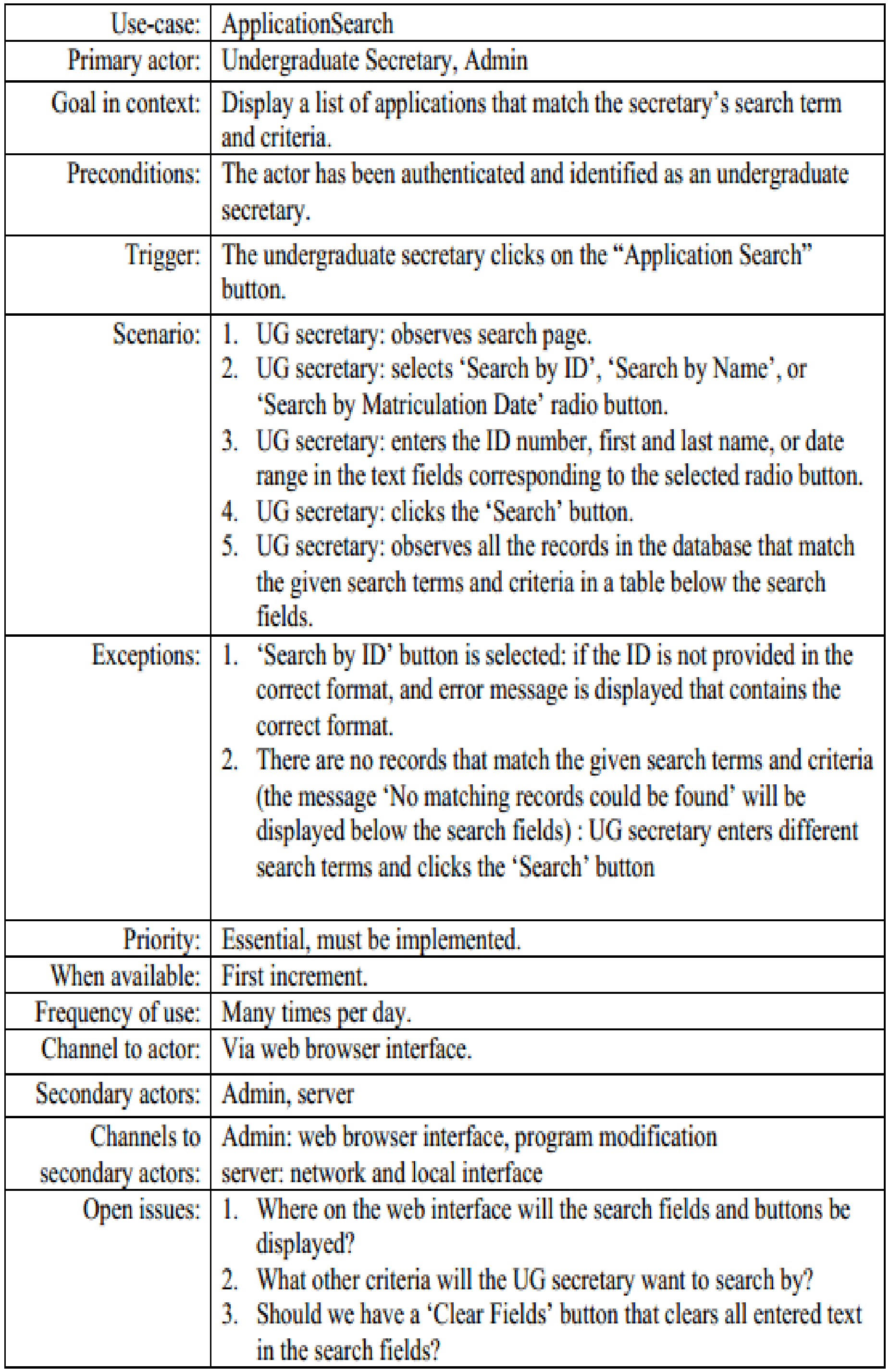
- Generalization: Solid line with triangular arrowhead from use case U1 to use case U2 means that U2 is a generalization of U1 (equivalently, U1 is a specialized version of U2). Generalization is similar to inheritance.



Use Case Diagram For ATM



## Formal UML Use Case Diagram



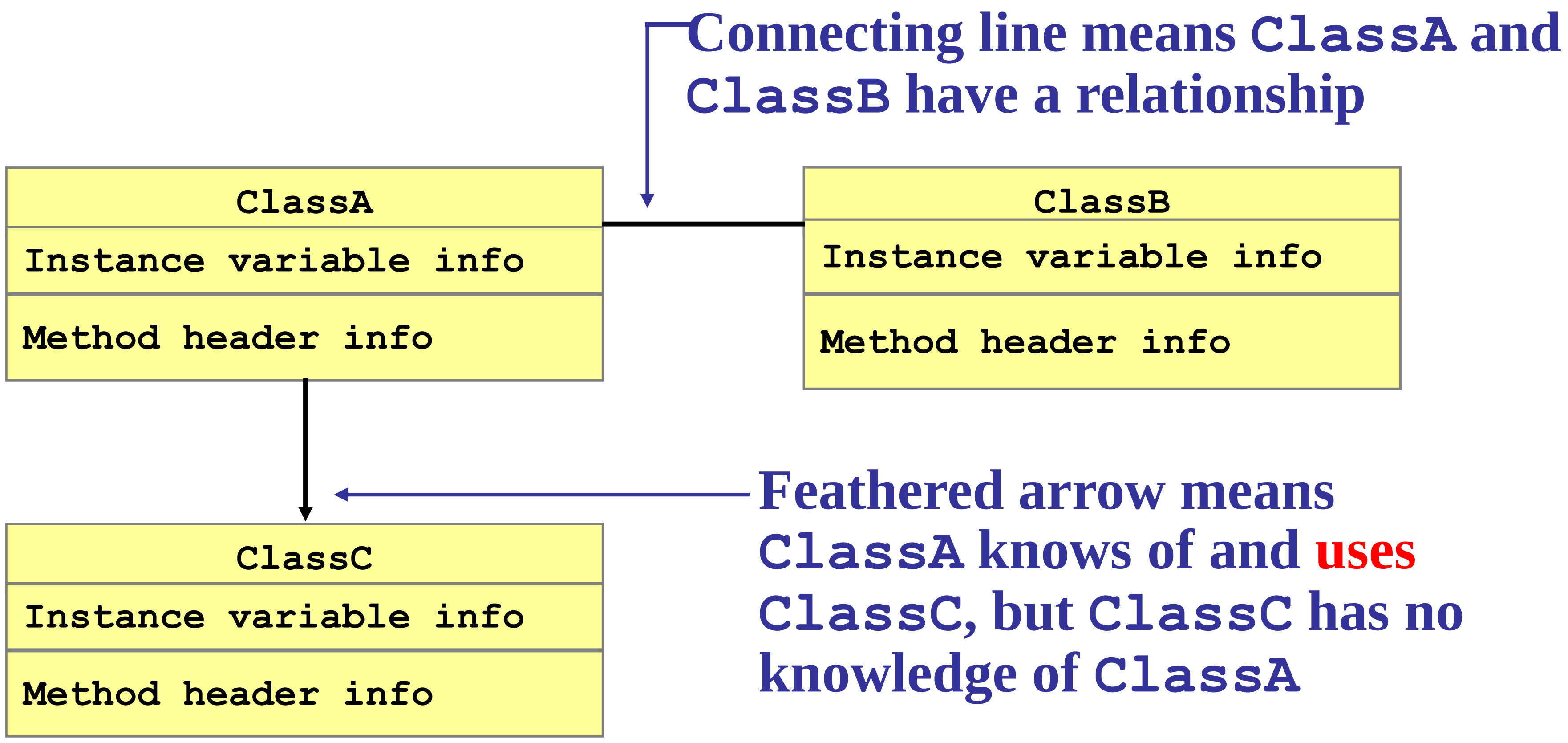
| | |
|---|---|
| Use-case: | ApplicationSearch |
| Primary actor: | Undergraduate Secretary, Admin |
| Goal in context: | Display a list of applications that match the secretary's search term and criteria. |
| Preconditions: | The actor has been authenticated and identified as an undergraduate secretary. |
| Trigger: | The undergraduate secretary clicks on the "Application Search" button. |
| Scenario: | 1. UG secretary: observes search page.<br>2. UG secretary: selects 'Search by ID', 'Search by Name', or 'Search by Matriculation Date' radio button.<br>3. UG secretary: enters the ID number, first and last name, or date range in the text fields corresponding to the selected radio button.<br>4. UG secretary: clicks the 'Search' button.<br>5. UG secretary: observes all the records in the database that match the given search terms and criteria in a table below the search fields. |
| Exceptions: | 1. 'Search by ID' button is selected: if the ID is not provided in the correct format, and error message is displayed that contains the correct format.<br>2. There are no records that match the given search terms and criteria (the message 'No matching records could be found' will be displayed below the search fields) : UG secretary enters different search terms and clicks the 'Search' button |
| Priority: | Essential, must be implemented. |
| When available: | First increment. |
| Frequency of use: | Many times per day. |
| Channel to actor: | Via web browser interface. |
| Secondary actors: | Admin, server |
| Channels to secondary actors: | Admin: web browser interface, program modification<br>server: network and local interface |
| Open issues: | 1. Where on the web interface will the search fields and buttons be displayed?<br>2. What other criteria will the UG secretary want to search by?<br>3. Should we have a 'Clear Fields' button that clears all entered text in the search fields? |

