Principles of Software Construction: Objects, Design, and Concurrency

Course Introduction

toad

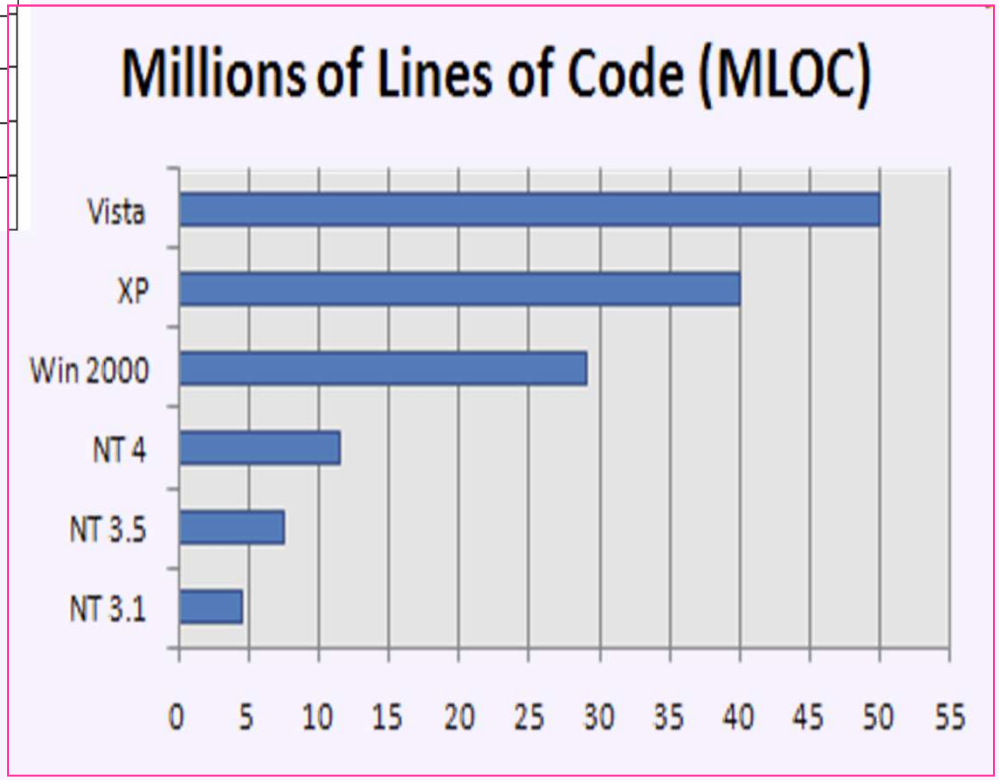
Fall 2014

Jonathan Aldrich Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

Growth of code—and complexity—over time

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)

Principles of Software Construction

- 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
  - Software **evolution**: design for change over time
  - Correctness: testing, static analysis

