Principles of Software Construction: Objects, Design, and Concurrency

Course Introduction

Jonathan Aldrich    Charlie Garrod

© 2012 W Scherlis and J Aldrich
Software and automobiles

1 Adaptive Cruise Control
2 Electronic Brake System MK60E
3 Sensor Cluster
4 Gateway Data Transmitter
5 Force Feedback Accelerator Pedal
6 Door Control Unit
7 Sunroof Control Unit
8 Reversible Seatbelt Pre-tensioner
9 Seat Control Unit
10 Brakes
11 Closing Velocity Sensor
12 Slide Satellites
13 Upfront Sensor
14 Airbag Control Unit

<table>
<thead>
<tr>
<th>Air-bag system</th>
<th>Antilock brakes</th>
<th>Automatic transmission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm system</td>
<td>Climate control</td>
<td>Collision-avoidance system</td>
</tr>
<tr>
<td>Cruise control</td>
<td>Communication system</td>
<td>Dashboard instrumentation</td>
</tr>
<tr>
<td>Electronic stability control</td>
<td>Engine ignition</td>
<td>Engine control</td>
</tr>
<tr>
<td>Electronic-seat control</td>
<td>Entertainment system</td>
<td>Navigation system</td>
</tr>
<tr>
<td>Power steering</td>
<td>Tire-pressure monitoring</td>
<td>Windshield-wiper control</td>
</tr>
</tbody>
</table>
Moore’s Law: transistors per chip

Similar curve for **memory**, slightly steeper
How much software?

<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)
The limits of exponentials

computing capability

human capacity

capability

time
Scaling Up: From Programs to Systems

• You’ve written small- to medium-size programs in 15-122

• This course is about managing software complexity
  ▪ **Scale** of code: KLOC -> MLOC
  ▪ Worldly **environment**: external I/O, network, asynchrony
  ▪ Software **infrastructure**: libraries, frameworks, components
  ▪ Software **evolution**: change over time

  ▪ Contrast: algorithmic complexity
    • Not an emphasis in this course
Our goal: understanding both the **building blocks** and also the **principles** for construction of software systems at scale
A framework for mobile app software (IOS)
The four course themes

- **Threads and Concurrency**
  - Concurrency is a crucial system abstraction
    - E.g., background computing while responding to users
  - Concurrency is necessary for performance
    - Multicore processors and distributed computing
  - *Our focus*: application-level concurrency
    - Cf. functional parallelism (150, 210) and systems concurrency (213)

- **Object-oriented programming**
  - Excels for flexible designs and reusable code
  - A primary paradigm in industry – basis for modern frameworks
  - Focus on Java – used in industry, some upper-division courses

- **Analysis and Modeling**
  - *Practical* specification techniques and verification tools
  - Address challenges of threading, correct library usage, etc.

- **Design**
  - Proposing and evaluating alternatives
  - Modularity, information hiding, and planning for change
  - Patterns: well-known solutions to design problems
Motivating example #1: GraphLayout

Source code: http://java.sun.com/applets/jdk/1.4/demo/applets/GraphLayout/example1.html
Screenshot from http://stackoverflow.com/questions/1318770/impressive-examples-in-java
Discussion: GraphLayout

• What does the design of GraphLayout look like, conceptually?

• What is most important about the design?

• How should the GUI be organized? Why?
Motivating example #2: Virtual Worlds
Discussion: Virtual Worlds

- How can the virtual world scale to thousands of users?

- How can we organize the system to easily add new things?

- How can we support different kinds of things, while taking advantage of their similarities? (can you think of an example?)
Considering the examples

- **Threads and Concurrency**
  - In the GUI-based app
  - On game clients
  - On the game servers

- **Object-oriented programming**
  - Organizing by object types, then actions

- **Analysis and Modeling**
  - How to gain confidence regarding *all* possible executions

- **Design**
  - How to organize systems that grow and evolve
  - How to define the interfaces between infrastructure and our code
Principles of Software Construction: Objects, Design, and Concurrency

Course Mechanics

Jonathan Aldrich    Charlie Garrod

© 2012 W Scherlis and J Aldrich
Course Preconditions

• Preconditions
  ▪ 15-122 or equivalent
    • 2 semesters of programming, knowledge of C-like languages
  ▪ That is:
    • Basic programming skills
    • Basic reasoning about programs
    • Basic algorithms and data structures
Course Post-conditions

• **Post-conditions**
  - **OO understanding**
    - Objects, classes, types
    - Java development skills
  - Understanding larger-scale software
    - Design patterns
    - Design and use of libraries and frameworks
  - Modeling and analysis
    - Use of development, testing, and analysis tools
  - Concurrent and distributed systems
    - Scaling and performance
    - Safe programming practices for explicit concurrency
Important features of this course

