Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Josh Bloch       Charlie Garrod
Growth of code—and complexity—over time

<table>
<thead>
<tr>
<th>System</th>
<th>Year</th>
<th>% of Functions Performed in Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-4</td>
<td>1960</td>
<td>8</td>
</tr>
<tr>
<td>A-7</td>
<td>1964</td>
<td>10</td>
</tr>
<tr>
<td>F-111</td>
<td>1970</td>
<td>20</td>
</tr>
<tr>
<td>F-15</td>
<td>1975</td>
<td>35</td>
</tr>
<tr>
<td>F-16</td>
<td>1982</td>
<td>45</td>
</tr>
<tr>
<td>B-2</td>
<td>1990</td>
<td>65</td>
</tr>
<tr>
<td>F-22</td>
<td>2000</td>
<td>80</td>
</tr>
</tbody>
</table>

**Millions of Lines of Code (MLOC)**

- Vista
- XP
- Win 2000
- NT 4
- NT 3.5
- NT 3.1

*(informal reports)*
Why Ford Just Became A Software Company

Ford is upgrading its in-vehicle software on a huge scale, embracing all the customer expectations and headaches that come with the development lifecycle.

Sometime early next year, Ford will mail USB sticks to about 250,000 owners of vehicles with its advanced touchscreen control panel. The stick will contain a major upgrade to the software for that screen. With it, Ford is breaking from a history as old as the auto industry, one in which the technology in a car essentially stayed unchanged from assembly line to junk yard.

Ford is significantly changing what a driver or passenger experiences in its cars years after they're built. And with it, Ford becomes a software company—with all the associated high customer expectations and headaches.
Normal night-time image  Blackout of 2003
Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Josh Bloch  Charlie Garrod
primes

binary tree

GCD

sorting

BDDs

graph search

sorting

BDDs
From programs to systems

Our goal: understanding both the building blocks and also the design principles for construction of software systems at scale
Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Josh Bloch         Charlie Garrod
Objects in the real world
Object-oriented programming

- Programming based on structures that contain both data and methods

```java
public class Bicycle {
    private int speed;
    private final Wheel frontWheel, rearWheel;
    private final Seat seat;
    ...

    public Bicycle(...) { ... }

    public void accelerate() {
        speed++;
    }

    public int getSpeed() { return speed; }
}
```
Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Josh Bloch  Charlie Garrod
Semester overview

- **Introduction**
  - Design goals, principles, patterns

- **Designing classes**
  - Design for change: Subtype polymorphism and information hiding
  - Design for reuse: inheritance and delegation

- **Designing (sub)systems**
  - What to build: Domain models, system sequence diagrams
  - Assigning responsibilities: GRASP patterns
  - Design for robustness: Exceptions, modular protection
  - Design for change (2): Façade, Adapter, Observer

- **Design case studies**
  - Graphical user interfaces
  - Streams, I/O
  - Collections

- **Design for large-scale reuse**
  - Libraries, APIs,
  - Frameworks
  - Product lines

- **Explicit concurrency**
- **Distributed systems**

- **Crosscutting topics:**
  - Modern development tools: IDEs, version control, build automation, continuous integration, static analysis
  - Modeling and specification, formal and informal
  - Functional correctness: Testing, static analysis, verification
void sort(int[] list, boolean ascending) {
    ...
    boolean mustSwap;
    if (ascending) {
        mustSwap = list[i] < list[j];
    } else {
        mustSwap = list[i] > list[j];
    }
    ...
}
interface Comparator {
    boolean compare(int i, int j);
}

class AscendingComparator implements Comparator {
    boolean compare(int i, int j) { return i < j; }
}
class DescendingComparator implements Comparator {
    boolean compare(int i, int j) { return i > j; }
}

void sort(int[] list, Comparator cmp) {
    ...
    boolean mustSwap =
        cmp.compare(list[i], list[j]);
    ...
}
interface Comparator {
    boolean compare(int i, int j);
}

final Comparator ASCENDING = (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

void sort(int[] list, Comparator cmp) {
    ...
    boolean mustSwap =
        cmp.compare(list[i], list[j]);
    ...
}
Which version is better?

Version A:
```java
void sort(int[] list, boolean ascending) {
    ...
    boolean mustSwap;
    if (ascending) {
        mustSwap = list[i] < list[j];
    } else {
        mustSwap = list[i] > list[j];
    }
    ...
}
```