  - In contrast: algorithmic complexity not an emphasis in 15-214

primes

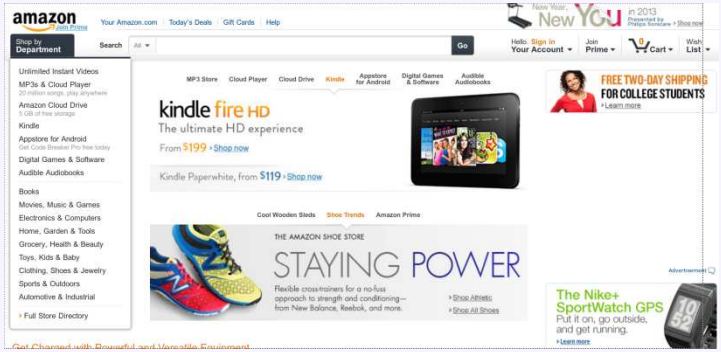
graph search

binary tree

GCD

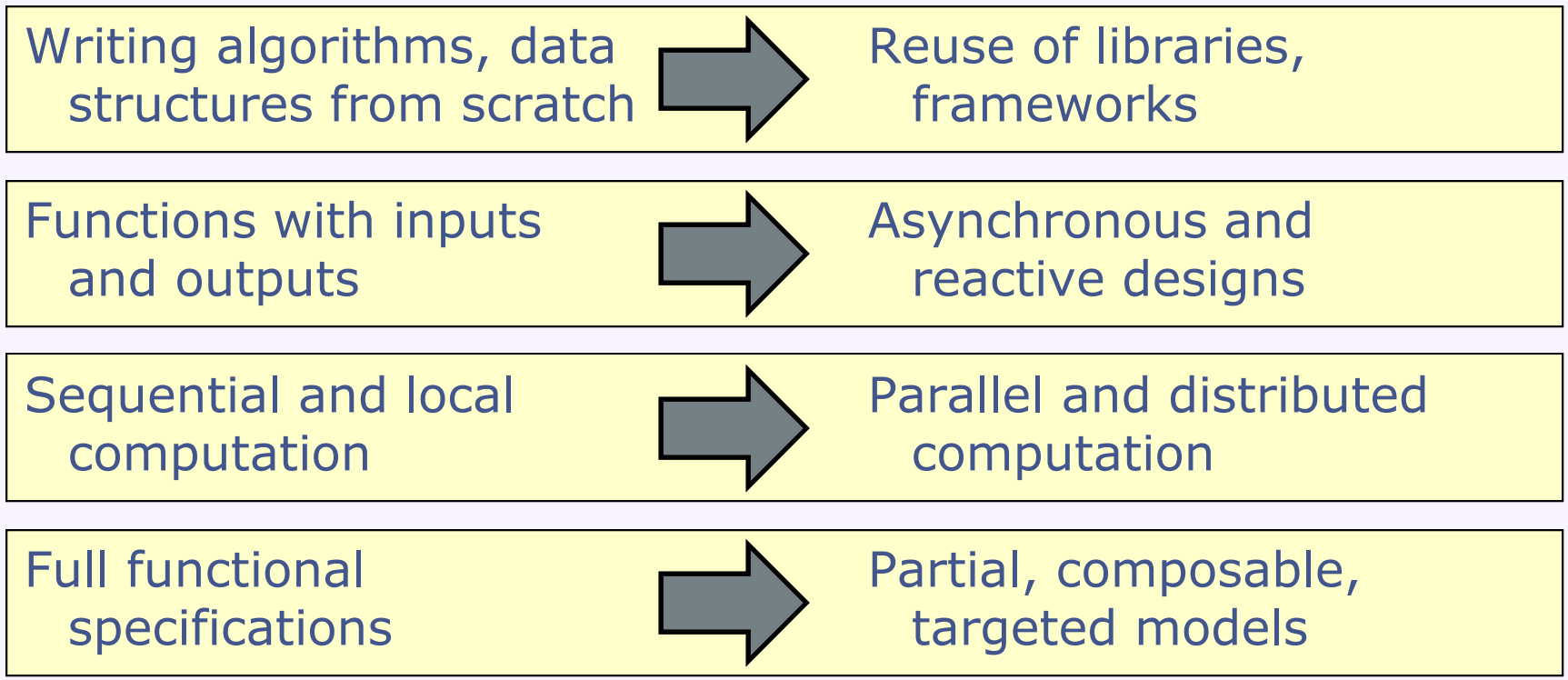
sorting

BDDs



institute for SOFTWARE RESEARCH

From Programs to Systems



Our goal: understanding both the **building blocks** and also the **principles** for construction of software systems at scale

The four course themes

- **T**hreads and Concurrency
  - System abstraction – background computing
  - Performance
  - *Our focus*: application-level concurrency
    - Cf. functional parallelism (150, 210) and systems concurrency (213)
- **O**bject-oriented programming
  - Evolveability, Reuse
  - Industry use – basis for frameworks
  - Vehicle is Java –industry, upper-division courses
- **A**nalysis and Modeling
  - *Practical* specification techniques and verification tools
- **D**esign
  - Process – how to start
  - Patterns – re-use conceptual solutions
  - Criteria – e.g. evolveability, performance

Principles of Software Construction: Objects, Design, and Concurrency

Course Organization

toad

Fall 2014

Jonathan Aldrich Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

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, verification of correctness
  - Basic algorithms and data structures (lists, graphs, sorting, binary search, …)

Course learning goals

1. Ability to **design** medium-scale programs
   - Design patterns and frameworks
   - Paradigms such as event-driven GUI programming
2. Understanding **object-oriented programming** concepts
   - Polymorphism, encapsulation, inheritance, object identity
3. Proficiency with basic **quality assurance** techniques
   - Unit testing
   - Static analysis
   - Verification
4. Fundamentals of **concurrency and distributed systems**

In addition:

