Exam Review

CSE 331 – Section 10
12/6/12

Slides by Kellen Donohue with material from Mike Ernst

Course Logistics

- All homework's done (except late days)
- HW8 returned
- HW7 being graded
- HW9 will be graded during finals week

- Final on Monday
- Review session Sunday, 3PM in our normal classroom

Final Exam

- 8:30 AM
- Full final (vs. some previous quarters)
- Cumulative over the quarter
- All section and lecture material
- May include project-related questions

Reasoning about code

- Forward reasoning
- Backward reasoning
  - Finding the weakest precondition
- If/else reasoning
- Loop development
  - Loop invariant
  - True before loop, re-established at end of each loop

- Practice: Do the problems from hw1 and hw2 again, google practice interview questions, answer and prove

Specifications

- Stronger vs. weaker specs.
  - How to prove
    - Strong/weaker pre/post conditions
    - One implementation satisfies another
  - Effect on client/implementer
- Javadoc -- requires, effects, modifies, etc.

- Practice: Review old midterms and finals

Daikon invariant detection

- Tool for automatically generating specifications
- Daikon uses a compiler front-end (not like a GUI front-end) to add instrumentation calls to your program
  - At beginning and end of every method
  - Says what the value of every variable is
- Running your program produces a program trace

Daikon Datatrace

```
HelloWorld.ArrayHolder.updateArray(System.Int32[]\_integers):::ENTER
this_invocation_nonce
2
this
4094363
1
this.baseArray
63208015
1
this.baseArray[..]
[]
1
this.expandedArray
41962596
1
this.expandedArray[..]
[]
1
this.GetType()
"HelloWorld.ArrayHolder"
1
integers
43527150
1
integers[..]
[-1 0 1]
1

HelloWorld.ArrayHolder.updateArray(System.Int32[]\_integers):::EXIT66
this_invocation_nonce
2
this
4094363
1
this.baseArray
63208015
1
this.baseArray[..]
[4 5 6]
1
this.expandedArray
41962596
1
this.expandedArray[..]
[3 4 5 6 7]
1
```

Daikon Invariants

- Daikon then analyzes the data trace and guesses invariants using machine learning
  - this.a > abs(y) array a is sorted
  - n.left.value < n.right.value
  - p != null ⇒ p.content in myArray
  - x = orig(x+1)
- Invariants can be encoded in the program with asserts, javadoc, etc.
- False positives can be removed by adding new tests

Abstract Data Types (ADT's)

- Abstraction vs. implementation/representation
- Representation Invariant
- Abstraction function
- Representation exposure

- Practice: Think about implementing a sample ADT, a PriorityQueue is a good example, write an AF and RI. Change implementation details and update the AF and RI.

Testing Theory

- Unit testing vs. other kinds
- Black box vs. white box
- Implementation vs. specification
- Revealing subdomains
- Boundary cases
- Coverage types

- Practice: Think about how you would test projects that you didn’t already write tests for (other CSE classes)

Testing Practice

- JUnit basics
- Test rules of thumb
  - Test only one function at a time if possible
  - Test only one data set per test
  - Use at least one assert per test
  - More in section slides

- Practice: Implement JUnit tests for projects that you didn't already write tests for (other CSE classes)

Interfaces & Classes

- Specification, how to comment
- Classes & Types
  - Coupling/Cohesion
- Including the right amount
  - Avoid god classes
  - Avoid writing a kitchen sink class

- Practice: Design the data model for a smartphone contacts application

Exceptions and assertions

- Rationale behind exceptions
- Basic Uses
- Exception vs. assertions
- Checked vs. unchecked exceptions
- Special values vs. exceptions

Debugging strategies

- Setting up experiments
- Use with testing
- Regression testing
- Binary search

Identity & Equality

- Properties of equality
- Reference equality
- hashCode() and equals()

Subtypes & Subclasses

- True subtypes vs. Java subtypes
  - Remember the Properties class that extends Hashtable but isn't a true subtype
- Composition/delegation vs. inheritance
  - Remember InstrumentedHashSet problems with inheritance
- Interfaces & abstract classes

Generics

- Use generic, not raw collections
- Remember generic data is erased at runtime
- Java subtyping is invariant subtyping
  - This is more restrictive than we want, (e.g. can't call a method taking List<Object> with a List<Integer>) so commonly use wildcards

Wildcards

- **?** indicates a wild-card type parameter, one that can be any type
  `List<?> list = new List<?>(); // anything`

- Difference between **`List`**<?> and **`List<Object>`**
  - **?** can become any particular type; **`Object`** is just one such type
  - **`List<Object>`** is restrictive; wouldn't take a **`List<String>`**