• The team
  ▪ Instructors
    • Jonathan Aldrich jonathan.aldrich@cs.cmu.edu
    • Charlie Garrod charlie@cs.cmu.edu
  ▪ TAs
    • Patrick Woody pwoody@andrew.cmu.edu [Section A]
    • YoungSeok Yoon youngseok@cs.cmu.edu [Section B]
    • Alex Duda aduda@andrew.cmu.edu [Section C]
    • Saagar Sethi saagars@andrew.cmu.edu [Section D]
    • Ivan Ruchkin iruchkin@cs.cmu.edu [Cross-Section TA]

• The schedule
  ▪ Lectures
    • Tues, Thurs 3:00 – 4:20pm HBH 1000
  ▪ Recitations
    • A: Weds 9:30-10:20am WEH 5310
    • B: Weds 10:30-11:20am WEH 5310
    • C: Weds 11:30-12:20pm WEH 5310
    • D: Weds 12:30-1:20pm WEH 5310
  ▪ Labs
    • Monday 9:30-1:30 in GHC 5201
  ▪ Office hours
    • To be announced – see course web page

Recitations are required
Important features of this course

• Course website
  ▪ Schedule, assignments, lecture slides, policy documents
    http://www.cs.cmu.edu/~aldrich/214

• Tools
  ▪ Subversion
    ▪ Assignment distribution, handin, and grades
  ▪ Piazza
    ▪ Discussion site – link from course page
  ▪ Eclipse
    ▪ Recommended for developing code

• Assignments
  ▪ The first is easy, but important
    ▪ Ensure all tools are working together
    ▪ Subversion, Java, Eclipse

• First recitation, tomorrow
  ▪ Introduction to Java and the tools in the course
  ▪ Bring your laptop, if you have one!
    ▪ Install Subversion, Java, Eclipse beforehand – instructions on Piazza
Policies

• Late days for assignments
  ▪ Five total for the semester
  ▪ Use a maximum of 2 late days per assignment
  ▪ No other late work accepted, except under extreme circumstances

• Recitations
  ▪ Fleshing out lecture with new material
  ▪ Practice of lecture material
  ▪ Discussion, presentations, etc.
  ▪ Attendance is required

• Labs
  ▪ Mini-lecture at the beginning
  ▪ Task that will help you on the homework
  ▪ Doing the task is required, attendance at lab is not required
  ▪ Not included every week

• No official lab on Labor Day
  ▪ But Pat and Alex will be in GHC 5201 at 9:30 to help with HW0
  ▪ This is in addition to regular office hours this week
Policies

• **Grading** (*subject to adjustment*)
  - 60% assignments
  - 35% midterm and final exam
  - 5% participation
    - Worksheets periodically used in class and recitation

• **Cheating**
  - Collaboration policy is on the course website
    - Read it—you are responsible for knowing what it says!
  - Ask if you have any questions
  - If you are feeling desperate, please reach out to us

• **Texts**
  - Alan Shalloway and James Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design* (2nd Ed).
  - Several free online texts (Java, etc.)
Principles of Software Construction: Objects, Design, and Concurrency

Objects

Jonathan Aldrich  Charlie Garrod

© 2012 W Scherlis and J Aldrich
Object orientation (OO)

• History
  ▪ Simulation – Simula 67, first OO language
  ▪ Interactive graphics – SmallTalk-76 (inspired by Simula)

• Object-oriented programming (OOP)
  ▪ Organize code bottom-up rather than top-down
  ▪ Focus on concepts rather than operations
  ▪ Concepts include both conventional data types (e.g. List), and other abstractions (e.g. Window, Command, State)

• Some benefits, informally stated
  ▪ Easier to reuse concepts in new programs
    ▪ Concepts map to ideas in the target domain
  ▪ Easier to extend the program with new concepts
    ▪ E.g. variations on old concepts
  ▪ Easier to modify the program if a concept changes
    ▪ Easier means the changes can be localized in the code base
Objects

- **Object**
  - A package of state (data) and behavior (actions)

- **Data and actions**
  - **Fields** in the object hold data values
    - Like the fields of a struct in C
    - Access to fields can be restricted
  - **Methods** describe operations or actions on that data
    - Like functions associated with an abstract data type
    - They have access to the all fields
    - Method calls can be thought of as “messages” to the object

- **Thus...**
  - **Methods** can control access to the fields
    - Best practice: Don’t allow fields to be seen from outside
  - The **object** can be thought of as a *service* that is accessed through a managed interface. The **class** described a family of similar services.
    - E.g., a particular button (object) vs. buttons in general (class)
Example: Concept of a Rectangle

• What do you need to know about a rectangle?