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

Important features of this course

- The team
  - Instructors
    - Jonathan Aldrich aldrich@cs.cmu.edu Wean 4216
    - Charlie Garrod charlie@cs.cmu.edu Wean 5101
  - TAs
    - Harry Zeng [Section A]
    - Matt Gode [Section B]
    - Ken Li [Section C]
    - Andrew Zeng [Section D,E]
    - Yada Zhai [Section F]
    - Siyu Wei
    - Aniruddh Chaturvedi
    - Omer Elhiraika
- The schedule
  - Lectures
    - Tues, Thurs 9:00 – 10:20pm DH 2210
  - 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
    - E: Weds 3:30-4:20pm WEH 5302
    - F: Weds 3:30-4:20pm SH 222
  - Office hours and emails
    - *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/~charlie/courses/15-214
- Tools
  - Git
    - Assignment distribution, hand-in, and grades
  - Piazza
    - Discussion site – link from course page
  - Eclipse
    - Recommended for developing code
  - Online quizzes (tool TBA)
    - Low-consequence way to check your understanding
- Assignments
  - Homework 0 available tonight
    - Ensure all tools are working together
    - Git, Java, Eclipse
- First recitation is tomorrow
  - Introduction to Java and the tools in the course
  - ***Bring your laptop, if you have one!***
    - Install Git, Java, Eclipse beforehand – instructions on Piazza

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 is on the course website
  - We expect your work to be your own
  - 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
- Texts
  - Alan Shalloway and James Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design* (2nd Ed).
  - Several free online texts (Java, etc.)

Course policies

- Late days for homework assignments
  - 5 total free late days for the semester
    - A separate budget of 2 late days for assignments done in pairs
    - Going over budget: penalty 1% per 5 minutes, max 10% per day
  - May use a maximum of 2 late days per assignment
    - penalty 1% per 5 minutes beyond 2 days, 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

Principles of Software Construction: Objects, Design, and Concurrency

Design and Objects

toad

Fall 2014

Jonathan Aldrich Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

This lecture

- 214: managing complexity, from programs to systems
  - **T**hreads and concurrency
  - **O**bject-oriented programming
  - **A**nalysis and modeling
  - **D**esign

- Learning Goals
  - Introduce the design process through an example
  - Understand what drives design
  - Motivate object-oriented programming
  - Understand basic object-oriented concepts and their benefits

Motivation: A Story of Pines and Beetles

Lodgepole Pine

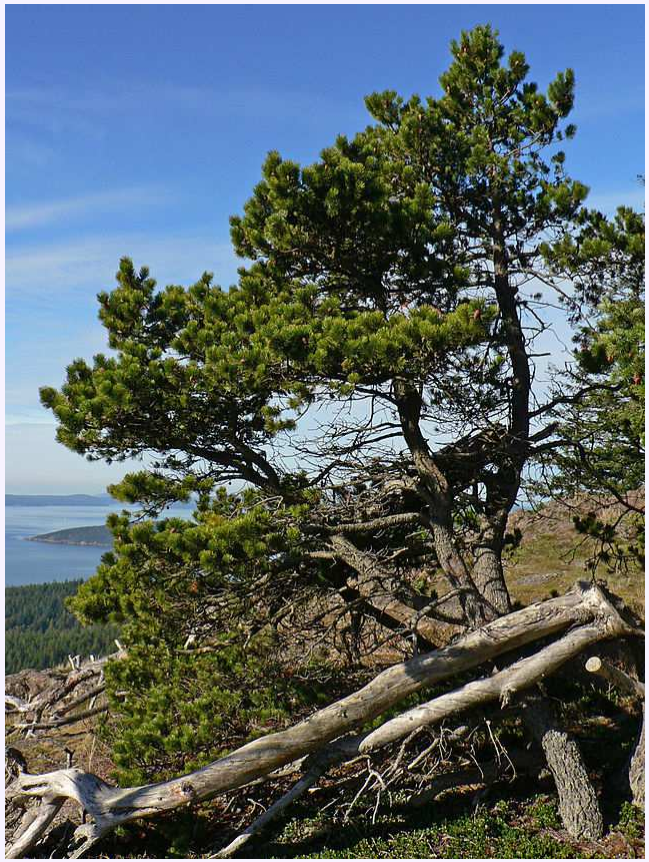


Photo by Walter Siegmund

Mountain Pine Beetle

Galleries carved in inner bark

Widespread tree death

Source: BC Forestry website

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

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
  - 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

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

Lodgepole agent

Infestation agent

Management agent

Douglas Fir agent

Observation agent

...