## UML *Class Diagrams*

- A UML class diagram consists of one or more classes, each with sections for:
  - class name
  - instance variables
  - methods
- Lines between classes represent associations
  - Uses
  - Aggregation (HAS-A)
    - Containment
  - Inheritance (IS-A)

# UML Class Diagrams

- Show relationships between classes
  - Class associations denoted by lines connecting classes
  - A feathered arrow denotes a one-directional association



Connecting line means ClassA and ClassB have a relationship
ClassA
Instance variable info
Method header info
ClassB
Instance variable info
Method header info
ClassC
Instance variable info
Method header info
Feathered arrow means ClassA knows of and uses ClassC, but ClassC has no knowledge of ClassA

## Method and Instance Variable Descriptions

- Instance Variables Format

  **`variableName : variableType`**

  - For example: **`upValue : int`**

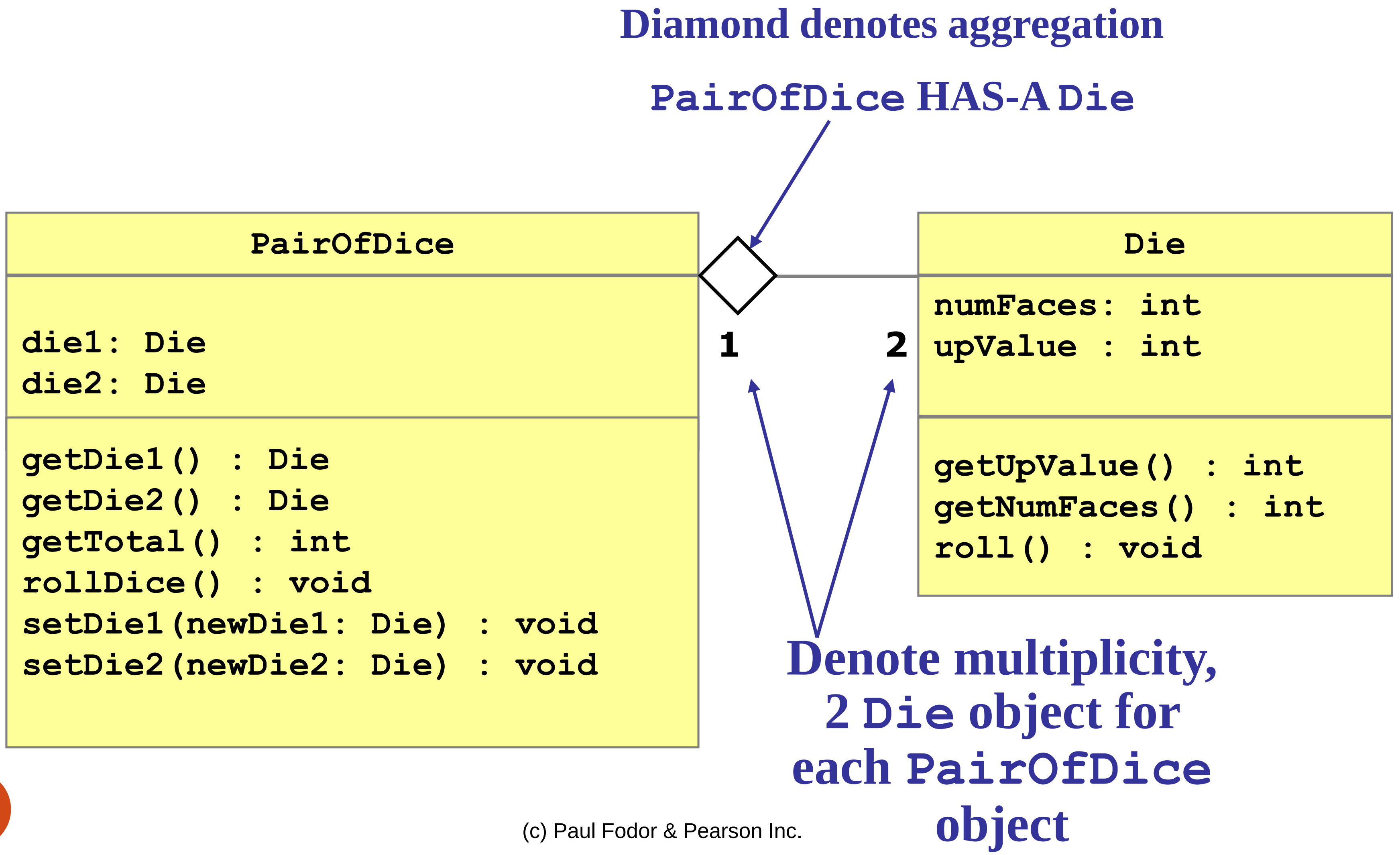
- Method Header Format

  **`methodName(argumentName:argumentType) :returnType`**

  - For example: **`setDie1(newDie1:Die):void`**
  - Underlined or $ denotes a static method or variable
    - For example: **`myStaticMethod(x:int):void`** (underlined)

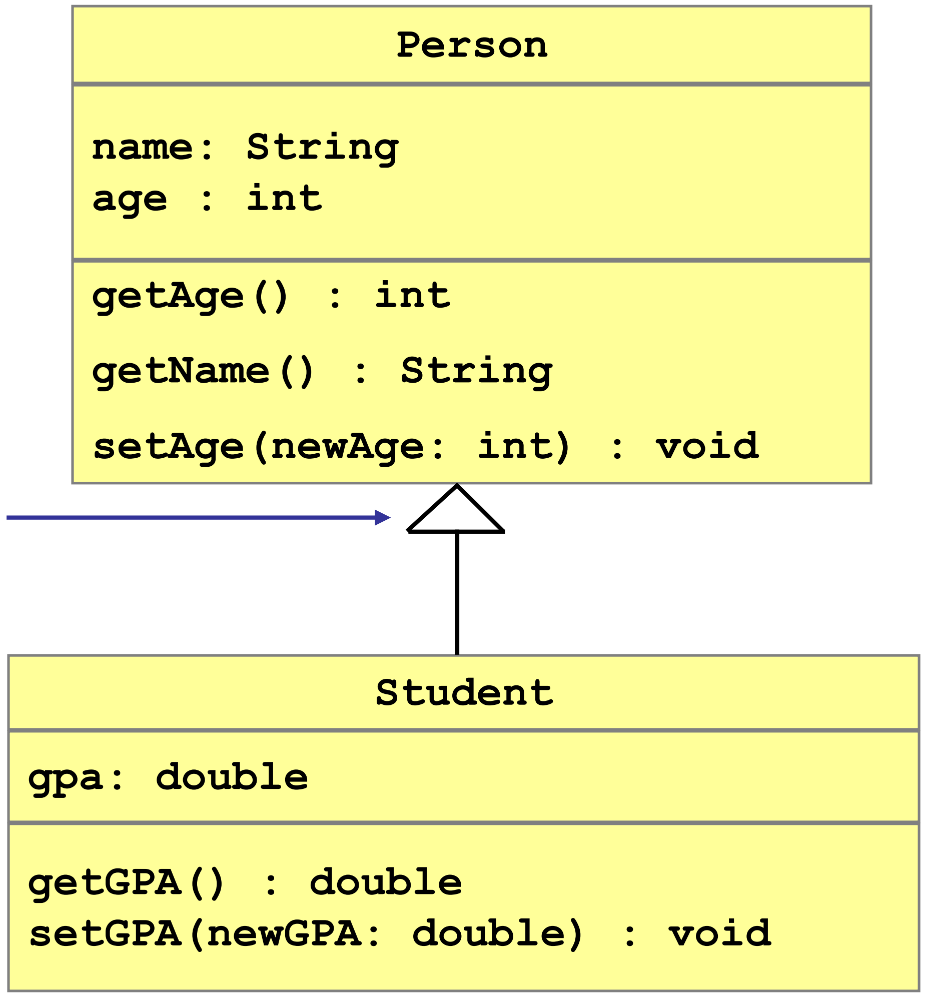
## UML Class Diagrams & Aggregation

- UML class diagram for **`PairOfDice`** & **`Die`**:



