6.005 Quiz 3 Review

May 5, 2008
Topics

• Object models (lec 15, 17)
• Event-based programming (lec 19)
• Equality, hashing, mutability (lec 16)
• Usability (lec 18)
• Relational databases (lec 20)
Object Models

• You should be able to create OMs:
  – Understand OM semantics and syntax
  – Express OM constraints and invariants
  – Recognize common OM patterns

• You should be able to implement OMs:
  – Know choices for implementing sets and relations
  – Know pros/cons and implications of choices
OM semantics

• Deals with abstract concepts, not implementation details
• Shows essential elements of problem
• Denotes invariant: set of states
  – Can help define structure of state in state machine
• All structure is in the relations!
OM syntax

- Boxes are (possibly empty) sets
- Ovals are singleton sets
- Subset relations are $\subseteq$ arrows
  - “set $A$ is-a set $B$”
- Other relations are labeled
  - often a variant of “set $A$ has-a set $B$”
  - can think of non-verb labels as field names (but this is NOT a requirement – purely semantic)
OM syntax

• Can classify with subset structure
  – Subsets of same set are disjoint
  – Subsets exhaust abstract sets
OM syntax

• Multiplicities show in and out degrees
  – These are relative counts, not absolute
  – Don’t mix them up! The number closer to a set describes the count of that set
    • “For each B, m A”
    • “For each A, n B”
OM syntax

• Know the multiplicity symbols
  – ! means exactly one
  – ? means at most one
  – + means one or more
  – * means zero or more
  – omitted means *
OM constraints

- OM alone can’t express everything; must generally augment with constraints
- Often used to define derived relations
  - e.g. c.photos = c.inserted + c.subs.photos
- OM shows invariant; constraints also show invariant (valid states and relations)
Common OM patterns

• Naming
  – If a set should be uniquely identifiable (e.g. with a name or number), need an identifier set
  – If identifier is unique across subsets, relate identifier set to the most abstract set
  – Multiplicity will always be:
    • exactly one (!) for identifier set
    • at most one (?) for object set
Common OM patterns

• Recursive relations (composite)
  – If a set has a relation to itself (e.g. “contains”), need to prevent cyclic behavior
  – Can do so with *abstract* top-level set, subsets and relations to/from this *abstract* set
  – Depending on design, may also have singleton subset to represent “root”, no relation to/from it
  – Usually need additional constraints
Chat example

• Basic sets: users, conversations, messages
• Users can be uniquely identified by username
  – So need an identifier set, usernames
• Conversations are subjective
  – Multiple users, or at most two?
  – Empty conversations?
  – Multiple conversations between same users?
Chat example

• Multiple ways of specifying the rules of the problem space ➔ multiple ways to make OM

• We’ll split this into two examples:
  – AIM style
  – IRC style
Chat example (AIM)

• Conversations:
  – Have exactly two users (though both can be you)
  – Are unique by their users
Chat example (AIM)
Chat example (AIM)

• Subtleties:
  – Messages are sent to usernames, not users
    • Can send to non-existent users; e.g. AIM gives error
  – Convo can’t exist without at least one message

• But plenty more is missing
  – OM can’t express everything
  – Need constraints!
Chat example (AIM)

• Some constraints, in English:
  – The first, second of a convo must match up to its’ messages’ senders and recipients
  – For two given users A and B, at most one conversation having A and B (in either order)
Chat example (IRC)

• Conversations are channels:
  – Have any number (even zero) of users
  – Are uniquely identified by their name
    • But not unique across usernames also
Chat example (IRC)
Chat example (IRC)

• Some constraints, in English:
  – A channel’s messages’ senders and recipients must match its users and its name
Formula example

• Formulas are recursive type:
  – Simple predicates
  – **Not** formula (takes 1 formula)
  – **And** formula (takes 2+ formulas)
  – **Or** formula (takes 2+ formulas)
Formula example

(parts) → "abs Form" → 2+ → target

(parts) → "abs Form" → 2+ → !

