Principles of Software Construction: Objects, Design, and Concurrency

Course Introduction

Jonathan Aldrich    Charlie Garrod
Construction of

Software Systems

at Scale
Libraries
Reuse
Design
Analysis
Concurrency
primes  graph search
binary tree  GCD
sorting  BDDs
Software and automobiles

<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
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
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 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**
  - 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 to 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

Objects

Jonathan Aldrich  Charlie Garrod
Object-Oriented Programming Languages

- C++
- Java
- C#
- Smalltalk
- Scala
- Objective-C
- JavaScript
- Ruby
- PHP5
- Object Pascal/Delphi
- OCaml
- ...
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 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

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

class Point {
    int x, y;
    int getX() { return this.x; } // a method; getY() is similar
    Point(int px, int py) {this.x = px; this.y = py; } // constructor for creating the object
}
class Rectangle {
    Point origin;
    int width, height;
    Point getOrigin() { return this.origin; }
    int getWidth() { return this.width; }
    void draw() {
        this.drawLine(this.origin.getX(), this.origin.getY(), // first line
                      this.origin.getX()+this.width, this.origin.getY());
    }
}

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(); // 5
What’s really going on?

Some Client Code

Point o = new Point(0, 10); // a Point is an instance of the class Point
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin().getX() + r.getWidth(); // 5
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