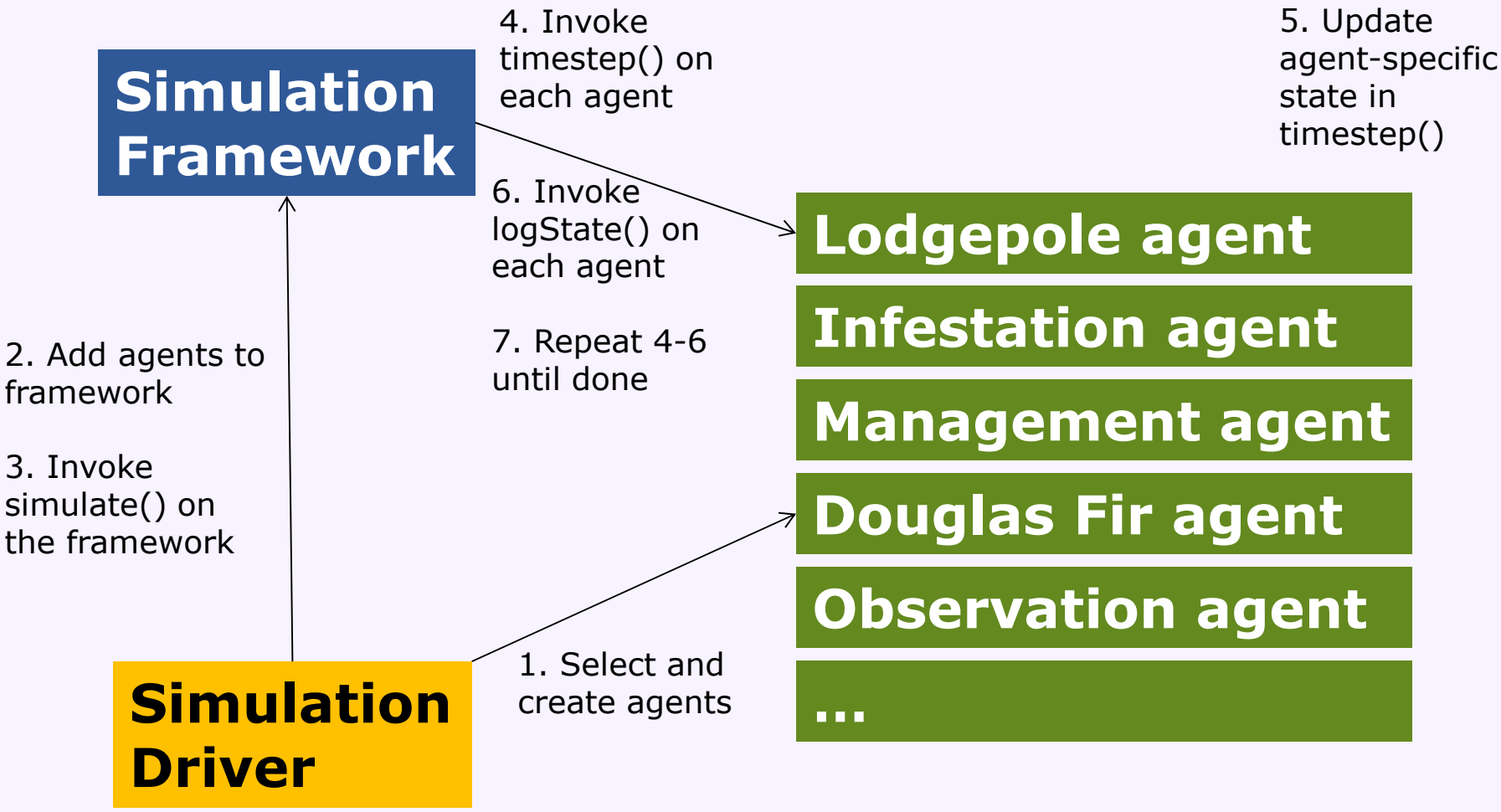
Choose any subset, or easily add new agents