AndForm → "abs Form"

OrForm → "abs Form"

NotForm → "abs Form"

Predicate → "abs Form"
Formula example

• There are other ways to express this
  – Trade-off between accuracy and simplicity
• None are fully expressive alone
  – Constraints needed to fill in gaps
Formula example

• Some constraints in English:
  – A Formula can be in at most one other Formula, e.g. not in both an AndFormula and an OrFormula
  – To prevent cycles, a Formula cannot be a part of itself, or be a part of a Formula that’s a part of it
File system example

• Thorough example showing:
  – Composite pattern
  – Linking subtleties
  – Derived relations

• At end, stay if interested
OM implementation

• Review lec17a slides 10-14!
  – These slides are succinct and say it all
  – If you have any questions on them, ask us after

• For this review session, we’ll cover:
  – General guidelines
  – Subtleties to think about
  – Chat example implementation
OM implementation

• Top-level sets become classes
• Subsets can be one of:
  – boolean field (for non-exhaustive subsets) or enum field (for exhaustive subsets) inside class
  – set (or instance for singleton) outside class
  – subclass (entirely static for singleton)
OM implementation

• Relations can be implemented as either:
  – field inside class, or
  – map outside class

• (Remember that a subset relation is just another type of relation)
OM implementation

• Other influencing factors:
  – Field inside class marginally more efficient
  – Immutability:
    • If relation never changes, encapsulate as part of immutable class
    • If relation can change, keep as map outside otherwise immutable class
OM implementation

• Relations are exposed as operations
• Operations constrain navigation
• Have a choice of implementing R, R’, or both
• Choice depends on need
  – e.g. chat server, implement user-convo both ways
    • To show what convos a given user is in
    • To send a message to all users in a given convo
OM implementation

• For sets or relations that can be derived from others, can choose whether to implement
  – If implement, you have an extra invariant (constraint) to maintain
  – If not implement, you have to compute on-the-fly when required (can be inefficient)
  – e.g. “all photos”, traversing sub-collections can be slow, is this important for the application?
OM implementation

• OM invariants need to be maintained in code
  – Whenever not automatically enforced (e.g. Java type checking), need to be explicitly enforced
  – In general, choose the implementation that lets you encapsulate OM invariants as rep invariants
  – e.g. for IRC chat, choose map implementation:
    • Map<User, Set<Channel>>, Map<Channel, Set<User>>
    • can encapsulate invariant in single checkRep()
    • but not necessary, can use getter methods too
Chat example (AIM)
Chat example (AIM)

Top-level classes:

- class User
- class Convo
- class Message

(Username is String)
Chat example (AIM)

• Design options:
  – Encapsulate immutable relations in classes, and have mutable relations outside classes; or
  – Encapsulate OM invariants into single checkRep
Chat example (AIM)
Chat example (AIM)

(First design)

class User {
    private final String name;
}
class Convo {
    private final User user1;
    private final User user2;
}
class Message {
    private final User sender;
    private final String recip;
}
Chat example (AIM)

Diagram:
- User
- Username
- Convo
- Message

Connections:
- User to Username: ! name
- Username to User: ?
- User to Convo: !
- Convo to User: !
- Username to Message: !
- Message to Username: !
- Convo to Message: messages +
- User to Message: first, second
- Username to Convo: sender
- Convo to Username: recipient
Chat example (AIM)

(First design)

// for efficiency mainly:
Map<String, User> users;    // username to user

// for mutability mainly:
Map<User, Set<Convo>> convos;
Map<Convo, Set<Message>> messages;
Chat example (AIM)

(First design)

Things to checkRep for:
- consistency between relations (maps, fields)
- at most one convo w/ same two users
- every convo has at least one message
- every convo’s messages’ senders/recipients match the convo’s users
Chat example (AIM)
Chat example (AIM)

(Second design)

Two-way relations:
- Users to Usernames
- Users to Convos
Chat example (AIM)

