CSE 219
COMPUTER SCIENCE III

OBJECT ORIENTED DESIGN

SLIDES COURTESY:

PROF. RICHARD MCKENNA

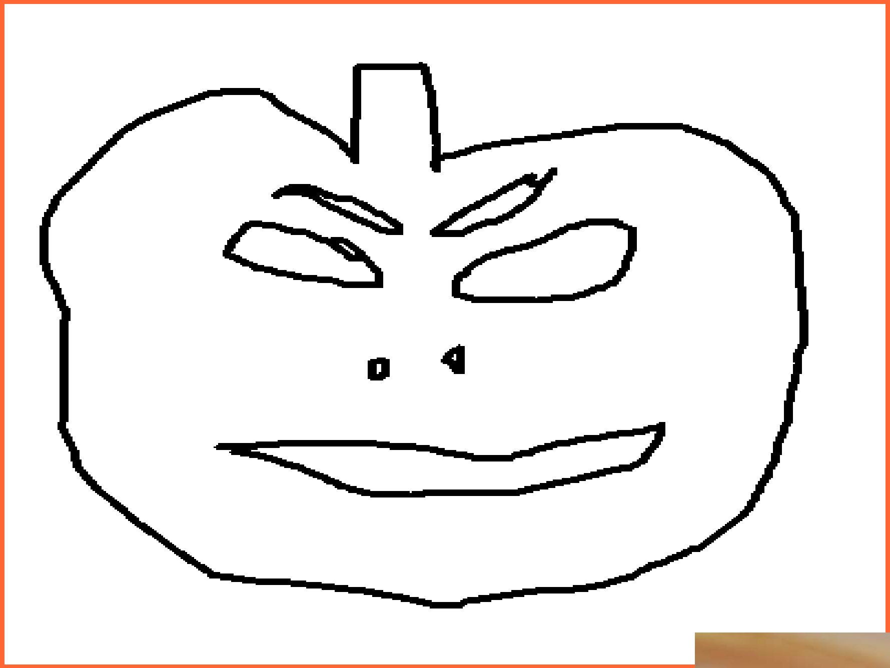
STONY BROOK UNIVERSITY

Halloween will be here before you know it

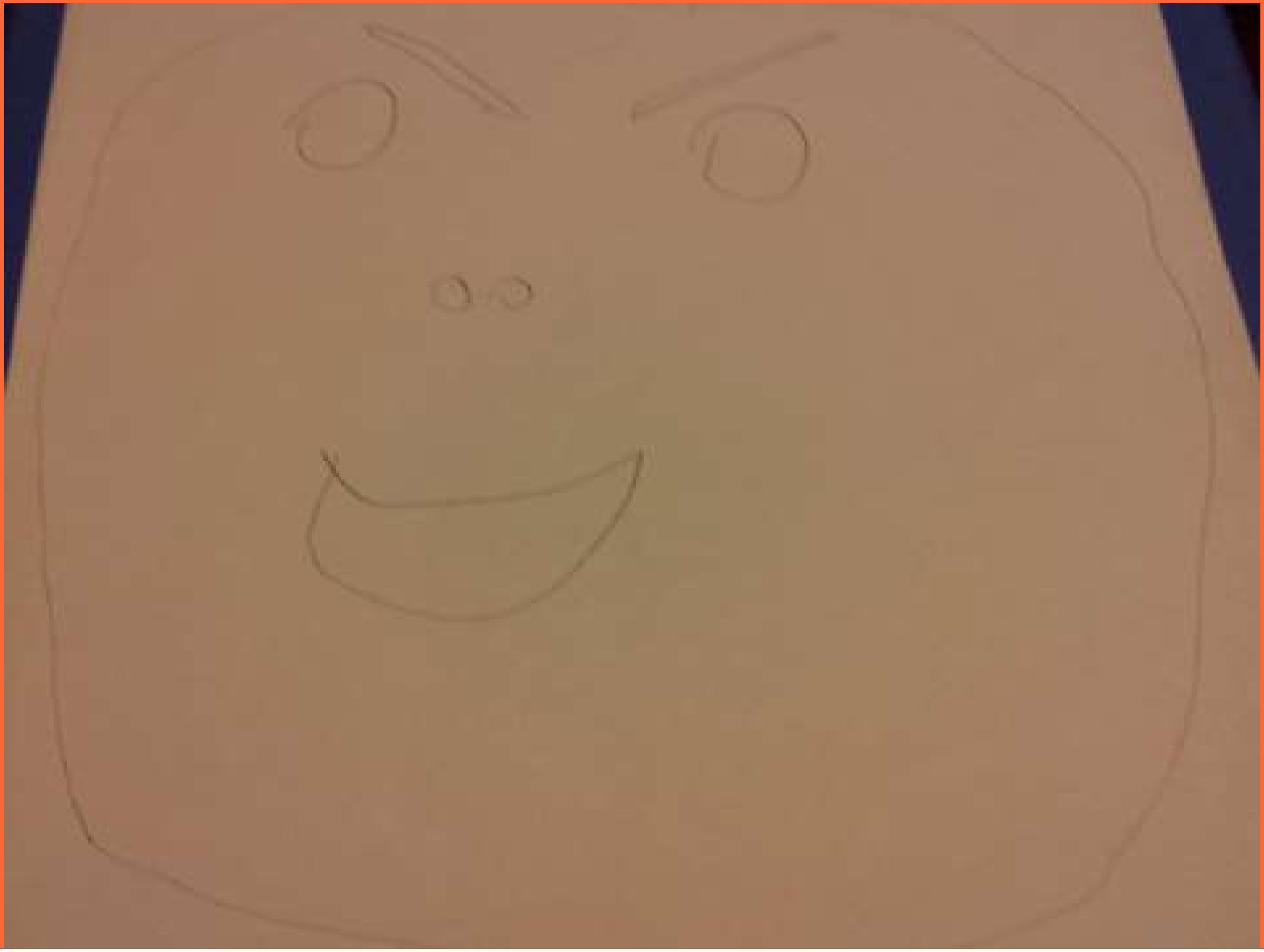
Time to Make a Jack O'Lantern

Without a plan, your Jack is doomed

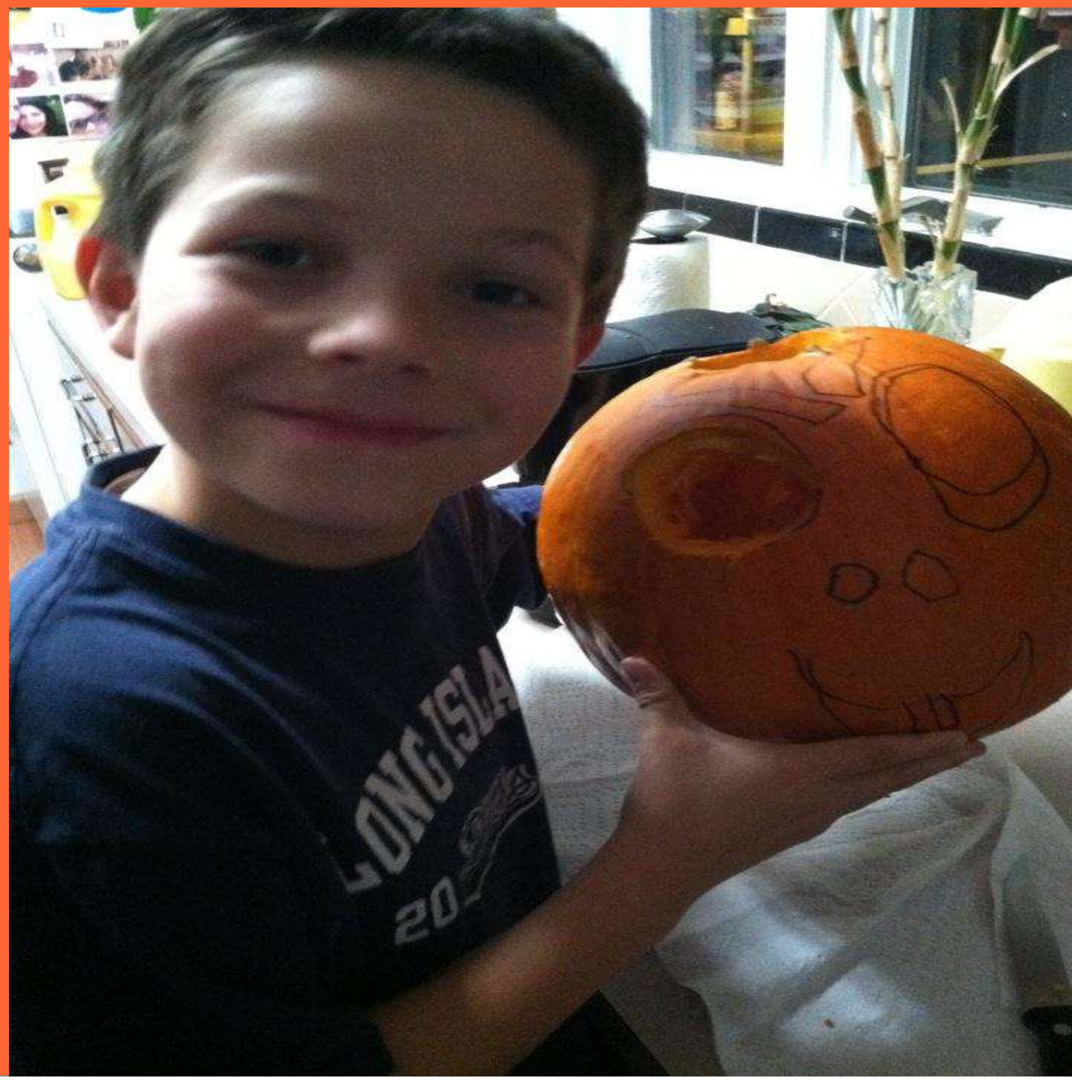
Design, prototype, then implement



Design



Prototype



Implement

Enjoy

And of course smash

High Quality Software Properties

- Correctness, Efficiency, Ease of use, Reliability/robustness, Reusability, Maintainability, Modifiability, Testability, Extensibility, Scalability

- When should we consider these properties?
  – the requirements analysis & design stages

- How about the implementation stages?
  – too late to make a big impact

UML Diagrams