## UML Class Diagrams & Inheritance

**`public class Student extends Person`**



**Triangle denotes inheritance**

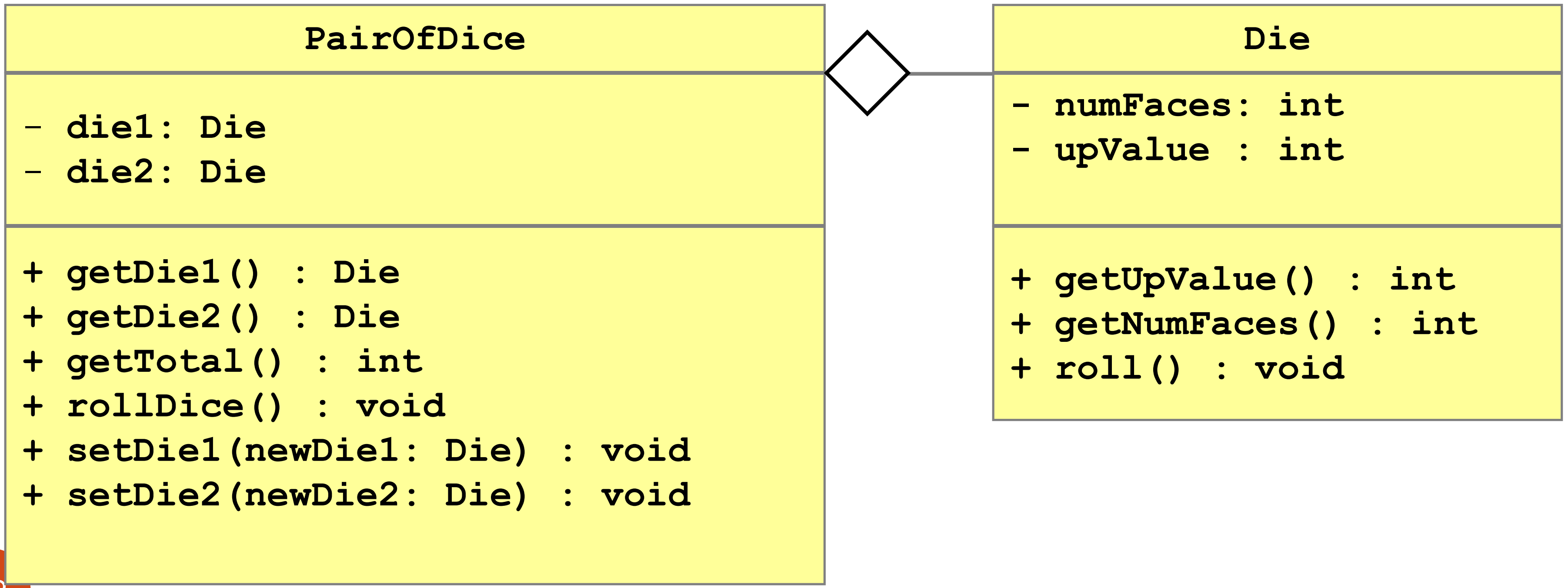
**Student IS-A Person**

# Encapsulation

- We can take one of two views of an object:
  - internal - the variables the object holds and the methods that make the object useful
  - external - the services that an object provides and how the object interacts
- From the external view, an object is an *encapsulated* entity, providing a set of specific services
- These services define the *Application Programming Interface* (*API*) to the object
  - *abstraction* hides details from the rest of the system

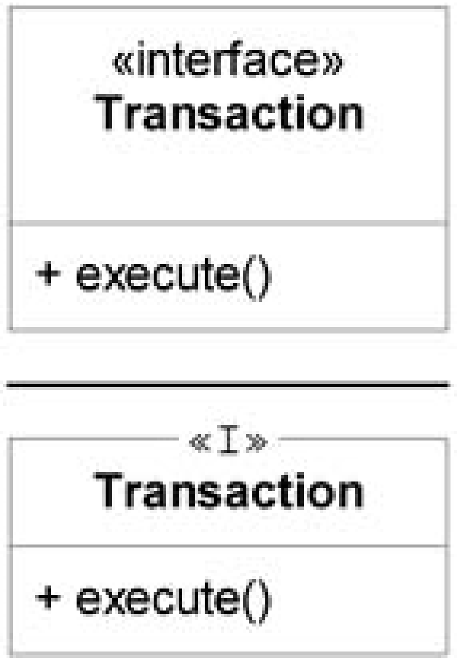
## Class Diagrams and Encapsulation

- In a UML class diagram:
  - public members can be preceded by +
  - private members are preceded by -
  - protected members are preceded by #



## Interfaces in UML

- 2 ways to denote an interface
  - <<interface>> (standard) OR <<I>>

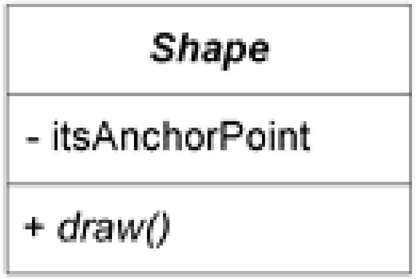


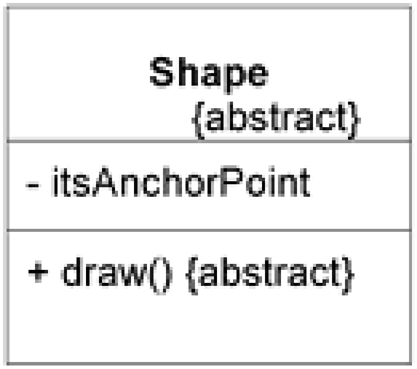
```
interface Transaction
{
   public void execute();
}
```

http://www.informit.com/articles/article.asp?p=336264&seqNum=3

# Abstract Classes in UML

- 2 ways to denote a class or method is abstract:
  - class or method name in italics, OR
  - {abstract} notation

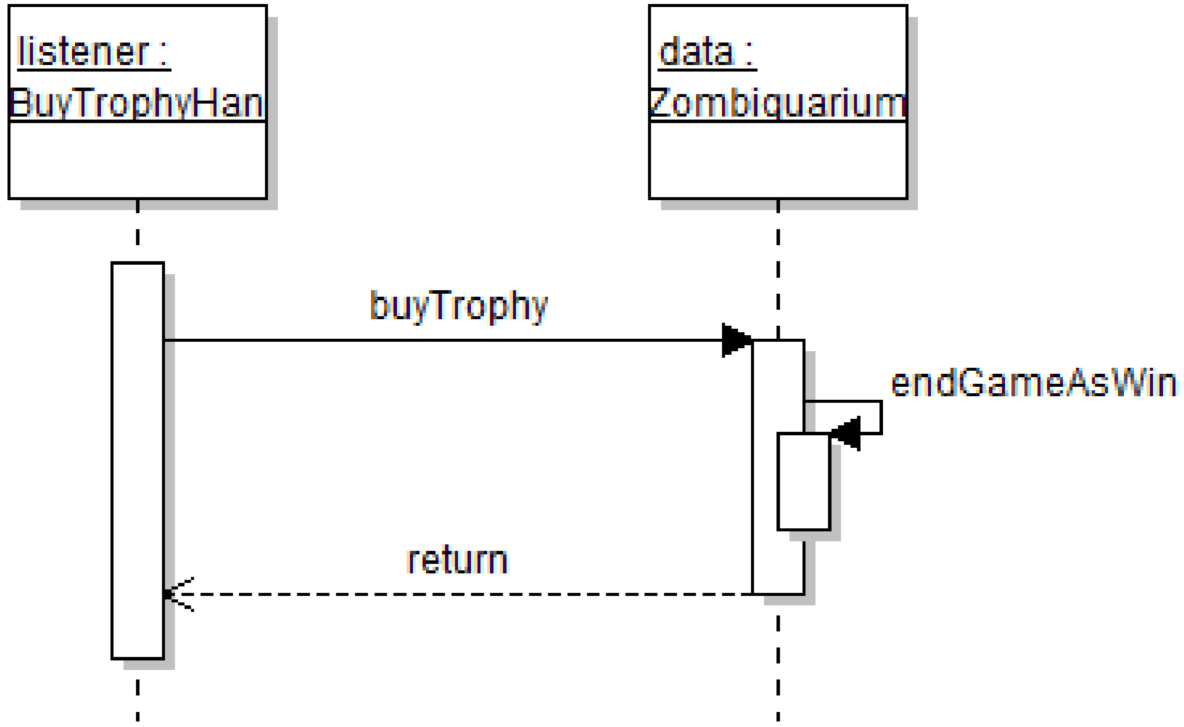




```
public abstract class Shape
{
   private Point itsAnchorPoint;
   public abstract void draw();
}
```

