# Software Engineering I

Based on materials by Ken Birman, Cornell

# Software Engineering

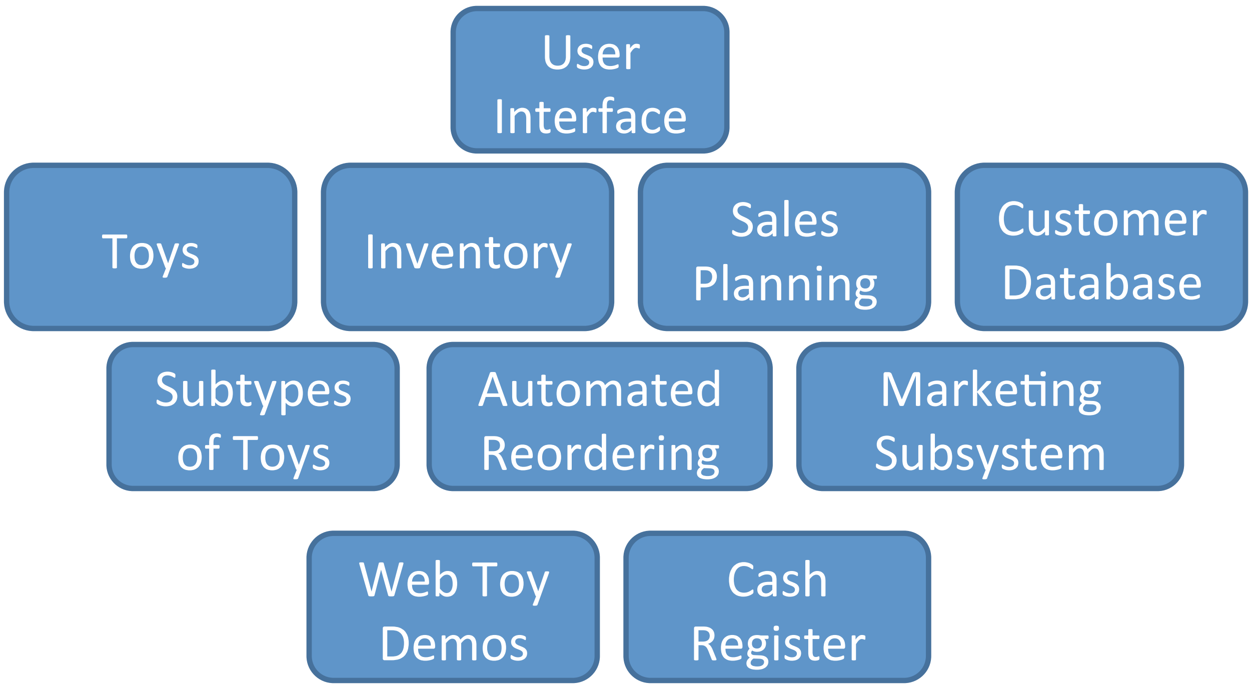
- The art by which we start with a problem statement and gradually evolve a solution
- There are whole books on this topic and most companies try to use a fairly uniform approach that all employees are expected to follow
- Interface design, class hierarchy are but two steps in this process

# The software design cycle

- Some ways of turning a problem statement into a program that we can debug and run
  - Top-Down, Bottom-Up Design
  - Software Process (briefly)
  - Modularity
  - Information Hiding, Encapsulation
  - Principles of Least Astonishment and “DRY”
  - Refactoring

# Top-Down Design

- Start with big picture:




- Invent abstractions at a high level
- Decomposition / "Divide and Conquer"

# Not a perfect, pretty picture

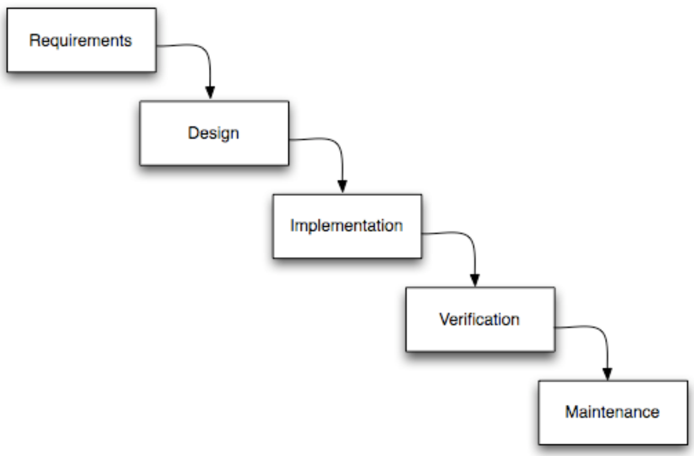
- It is often easy to take the first step but not the second one
- Large abstractions come naturally. But details often work better from the ground up
- Many developers work by building something small, testing it, then extending it
  - It helps to not be afraid of needing to recode things