- UML - Unified Modeling Language

- UML diagrams are used to *design* object-oriented software systems
  - represent systems *visually*
  - 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
  - CASE tools

Types of UML Diagrams

- Types we'll make:
  - Use Case Diagram
  - Class Diagram
  - Sequence Diagram

- Others:
  - 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
  - – class names, relationships, instance variables, method signatures

- Sequence Diagrams
  - – describe all event handling
  - – method invocation chains

What tools should we use?

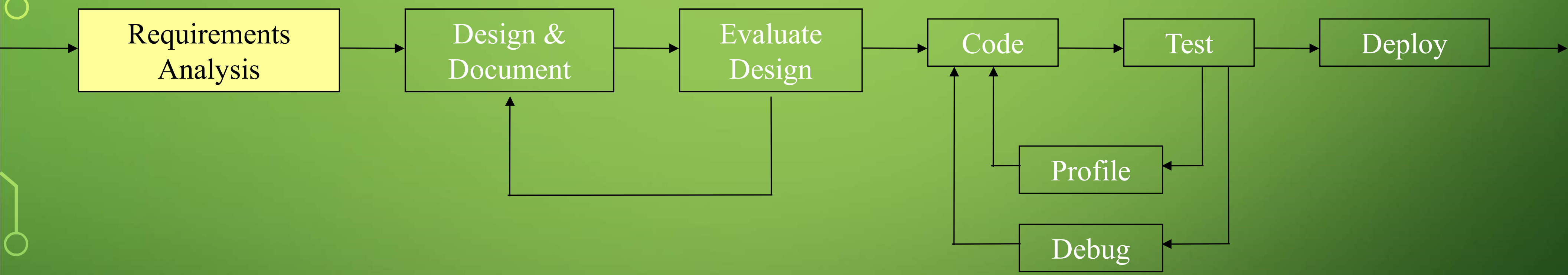
- UML modeling software

- Violet UML Editor (nice simple option)

http://alexdp.free.fr/violetumleditor/page.php

How can these properties be achieved?

- By using well proven, established processes
