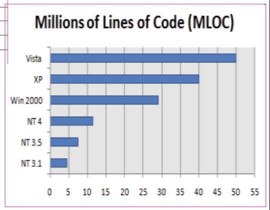
Principles of Software Construction: Objects, Design, and Concurrency

Introduction, Overview, and Syllabus

Christian Kästner Charlie Garrod

Growth of code—and complexity—

n System	Year	% of Functions Performed in Software
F-4	1960	8
A-7	1964	10
F-111	1970	20
F-15	1975	35
F-16	1982	45
B-2	1990	65
F-22	2000	80



(informal reports)

COMMENTARY

Chris Murphy
Editor, InformationWeek

See more from this author

Tweet 164 Like 548 Share +1 21 Permalink

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.

6 Comments | Chris Murphy | November 14, 2011 09:31 AM

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

- 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

primes **graph search**
binary tree **GCD**
sorting
BDDs

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 **design principles** for construction of software systems at scale

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)

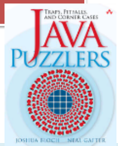
This is not a Java course

but you will write a lot of Java code


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



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



Sorting with configurable order, variant A

```
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];
  }
  …
}
```

Sorting with configurable order, variant B

```
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")

Tradeoffs?

```
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];
  }
  …
}
```

```
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; }}
```


it depends

(see context)

depends on what?
what are scenarios?
what are tradeoffs?

"**Software engineering** is the branch of computer science that creates practical, cost-effective solutions to computing and information processing problems, preferentially 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

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/

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 system
- 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

COURSE ORGANIZATION

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., 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
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, …

(See course web site for a full list of learning goals)

Important features of this course

- The team
  - Instructors
    - Christian Kästner kaestner@cs.cmu.edu Wean 5122
    - Charlie Garrod charlie@cs.cmu.edu Wean 5101
  - TAs: Jonathan, Matt, Nora, Pratik, Terence, Yongjin, Shurui
- The schedule
  - Lectures: Tues, Thurs 3:00 – 4:20pm DH 2315
  - Recitations: A-E: Weds 9:30 - … - 2:20pm WEH 5310
  - Office hours and emails see course web page
    - https://www.cs.cmu.edu/~charlie/courses/15-214/2015-spring/

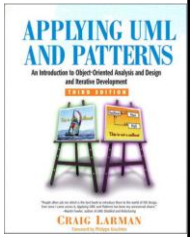
Recitations are required

Course 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 site – link from course page
  - Eclipse: Recommended for developing code
  - Gradle, Travis-CI, Checkstyle, Findbugs: Practical development tools
- Assignments
  - Homework 1 available tomorrow morning
    - Ensure all tools are working 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 – instructions on Piazza

Text books

- Required course textbook:
  - **Craig Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3rd Edition. Prentice Hall. 2004. ISBN 0-13-148906-2**
  - 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

Course policies

- Late days for homework assignments
  - 2 possible late days per deadline (exceptions will be announced)
    - 5 total free late days for semester (+ 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

INTRODUCTION TO SOFTWARE DESIGN

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?

Source: Braude, Bernstein, Software Engineering. Wiley 2011

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 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

Object-Design Challenges

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

Source: BC Forestry website

Case Study: Pines and Beetles

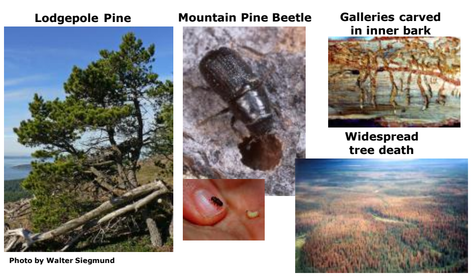


Photo by Walter Siegmund

Further reading: Liliana Péreza and Suzana Dragićević. Exploring Forest Management Practices Using an Agent-Based Model of Forest Insect Infestations. International Congress on Environmental Modelling and Software Modelling for Environment's Sake, 2010.

How to save the trees?

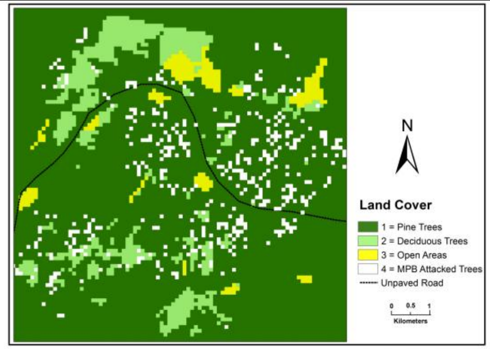
- Causes
  - Warmer winters → fewer beetles die
  - Fire suppression → more old (susceptible) trees
- Can management help? And what form of management?
  - Sanitation harvest
    - Remove highly infested trees
    - Remove healthy neighboring trees above a certain size
  - Salvage harvest
    - Remove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

Further reading: Liliana Péreza and Suzana Dragićević. Exploring Forest Management Practices Using an Agent-Based Model of Forest Insect Infestations. International Congress on Environmental Modelling and Software Modelling for Environment's Sake, 2010.

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

Further reading: Liliana Péreza and Suzana Dragićević. Exploring Forest Management Practices Using an Agent-Based Model of Forest Insect Infestations. International Congress on Environmental Modelling and Software Modelling for Environment's Sake, 2010.



Further reading: Liliana Péreza and Suzana Dragićević. Exploring Forest Management Practices Using an Agent-Based Model of Forest Insect Infestations. International Congress on Environmental Modelling and Software Modelling for Environment's Sake, 2010.

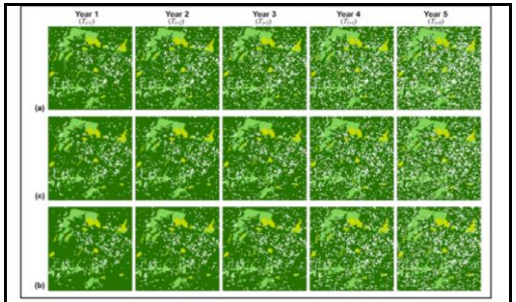


Figure 2. Five year simulation of lodgepole pine stands' mortality patterns using three different management strategies (a) MPB dispersion under no management, (b) MPB dispersion under a *sanitation harvest* strategy, and (c) MPB dispersion under a *salvage harvest* strategy. Legend for land uses are the same as on Fig. 1.

Further reading: Liliana Péreza and Suzana Dragićević. Exploring Forest Management Practices Using an Agent-Based Model of Forest Insect Infestations. International Congress on Environmental Modelling and Software Modelling for Environment's Sake, 2010.

Simulating Pines and Beetles

- Pine trees
  - Track size/age—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 susceptable)
- Agent operations
  - Simulation of a time step
  - Logging (and perhaps restoring) 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

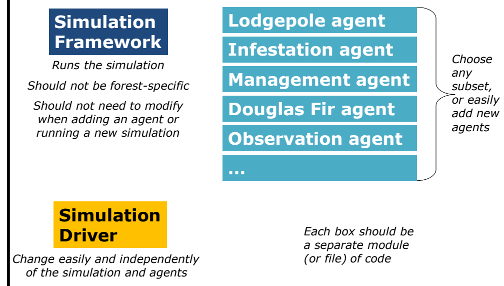
Exercise (small groups, on paper)

- 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

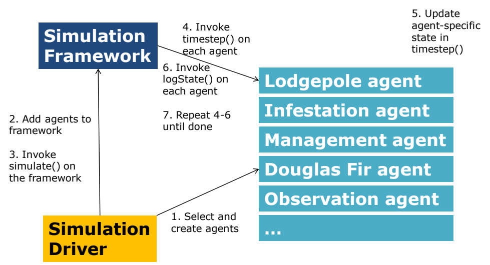
Design Exercise - Reflection

- "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 Behavior Model



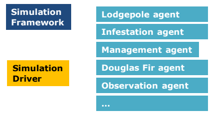
Idea: Managing the Agents

- 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" it's 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

Simulation Driver

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

** some keywords left out for simplicity*

Simulation Framework

