Software Development Life Cycle

• Using well proven, established processes
  – preferably while taking advantage of good tools
Design Approaches

- Have other “similar” problems been solved?
  - Do design patterns exist to help?
- What are the “easy” and “hard” parts?
  - Why is this important?
    - work measurement
- Employ:
  - data-driven design
    - Note: 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?
  - 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?
  • Separate Data from Mechanics
  • Separate Functionality from Presentation
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)
  - 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?
  - Which access algorithms?
  - Ordered by what?
UML Diagrams

- UML - Unified Modeling Language
- UML 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 CSE219:

- 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
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)
Class Name: Die

State Info:
- number of faces
- value facing up

Responsibilities:
- access instance variables
- roll die

Class Name: PairOfDice

State Info:
- die1: Die
die2: Die

Responsibilities:
- access instance variables
- roll dice
- calculate total

(c) Paul Fodor
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

Connecting line means ClassA and ClassB have a relationship

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`
UML Class Diagrams & Aggregation

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

  **PairOfDice**
  
  - die1: Die
  - die2: Die

  **Die**
  
  - numFaces: int
  - upValue : int

  **Methods**
  
  - getDie1() : Die
  - getDie2() : Die
  - getTotal() : int
  - rollDice() : void
  - setDie1(newDie1: Die) : void
  - setDie2(newDie2: Die) : void

  **Description**
  
  Diamond denotes aggregation:
  
  **PairOfDice HAS-A Die**

  Denote multiplicity, 2 Die object for each PairOfDice object.
public class Student extends Person

Person

name: String
age : int
getAge() : int
getName() : String
setAge(newAge: int) : void

Student

gpa: double
getGPA() : double
setGPA(newGPA: double) : void

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
In a UML class diagram:

- public members can be preceded by +
- private members are preceded by -
- protected members are preceded by #

```plaintext
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

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

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


(c) Paul Fodor
Abstract Classes in UML

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

```java
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
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
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
  • 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 😊
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