  - preferably while taking advantage of good tools



- Software Development Life Cycle

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
  - I'll do this for you this semester

Requirements Analysis

- i.e. Software Specification (spec.)
- Also called Software Requirements Specification (SRS)
- This document 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 an SRS/RA document?

- 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?

What *really* goes in an SRS/RA?

- Detailed descriptions of all:
  - – necessary data
  - – program input and output
  - – GUI screens & controls
  - – user actions and program reactions

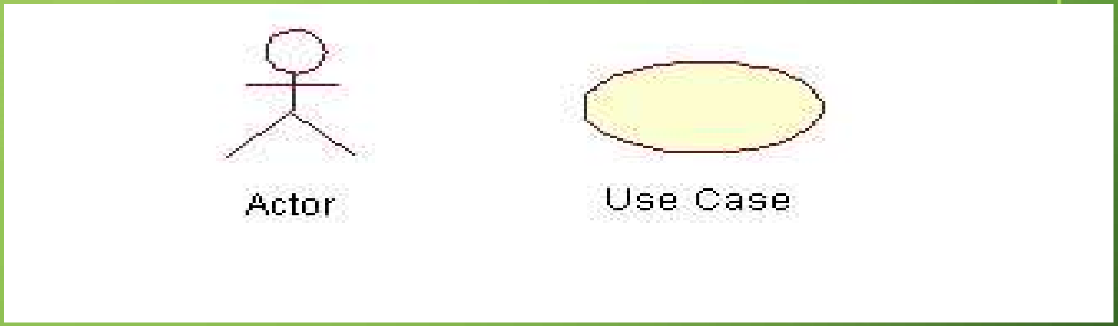
- For a database:
  - – necessary forms & views

Where do you start?

- Interviews (really)
- Who do you interview?
  - end users
- What do they need?
- What do they want?

