Software Design

Why Design?

- "If you can't describe what you are doing as a process, you don't know what you're doing."
  --- W. Edwards Deming
- "The quality of a software system is governed by the quality of the process used to develop it."
  --- Watts Humphrey
- "When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science."
  --- Lord Kelvin

Phases in Software Development

- Requirements Engineering (a.k.a. Analysis)
  - *Requirements Specification*
- Design
  - *Specification of the solution*
- Implementation
  - *Programming*
- Testing
  - *Working program*
- Maintenance
  - *Enhanced program and bug fixes*

Requirements Engineering

- Yields a description of the desired system
  - Functionality
  - Performance needs
  - 'Wish List' extensions
  - Documentation needs
  - Feasibility study
- Requirements Specification Document
  - All stakeholders involved
  - No room for ambiguity…this might be legally binding

Design

- ***What*** will be done (not how)
- Technical in nature
- Decomposes system into components
  - Defines interfaces btwn major components
- Documents decisions made/evaluated
  - Changes made here are relatively cheap
- Technical Specification
- Don't rush to code!

Implementation

- Individual components
- Goal
  - Well documented
  - Reliable
  - Flexible
  - Correct software
- Document code and decisions made!
- Conform to specification

Testing

- When does testing begin?
  - should be done from day 1
- Early errors are cheaper to fix
  - Errors in concept
  - Errors in implementation
  - Overpromising…
- **Validation**
  - are we building the right system?
    - the system the user asked for
- **Verification**
  - are we building the system right?
    - is the system flawed?

Maintenance

- Ongoing fixes
- Upgrades
  - new features
  - adapting to changing environment
- Continues until software is retired
- Refactoring

Design Principles

Abstraction

Modularity (coupling and cohesion)

Information Hiding

Limit complexity

Abstraction Techniques

Abstraction tames complexity

Procedural Abstraction

Stepwise refinement

Top down decomposition

Data Abstraction

Find a hierarchy in the data

Modularity

Coupling

Amount of connection between modules

Cohesion

Singularity of purpose

GOAL: Low coupling with High cohesion

Information Hiding

Hide design decisions

Hide things that are likely to change!

Need to know basis only

Hiding:

decreases coupling

increases abstraction (decreases complexity)

indicates module decomposition (cohesion)

Essence of what's nice about O.O.

Limit Complexity

"I'll know it when I see it"

How would you respond to programming 'guidelines' like:

- "Inheritance hierarchies should not exceed 5 levels"
- "Methods should be less than one page in size"
- "Classes should be less than 20 methods"

Abstraction is our natural method for controlling complexity

Design Principles

Abstraction
Modularity (coupling and cohesion)
Information Hiding
Limit complexity

Object Oriented Design addresses these well

however, anything can be mis-used…
StarShip → Freighter hierarchy…

Object Oriented Design

Modularity and Encapsulation
- Simplify
- compartmentalize !
- Access modifiers (public, private,…)

Abstraction via a public interface (API)
- restrict access through well-defined interface - "what" not "how"

Inheritance hierarchy
- limit complexity

Object Oriented Design

Identify Objects
- Nouns of the problem

Determine public attributes and actions
- Attributes are the state
- Actions are the methods

Determine relationships between objects

O.O. Class Guidelines

ENCAPSULATE

Declare attributes private
- Controls all access (hence allows for verification)
- Restricts changes

Constructors of ***valid*** objects
- Clients might forget to use setters properly
- Initialize everything

Minimize coupling (dependence *between* classes)

Notes from Lab

class is primary structure
- all methods are inside of some class

public attributes -vs- getter( ) and setter( )

Style: capitalize Class; camelCase var and methods: class Die … int roll( )

Think of physical analogy…build accordingly

Notes from Lab

Goal

- Well documented *(or extremely obvious)*
- Reliable *(always works)*
- Flexible *(adaptable as needed)*
- Correct *(does what it was asked to do)*

Notes from Lab

Java Classes - Encapsulate everything

```
public Class DieTester {
  public static void main (String[] args){
     Die dice1 = new Die (6);
     Die dungeonDice = new Die (12);

     int result = dice1.roll();
     int faces = dice1.numberOfFaces();

     if (dice1.roll() + result == 2)
         System.out.println ("Snake Eyes!");
  }
}
```


Notes from Lab

Die Class

- Attributes *(internal "state" information)*
  - public view will use getters/setters
- Actions *(what do we DO with a die)*
  - Contructors?
  - Default constructor?

Notes from Lab

Missing specifications

What if…

- asked to construct 56-sided die?

Design Patterns

Proven solution to recurring problems

Common vocabulary

Speeds implementation

"Anti-patterns"

- *God Class* - too much
- *Poltergeist* - too little
- *Golden hammer* - bad fit
- Reasons to refactor!

Additional Patterns

Creational Patterns
- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural Patterns
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- **Observer**
- State
- Strategy
- Template Method
- Visitor

J2EE Patterns
- MVC
- Business Delegate
- Composite Entity
- Data Access Object
- Front Controller
- Intercepting Filter
- Service Locator
- Transfer Object

Misc
- typesafe enum
- RESTful WS

What drives design?

Patterns - familiar solutions that work

Theory --> Patterns --> Implementation

Experience with alternatives

  Learning from prior mis-steps

Balancing priorities

Managing risks

Game Design

Furry Hurry

  6x9 grid with obstacles

  2-4 players with 4 furry critters each

  Take turns

    Roll die

    Optionally move yourself N / S by one square

    Must move East, any critter from row # given by die

First with 3 critters in East-most column wins

May not move onto obstacles

Moving on top of another player, prevents them from moving

Play starts by taking turns placing all critter pieces in column 1. Must fill all rows before stacking pieces