- Wildcards can be bounded with extends of super

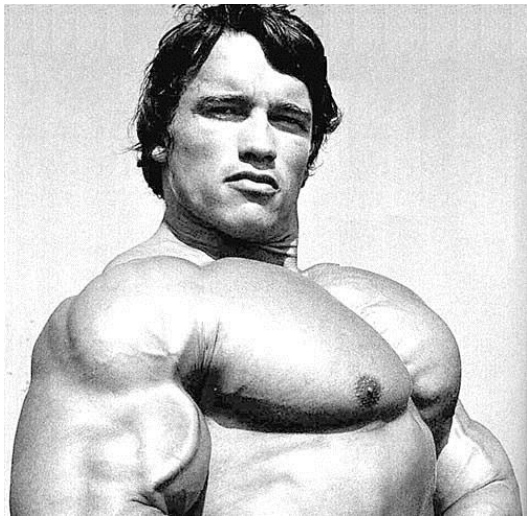
- Difference between **`List<Foo>`** and **`List<? extends Foo>`**
  - The latter binds to a particular **`Foo`** subtype and allows ONLY that
    - Ex: **`List<? extends Animal>`** might store only **`Giraffes`** but not **`Zebras`**
  - The former allows anything that is a subtype of **`Foo`** in the same list
    - Ex: **`List<Animal>`** could store both **`Giraffes`** and **`Zebras`**

PECS: Producer Extends, Consumer Super

Where should you insert wildcards?
Should you use **`extends`** or **`super`** or neither?

- Use **`? extends T`** when you *get* values from a producer
- Use **`? super T`** when you *put* values into a consumer
- Use neither (just **`T`**, not **`?`**) if you do both

```
<T> void copy(
    List<? super T> dst,
    List<? extends T> src)
```



Legal operations on wildcard types

```
Object o;
Number n;
Integer i;
PositiveInteger p;

List<? extends Integer> lei;
```

First, which of these is legal?

~~`lei = new ArrayList<Object>;`~~
~~`lei = new ArrayList<Number>;`~~
`lei = new ArrayList<Integer>;`
`lei = new ArrayList<PositiveInteger>;`
`lei = new ArrayList<NegativeInteger>;`

Which of these is legal?

~~`lei.add(o);`~~
~~`lei.add(n);`~~
~~`lei.add(i);`~~
~~`lei.add(p);`~~
`lei.add(null);`
`o = lei.get(0);`
`n = lei.get(0);`
`i = lei.get(0);`
~~`p = lei.get(0);`~~

Legal operations on wildcard types

```
Object o;
Number n;
Integer i;
PositiveInteger p;

List<? super Integer> lsi;
```

First, which of these is legal?

```
lsi = new ArrayList<Object>;
lsi = new ArrayList<Number>;
lsi = new ArrayList<Integer>;
lsi = new ArrayList<PositiveInteger>;
lsi = new ArrayList<NegativeInteger>;
```

Which of these is legal?

```
lsi.add(o);
lsi.add(n);
lsi.add(i);
lsi.add(p);
lsi.add(null);
o = lsi.get(0);
n = lsi.get(0);
i = lsi.get(0);
p = lsi.get(0);
```

Events, listeners, and callbacks

- Register to be called back when an event occurs
- Useful for inverting dependency
- Review the Observer pattern

MVC

- Model covers everything related to loading, managing the data, performing computations, etc.
- View shows the model to the user in one of many ways (may use Observer pattern to be notified of updates)
- Controllers are how the user interacts with the data and customizes the view
- Practice: Design views and controllers for earlier Contacts app

Design Patterns

- Need & purpose
- Creational Patterns
  - Singleton
  - Interning
  - Factory
- Structural Patterns
  - Adaptor
  - Proxy
- Behavioral Patterns
  - Composite
  - Visitor
- Know what patterns are useful for

Swing GUI

- Usability
- Swing vs. AWT
- JFrame & JPanel for layout
- Using paintComponent() for drawing
- Interaction with Events, Listeners

- Practice: Implement earlier Contacts app

System integration

- Architecture
- Tools
  - Source control
  - Bug tracking
- Schedule
  - Potential problems
  - How to deal with slippage
- Implementation / test order
  - Top-down or bottom-up
  - Test drivers or stubs
  - Pros and cons of each

Final Topics

- Reasoning
- Specifications
- ADTs
- Testing
- Class design
- Exceptions & assertions
- Debugging
- Identity & equality
- Generics
- Events, callbacks
- MVC
- Design patterns
- Swing GUIs
- System Integration

Course Evals