(Second design)

class User { }

class Convo { }

class Message { private final User sender;
              private final String recip; }
Chat example (AIM)
Chat example (AIM)

(Second design)

Map<User, String> usersToUsernames;
Map<String, User> usernamesToUsers;

Map<User, Set<Convo>> usersToConvos;
Map<Convo, User> convosToFirsts;
Map<Convo, User> convosToSeconds;
Chat example (AIM)

(Second design)

Map<Convo, Set<Message>> messages;

// Alternately, instead of two Convo-User maps:
Map<Convo, Set<User>> convosToUsers;
    // then need to checkRep exactly 2 users each
Chat example (AIM)

(Second design)

Things to checkRep for:
- consistency between relations (maps)
- at most one convo w/ same two users
- every convo has at least one message
- every convo’s messages’ senders/recipient match the convo’s users
Chat example (AIM)

• Things to note:
  – Same checkRep in both designs
  – User class is redundant in second design
    • Still want it for abstraction?
  – First design can enforce immutability better
Event-Based Programming

• You should know and understand:
  – The observer (pub/sub) vs. polling patterns
  – Sequential vs. event-based programming
  – The model-view-controller paradigm and how it can be combined with the observer pattern
  – Java Swing techniques for MVC and observer
  – Java Swing threading model
Polling pattern

- Client of class continuously polls class
  - Either while (true) or with a timer
  - Via observer (getter) methods

- Dependencies:
  - From client to class (strong)

- Example:
  - Repeatedly refreshing your inbox
Observer pattern

- Also called pub/sub or listener pattern
- Client of class registers as a listener
  - Must implement some listener interface
- Upon change, class notifies listeners
  - Calls listener’s handle methods on same thread
- When handling, client polls class
  - Again via observer (getter) methods
Observer pattern

• Concurrent pattern, multiple threads

• Dependencies:
  – From client to class (strong)
  – From class to client (weak)
    • Dependency to listener interface, not to client itself

• Example:
  – RSS feed, email alert
Sequential programming

• Applying polling pattern to user input
  – Enter continuous loop reading input
  – Upon input, call appropriate handlers

• Okay for e.g. command line
  – Blocking output until next user input

• Not okay for real-time applications, e.g. GUls
  – Want to be able to update output always
Event-based programming

• Solution: observer pattern
  – Listeners register themselves for different inputs
  – Thread dedicated solely to reading user input
  – On input, listeners notified and input handled

• Event loop is a variant of polling pattern
  – This is why listeners should be fast!
  – e.g. don’t start polling for more input when handling an input; add another listener instead.
MVC

• Can take event-based programming further
• Model-view-controller paradigm is a way to decouple input, output, backend
  – Model: backend of system
  – View: output to user
  – Controller: input from user
MVC

- Tied together by observer pattern
  - **View** listens to **model** to update its **output**
  - **Controller** listens to **view's input** to update **model**
Concurrency risks

- Because event-based programming relies on concurrency, need good concurrency model!
  - e.g. view being updated by both model event and controller modifying
  - e.g. model being updated by both backend source and controller modifying
Concurrency risks

• Listener calls get(), event in middle of set()
  – Solution: don’t raise events until end of set()

• Listener calls set()
  – e.g. two listeners synchronizing to each other
  – Solution: only raise event if set() caused change

• Listener removes itself
  – Solution: use a concurrent list, or copy before
Java Swing

• View is a hierarchy of components
• Follows composite pattern,
  – some components can contain other components
Java Swing

• Built-in support for event-based programming
  – Event loop continuously reading input
  – Can add listeners for input
    • To specific view components, **not** general input

• Built-in support for MVC
  – Event and EventListener interfaces
  – Components themselves support MVC
    • e.g. JTextField, JList, JTabPane, ...
Java Swing

• Swing’s concurrency model:
  – Event queue for handling all events
  – Not thread-safe off event-handling thread
Java Swing

• This means it’s okay to read/write components on the event-handling thread
  – In response to events (in event handler code)

• But it’s **not** okay to do so off that thread
  – Use SwingUtilities.invokeLater(Runnable) instead