# UML *Sequence Diagrams*

- Demonstrate the behavior of objects in program
  - describe the objects and the messages they pass
  - diagrams are read left to right and descending



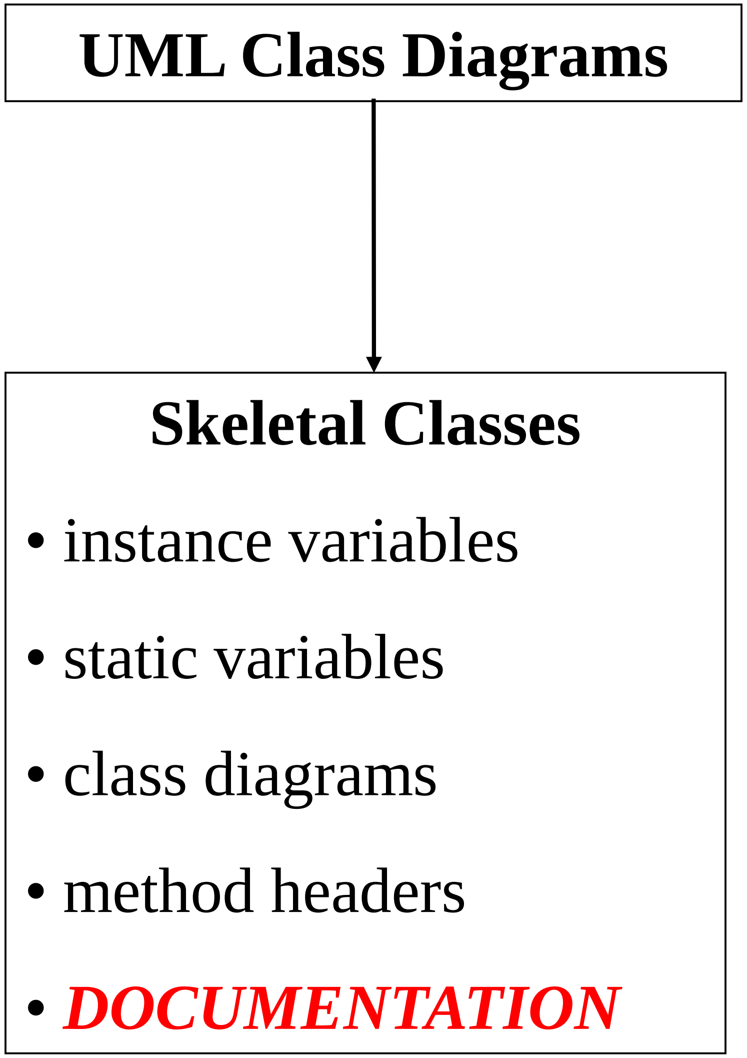
# Top-down class design

- Top-down class design strategy:
  - Decompose the problem into sub-problems (large chunks).
  - Write skeletal classes for sub-problems.
  - Write skeletal methods for sub-problems.
  - Repeat for each sub-problem.
- If necessary, go back and redesign higher-level classes to improve:
  - modularity,
  - information hiding, and
  - information flow

# Designing Methods

- Decide method signatures
  - numbers and types of parameters and return values
- Write down what a method should do (spec)
  - use top-down design
    - decompose methods into helper methods
- Use javadoc comments to describe methods
- Use method specs for implementation

# Results of Top-down class design



**UML Class Diagrams**

**Skeletal Classes**

- instance variables
- static variables
- class diagrams
- method headers
- ***DOCUMENTATION***

# Software Longevity

- The FORTRAN & COBOL programming languages are ~50 years old
  - many mainframes still use code written in the 1960s
  - software maintenance is more than ½ a project
- Moral of the story:
  - the code you write may outlive you, so make it:
    - Easy to understand
    - Easy to modify & maintain
  - software must be ready to accommodate change

# Software Maintenance

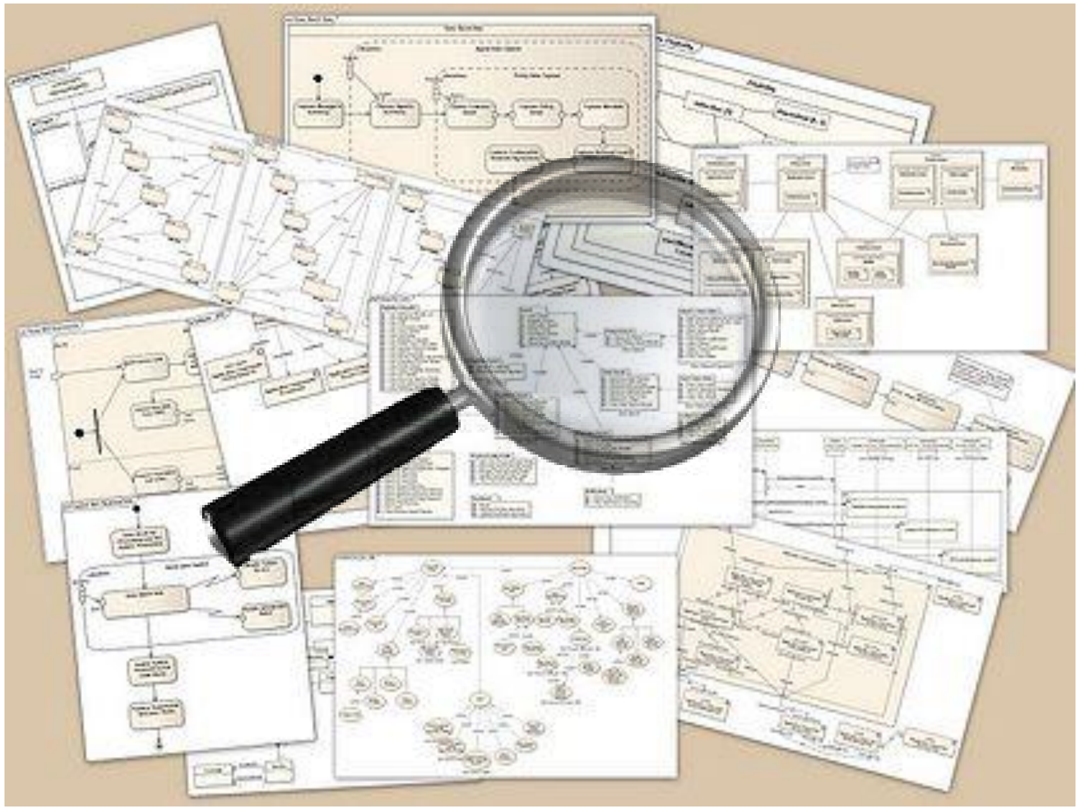
- What is software maintenance?
- Improving or extending existing software
  - incorporate new functionality
  - incorporate new data to be managed
  - incorporate new technologies
  - incorporate new algorithms
  - incorporate use with new tools
  - incorporate things we cannot think of now ☺

# Software Engineering

- Always use data driven & top-down design:
  - identify and group system data
  - identify classes, their methods and method signatures
  - determine what methods should do
  - identify helper methods
    - Write down step by step algorithms inside methods to help you!
  - document each class, method and field
  - specify all conditions that need to be enforced or checked
    - decide where to generate exceptions
    - add to documentation
  - evaluate design, and repeat above process
    - until implementation instructions are well-defined

# Evaluating a Design

- During the design of a large program, it is worthwhile to step back periodically & attempt a comprehensive evaluation of the design so far
  - called a ***design review***