# Top-Down vs. Bottom-Up

- Is one way better? Not really!
  - It's sometimes good to alternative
  - By coming to a problem from multiple angles you might notice something you had previously overlooked
  - Not the only ways to go about it
- Top-Down: harder to test early because parts needed may not have been designed yet
- Bottom-Up: may end up needing things different from how you built them

# Software Process

- For simple programs, a simple process…

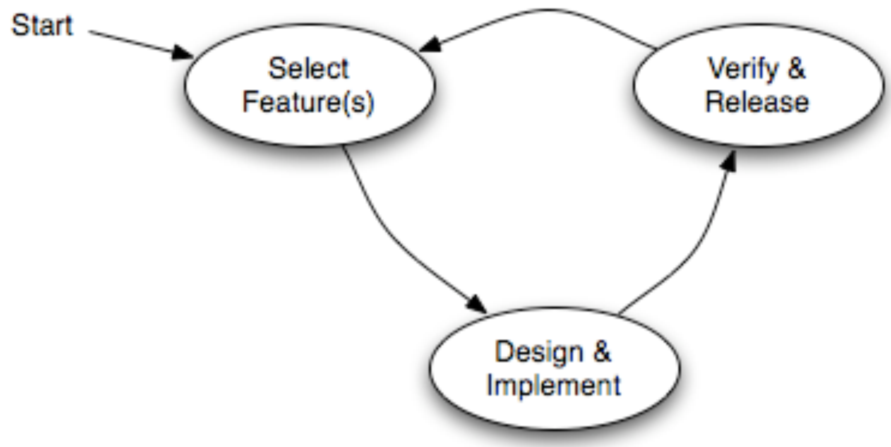



"Waterfall"

- But to use this process, you need to be sure that the requirements are fixed and well understood!
  – Many software problems are not like that
  – Often customer refines requirements when you try to deliver the initial solution!

# Incremental & Iterative

- Deliver versions of system in several small cycles




- Recognizes that for some settings, software development is like gardening
- You plant seeds… see what does well… then replace the plants that did poorly

# Information Hiding

- What "information" should we try to hide?
  - "Internal" design decisions.
- interface: everything that is externally accessible
- What OOP concept(s) relates to the idea of information hiding?

# Degenerate Interfaces

- Public fields and global variables are usually a Bad Thing:

```
double totalCount__SallysVariable_DoNotTouch = 0;

int main() {
        ...
}
```

- Anybody can change them; we don't maintain control

# Use of interfaces?

- When team builds a solution, interfaces can be valuable!
  - Rebecca agrees to implement the code to extract genetic data from files
  - Tom will implement the logic to compare DNA
  - Willy is responsible for the GUI
- By agreeing on the interfaces between their respective modules, they can all work on the program simultaneously

# Principle of Least Astonishment

- Interface should “hint” at its behavior

Bad:

```
int product(int a, int b) {
    return a*b > 0 ? a*b : -a*b;
}
```

# Principle of Least Astonishment

- Interface should "hint" at its behavior

Bad:

```
int product(int a, int b) {
    return a*b > 0 ? a*b : -a*b;
}
```

Better:

```
/** Return absolute value of a * b */
int absProduct(int a, int b) {
    return a*b > 0 ? a*b : -a*b;
}
```

- Names and comments matter!

