Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Josh Bloch       Charlie Garrod       Darya Melicher
Software is everywhere
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>

(informal reports)
Normal night-time image  Blackout of 2003
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.
Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Josh Bloch    **Charlie Garrod**    Darya Melicher
From programs to systems

Writing algorithms, data structures from scratch → Reuse of libraries, frameworks

Functions with inputs and outputs → Asynchronous and reactive designs

Sequential and local computation → Parallel and distributed computation

Full functional specifications → Partial, composable, targeted models

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

Part 1: Introduction

Course overview and introduction to software design

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

- Programming based on structures that contain both data and methods

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

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

    public void accelerate() {
        speed++;
    }

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

Part 1: Introduction

Course overview and introduction to software design

Josh Bloch  Charlie Garrod  Darya Melicher
Semester overview

• Introduction to Java and O-O
• Introduction to design
  – Design goals, principles, patterns
• Designing classes
  – Design for change
  – Design for reuse
• Designing (sub)systems
  – Design for robustness
  – Design for change (cont.)
• Design case studies
• Design for large-scale reuse
• Explicit concurrency

• 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
static void sort(int[] list, boolean ascending) {
    ...
    boolean mustSwap;
    if (ascending) {
        mustSwap = list[i] > list[j];
    } else {
        mustSwap = list[i] < list[j];
    }
    ...
}
interface Order {
    boolean lessThan(int i, int j);
}

class AscendingOrder implements Order {
    public boolean lessThan(int i, int j) { return i < j; }
}
class DescendingOrder implements Order {
    public boolean lessThan(int i, int j) { return i > j; }
}

class AscendingOrder implements Order {
    public boolean lessThan(int i, int j) { return i < j; }
}

static void sort(int[] list, Order order) {
    ...
    boolean mustSwap =
        order.lessThan(list[j], list[i]);
    ...
}
interface Order {
    boolean lessThan(int i, int j);
}

final Order ASCENDING = (i, j) \rightarrow i < j;
final Order DESCENDING = (i, j) \rightarrow i > j;

static void sort(int[] list, Order order) {
    ...
    boolean mustSwap =
        order.lessThan(list[j], list[i]);
    ...
}
Which version is better?

Version A:

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

```
interface Order {
    boolean lessThan(int i, int j);
}
final Order ASCENDING = (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;
```

Version B':

```java
static void sort(int[] list, Order order) {
    ...
    boolean mustSwap =
        order.lessThan(list[j], list[i]);
    ...
}
```
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 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 for the 21st Century: A basis for rethinking the curriculum Manifesto, CMU-ISRI-05-108
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 create a variant with desired properties?
## Metrics of software quality, i.e., *design goals*

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional correctness</td>
<td>Adherence of implementation to the specifications</td>
</tr>
<tr>
<td>Robustness</td>
<td>Ability to handle anomalous events</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Ability to accommodate changes in specifications</td>
</tr>
<tr>
<td>Reusability</td>
<td>Ability to be reused in another application</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Satisfaction of speed and storage requirements</td>
</tr>
<tr>
<td>Scalability</td>
<td>Ability to serve as the basis of a larger version of the application</td>
</tr>
<tr>
<td>Security</td>
<td>Level of consideration of application security</td>
</tr>
</tbody>
</table>

*Source: Braude, Bernstein, Software Engineering. Wiley 2011*
A typical Intro CS design 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: broadly consider quality attributes
  – Maintainability, extensibility, performance, ...

• Propose, consider design alternatives
  – Make explicit design decisions
Using a design process

• A design process organizes your work
• A design process structures your understanding
• A design process facilitates communication
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
Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Josh Bloch    Charlie Garrod    Darya Melicher
Concurrency

- Roughly: doing more than one thing at a time
Summary: Course themes

• Object-oriented programming
• Code-level design
• Analysis and modeling
• Concurrency
Software Engineering (SE) at CMU

• 17-214: Code-level design
  – Extensibility, reuse, concurrency, functional correctness
• 17-313: Human aspects of software development
  – Requirements, teamwork, scalability, security, scheduling, costs, risks, business models
• 17-413 Practicum, 17-415 Seminar, Internship
• Various courses on requirements, architecture, software analysis, SE for startups, etc.
• SE Minor: [http://isri.cmu.edu/education/undergrad](http://isri.cmu.edu/education/undergrad)
COURSE ORGANIZATION
Preconditions

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

- **21-127 or equivalent**
  - Familiarity with basic discrete math concepts

- **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** and implement medium-scale programs
• Understanding **OO programming** concepts & design decisions
• Proficiency with basic **quality assurance** techniques for functional correctness
• Fundamentals of **concurrency**
• Practical skills
Course staff

• Josh Bloch  
jbloch@andrew.cmu.edu  
Wean 5311

• Charlie Garrod  
charlie@cs.cmu.edu  
Wean 5120

• Darya Melicher  
darya@cs.cmu.edu  
Wean 4105

• Teaching assistants: Caroline, Dan, Diego, Echo, Megan, Rosie
Course meetings

• Lectures: Tuesday and Thursday, noon – 1:20pm, Wean 7500
  – Electronic devices discouraged

• Recitations: Wednesdays 9:30 - ... - 2:20pm
  – Supplementary material, hands-on practice, feedback
  – Bring your laptop

• Office hours: see course web page
Infrastructure

• Course website: http://www.cs.cmu.edu/~charlie/courses/17-214
  – Schedule, office hours calendar, lecture slides, policy documents
• Tools
  – Git, Github: Assignment distribution, hand-in, and grades
  – Piazza: Discussion board
  – IntelliJ or Eclipse: Recommended for code development (other IDEs are fine)
  – Gradle, Travis-CI, Checkstyle, Findbugs: Practical development tools
• Assignments
  – Homework 1 available tomorrow
• First recitation is tomorrow
  – Introduction to Java and the tools in the course
  – Install Git, Java, some IDE, Gradle beforehand
Textbooks

- Required course textbooks (electronically available through CMU library):

- Additional readings on design, Java, and concurrency on the course web page
Approximate grading policy

• 50% assignments
• 20% midterms (2 x 10% each)
• 20% final exam
• 10% quizzes and participation

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 see the course syllabus)

• *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

• You must sign and return a copy of the collaboration policy before we will grade your work: [https://goo.gl/CBXKQK](https://goo.gl/CBXKQK)
Late day policy

• You may turn in each* homework up to 2 days late
• You have five free late days per semester
  – 10% penalty per day after free late days are used
• We don't accept work 3 days late
• See the syllabus for additional details
• Got extreme circumstances? Talk to us
10% quizzes and participation

- Recitation participation counts toward your participation grade
- Lecture has in-class quizzes
Summary

• Software engineering requires decisions, judgment
• Good design follows a process
• You will get lots of practice in 17-214!