• Swing treats Runnables like regular input events and processes them from the queue
Chat GUI example

- Two abstractions:
  - client/server
  - model/view/controller
Chat GUI example

• Client and server communicate via sockets
  – Pre-specified text protocol

• Essentially polling pattern between the two, but can mimic observer pattern
  – Thread devoted to checking and reading socket
  – When message received, call a listener’s method
  – Now an event-handling thread for the backend
Chat GUI Example

• General flow of server to client:
  – Model listens to server (via server reading thread), fires event on update
  – View listens to model, updates itself

• General flow of client to server:
  – View listens to user (invisibly), fires event on input
  – Controller listens to view, updates model
  – Model’s setter methods send message to server
Chat GUI Example

public interface ModelListener {
    // some useful events
    public void userSignedOnline(...);
    public void userSignedOffline(...);
    public void messageReceived(...);
    // ...
}

Chat GUI Example

public class Model {
    ...
    public String[] getOnlineUsers() {...}
    public Message[] getMessages() {...}
    ...
    public void send(Message msg) {...}
    public void addModelListener(ModelListener ml) {...}
    ...
    // inner class shown on right
    ...
}

private class ServerBridge implements Runnable {
    private BufferedReader reader;
    ...
    public void run() {
        while (true) {
            String msg = reader.readLine();
            ...
            // parse and update model
            for (ML ml : listeners) {
                // call event handler
            }
        }
    }
}
Equality, Hashing, Mutability

• You need to know and understand:
  – Heap semantics in Java
  – Mutability and common mutable types
  – Properties of equality
  – Object and the Object contract
  – Hashing and hash maps
  – Consistency and mutability in hash maps
  – Non-determinism in hash maps
Relational Databases

• You need to know and understand:
  – Relations, tables and keys
  – Patterns for implementing OMs as RDBs
  – Basic relational queries expressed in SQL
  – Database connection in Java
RDB basics

• RDBs exhibit ACID:
  – Atomicity (it happens or it doesn’t)
  – Consistency (always satisfies rep invariant)
  – Isolation (no concurrent actions)
  – Durability (effects never lost)
RDB semantics

• RDB is a set of **tables**
  – Each table represents an OM **relation**
  – Each table made of rows (records, tuples) and columns (fields, attributes)
RDB implementation

• Pure/relational view:
  – Every table is a binary relation

• Class/relational view:
  – Binary tables for a class are combined into one

• Either way, binary relations should not be combined with other multiplicites (?, +, *)
  – Breaks the rep invariant
  – Use separate tables instead
RDB queries

• Most queries can be expressed with SELECT

• Syntax:
  – SELECT [columns]
    FROM [relations/tables]
    WHERE [predicate]
  – optionally,
    GROUP BY [column]
RDB queries

• Recognize how to do the four basic operations on relations, using SELECT:
  – PROJECT (filters the columns)
  – SELECT (filters the rows)
  – PRODUCT (combines two relations)
  – JOIN (joins only select rows of two relations)

• See lec20 slides 20-24 if unsure
RDB queries

• Other queries:
  – INSERT INTO [table]
    VALUES [values]
  – UPDATE [table]
    SET [assignment]
    WHERE [predicate]
  – DELETE FROM [table]
    WHERE [predicate]
RDB concurrency

• Each basic transaction is thread-safe
• But sometimes a transaction needs to be ACID across multiple queries:
  – BEGIN TRANSACTION
    [queries]
  COMMIT
Usability

• Be sure to know and understand:
  – Iterative design process + spiral model
  – Dimensions of usability
    • Learnability
    • Efficiency
    • Visibility
    • Errors
    • Simplicity
• Basic sets: files, folders
• Files and folders can be uniquely identified
  – So need an identifier set, e.g. paths
• Folders contain both files and other folders
  – So need a singleton set to prevent cycles
  – Tricky, will attack relation later
File system example
File system example

• Also have concept of links (shortcuts)
  – Can point to both files and folders
  – And to other links