Simulation Driver

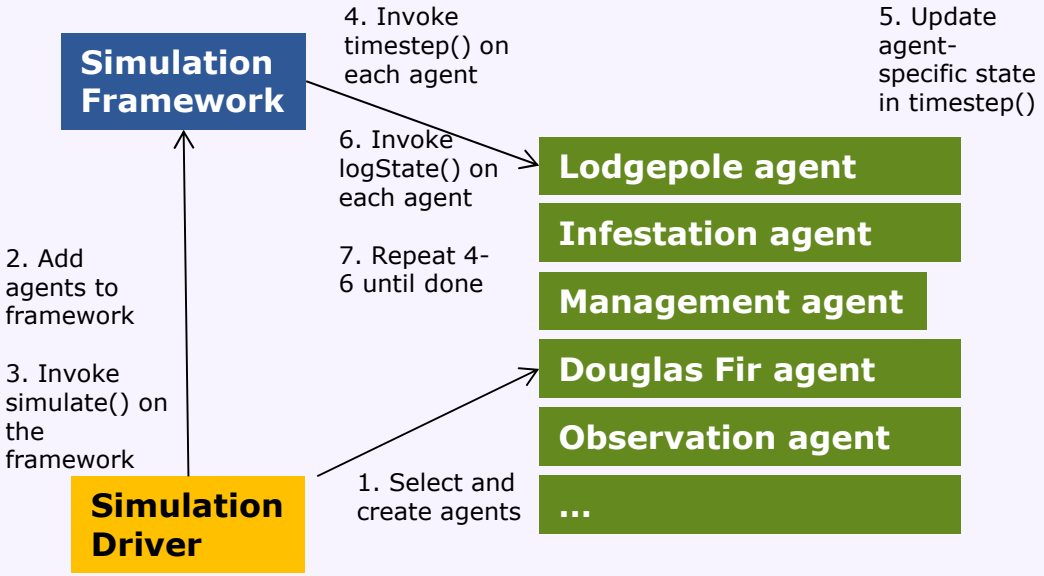
Change easily and independently of the simulation and agents

Each box should be a separate module (or file) of code

Simulation Framework Behavior Model



Exercise (small groups, on paper)



Sketch the design of the simulation framework

- Each box is a separate module / code file
- Can add new agents w/o changing Simulation Framework

Key question: how can the framework call timestep() on agents?

*If you already know OOP, think about how you would do this **without** objects*

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

- "You can't solve that problem in C / without OO!"
  - Actually, it's hard, though not impossible
  - The secret is to simulate objects in C – more later

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

Who is Responsible for…

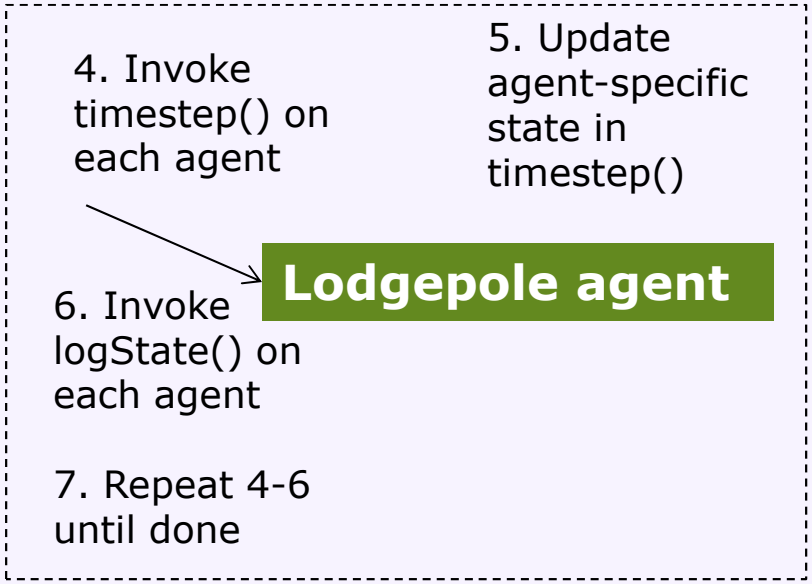
- Creating the list of agents?
- Storing the list of agents?
- Running the simulation?
- Implementing agent behavior?
- Storing agent state?

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

Designing the Agent Interface

- Agent Responsibilities
  - Implementing agent behavior
  - Storing agent state
- Interface to agent behavior?
- Interface to agent state?
  - *HINT: think about what other agents need to know*