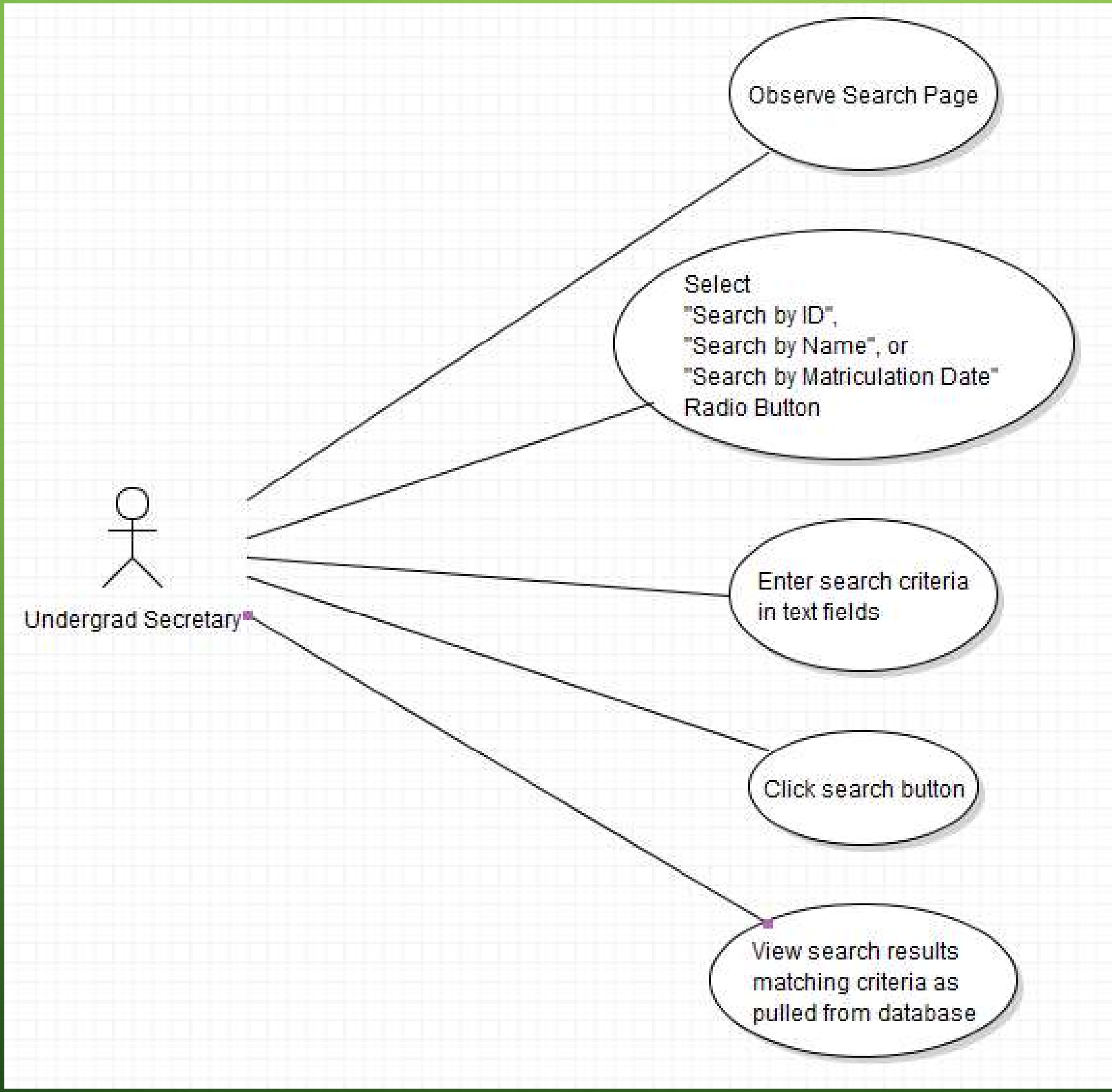
UML Use Case Diagrams

- 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

Informal UML Use Case Diagram



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. 2. UG secretary: selects 'Search by ID', 'Search by Name', or 'Search by Matriculation Date' radio button. 3. UG secretary: enters the ID number, first and last name, or date range in the text fields corresponding to the selected radio button. 4. UG secretary: clicks the 'Search' button. 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. 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 server: network and local interface
Open issues:	1. Where on the web interface will the search fields and buttons be displayed? 2. What other criteria will the UG secretary want to search by? 3. Should we have a 'Clear Fields' button that clears all entered text in the search fields?

