6.005 Quiz 1 Review

March 3, 2008
Aseem, PJ
Topics

• State machines
• Modularity
• Testing
• Invariants
• Concurrency
State Machines

• You should be able to design a state machine:
  – Construct a state machine model
  – Use and understand state machine diagrams
  – Use and understand textual syntax
  – Use and understand concurrent/parallel machines

• And implement a state machine:
  – Apply key patterns
  – Understand advantages/disadvantages of each
State Machine Model

• A state machine is:
  – A set of states (including initial state!)
  – A set of events
  – A set of transitions (state, event, state)

• A state machine is NOT a flow chart:
  – No behavior inside states
  – No question nodes
  – No decision edges
State Machine Diagrams

• Think about events first!
  – What can the user do? What can happen anytime? (often same answer)
  – Should be atomic, simple, recognizable

• Then go on to states:
  – All possible causes/effects for each event
  – Which events get ignored/blocked?

• Clearly label all transitions (including init)!
State Machine Textual Syntax

• Textual syntax consists of:
  – State components, including initializations
  – State operations (transitions in diagram)

• Operations should show:
  – Precondition (“when [boolean]”)
  – Postcondition (“do [statements]”)

L1 Exercise: Call-Waiting

• Events (assuming cell phone):
  – send (to initiate a call), end (to hang up)
  – incoming call waiting, tick (call waiting ends)
  – flash (to switch between calls), hangup (other line)

• States:
  – idle, one line only, call waiting,
    (line A active, B hold), (line A hold, B active)

• Transitions?
L1 Exercise: Call-Waiting

- **idle**
  - Transitions:
    - Send
    - End

- **one line only**
  - Transitions:
    - End
    - Incoming

- **call waiting**
  - Transitions:
    - Tick
    - Flash

- **line A active, line B holding**
  - Transitions:
    - Hangup
    - Flash
    - End

- **line B active, line A holding**
  - Transitions:
    - Hangup
    - Flash
    - End
L1 Exercise: Call-Waiting

idle  send  one line only  incoming  call waiting

line A active, line B holding  tick  hangup  flash

line B active, line A holding  flash  hangup
L1 Exercise: Textual Syntax

- Question:
  - construct machine with inputs
    - k-pr, k-rel
  - that on R-pr toggles state component
    - boolean isRecording
  - but ignores R-pr when any key is down

- Components? Operations?
- Preconditions? Postconditions?
L1 Exercise: Textual Syntax

state
   boolean isRecording = false
   int numKeysDown = 0

op R-pr:
   when numKeysDown is 0
   do isRecording = !isRecording

op pr(k):
   when true do numKeysDown++

op rel(k):
   when true do numKeysDown--
State Machine Implementation

• Two patterns for implementing machine:
  – Singleton (machine as class)
  – Object (machine as object)

• Two patterns for implementing states:
  – Enum (state as enum)
  – State (state as object)
Singleton Pattern

• Everything is **static** – one instance, global

• Usually avoid for machines

• Advantages:
  – Efficient space (no allocation)
  – Namespace access (everything is global)

• Disadvantages:
  – Limited to one machine
  – No modularity support in Java
Object Pattern

• Each machine is an instance of the class
• Standard Java paradigm, most useful
• Advantages:
  – Multiple machines
  – Dynamic access (can pass it around, etc.)
  – Modularity (can decouple with interfaces, etc.)
Enum Pattern

• Each state is a constant in an enum
• Remember “switch” fall-through (use breaks)
• Advantages:
  – Explicit focus on transitions, ensure correctness
  – Code readability
• Disadvantages:
  – Can’t handle orthogonal and nested states
State Pattern

• Each (major) state is its own object
• Even though each state implements its own transitions, still use wrapper machine class

• Advantages:
  – Abstract away complex (minor) states
  – Focus on transitions for simpler (major) states

• Disadvantages:
  – Can’t handle orthogonal states
Modularity

• You should be able to understand modularity:
  – Recognize dependencies between modules and the source of each dependency
  – Design and insert interfaces to reduce coupling
  – Recognize standard uses of interfaces
  – Use and understand inheritance in Java
Dependencies