## Design Reviews are not just for Software

## Who performs the design review?

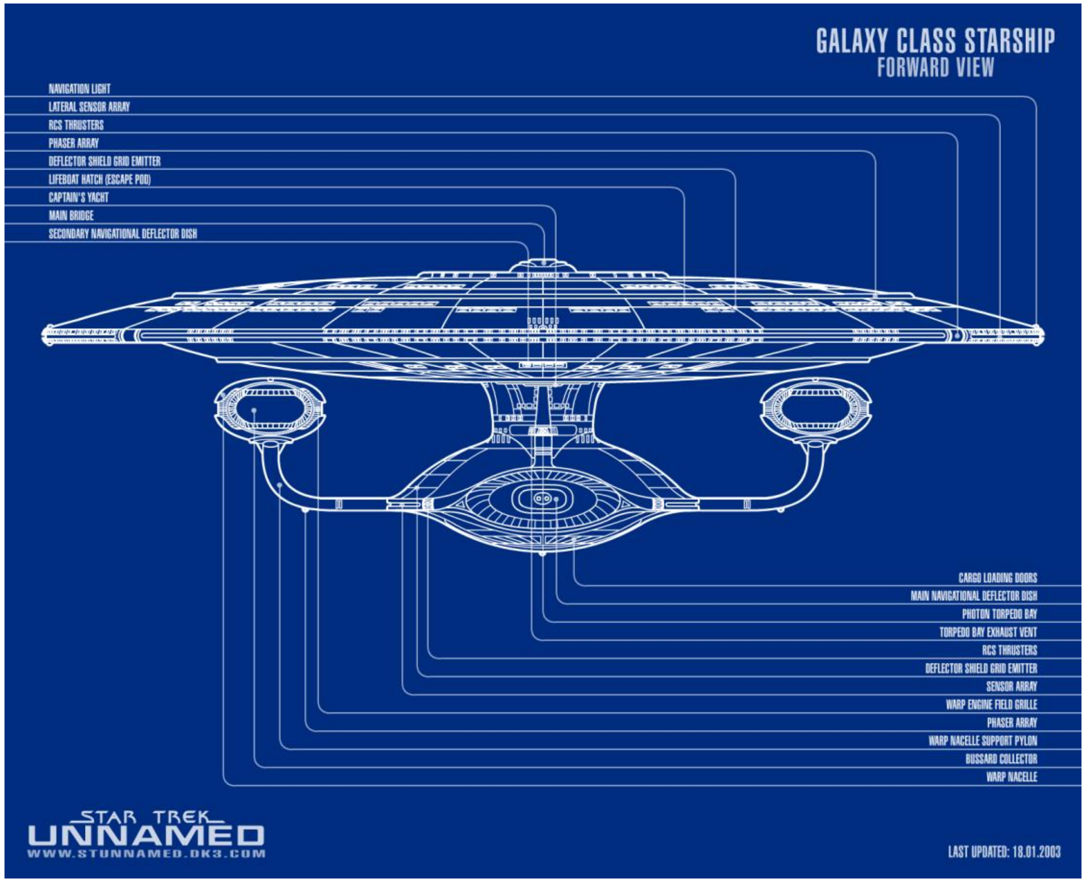
- Design review committee

- Members should include:
  - varied perspectives
  - some from the project
  - some external to the project

- All should be familiar with the design itself

# There is no perfect design

- Is the design adequate?
- Will do the job with adequate performance & cost?



# Critical Design Issues

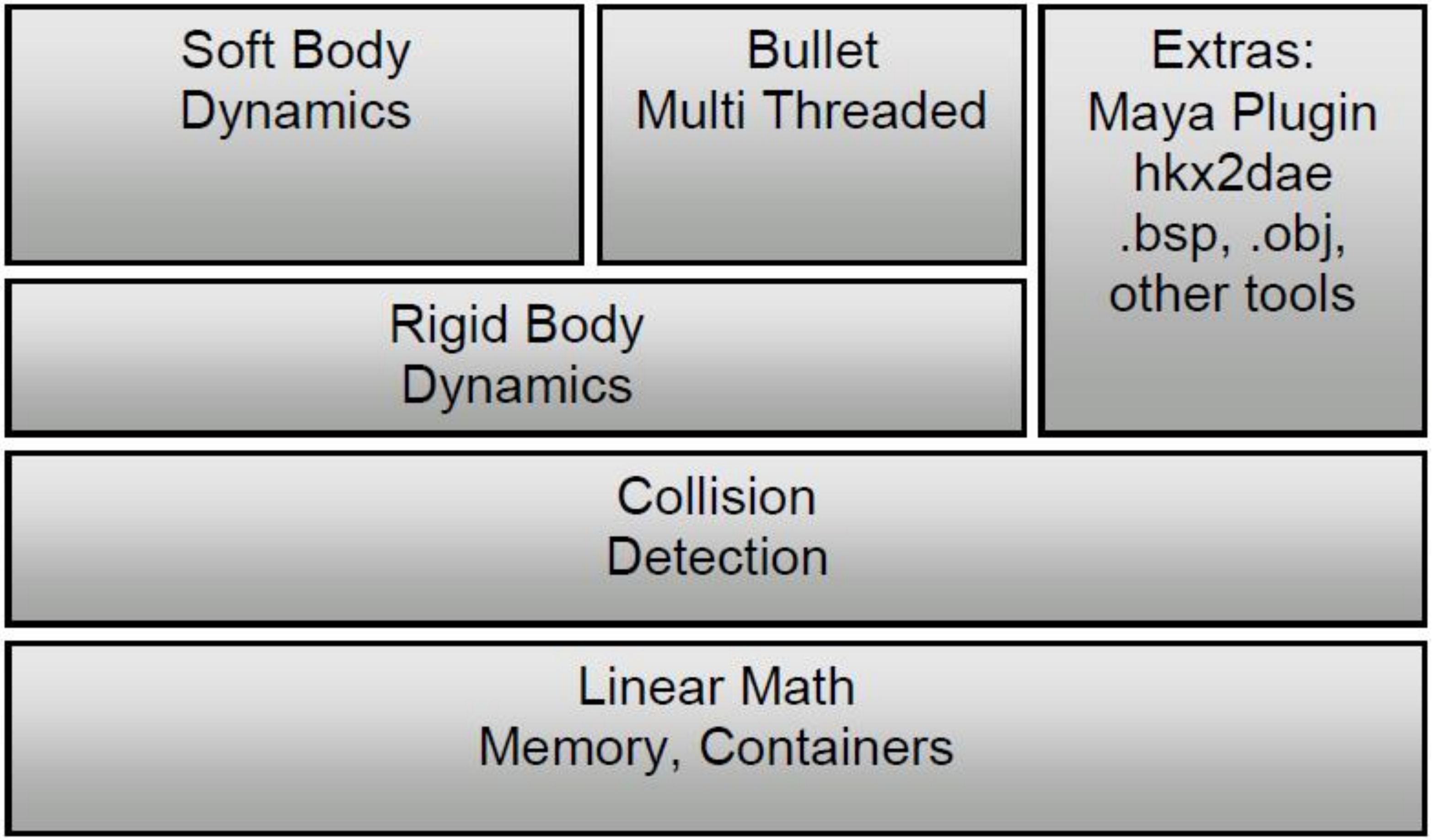
- Is it correct?
  - Will all implementations of the design exhibit the desired functionality?
- Is it efficient?
  - Are there implementations of the design that will be acceptably efficient?
- Is it testable & maintainable?
  - Does the design describe a program structure that will make implementations reasonably easy to build, test and maintain?
- Is it modifiable, extensible, & scalable?
  - How difficult will it be to enhance the design to accommodate future modifications?

# Other Considerations

- Are the classes independent?
- Is there redundancy?
- Do they manage & protect their own data?
- Can they be tested individually?
- Do they promote code reuse?
- Is data and control flow clear or complex?

