Principles of Software Construction: Objects, Design, and Concurrency

Introduction, Overview, and Syllabus

Christian Kästner  Charlie Garrod

Growth of code—and complexity—

<table>
<thead>
<tr>
<th>System</th>
<th>Year</th>
<th>% of Functions Referenced in Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>F4</td>
<td>1960</td>
<td>10</td>
</tr>
<tr>
<td>F6</td>
<td>1961</td>
<td>30</td>
</tr>
<tr>
<td>F6.1</td>
<td>1970</td>
<td>30</td>
</tr>
<tr>
<td>F2.2</td>
<td>1975</td>
<td>33</td>
</tr>
<tr>
<td>E2.1</td>
<td>1982</td>
<td>47</td>
</tr>
<tr>
<td>E2</td>
<td>1989</td>
<td>60</td>
</tr>
<tr>
<td>F.2</td>
<td>2000</td>
<td>80</td>
</tr>
</tbody>
</table>

Principles of Software Construction

• You've written small- to medium-size programs in 15-122
• This course is about software design and managing software complexity
  – Scale of code: KLOC -> MLOC, design at scale
  – Worldly environment: external I/O, network, asynchrony
  – Software infrastructure: libraries, frameworks, design for reuse
  – Software evolution: design for change over time
  – Correctness: testing, static analysis tools, automation
  – In contrast: algorithmic complexity not an emphasis in 15-214
Our goal: understanding both the building blocks and also the design principles for construction of software systems at scale.

**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

**Course themes**

- **Code-level Design**
  - Process – how to start
  - Patterns – re-use conceptual solutions
  - Criteria – e.g., evolveability, understandability
- **Analysis and Modeling**
  - Practical specification techniques and verification tools
- **Object-oriented programming**
  - Evolveability, Reuse
  - Industry use – basis for frameworks
  - Vehicle is Java – industry, upper-division courses
- **Threads and Concurrency**
  - System abstraction – background computing
  - Performance
  - Our focus: explicit, application-level concurrency
  - Cf. functional parallelism (150, 210) and systems concurrency (213)

**This is not a Java course but you will write a lot of Java code**

```java
int a = 010 + 3;
System.out.println("A" + a);
```

```java
void sort(int[] list, String order) {
    boolean mustswap;
    if (order.equals("up")) {
        mustswap = list[i] < list[j];
    } else if (order.equals("down")) {
        mustswap = list[i] > list[j];
    } else {
        return;
    }
}
```

**Sorting with configurable order, variant A**
void sort(int[] list, Comparator cmp) {
    ... boolean mustswap;
    mustswap = cmp.compare(list[i], list[j]);
    ...}
}
interface Comparator {
    boolean compare(int i, int j);}
class UpComparator implements Comparator {
    boolean compare(int I, int j) { return I<j; }}
class DownComparator implements Comparator {
    boolean compare(int I, int j) { return I>j; }}

(by the way, this design is called "strategy pattern")

Software Engineering at CMU

- 15-214: "Code-level" design
  - extensibility, reuse, concurrency, functional correctness
- 15-313: "Human aspects" of software development
  - requirements, team work, scalability, security, scheduling, costs, risks, business models
- 15-413, 17-413 Practicum, Seminar, Internship
- Various master-level courses on requirements, architecture, software analysis, etc
- SE Minor: http://isri.cmu.edu/education/undergrad/
Course preconditions

• 15-122 or equivalent
  – 2 semesters of programming, knowledge of C-like languages

• Specifically:
  – Basic programming skills
  – Basic (formal) reasoning about programs with pre/post conditions, invariants, formal verification of correctness
  – Basic algorithms and data structures (lists, graphs, sorting, binary search, …)

High-level learning goals

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

2. Understanding object-oriented programming concepts and how they support design decisions
   – Polymorphism, encapsulation, inheritance, object identity

3. Proficiency with basic quality assurance techniques for functional correctness
   – Unit testing
   – Static analysis
   – Verification

4. Fundamentals of concurrency and distributed systems

5. Practical skills
   – Ability to write medium-scale programs in Java
   – Ability to use modern development tools, including VCS, IDEs, debuggers, build and test automation, static analysis

Text books

• Required course textbook:
  – Covers the design process and most design patterns
  – Regular reading assignments + in-class quizzes
  – Read chapters 14 and 16 until next Tuesday