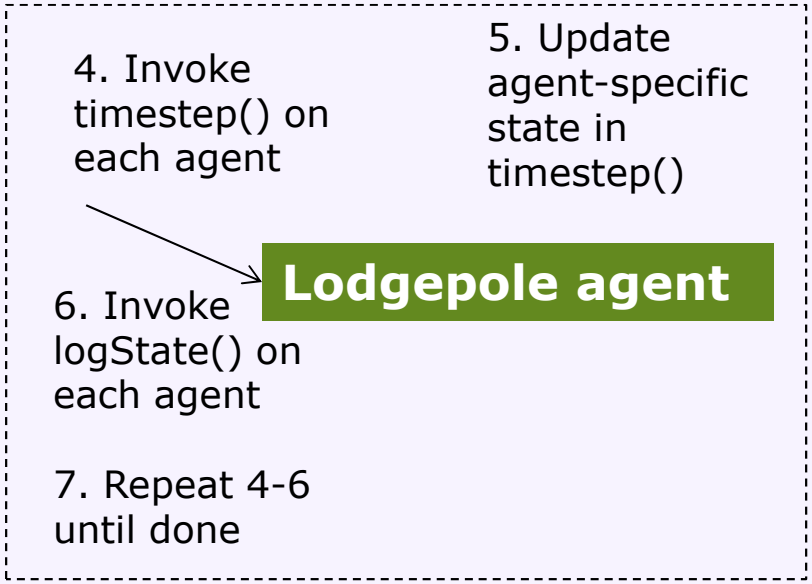


Part of the Behavioral Model

Designing the Agent Interface

- Agent Responsibilities
  - Implementing agent behavior
  - Storing agent state

- Interface to agent behavior?
  - **void** timeStep(Simulation s)
  - **void** logState()



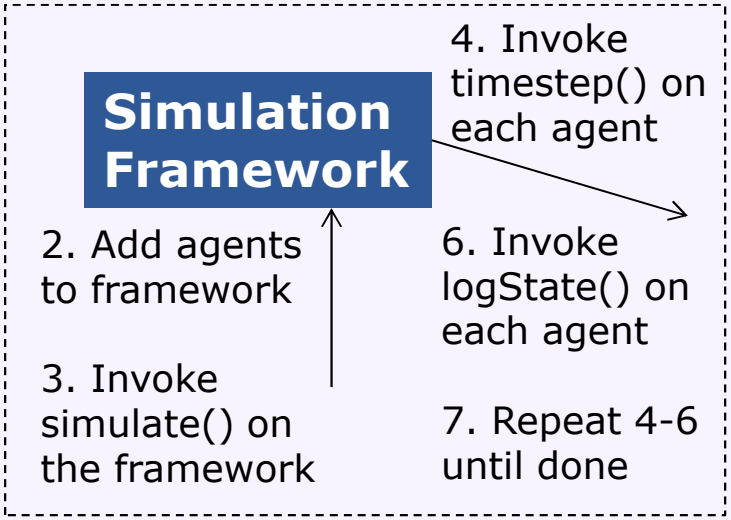
Part of the Behavioral Model

- Interface to agent state?
  - *HINT: think about what other agents need to know*
  - **boolean** isLodgepolePine()
  - **boolean** isInfested()
  - **int** getAge()
  - **int** getInfestation()
  - Location getLocation()
  - String getStateDescription()

Note: this agent interface is specific to tree infestation simulations. We'll discuss later how to make it generic.

Designing the Framework Interface

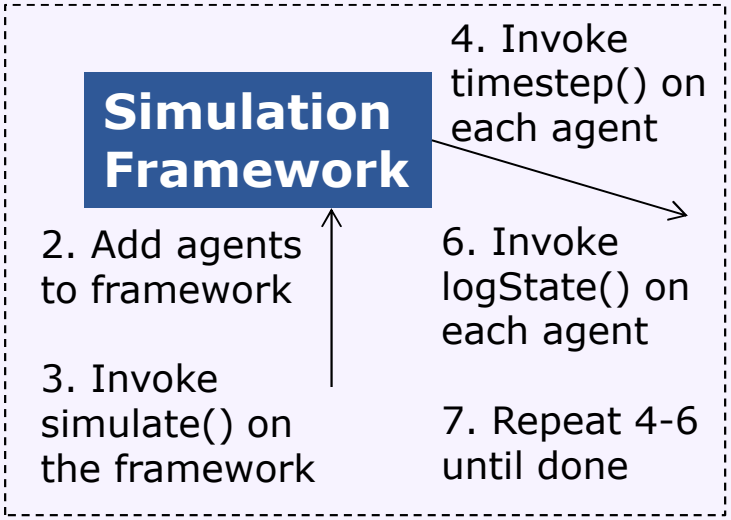
- Framework Responsibilities
  - Running the simulation
  - Storing the list of agents
- Framework interface?