```
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();
      }
  } }
  // other methods, such as add(Agent a)…
}
```

A two-dimensional array of Agents

Let's Run the Code!

Extending with Infestations

Simulation Driver

```
void main(…) {
    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()
}
```

We simply add InfectedPine objects to the Agents in the Simulation.

Separately, we implement an InfectedPine class.

** some keyword…*

Simulation Framework

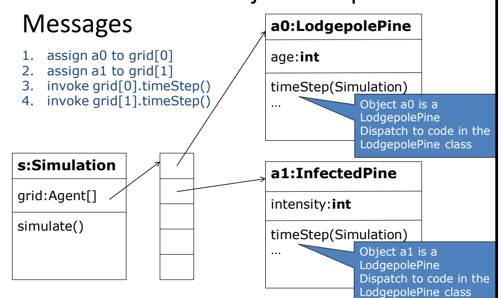
```
class Simulation {
  Agent grid[][];
  int xSize;
  int ySize;
  void simulate() {
   for (int i=0; i<NUM_ST…
    for (int x=0; x<xSiz…
     for (int y=0; y…
      Agent a = …
      if (a !…
       a…
  } }
  // other methods, such as add(Agent a)…
}
```

Unchanged!

Let's Run the Code Again!

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()



**simplification: we consider a 1-dimensional grid in this diagram*

Historical Note: Simulation and the Origins of Objects

Dahl and Nygaard at the time of Simula's development

- 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**