• What might you want to do with a rectangle?
Example: Points and Rectangles

```java
class Point {
    int x, y;
    int getX() { return x; } // a method; getY() is similar
    Point(int px, int py) { x = px; y = py; } // constructor for creating the object
}
class Rectangle {
    Point origin;
    int width, height;
    Point getOrigin() { return origin; }
    int getWidth() { return width; }
    void draw() {
        drawLine(origin.getX(), origin.getY(), // first line
                origin.getX()+width, origin.getY());
        ... // more lines here
    }
    Rectangle(Point o, int w, int h) {
        origin = o; width = w; height = h;
    }
}
```
Example: Points and Rectangles

```java
class Point {
    int x, y;
    int getX() { return x; }
    // a method; getY() is similar
    Point(int px, int py) { x = px; y = py; }
    // constructor for creating the object
}

class Rectangle {
    Point origin;
    int width, height;
    Point getOrigin() { return origin; }
    int getWidth() { return width; }
    void draw() {
        drawLine(origin.getX(), origin.getY(), origin.getX()+width, origin.getY());
        ... // more lines here
    }
    Rectangle(Point o, int w, int h) {
        origin = o; width = w; height = h;
    }
}

Some Client Code

Point o = new Point(0, 10); // allocates memory, calls ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin.getX() + r.getWidth(); // 15
```
Controlling access by client code

```java
class Point {
    private int x, y;
    public int getX() { return x; } // a method; getY() is similar
    public Point(int px, int py) { x = px; y = py; } // constructor for creating the object
}
class Rectangle {
    private Point origin;
    private int width, height;
    public Point getOrigin() { return origin; }
    public int getWidth() { return width; }
    public void draw() {
        drawLine(origin.getX(), origin.getY(), // first line
            origin.getX()+width, origin.getY());
        ... // more lines here
    }
    public Rectangle(Point o, int w, int h) {
        origin = o; width = w; height = h;
    }
}
```
Hiding interior state

```java
class Point {
    private int x, y;
    public int getX() { return x; }
    public Point(int px, int py) { x = px; y = py; }
    // constructor for creating the object
}
class Rectangle {
    private Point origin;
    private int width, height;
    public Point getOrigin() { return origin; }
    public int getWidth() { return width; }
    public void draw() {
        drawLine(origin.getX(), origin.getY(),
        // first line
        origin.getX()+width, origin.getY());
        // more lines here
    }
    public Rectangle(Point o, int w, int h) {
        origin = o; width = w; height = h;
    }
    // Client Code
    Point o = new Point(0, 10); // allocates memory, calls ctor
    Rectangle r = new Rectangle(o, 5, 10);
    r.draw();
    int rightEnd = r.getOrigin.getX() + r.getOrigin.getY(); // 15
    // Client Code that will not work in this version
    Point o = new Point(0, 10); // allocates memory, calls ctor
    Rectangle r = new Rectangle(o, 5, 10);
    r.draw();
    int rightEnd = r.origin.x + r.width; // trying to “look inside”
}
```

Some Client Code

```java
Point o = new Point(0, 10); // allocates memory, calls ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin.getX() + r.getOrigin.getY(); // 15
```
Hiding interior state

```java
class Point {
  private int x, y;
  public int getX() { return x; }
  public Point(int px, int py) { x = px; y = py; }
}

class Rectangle {
  private Point origin;
  private int width, height;
  public Point getOrigin() { return origin; }
  public int getWidth() { return width; }
  public void draw() {
    drawLine(origin.getX(), origin.getY(), // first line
       origin.getX()+width, origin.getY());
    ... // more lines here
  }
  public Rectangle(Point o, int w, int h) {
    origin = o; width = w; height = h;
  }
}
```

**Discussion:**
- What are the benefits of private fields?
- Methods can also be private – why is this useful?
The keyword **this** refers to the “receiver”

```java
class Point {
    private int x, y;
    public int getX() { return x; }
    public Point(int px, int py) { x = px; y = py; }
}
```

*can also be written in this way:*

```java
class Point {
    private int x, y;
    public int getX() { return x; }
    public Point(int x, int y) { this.x = x; this.y = y; }
}
```
Toad’s Take-Home Messages

• 214: managing complexity, from programs to systems
  ▪ Threads and concurrency
  ▪ Object-oriented programming
  ▪ Analysis and modeling
  ▪ Design

• GraphLayout and virtual worlds illustrate some challenges

• Object-oriented programming organizes code around concepts
  ▪ Methods capture behavior, fields capture state
  ▪ As we will see, this organization allows
    ▪ Greater reuse of concepts
    ▪ Better support for change when concepts vary