• Caused when a change in one will require a change in another – aka coupling
• Want as few edges as possible in dep. diagram
• How to decouple?
  – Interfaces and inheritance
  – Refactor coupled classes into common module
  – Change users of coupled classes to use module
  – Specify at high level which class to use
Interfaces and Inheritance

- Interfaces specify a **contract**
  - What to do, but not how to
- Contract not always enforceable!
  - e.g. can’t specify what constructor should look like
- Inheritance a means for **code sharing**
  - Can encapsulate common functionality
- Use inheritance with care, but can be useful
Uses For Interfaces

• **Plugin** model
  – Client uses plugin interface, given specific plugin

• **Hiding** implementation (choice of rep)
  – Client only knows enough to use contract
  – Can change implementation to suit application

• **Marker** interfaces
  – Declare no methods
  – Used to expose spec properties (or as hack)
Inheritance in Java

• Single inheritance, but multiple interfaces
  – An interface can extend multiple interfaces

• Overridden methods cannot be weakened
  – e.g. cannot have a broader return type

• Methods are **verified at compile-time**
  – Declared type has method
  – Classes have methods of all interfaces

• Methods are **chosen at runtime**
  – Search from runtime type up the type hierarchy
L3 Exercise: File Output

• How to decouple generators from file output?

```java
public class RTFGenerator implements Generator {

    boolean italic, bold;
    String filename;
    PrintStream stream;

    public RTFGenerator (String filename) {
        this.filename = filename;
    }

    public void open() throws FileNotFoundException {
        FileOutputStream fos = new FileOutputStream(filename);
        stream = new PrintStream(fos);
        stream.println("{\rtf1\mac");
    }

    ...
}
```
L3 Exercise: File Output

• Source of coupling:
  – Constructor takes in String filename

• Necessary?
  – write() and toggle methods only need PrintStream

• Where does conversion occur?
  – In open(), creates new PrintStream from filename

• Decouple by defining “Printable” interface OR by taking a PrintStream in constructor (Java)
You should be able to test state machines:

- Understand test coverage metrics
- Select appropriate coverage for a problem
- Partition input spaces into discrete domains
Test Types

• Unit testing
  – Does each module work correctly?
• Integration testing
  – Do modules work together correctly?
• Validation testing
  – Does the application meet the requirements?
• System testing
  – Does the application fit within the system?
Unit Tests

• Can be **black box** or **glass box**

• Black box depends only on contract
  – Robust through implementation changes
  – Allows for independent testers

• Glass box based on implementation
  – Ensure tests cover all lines of program
  – Can catch bugs black box would miss
Test Coverage

• Exhaustive tests try to cover all (or as much as possible of) inputs
  – Quickly becomes impractical

• Boundary tests rely on no changes in behavior
  – Must partition the input space into domains
  – Test at all edges between domains
  – Glass box tests often show new edges!
L4 Exercise: Efficient Thermostat

- Four states, use outdoor intake efficiently
- When off?
  \[ T_{\text{cur}} = T_{\text{des}} \] (\( T_{\text{out}} \) irrelevant)
- When heating?
  \[ T_{\text{out}} \leq T_{\text{cur}} < T_{\text{des}} \]
- When cooling?
  \[ T_{\text{des}} < T_{\text{cur}} \leq T_{\text{out}} \]
- When intake?
  - When heat with intake?
    \[ T_{\text{cur}} < T_{\text{des}} \text{ and } T_{\text{cur}} \leq T_{\text{out}} \]
  - When cool with intake?
    \[ T_{\text{des}} < T_{\text{cur}} \text{ and } T_{\text{out}} \leq T_{\text{cur}} \]
• You should be able to use and understand state invariants:
  – Express a desired property as a state invariant
  – Reason about invariant preservation informally (diagram) and formally (textual syntax)
  – Reason about invariants crossing multiple concurrent state machines
  – Reason inductively about invariants, or when it’s not, why it’s not, and how you can strengthen it
Invariants

• Express desired property of system
• Ensure correctness of system
• Can be enforced with interlocks
  – Check invariant before each operation
  – Aren’t always practical or usable
• Can be reasoned about through state diagrams or through formal reasoning
Induction Reasoning

• Show that invariant holds at initial state
• Show that:
  – If invariant holds before an operation
  – Invariant holds after that operation
  – For every operation
• This is an inductive proof
• When will it not work? Weak invariant
Induction For Operations

• Apply same reasoning for operations:
  – Assume invariant is true before operation
  – Assume precondition of operation is true
  – Does postcondition satisfy invariant?