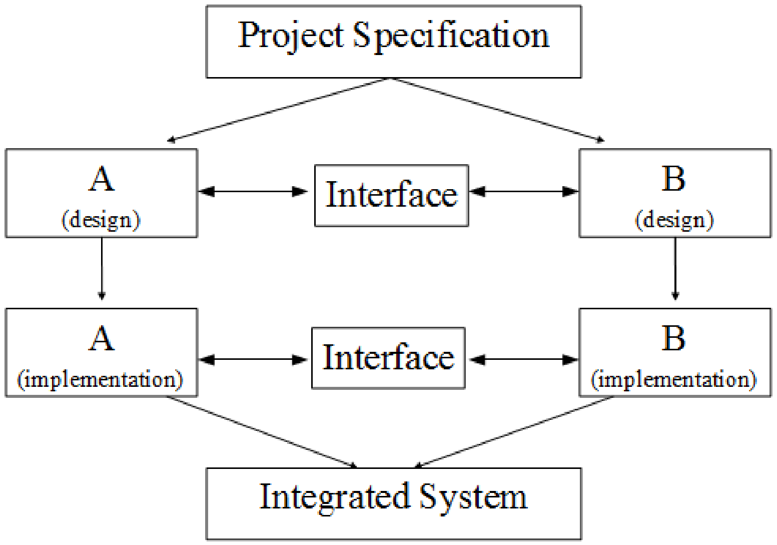
# It all starts with a Modular Design

- Large software projects are divided up into separate modules
  - i.e. groups of related classes


Soft Body Dynamics
Bullet Multi Threaded
Extras: Maya Plugin hkx2dae .bsp, .obj, other tools
Rigid Body Dynamics
Collision Detection
Linear Math Memory, Containers

# Modular Design Methodology

- Decompose
  - large programming problems into smaller ones
  - i.e. sub-problems
- Solve
  - the sub-problems independently
  - modules solve sub-problems
- Assemble
  - the modules to build full system
  - called system integration
    - scariest parts of software development
    - serious design flaws can be exposed

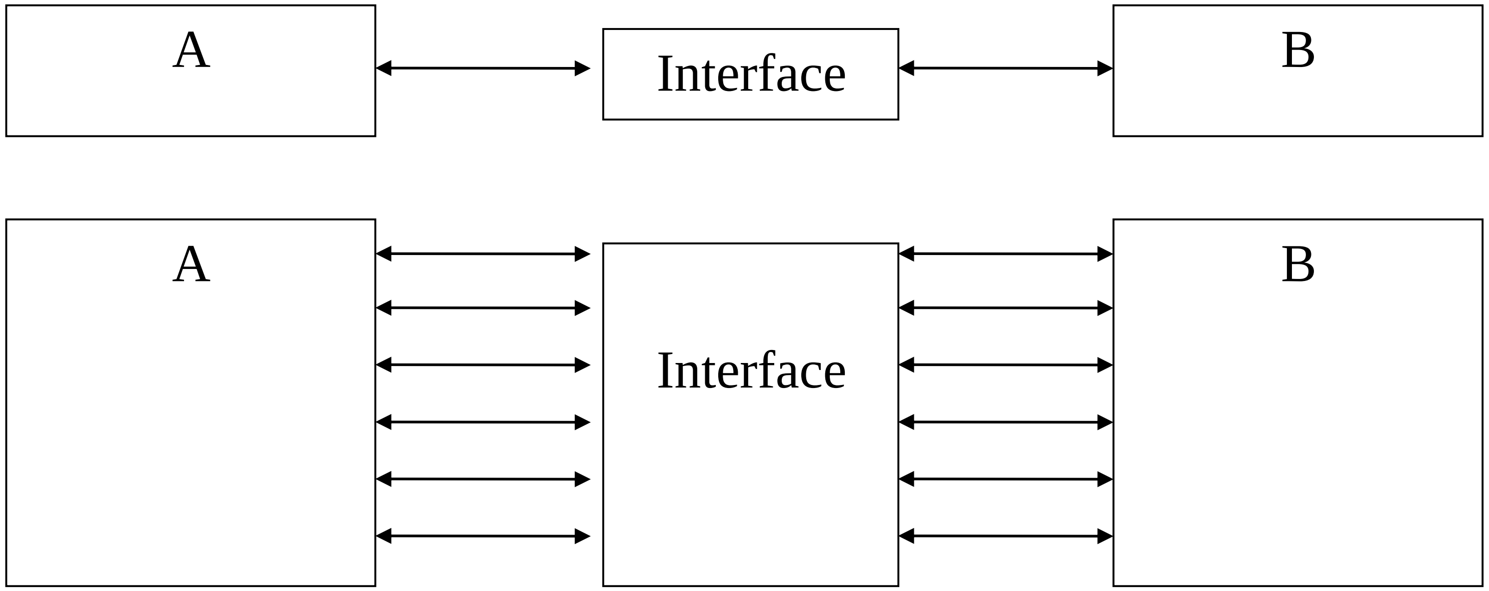


## What makes a good modular design?

- Connections between modules are ***explicit***
- Connections between modules are ***minimized***
  - called ***narrow interfaces***
- Modules use ***abstraction*** well
- Implementation of modules can be done **independently**
  - modules avoid duplication of effort

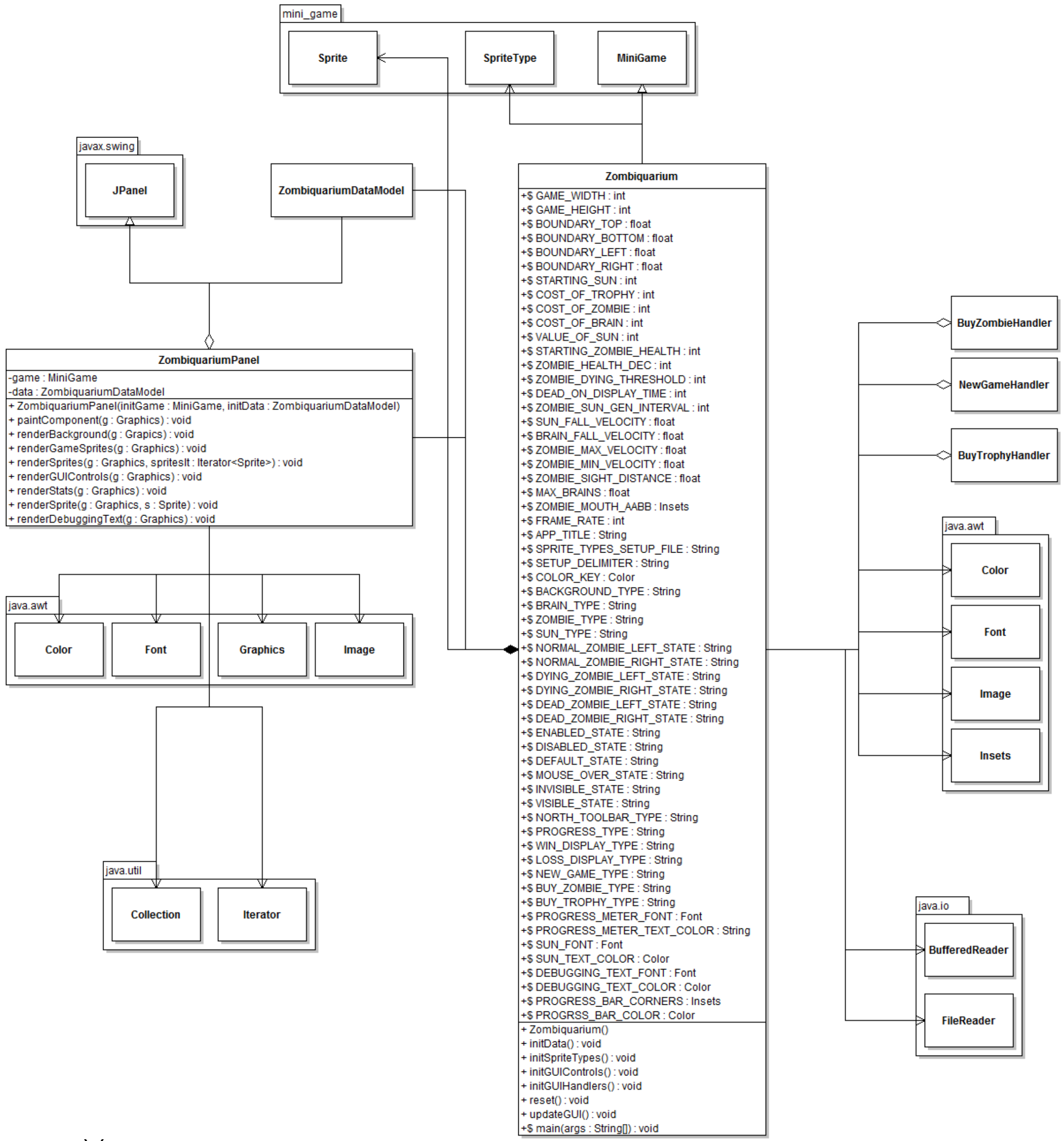
## More on Narrow Interfaces

- A module should have access to only as much information as it **needs** to work (no more than that)
  - less chance of misuse
  - less coordination needed between team members
    - fewer meetings necessary