Version B':
```java
interface Comparator {
    boolean compare(int i, int j);
}
final Comparator ASCENDING = (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

given void sort(int[] list, Comparator cmp) {
    ...
    boolean mustSwap =
        cmp.compare(list[i], list[j]);
    ...
}```
It depends?
Software engineering is the branch of computer science that creates practical, cost-effective solutions to computing and information processing problems, preferably by applying scientific knowledge, developing software systems in the service of mankind.

Software engineering entails making **decisions** under constraints of limited time, knowledge, and resources. [...] 

Engineering quality resides in engineering judgment. [...] Quality of the software product depends on the engineer’s faithfulness to the engineered artifact. [...] Engineering requires reconciling conflicting constraints. [...] Engineering skills improve as a result of careful systematic reflection on experience. [...] Costs and time constraints matter, not just capability. [...]
Software Engineering (SE) at CMU

• 15-214: Code-level design
  – Extensibility, reuse, concurrency, functional correctness
• 15-313: Human aspects of software development
  – Requirements, teamwork, scalability, security, scheduling, costs, risks, business models
• 15-413 Practicum, 17-413 Seminar, Internship
• Various Master's level courses on requirements, architecture, software analysis, etc.
• SE Minor: http://isri.cmu.edu/education/undergrad
Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Josh Bloch  Charlie Garrod
Concurrency

- Simply: doing more than one thing at a time
 Semester overview

- Introduction
  - Design goals, principles, patterns
- Designing classes
  - Design for change: Subtype polymorphism and information hiding
  - Design for reuse: inheritance and delegation
- Designing (sub)systems
  - What to build: Domain models, system sequence diagrams
  - Assigning responsibilities: GRASP patterns
  - Design for robustness: Exceptions, modular protection
  - Design for change (2): Façade, Adapter, Observer
- Design case studies
  - Graphical user interfaces
  - Streams, I/O
  - Collections
- Design for large-scale reuse
  - Libraries, APIs,
  - Frameworks
  - Product lines

- Explicit concurrency
- Distributed systems

- Crosscutting topics:
  - Modern development tools: IDEs, version control, build automation, continuous integration, static analysis
  - Modeling and specification, formal and informal
  - Functional correctness: Testing, static analysis, verification
Summary: Course themes

- Object-oriented programming
- Code-level design
- Analysis and modeling
- Concurrency and distributed systems
COURSE ORGANIZATION
Preconditions

• 15-122 or equivalent
  – Two semesters of programming
  – Knowledge of C-like languages

• Specifically:
  – Basic programming skills
  – Basic (formal) reasoning about programs
    • Pre/post conditions, invariants, formal verification
  – Basic algorithms and data structures
    • Lists, graphs, sorting, binary search, etc.
Learning goals

• Ability to **design** medium-scale programs
  – Design goals (e.g., design for change, design for reuse)
  – Design principles (e.g., low coupling, explicit interfaces)
  – Design patterns (e.g., strategy pattern, decorator pattern), libraries, and frameworks
  – Evaluating trade-offs within a design space
  – Paradigms such as event-driven GUI programming

• Understanding **OO programming** concepts & how they support design decisions
  – Polymorphism, encapsulation, inheritance, object identity

• Proficiency with basic **quality assurance** techniques for functional correctness
  – Unit testing
  – Static analysis
  – (Verification)

• Fundamentals of **concurrency and distributed systems**

• Practical skills
  – Ability to write medium-scale programs in Java
  – Ability to use modern development tools
    • VCS, IDE, debugger, build and test automation, static analysis, ...
Course staff

- Josh Bloch  
  [jjb@cs.cmu.edu](mailto:jjb@cs.cmu.edu)  
  Wean 4213

- Charlie Garrod  
  [charlie@cs.cmu.edu](mailto:charlie@cs.cmu.edu)  
  Wean 5101

- Teaching assistants: Jacob, Jordan, Kathleen, Kirn, Marcel, Nora, Reid, Tianyu
Course meetings

- Lectures: Tuesday and Thursday 3:00 – 4:20pm WEH 7500
- Recitations: Wednesdays 9:30 - … - 2:20pm
  - Supplementary material, hands-on practice, feedback
- Office hours: see course web page
Infrastructure

• Course website http://www.cs.cmu.edu/~charlie/courses/15-214
  – Schedule, assignments, lecture slides, policy documents
• Tools
  – Git, Github: Assignment distribution, hand-in, and grades
  – Piazza: Discussion board
  – Eclipse: Recommended for code development
  – Gradle, Travis-CI, Checkstyle, Findbugs: Practical development tools
• Assignments
  – Homework 1 available tomorrow in recitation
    • Ensure all tools work together: Git, Java, Eclipse, Gradle, Checkstyle
• First recitation is tomorrow
  – Introduction to Java and the tools in the course
  – Bring your laptop, if you have one!
  – Install Git, Java, Eclipse, Gradle beforehand
Textbooks

• Required course textbook:
  – Read chapters 14 and 16 by next Tuesday

• Additional texts on Java, concurrency, and design patterns recommended on the course web page
Grading policy (subject to change)

- 50% assignments
- 20% midterms (2 x 10% each)
- 20% final exam
- 10% quizzes and participation
  - *Bring paper and a pen/pencil to class!*

This course does not have a fixed letter grade policy; i.e., the final letter grades will not be A=90-100%, B=80-90%, etc.
Collaboration policy (also on course web page)

• *We expect your work to be your own*