• Additional texts on Java, concurrency, and design patterns recommended on the course web page
Course policies
• Grading (subject to adjustment)
  – 50% assignments
  – 20% midterms (2 x 10% each)
  – 20% final exam
  – 10% quizzes and participation
  • Bring paper and a pen/pencil to class!
• Collaboration policy on the course website
  – We expect your work to be your own
  – Do not release your solutions (not even after end of semester)
  – Ask if you have any questions.
  – If you are feeling desperate, please reach out to us
  – Always turn in any work you’ve completed before the deadline.
  – We run cheating detection tools. Trust us, academic integrity meetings are painful for everybody.
• Late days for homework assignments
  – 2 possible late days per deadline (exceptions will be announced)
  • 5 total free late days for semester (1 separate 2 late days for assignments done in pairs)
  • Beyond 5 free late days, penalty 1% per 5 minutes, up to 10% per day
  – After 2 possible late days: Penalty 1% per 5 minutes, up to 100%
  – Extreme circumstances – talk to us
• Recitations
  – Practice of lecture material
  – Presentation of additional material
  – Discussion, presentations, etc.
  – Attendance is required
  – In general, bring a laptop if you can

Today’s Learning Goals
• Introduce the design process through an example
• Understand what drives design

Goal of Software Design
• For each desired program behavior there are infinitely many programs that have this behavior
  – What are the differences between the variants?
  – Which variant should we choose?
• Since we usually have to synthesize rather than choose the solution...
  – How can we design a variant that has the desired properties?

Software Quality
• Sufficiency / Functional Correctness
  • Fails to implement the specifications ...
  • Satisfies all of the specifications
• Robustness
  • Will crash on any anomalous event ...
  • Functions from all anomalous events
• Flexibility
  • Must 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 applications without modification
• Efficiency
  • Fails to satisfy speed or data storage requirement ...
  • Satisfies speed or data storage 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
Why a Design Process?

- Without a process, how do you know what to do?
  - A process tells you what is the next thing you should be doing
- A process structures learning
  - We can discuss individual steps in isolation
  - You can practice individual steps, too
- If you follow a process, we can help you better
  - You can show us what steps you have done
  - We can target our advice to where you are stuck

A simple process

1. Discuss the 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

Software Design

- Think before coding
- Consider quality attributes (maintainability, extensibility, performance)
- Consider alternatives and make conscious design decisions

Preview: Goals, Principles, Patterns

- Design goals enable evaluation of designs and discussion of tradeoffs
- Design requires experience, learn and generalize from examples, discover good solutions
- Principles describe best practices
- Patterns codify experiences: established solutions for common problems; building blocks and vocabulary

Preview: The design process

- Object-Oriented Analysis
  - Understand the problem
  - Identify the key concepts and their relationships
  - Build a (visual) vocabulary
  - Create a domain model (aka conceptual model)
- Object-Oriented Design
  - Identify software classes and their relationships with class diagrams
  - Assign responsibilities (attributes, methods)
  - Explore behavior with interaction diagrams
  - Explore design alternatives
  - Create an object model (aka design model and design class diagram) and interaction models
- Implementation
  - Map designs to code, implementing classes and methods

Case Study: Pines and Beetles

- Lodgepole Pine
- Mountain Pine Beetle
- Galleries carved in inner bark
- Widespread tree death

How to save the trees?

- Causes
  - Warmer winters → fewer beetles die
  - Fire suppression → more old (susceptible) trees
- Can management help? And what form of management?
  - Sanitation harvest
    - Remove highly infested trees
    - Remove healthy neighboring trees above a certain size
  - Salvage harvest
    - Remove healthy trees that have several infested neighbors

Applying Agent-Based Modeling to the Pine Beetle Problem

- 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


Simulating Pines and Beetles

- Pine trees
  - Track growth—beetles only infect trees with thick enough bark
  - Seedling germination and natural tree death
- Infestations
  - Growth in the number of beetles per tree
  - Spreads to nearby trees once the infestation is strong enough
  - Kills the tree once there are enough beetles
- Forest manager
  - Applies sanitation or salvage harvest
- Others?
  - Statistics gathering agent?
  - Climate? (cold winters kill beetles)
  - Competing trees? (the Douglas Fir is not susceptible)
- Agent operations
  - Simulation of a time step
  - Logging (and perhaps reseeding) state