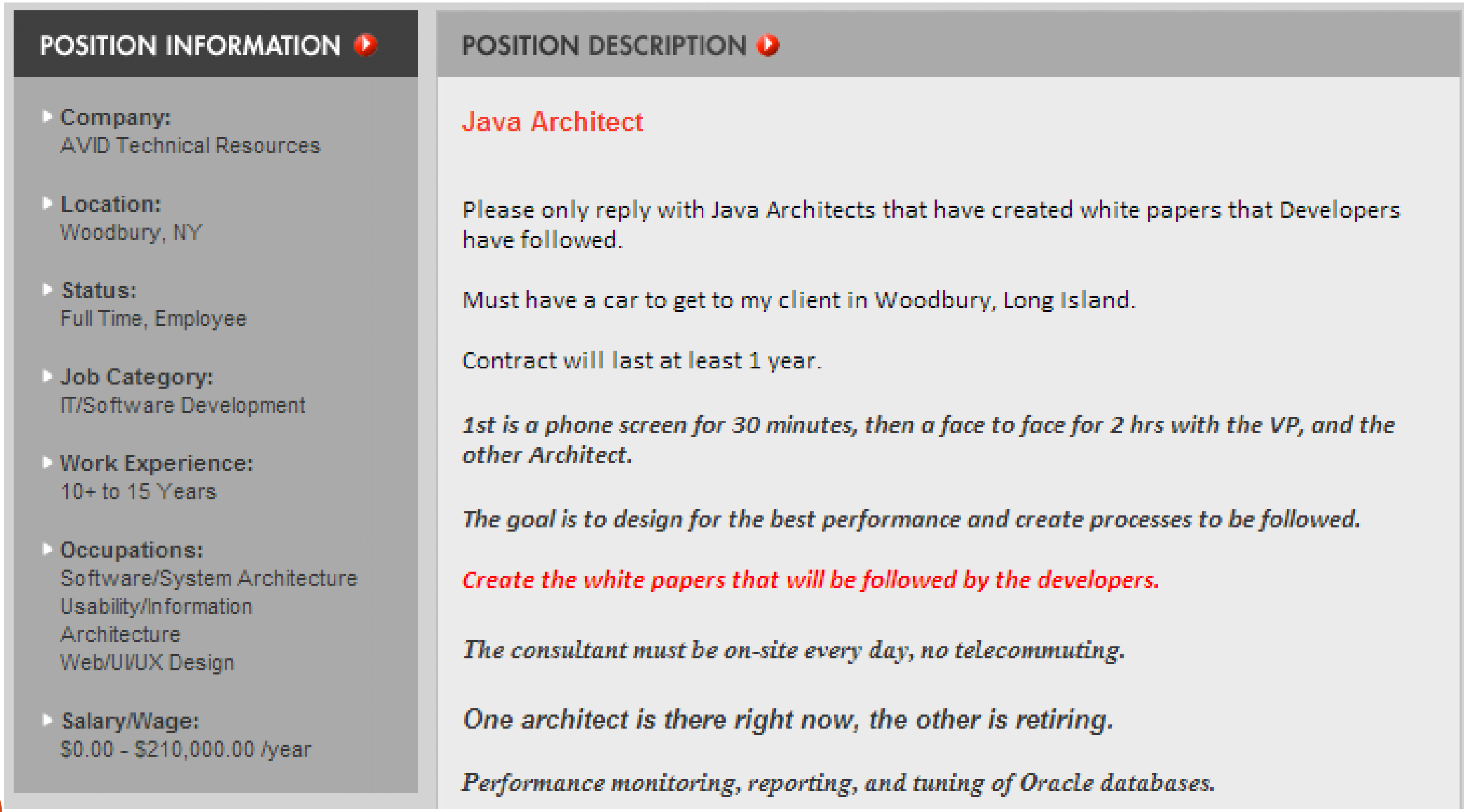
# Design is Difficult

- Where do you begin?
- When is the design complete?


mini_game
Sprite
SpriteType
MiniGame
javax.swing
JPanel
ZombiquariumDataModel
ZombiquariumPanel
-game : MiniGame
-data : ZombiquariumDataModel
+ ZombiquariumPanel(initGame : MiniGame, initData : ZombiquariumDataModel)
+ paintComponent(g : Graphics) : void
+ renderBackground(g : Grapics) : void
+ renderGameSprites(g : Graphics) : void
+ renderSprites(g : Graphics, spritesIt : Iterator<Sprite>) : void
+ renderGUIControls(g : Graphics) : void
+ renderStats(g : Graphics) : void
+ renderSprite(g : Graphics, s : Sprite) : void
+ renderDebuggingText(g : Graphics) : void
java.awt
Color
Font
Graphics
Image
java.util
Collection
Iterator
Zombiquarium
+$ GAME_WIDTH : int
+$ GAME_HEIGHT : int
+$ BOUNDARY_TOP : float
+$ BOUNDARY_BOTTOM : float
+$ BOUNDARY_LEFT : float
+$ BOUNDARY_RIGHT : float
+$ STARTING_SUN : int
+$ COST_OF_TROPHY : int
+$ COST_OF_ZOMBIE : int
+$ COST_OF_BRAIN : int
+$ VALUE_OF_SUN : int
+$ STARTING_ZOMBIE_HEALTH : int
+$ ZOMBIE_HEALTH_DEC : int
+$ ZOMBIE_DYING_THRESHOLD : int
+$ DEAD_ON_DISPLAY_TIME : int
+$ ZOMBIE_SUN_GEN_INTERVAL : int
+$ SUN_FALL_VELOCITY : float
+$ BRAIN_FALL_VELOCITY : float
+$ ZOMBIE_MAX_VELOCITY : float
+$ ZOMBIE_MIN_VELOCITY : float
+$ ZOMBIE_SIGHT_DISTANCE : float
+$ MAX_BRAINS : float
+$ ZOMBIE_MOUTH_AABB : Insets
+$ FRAME_RATE : int
+$ APP_TITLE : String
+$ SPRITE_TYPES_SETUP_FILE : String
+$ SETUP_DELIMITER : String
+$ COLOR_KEY : Color
+$ BACKGROUND_TYPE : String
+$ BRAIN_TYPE : String
+$ ZOMBIE_TYPE : String
+$ SUN_TYPE : String
+$ NORMAL_ZOMBIE_LEFT_STATE : String
+$ NORMAL_ZOMBIE_RIGHT_STATE : String
+$ DYING_ZOMBIE_LEFT_STATE : String
+$ DYING_ZOMBIE_RIGHT_STATE : String
+$ DEAD_ZOMBIE_LEFT_STATE : String
+$ DEAD_ZOMBIE_RIGHT_STATE : String
+$ ENABLED_STATE : String
+$ DISABLED_STATE : String
+$ DEFAULT_STATE : String
+$ MOUSE_OVER_STATE : String
+$ INVISIBLE_STATE : String
+$ VISIBLE_STATE : String
+$ NORTH_TOOLBAR_TYPE : String
+$ PROGRESS_TYPE : String
+$ WIN_DISPLAY_TYPE : String
+$ LOSS_DISPLAY_TYPE : String
+$ NEW_GAME_TYPE : String
+$ BUY_ZOMBIE_TYPE : String
+$ BUY_TROPHY_TYPE : String
+$ PROGRESS_METER_FONT : Font
+$ PROGRESS_METER_TEXT_COLOR : String
+$ SUN_FONT : Font
+$ SUN_TEXT_COLOR : Color
+$ DEBUGGING_TEXT_FONT : Font
+$ DEBUGGING_TEXT_COLOR : Color
+$ PROGRESS_BAR_CORNERS : Insets
+$ PROGRSS_BAR_COLOR : Color
+ Zombiquarium()
+ initData() : void
+ initSpriteTypes() : void
+ initGUIControls() : void
+ initGUIHandlers() : void
+ reset() : void
+ updateGUI() : void
+$ main(args : String[]) : void
BuyZombieHandler
NewGameHandler
BuyTrophyHandler
java.awt
Color
Font
Image
Insets
java.io
BufferedReader
FileReader

# Good Design comes with Experience

- It takes time to become an expert



POSITION INFORMATION

- **Company:** AVID Technical Resources
- **Location:** Woodbury, NY
- **Status:** Full Time, Employee
- **Job Category:** IT/Software Development
- **Work Experience:** 10+ to 15 Years
- **Occupations:** Software/System Architecture; Usability/Information Architecture; Web/UI/UX Design
- **Salary/Wage:** $0.00 - $210,000.00 /year

POSITION DESCRIPTION

Java Architect

Please only reply with Java Architects that have created white papers that Developers have followed.