# Outsmarting yourself

- A useful shorthand... Instead of

```
something = something * 2;
```

... use

```
something *= 2;
```

- All such operators:

```
+=    -=    *=    /=    %=    ^=
```

# Principle of Least Astonishment

- Unexpected side effects are a Bad Thing

```
int times(int& value, int factor) {
     value *= factor;
     return value;
}
...
int i = 1;
int j = times(i,10);
```

**Developer trying to be clever. But what does code do to i?**

# Duplication

- It is common to find some chunk of working code, make a replica, then edit the replica
- But this makes your software fragile: later, when code you copied needs to be revised, either
  - The person doing that changes all instances, or
  - some become inconsistent
- Duplication can arise in many ways:
  - constants (repeated "magic numbers")
  - code vs. comment
  - within an object's state
  - ...

# “DRY” Principle

- Don’t Repeat Yourself

- Nice goal: have each piece of knowledge live in one place
- But don’t go crazy over it
  - DRYing up at any cost can increase dependencies between code
  - “3 strikes and you refactor” (i.e. clean up)

# Refactoring

- Refactor: improve code's internal structure without changing its external behavior
- Most of the time we're modifying existing software
- "Improving the design after it has been written"
- Refactoring steps can be very simple:

```
double weight(double mass) {
  return mass * 9.80665;
}
```

```
#define GRAVITY = 9.80665;
public double weight(double mass) {
  return mass * GRAVITY;
}
```

- Other examples: renaming variables, methods, classes

# Why is refactoring good?

- If your application later gets used as part of a NASA mission to Mars, it won't make mistakes
- Every place that the gravitational constant shows up in your program a reader will realize that this is what they are looking at
- The compiler may actually produce better code

# Common refactorings

- Rename something
  - IDE's like Eclipse will do it all through your code
  - Warning: Eclipse doesn't automatically fix comments!

- Take a chunk of your code and turn it into a method
  - Anytime your "instinct" is to copy lines of code from one place in your program to another and then modify, consider trying this refactoring approach instead...
  - ... even if you have to modify this new method, there will be just one "version" to debug and maintain!

# Extract Method

- A comment explaining what is being done usually indicates the need to extract a method

```
double totalArea() {
   ...
   // add the circle
   area +=
         PI * pow(radius,2);
   ...
}
```

```
double totalArea() {
   ...
   area += circleArea(radius);
   ...
}

double circleArea (double radius) {
   return PI * pow(radius, 2);
}
```

- One of most common refactorings

# Extract Method

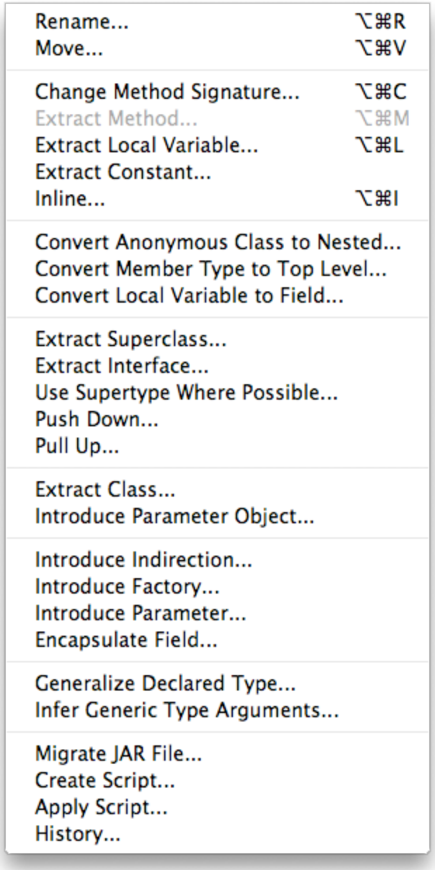
Before

```
if (date.before(SUMMER_START) ||
                date.after(SUMMER_END)) {
    charge = quantity * winterRate + winterServiceCharge;
}
else {
    charge = quantity * summerRate;
}
```

After

```
if (isSummer(date)) {
      charge = summerCharge(quantity);
}
else {
      charge = winterCharge(quantity);
}
```

# Refactoring & Tests




- Eclipse supports various refactorings
- You can refactor manually
  - Automated tests are essential to ensure external behavior doesn't change
  - Don't refactor manually without retesting to make sure you didn't break the code you were "improving"!
- More about unit testing later...

# Summary

- Our challenge is to use the features of programming languages to build clean, elegant software that doesn't duplicate functionality in confusing ways

- The developer's job is to find abstractions and use their insight to design better code!