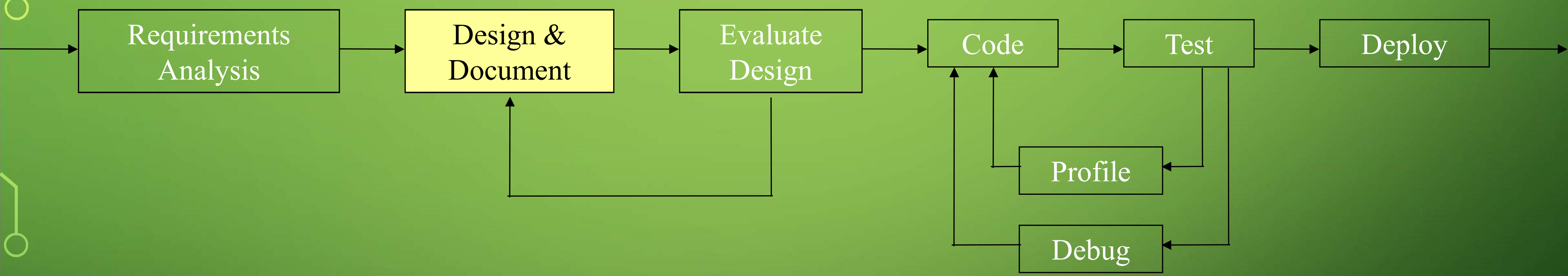
UML Use Case Diagrams

- Fed as input to the next step

- What's that?
  - class, data, and function design
    - UML Class Diagrams
    - UML Sequence Diagrams

How can these properties be achieved?

- By using well proven, established processes
  - preferably while taking advantage of good tools



- Software Development Life Cycle

First things first

- Have other “similar” problems been solved?
  - Do design patterns exist to help?
  - Does a framework exist to help us

- Will other “similar” problems need to be solved?
  - Should we make a framework?

Class Design Approaches

- Important Approaches:
  - Data-Driven design
  - Top-Down design (employing software ***decomposition***)

- What are the “easy” and “hard” parts?
  - Why is this important?
    - work measurement

Data-driven Design

- From the problem specification, extract
  - nouns (objects, attributes of objects)
  - verbs (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

Class relationships

- Think data flow:
  - What HAS what?
  - What IS what?
  - What USES what?
  - Where should data go?
  - How will event handler X change data in class Y?
  - 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
- Data vs. Mechanics
- Functionality vs. Presentation

Functionality vs. Presentation

- What is a game state manager (GSM)?
  - classes that do the work of managing data & enforcing rules on that data
- Why separate the GSM and the UI?
  - so we can change the GSM without changing the UI
  - so we can change the UI without changing the GSM
  - so we can design several different UIs for an GSM
  - reuse code that is proven to work
- This is a common principle throughout GUI design
  - even for Web sites (separate content)
  - different programmers for each task

Choosing Data Structures

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

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),* also known as containment
  - *Inheritance (IS-A)*

UML Class Responsibilities Diagrams

Class Name

Die
State Info: number of faces value facing up
Responsibilities: access instance variables roll die

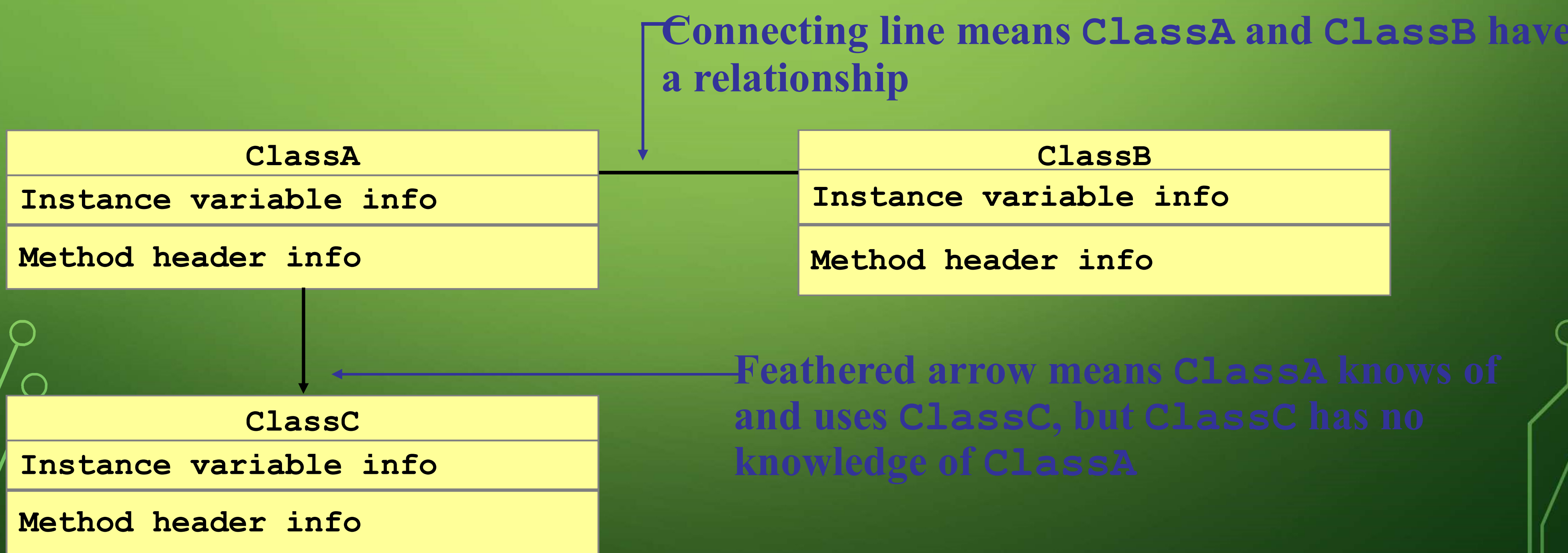
State info to be translated into instance variables