Part of the Behavioral Model

Designing the Framework Interface

- Framework Responsibilities
  - Running the simulation
  - Storing the list of agents
- Framework interface?
  - **void** simulate()
  - Agent[] getAgents()



Part of the Behavioral Model

Some Pseudo-code

Simulation Driver

void main(…)

create a simulation

create and add agents for trees

add agents for infestations, etc.

call simulate() on the framework

Simulation Framework

void simulate()

loop *// until done*

for each agent a

call a's timeStep(simulation)

call a's logState()

Lodgepole Pine Agent

void timeStep(Simulation s)

increment age

chance to die

chance to spawn seedlings nearby

String logState()

return a String representation of the agent's state

The Lodgepole Pine Agent is an **Object**

- An **Object** is a first-class package of behavior and state
  - **First-class:** we can create it and pass it around at run time

```
Agent a = new LodgepolePine();

simulate(a);
```

creates a LodgepolePine object, which we will call "a"

passes the object "a" to a function

  - **State:** data fields of the object

```
int age;
Location location;
```

so far an object is like a record or struct

  - **Behavior:** the object "knows" how to respond to requests

```
a.timeStep();        // the agent knows how to do a time step
                     // since the agent is a Lodgepole Pine,
                     // it will behave as in the previous slide
```

a is the *receiver* of the message

sends the timeStep *message* to the agent a

The Agent Interface

- An **interface** is a type describing the set of messages an object understands
- What messages does Agent understand?

```
interface Agent {
        void timeStep(Simulation s);
        void logState();

        boolean isLodgepolePine();
        boolean isInfested();
        int getAge();
        int getInfestation();
        Location getLocation();
}
```

The LodgepolePine Class

- A **class** is a construct describing the implementation of a certain kind of object
- We'll use a class to implement LodgepolePine objects:

```
class LodgepolePine implements Agent {
        int age;
        Location location;

        void timeStep(Simulation s) { … }
        void logState() { … }

        boolean isLodgepolePine() { … }
        boolean isInfested() { … }
        int getAge() { … }
        int getInfestation() { … }
        Location getLocation() { … }
}
```

* *some keywords left out for simplicity*

LodgepolePine can respond to the messages in the Agent interface

Each LodgepolePine object stores information about the pine's age and location in ***fields***

LodgepolePine defines how it responds to each message in the Agent interface with a ***method***

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()
}
```

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

The keyword **this** always refers to the current method's receiver

* *some keywords left out for simplicity*

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()
}
```

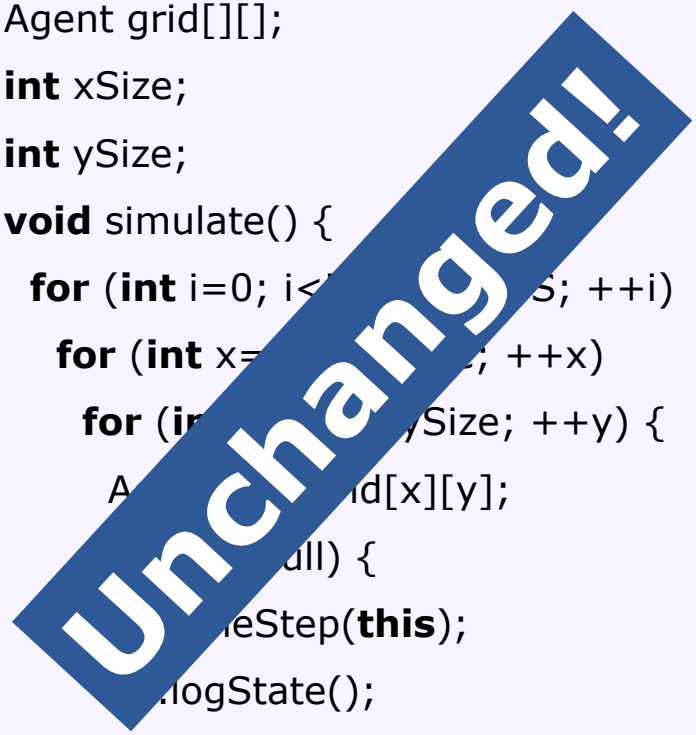
We simply add InfectedPine objects to the Agents in the Simulation.

Separately, we implement an InfectedPine class.