A Design Problem

- How should we organize our simulation code?
- Considerations (“Quality Attributes”)
  - Separate the simulation infrastructure from forest agents: We may want to reuse it in other studies and have multiple developers work in parallel
  - Make it easy to change the simulation setup: We want need to adjust the parameters before getting it right
  - Make it easy to add and remove agents: New elements may be needed for accurate simulation
Exercise (small groups, on paper)
• Sketch a design for the simulator
  – Ideally such that it can be extended (e.g., adding new agents without changing the simulation logic)
  – Such that work is decomposed into several modules/files
  – Use whatever notation (lines and boxes, code, etc) seems convenient

Design Exercise - Reflection
• “I didn’t know how to get started”
  – 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 that problem in C / without OO!”

The Simulation Architecture
Simulation Framework
Runs the simulation
Should not be forest specific
Should not need to modify when adding an agent or running a new simulation
Choose any subset, or easily add new agents
Simulation Driver
Change easily and independently of the simulation and agents
Each box should be a separate module (or file) of code

Simulation Framework Behavior Model

Idea: Managing the Agents
• Problem constraints
  – Functionality: framework invokes agents
  – Extension: add agents without changing framework code
• Consequence: framework must keep a list of agents
  – E.g. one per tree, or one for all Lodgepole trees
  – List must be open-ended, for extensibility
  – List must be populated by simulation driver
• Consequence: behavior tied to each agent
  – Framework invokes time step or logging actions
  – Each agent does timestep() and logState() differently
  – Framework can’t “know” which agent is which
  – So agent must “know” its own behavior

Design Questions: Who is Responsible for...
• Creating the list of agents?
• Storing the list of agents?
• Running the simulation?
• Implementing agent behavior?
• Storing agent state?
Who is Responsible for...?

- Creating the list of agents?
  - The Simulation Driver, because it is the only thing that should change when we add or remove an agent.

- Storing the list of agents?
  - The Simulation Framework, because it invokes them.

- Running the simulation?
  - The Simulation Framework, because it is the reusable code.

- Implementing agent behavior?
  - Each agent, because we must be able to add new agents and their behavior together.

- Storing agent state?
  - Each agent, because the state to be stored depends on the agent's behavior.

The Simulation Framework and Driver Code

```java
class Simulation {
    Agent[][] grid;
    int xSize;
    int ySize;

    void simulate() {
        for (int i = 0; i < NUM_STEPS; ++i) {
            for (int x = 0; x < xSize; ++x) {
                for (int y = 0; y < ySize; ++y) {
                    Agent a = grid[x][y];
                    if (a != null) {
                        a.timeStep(this);
                        a.logState();
                    }
                }
            }
        }
    }
}
```

Let's Run the Code!

```java
Simulation s = new Simulation();
for (int i = 0; i < NUM_TREES; ++i) {
    s.add(new LodgepolePine(...));
}
s.simulate();
```

Extending with Infestations

```java
Simulation Driver
void main(...) {
    Simulation s = new Simulation();
    for (int i = 0; i < NUM_TREES; ++i) {
        s.add(new LodgepolePine(...));
    }
    s.simulate();
}
```

Let's Run the Code Again!

```java
Simulation s = new Simulation();
for (int i = 0; i < NUM_TREES; ++i) {
    s.add(new LodgepolePine(...));
}
for (int i = 0; i < NUM_INFECT; ++i) {
    s.add(new InfectedPine(...));
}
s.simulate();
```

Next Week: How Objects Respond to Messages

1. assign a0 to grid[0]
2. assign a1 to grid[1]
3. invoke grid[0].timeStep()
4. invoke grid[1].timeStep()
Historical Note: Simulation and the Origins of Objects

• Simula 67 was the first object-oriented programming language
• Developed by Kristin Nygaard and Ole-Johan Dahl at the Norwegian Computing Center
• Developed to support discrete-event simulations
  – Much like our tree beetle simulation
  – Application: operations research, e.g. for traffic analysis
  – Extensibility was a key quality attribute for them
  – Code reuse was another—which we will examine later

Takeaways: Design and Objects

• Design follows a process
  – Structuring design helps us do it better
• Quality attributes drive software design
  – Properties of software that describe its fitness for further development and use
• Objects support extensibility, modifiability
  – Interfaces capture a point of extension or modification
  – Classes provide extensions by implementing the interface
  – Method calls are dispatched to the method’s implementation in the receiver object’s class