• Be very explicit about this; don’t assume that operations only occur in specific contexts
L5 Exercise: Vending Machine

• Clarify the problem:
  – Machine delivers items (chocolate, soda, ...)
  – Each item costs $1 (for simplicity)
  – Machine accepts only coins or $1 bills

• What invariants must it preserve?
  – numItems ≥ 0, balance ≥ $0
  – balance + (numItems x $1) ≤ k

• Want to show $0 ≤ balance < $2 always
L5 Exercise: Vending Machine

state

int numItems = N
float balance = $0

op deliverItem():
    when balance ≥ $1 and numItems ≥ 1
    do balance -= $1, numItems -= 1

op insertCoin(x):
    when balance < $1 and numItems ≥ 1 and x ≤ $1
    do balance += x
L5 Exercise: Vending Machine

• Does invariant hold at initial state?
  – Yes, balance is $0 initially

• Does invariant hold for op insertCoin(x)?
  – Assume precondition (when), postcondition (do)
  – Yes, most balance can be is 99¢ + $1

• Does invariant hold for op deliver()?
  – Assume precondition (when), postcondition (do)
  – Assume invariant is true before op! Then yes

• Initial state + operations – invariant holds
L6 Exercise: Type Casting

- Invariant we want to prove after every line:
  \( \text{runType}(u) \leq \text{declType}(u) \)
- Can we prove it after a type cast?
  \( u = (t) \ v \)
- What are the types? How do they relate?
- What are the preconditions? Compile-time conditions? Run-time conditions?
- To solidify, think of types you know:
  – e.g. Collection, List, Queue, LinkedList
L6 Exercise: Type Casting

• Preconditions:
  \( \text{runType}(v) \leq \text{declType}(v) \) \[1\]

• Compile-time conditions:
  \( t \leq \text{declType}(v) \) \[2\]
  \( t \leq \text{declType}(u) \) \[3\]

• Run-time conditions:
  \( \text{runType}(v) \leq t \) \[4\]

• Execution semantics:
  \( \text{runType}(u) = \text{runType}(v) \) \[5\]
Concurrency

• You should be able to design and implement concurrent state machines:
  – Construct a product machine for parallel ones
  – Understand race conditions and deadlock
  – Understand how synchronization establishes a mutual exclusion invariant
  – Express and add locking primitives to models
  – Understand how lock order avoids deadlock
Concurrency Defined

- **Sequential** programming runs line by line
- **Concurrent** programming runs multiple lines “in parallel” / “at the same time”
  - Multiprogramming: all threads on same processor, system alternates control between each thread
  - Multiprocessing: each thread on its own processor
  - Distributed processing: across several machines
- Can model as product machine
  - n machines, k states – $k^n$ total states
Race Conditions

• Occur when two or more threads attempt to write to the same data (reads are ok)
• Visible in product machine representation
• Java Tutorial example (link):
  – Two threads calling functions that write to counter
  – Don’t know which value write is based on
  – Unexpected result
• How to avoid race conditions?
  – Locking (synchronization in Java)
Locking (Synchronisation)

- Prevent multiple concurrent access to data
- Invariant of mutual exclusion
- To model, add parallel state machine for lock state, specify blocked and ignored events
- Can lock entire machine (class) or individual state components (methods or fields)
- Problems locking can introduce?
  - Deadlock!
Deadlock

• Occurs when two or more threads have each locked a piece of data and need another piece

• How to avoid deadlock?
  – Enforce locking order (not always easy)