  – You must clearly cite external resources so that we can evaluate your own personal contributions.

• Do not release your solutions (not even after end of semester)

• Ask if you have any questions

• If you are feeling desperate, please mail/call/talk to us
  – Always turn in any work you've completed *before* the deadline

• We use cheating detection tools
Late day policy

• You may turn in each* homework up to 2 days late
  – 5 free late days per semester
  – 10% penalty per day after free late days are used
    • ...but we don't accept work 3 days late
• See the syllabus for additional details
10% quizzes and participation

- Recitation participation counts toward your participation grade
- Lecture has in-class quizzes and faux participation quizzes
INTRODUCTION TO SOFTWARE DESIGN
Goal of software design

• For each desired program behavior there are infinitely many programs
  – What are the differences between the variants?
  – Which variant should we choose?
  – How can we synthesize a variant with desired properties?
Metrics of software quality

- **Sufficiency / functional correctness**
  - Fails to implement the specifications ... Satisfies all of the specifications

- **Robustness**
  - Will crash on any anomalous even ... Recovers from all anomalous events

- **Flexibility**
  - Will have to be replaced entirely if specification changes ... Easily adaptable to reasonable changes

- **Reusability**
  - Cannot be used in another application ... Usable in all reasonably related apps without modification

- **Efficiency**
  - Fails to satisfy speed or data storage requirement ... satisfies requirement with reasonable margin

- **Scalability**
  - Cannot be used as the basis of a larger version ... is an outstanding basis...

- **Security**
  - Security not accounted for at all ... No manner of breaching security is known

Source: Braude, Bernstein, Software Engineering. Wiley 2011
Why use a design process?

• A design process organizes your work
• A design process structures your understanding
• A design process facilitates communication
One simple process

1. Discuss software that needs to be written
2. Write some code
3. Test the code to identify the defects
4. Debug to find causes of defects
5. Fix the defects
6. If not done, return to step 1
Better software design

• Think before coding
• Consider non-functional quality attributes
  – Maintainability, extensibility, performance, ...
• Consider design alternatives
  – Make conscious design decisions
A design case study: Simulating an invasive species

Lodgepole Pine  
Mountain Pine Beetle  
Galleries carved in inner bark

Widespread tree death

Photo by Walter Siegmund

Real-world goal: Determine how best to save the trees

• Causes of the problem
  – Warmer winters: fewer beetles die
  – Fire suppression: more old (susceptible) trees

• Can management help? What form(s)?
  – Sanitation harvest
    • Remove highly infested trees
    • Remove healthy neighboring trees above a certain size
  – Salvage harvest
    • Remove healthy trees with several infested neighbors

Applying agent-based modeling

- **Goal:** evaluate different forest management techniques
  - Use a simulated forest based on real scientific observations
- **An agent-based model**
  - Create a simulated forest, divided into a grid
  - Populate the forest with agents: trees, beetles, forest managers
  - Simulate the agents over multiple time steps
  - Calibrate the model to match observations
  - Compare tree survival in different management strategies
    - and vs. no management at all

A design problem  (Version 1: Talk amongst yourselves)

• How should we organize our simulation code?
• Goals:
  – Separate simulation infrastructure from problem-specific details
    • May want to reuse infrastructure in other studies
    • Allows multiple developers to work in parallel
  – Make it easy to change the simulation setup
    • May need to adjust parameters before getting it right
  – Make it easy to add and remove problem-specific details
    • New elements may be needed for accurate simulation
A design problem (Version 2: Write your design on paper)

- How should we organize our simulation code?
- Goals:
  - Separate simulation infrastructure from problem-specific details
    - May want to reuse infrastructure in other studies
    - Allows multiple developers to work in parallel
  - Make it easy to change the simulation setup
    - May need to adjust parameters before getting it right
  - Make it easy to add and remove problem-specific details
    - New elements may be needed for accurate simulation
Common problems

• “I don’t even know how to start”
  – This course will help:
    • A process for design
    • Design patterns that you can apply
    • Principles for selecting among design alternatives
    • Techniques for documenting design for others

• “Is my design any good?”
• “You can’t solve problem in C / without OO!”
Preview: Design goals, principles, and patterns

- **Design goals** enable evaluation of designs
  - e.g. maintainability, reusability, scalability
- **Design principles** are heuristics that describe best practices
  - e.g. high correspondence to real-world concepts
- **Design patterns** codify repeated experiences, common solutions
  - e.g. template method pattern
Preview: The design process

- **Object-oriented analysis**
  - Understand the problem

- **Object-oriented design**
  - Cheaply create and evaluate plausible alternatives

- **Implementation**
  - Convert design to code
Summary: Design and objects

• Design follows a process
  – Structuring design helps us do it better

• Quality attributes drive software design
  – Properties that describe fitness for further development and use