PairOfDice
State Info: die1: Die die2: Die
Responsibilities: access instance variables roll dice calculate total

Responsibilities to be translated into methods

UML Class Diagrams

- Derived from class responsibilities diagrams
- Show relationships between classes
  - Class associations denoted by lines connecting classes
  - A feathered arrow denotes a one-directional association



Method and Instance Variable Descriptions

- Instance Variables Format
  - `variableName : variableType`
  - For example,

    `upValue : int`
- Method Header Format
  - `methodName(argumentName :`

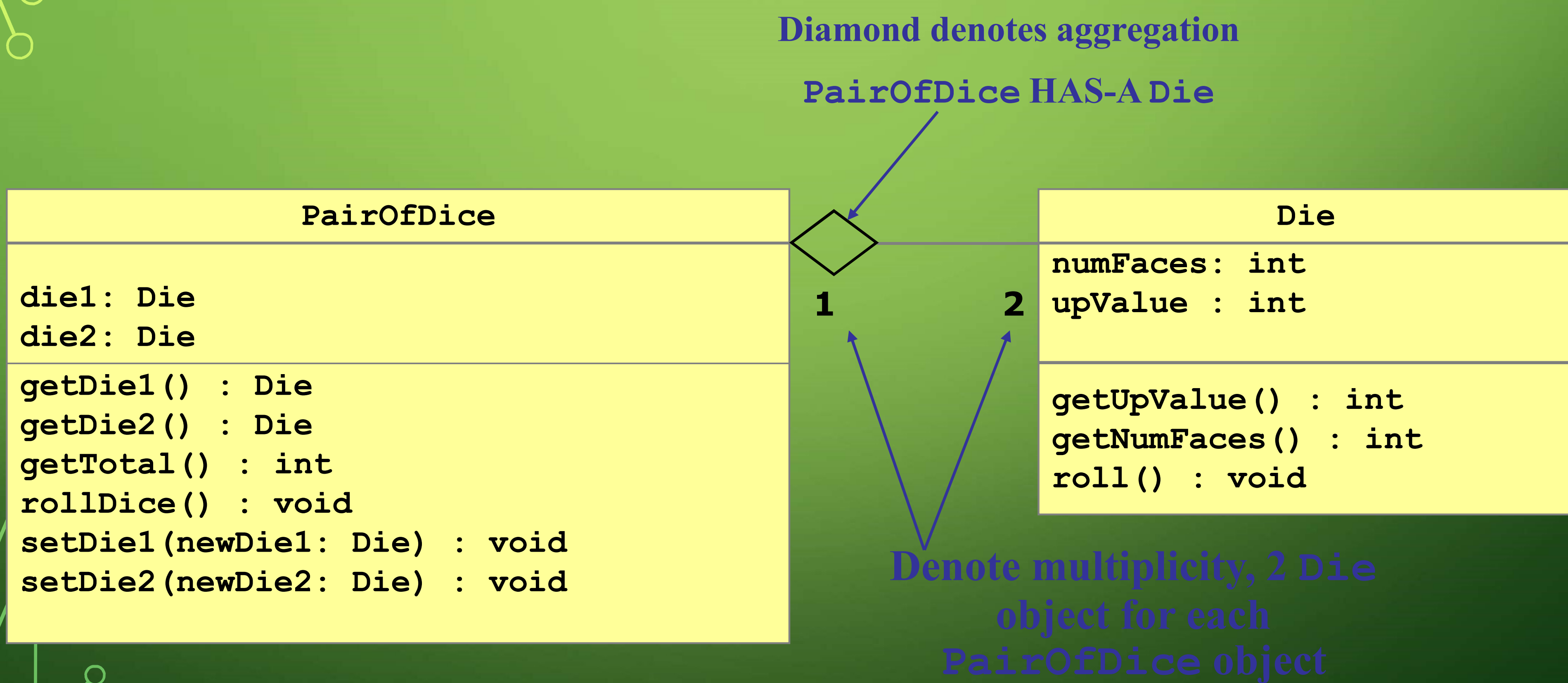
    `argumentType): returnType`
  - For example,