Must have a car to get to my client in Woodbury, Long Island.

Contract will last at least 1 year.

*1st is a phone screen for 30 minutes, then a face to face for 2 hrs with the VP, and the other Architect.*

*The goal is to design for the best performance and create processes to be followed.*

*Create the white papers that will be followed by the developers.*

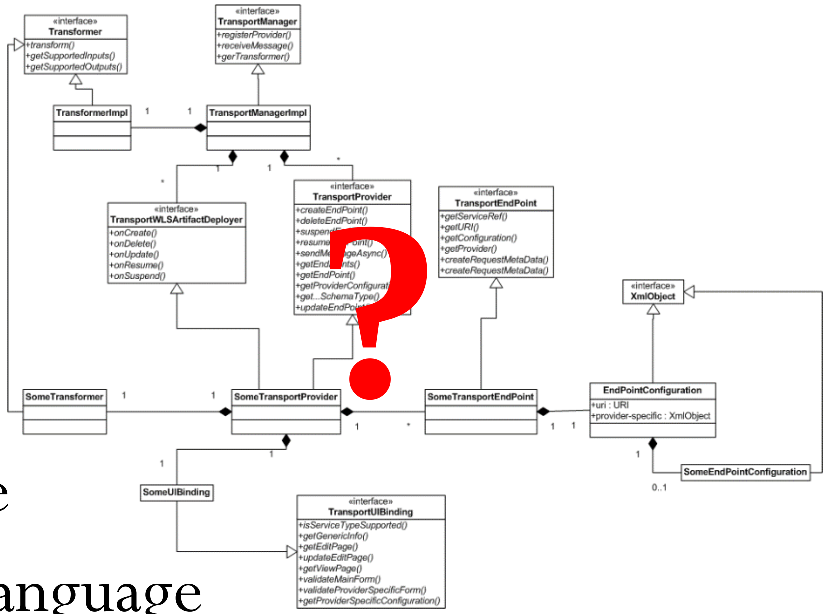
*The consultant must be on-site every day, no telecommuting.*

*One architect is there right now, the other is retiring.*

*Performance monitoring, reporting, and tuning of Oracle databases.*

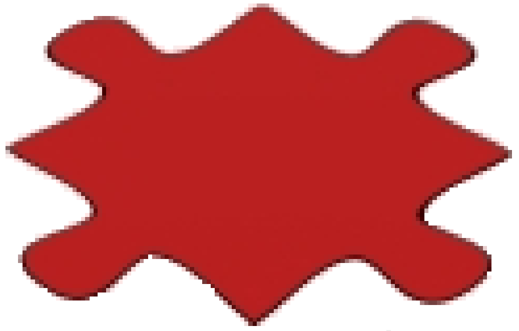
## How can a design be reviewed for correctness?

- Testing is not possible



- Verification is not possible
  - unless it uses a formal language
  - typically not practical
- Use proven, systematic procedure
  - **examine both local & global properties of the design**

# Local Properties

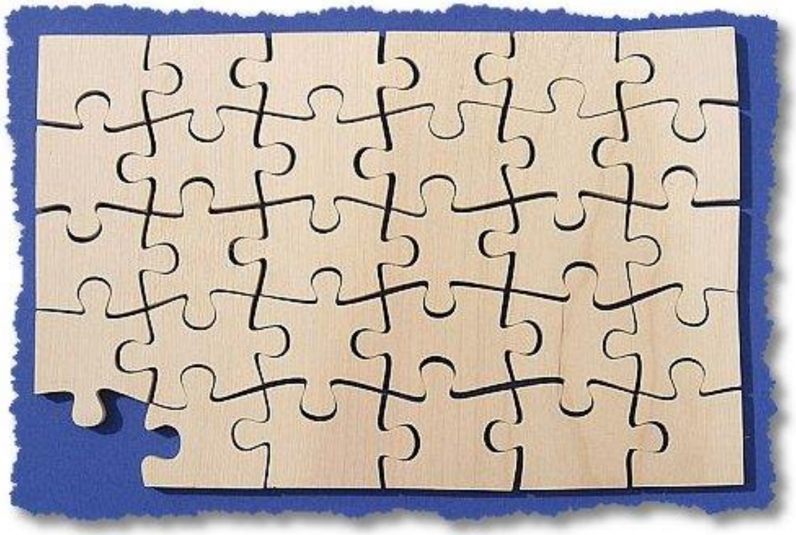


- Studying individual modules
- Important local properties:
  - consistency
    - everything designed was as specified
  - completeness
    - everything specified was designed
  - performance
    - running time
    - storage requirements

## Global Properties

- Studying how modules fit together
  - after examining local properties

## Global Properties to Consider



- Is all the data accounted for?
  - from original SRS
    - exists properly in a module
    - rules are properly enforced
- Trace paths through the design
  - walk-through
  - select test data
    - Does control flow properly through the design?
    - Does data flow properly through the design?

## Reviewing Design Structure

- Two key questions:
  - Is there an abstraction that would lead to a better modularization?
  - Have we grouped together things that really do not belong in the same module?
- Structural Considerations
  - Coherence of procedures
  - Coherence of types
  - Communication between modules
  - Reducing dependencies

## Coherence of procedures

- A procedure (method) in a design should represent a single, coherent abstraction
- Indicators of lack of coherence:
  - if the best way to specify a procedure is to describe how it works
  - if the procedure is difficult to name
- Arbitrary restrictions:
  - length of a procedure
  - method calls in a procedure

## Coherence of Types

- Examine each method to see how crucial it is for the data type
    - does it need to access instance or static variables of the class
- Move irrelevant methods out to another location
- Common with static functions

# Communication between Modules

- Careful examination can uncover important design flaws
  - think of handing your project design to another student for inspection
    - Do these pieces really fit together?
  - to improve any design:
    - act like a jerk when examining your own design
    - ask questions that a jerk would ask
    - make sure your design addresses these jerky questions

# Reducing Dependencies

- A design with fewer dependencies is generally better than one with more dependencies.
- What does this mean?
  - Make the design of each component dependent on as few other components as necessary
  - Example of bad framework design:
    - Every class in your framework uses every other class in your framework in one way or another
      - This would be terribly complex to test & modify

## Look for Antipatterns

- Common patterns in programs that use **poor** design concepts
  - make reuse very difficult
  - source: http://www.antipatterns.com
- Examples:
  - The Blob
  - Spaghetti Code
- We must correct these design errors

*Development AntiPattern:*

## The Blob

- **Symptoms**
  - Single class with many attributes & operations
  - Controller class with simple, data-object classes.
  - Lack of OO design.
  - A migrated legacy design

- **Consequences**
  - Lost OO advantage
  - Too complex to reuse or test.
  - Expensive to load

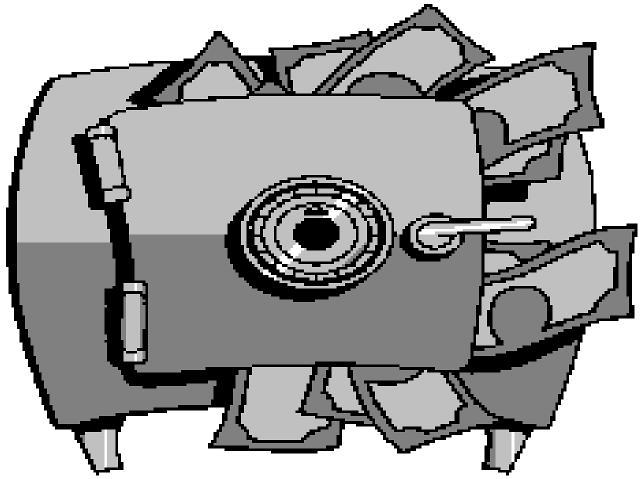
MITRE

*Development AntiPattern:*

## Spaghetti Code

- **spa·ghet·ti code** [Slang] an undocumented piece of software source code that cannot be extended or modified without extreme difficulty due to its convoluted structure.

Un-structured code
is a liability



Well structured code
is an investment.

MITRE