** some keywords left out for simplicity*

Simulation Framework

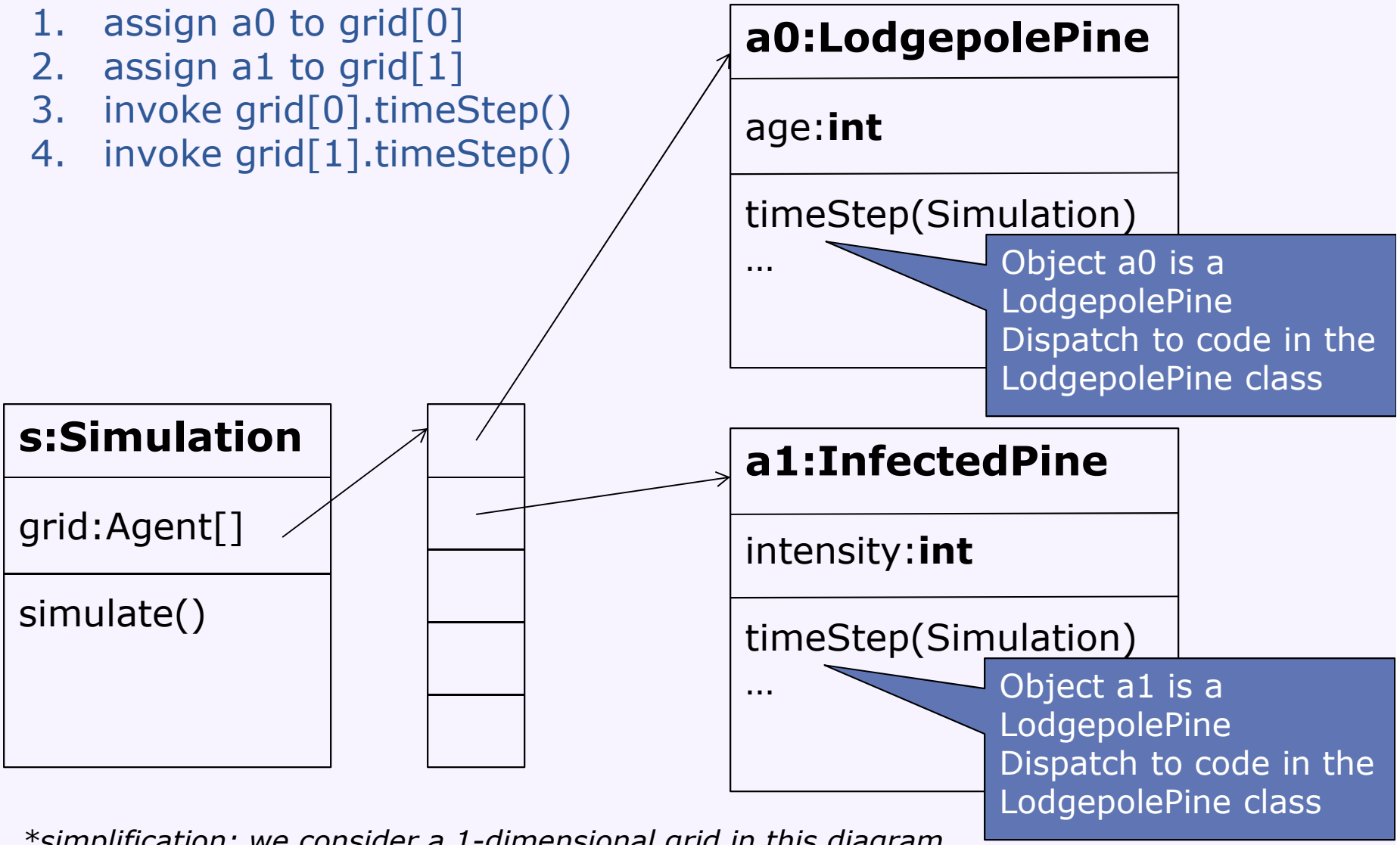
```
class Simulation {
  Agent grid[][];
  int xSize;
  int ySize;
  void simulate() {
    for (int i=0; i<…S; ++i)
      for (int x=…; ++x)
        for (in…ySize; ++y) {
          A…id[x][y];
          …ull) {
            …eStep(this);
            …logState();
          }
  } }
  // other methods, such as add(Agent a)…
}
```



Let's Run the Code Again!

Dispatch: 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

- **Simula 67** was the first object-oriented programming language
- Developed by **Kristin Nygaard** and **Ole-Johan Dahl** at the Norwegian Computing Center

Dahl and Nygaard at the time of Simula's development

- 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

Toad's 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 were invented to support **simulation**
  - Domain quality attributes: extensibility, modifiability
- 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**