• Two variations:
  – Soft links: refer to paths
  – Hard links: refer to actual files/folders
File system example
File system example

- Folders can contain files, other folders, links
- So folders contain nodes?
  - No, can’t contain root
  - This is why we had a root to begin with
- How we organized our sets now determines how we organize this relationship
File system example

• Alternate design for “contains”:
  – *abs* Link set unnecessary, scrap it
  – create *abs* Element set, folders contain elements
  – files and links become subsets of Element
File system example

\[ abs \text{ Dir} \rightarrow \text{Folder} \rightarrow \text{Root} \rightarrow \text{abs Elmt} \]

\[ abs \text{ Node} \rightarrow \text{Path} \]

contains

\[ \text{File} \]

\[ \text{File} \rightarrow \text{HL} \rightarrow \text{SL} \]

\[ \text{Path} \rightarrow \text{target} \]

\[ \text{Target} \]

HL = Hard Link, SL = Soft Link
File system example

• There are still lots of subtleties that our object model alone does not cover:
  – Not all paths are legal
  – Paths actually relate to directory structure
  – Directories can’t contain themselves
  – Subdirectories can’t contain their parents
  – All directories must be reachable from the root
  – Links can’t refer to themselves
File system example

• Can create a new relation:
  – all: Directory \(\rightarrow\) (Folder or Element)
  – assuming “contains” maps only to direct children

• Can now create constraints:
  – for all Dir \(x\) and Dir \(y\) in \(x\).contains:
    \(x\.all = x\.contains + y\.all\)
  – for all Folder \(f\): \(f\) in \(\text{Root}.all\)

• Actually fixes all three directory issues!
  – No cycles (itself or parent), reachable from root
File system example

• Paths are a little trickier
• Introduce new set for names
  – Must be unique inside parent directory, but not necessarily unique across all nodes
  – Every node (including root?) has one
  – Can now constrain paths and names
File system example

• Can now create constraints:
  – for all Dir $d$ and all $x, y$ in $d$.contains:
    $x$.name $\neq$ $y$.name
  – for all Dir $d$ and all $x$ in $d$.contains:
    $x$.path = $d$.path + ‘/’ + $x$.name
  – Root.path = e.g. “C:”
  – no constraint on Root.name
File system example

• Finally, constraints for links:
  – for all HardLink $h$: $h.target \neq h$
  – for all SoftLinks: $s.target \neq s.path$
File system example

Constraints on next slide.
HL = Hard Link, SL = Soft Link
File system example

- Final constraints:
  - for all Dir $x$ and all Dir $y$ in $x$.contains:
    $x$.all = $x$.contains + $y$.all
  - for all Folder $f$: $f$ in $Root$.all
  - for all Dir $d$ and all $x, y$ in $d$.contains: $x$.name $\neq$ $y$.name
  - for all Dir $d$ and all $x$ in $d$.contains:
    $x$.path = $d$.path + ‘/’ + $x$.name
  - $Root$.path = e.g. ‘C:’
  - for all HardLink $h$: $h$.target $\neq$ $h$
  - for all SoftLinks: $s$.target $\neq$ $s$.path
File system example

• Top-level classes:
  – abstract class Node
  – abstract class Elmt extends Node
  – class File, HL, SL extend Elmt

• Names, paths trivial
  – Not implemented as classes
File system example

• Face a choice for directories/folders/root:
  – abstract class Dir? class Folder extends Dir, class Root (entirely static) extends Dir
  – class Dir? Keep track of root, folders becomes derived set (just check \(\neq\) root)

• Choose second option, fewer classes
  – abstract class Dir extends Node
  – Dir root field outside Dir
File system example

• Relations:
  – abs class Node { String name; ... }
  – Map<String, Node> pathToNode; // enforcement
    Map<Node, String> nodeToPath; // encapsulation
  – class Dir extends Node { Set<Node> contains; ... }
  – class HL extends Node { Node target; }
  – class SL extends Node { String target; }