    `setDie1(newDie1 : Die) : void`
  - $ denotes a static method or variable, for example:

    `$ myStaticMethod(x : int) : void`

UML Class Diagrams & Aggregation

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

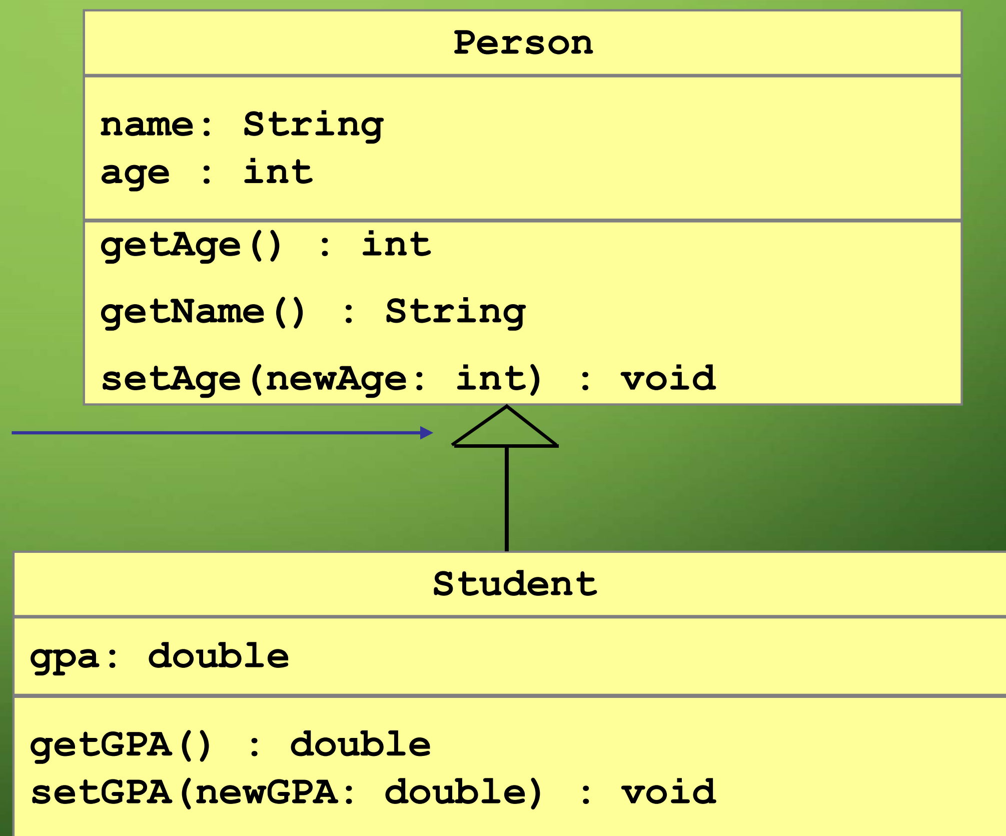


UML Class Diagrams & Inheritance

```
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 *interface* 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 #

```
PairOfDice
-----------------------------
- die1: Die
- die2: Die
-----------------------------
+ getDie1() : Die
+ getDie2() : Die
+ getTotal() : int
+ rollDice() : void
+ setDie1(newDie1: Die) : void
+ setDie2(newDie2: Die) : void
```

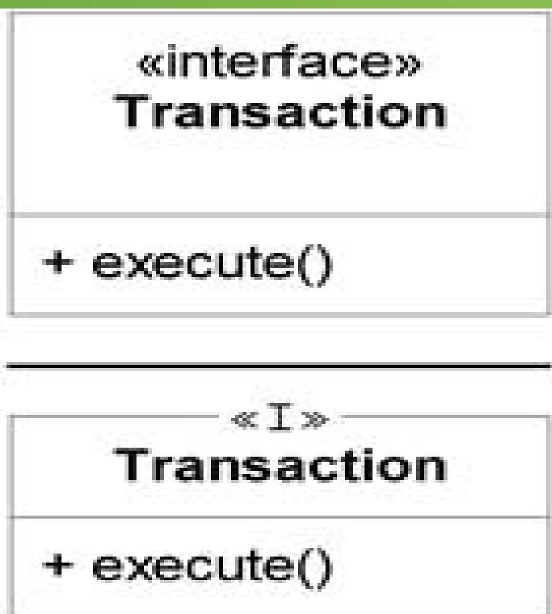
```
Die
-----------------------------
- numFaces: int
- upValue : int
-----------------------------
+ getUpValue() : int
+ getNumFaces() : int
+ roll() : void
```

Interfaces in UML

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

- 2 ways to denote an interface

1. <<interface>>, OR
2. <<I>>



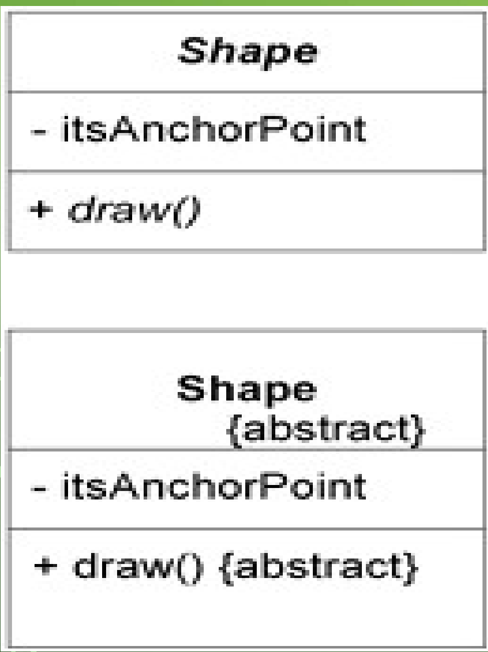

