Quiz 1 Review

**Today**

- Quiz review

**Big Picture**

Let’s take a moment to review how all the pieces we’ve looked at so far fit together into recipes for tackling different kinds of software construction problems.

<table>
<thead>
<tr>
<th></th>
<th>methods</th>
<th>mutable objects</th>
<th>ADTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>design</td>
<td>specs</td>
<td>state machine diagram</td>
<td>mutability &amp; immutability operations (C,P,O,M)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>grammar + method specs</td>
<td>datatype definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>functions over datatypes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>abstraction function &amp; rep invariant</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>equality in ADTs</td>
</tr>
<tr>
<td>test</td>
<td>unit testing</td>
<td>state machine coverage: all-states, all-actions, all-transitions, all paths</td>
<td></td>
</tr>
<tr>
<td></td>
<td>black box testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input space</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>partitioning</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>boundary values</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>glass box testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>code coverage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>code (design patterns)</td>
<td>static checking</td>
<td>enum</td>
<td>abstract syntax tree</td>
</tr>
<tr>
<td></td>
<td>arrays, lists, maps</td>
<td>Interpreter pattern</td>
<td>Interpreter pattern</td>
</tr>
<tr>
<td></td>
<td>immutable strings</td>
<td>Lexer/Parser pattern</td>
<td>Visitor pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>checkRep &amp; assertions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>composition &amp; delegation</td>
</tr>
</tbody>
</table>
Static Checking

A type is a set of values and operations on those values.

**Static type checking** guarantees that operations are applied only types for which they are defined. The compiler provides this guarantee, statically, before you even run your program.

```
true + true  type error!  + is not defined for bool x bool
```

```
String s = "";
s.gotcha();  type error! gotcha method is not defined on String
```

```
int x = "5"  type error! x is declared as an int, so it can’t point to a String, because that might produce a type error somewhere down the line when x is used with an int operation
```

Also part of static type checking is ensuring that a class implements an interface provides a method body for every method of the interface.

```java
interface ImList<E> {  E first(); ImList<E> rest(); }
class EmptyImList<E> implements ImList<E> { ... }
```

Actual type vs. declared type: variables and expressions have a **declared** type. At runtime, the variables point to an object which has an **actual** type. Those types don’t have to be identical, but they have to be “assignment compatible.” Every method on the declared type has to be found on the actual type.

```
List<Integer>
Integer
```

Testing

Testing against the spec

Blackbox vs. glassbox

- picking input values to powerMod that might cause intermediate steps to overflow int is a glassbox test
- but it still has to obey the spec! Tests can’t break the contract.

Oracles

- known answer (pi digits from the web)
- slow brute-force method (powerMod)
• comparison that ignores irrelevant details (e.g. order of items, accuracy threshold for answer)

Unit testing vs. subsystem/integration/system testing

Input space partitioning

Boundary value testing
  reverse: String -> String

State machine testing
  all-transitions coverage

  stack testing

Specs

Methods have a contract, that separates the caller from the method body

  precondition: set of legal arguments and states of the world in which method can be called
  postcondition: set of legal return values/exceptions/post-states, given the arguments

(More generally, a class has a contract, that separates its users from the internal representation & method behavior)

precondition is expressed as requires clauses. Preconditions are less friendly to clients, because method can do ANYTHING when the precondition isn’t satisfied. e.g.

char getKeypress();
  // requires: user must press a letter key

There’s no way for a caller to guarantee this! And if they don’t, what’s the state of the program? Destroyed? Exception thrown? Returns the letter “A”? Returns -1? What?

Some preconditions are unavoidable:

  int binarySearch(int x, int[] a)
  // requires: a is sorted in increasing order

postcondition can include exceptions – which are essentially like adding to the set of values of the return type. e.g.:

  String readLine() throws IOException
  is similar to returning the union type (using our datatype definition syntax) String + IOException.
  But note that the exceptional return value also has special control behavior – it can bubble up through levels of method calls until somebody handles it. Regular return values don’t behave that way. If they’re not captured right at the call, they’re ignored.

checked vs. unchecked exceptions

mutators have a frame condition that specifies what was modified. Think about mutable state in the rep, and think about mutable objects being passed in.

  int countElements(Iterator iter);
  // modifies: iter
  // returns: number of elements that were remaining in iter
State Machines

risks of aliasing
how to break an iterator
states, transitions
events, traces
all-transitions
the rep of a mutable object is its state
states are usually sets of different rep values, abstracting away from the particular rep value
draw a picture of the stack’s state machine

Regexes & Grammars

Lexer – takes a sequence of characters and produces a sequence of tokens instead
   a token is a unit more useful for parsing, e.g. number, word, identifier, operator
   lexer often removes whitespace
Parser – takes a sequence of tokens and produces an abstract syntax tree
   an abstract syntax tree is a datatype (usually recursive) representing a phrase in the language
a grammar describes legal phrases in the language using a set of productions (rules)
a regular expression is a single production

ADTs

an ADT is a datatype defined by its operations
the values of an ADT represent abstract mathematical or physical-world objects
abstraction function
   how the rep fields are mapped to an abstract value
   RatNum
   Complex: polar representation (r, theta) vs. real/imaginary representation
   Card: Suit/int rep vs. Suit/Rank rep vs. int rep (0-51)
rep invariant – what must always be true of the rep
   note that the declared types of the rep already express an invariant – which is checked statically
   so we’re documenting additional assumptions, and coding them in a runtime check
   usually != null is a critical part of every RI
but only for object types, not primitive types
Recursive Data Types

datatype definition shows how an abstract data type (on the left) is expressed in terms of a rep (on the right)
simple (non recursive) ADT:
Stack<E> = List<E>

recursive ADT has variants, which are written like constructors:
    Formula = Var(s:String) + Not(f:Formula)
we write an abstract value of a recursive ADT as an expression of constructors
    Not(Not(“a”))

functions can be defined over a recursive data type
    simplify: Formula -> Formula
    simplify(Var(s)) = Var(s)
    simplify(Not(Not(x))) = x
    simplify(Not(Var(s))) = Not(Var(s))

Interpreter pattern implements a function as a method on the interface
Visitor pattern implements a function as an object
both patterns are statically checked, unlike instanceof

Equality

equivalence relations: reflexive, symmetric, transitive
abstraction function meaning of equals()
immutable type must override equals()
    ex. define equals() for Formula
mutable type should not
    ex. define equals() for Stack<E>
relationship between equals() and hashCode()
how to break a HashSet using mutable objects that implement the wrong equals()

Inheritance

use composition, not inheritance
substitution principle