```
interface Transaction
{
   public void execute();
}
```

Abstract Classes in UML

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

- 2 ways to denote a class or method is abstract:

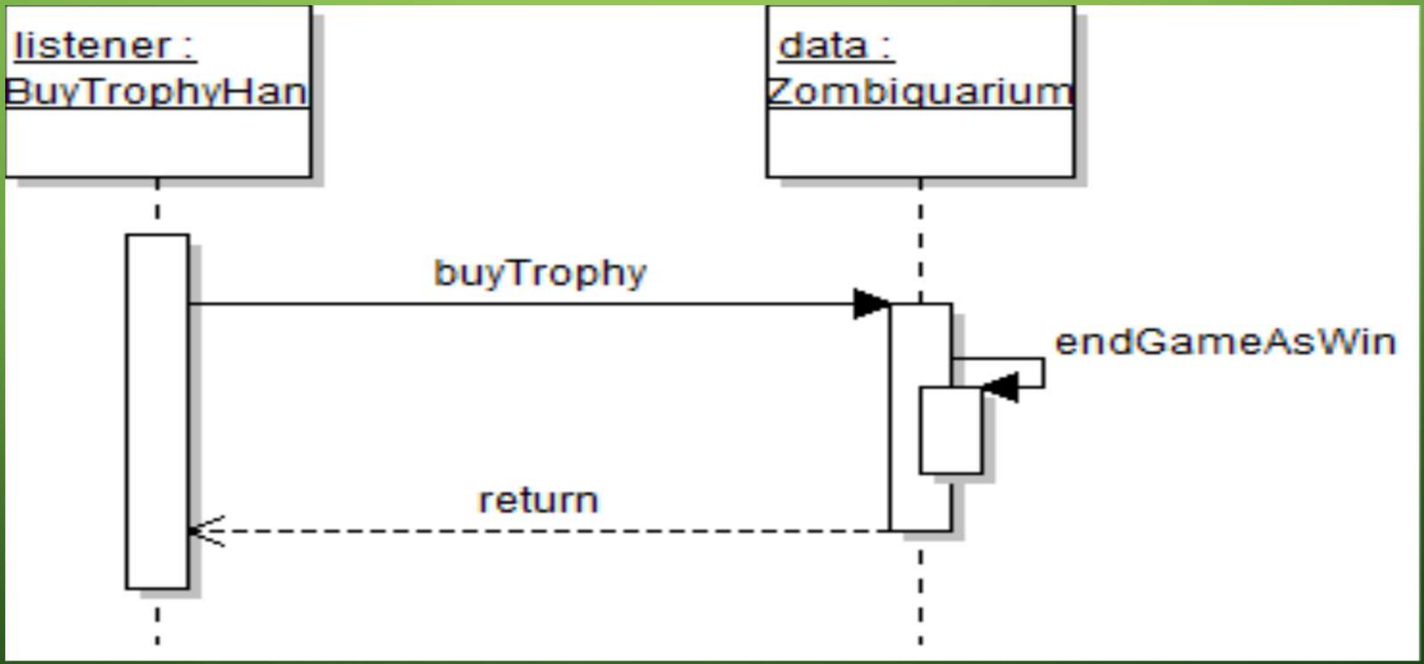
1. class or method name in italics, OR
2. {abstract} notation




```
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)
    - ***software decomposition***
  - 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
  - information flow
  - etc.

Designing Methods

- Decide method signatures
  - numbers and types of parameters and return values
- Write down what a method should do
  - 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 almost 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

Summary

- 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