CSE 331 Notes

Contents

1 **Hoare Logic** 3
- 1.1 Hoare Triple 3
- 1.2 Comparing Assertions 3
- 1.3 Modifying Hoare Triples 3
- 1.4 Forward Reasoning 3
- 1.5 Backward Reasoning 3
- 1.6 Combining Forward and Backward Reasoning 3
- 1.7 IF Statements 4
- 1.8 Loop Invariants 4

2 **Specifications** 5
- 2.1 Javadoc 5
- 2.2 Comparing Specifications 5
- 2.3 Modifying Specifications 5

3 **Abstract Data Types** 6
- 3.1 Abstract Data Types 6
- 3.2 Components of an ADT 6
- 3.3 Abstraction State 6
- 3.4 Abstraction Function 6
- 3.5 Representation Invariant 6
- 3.6 Representation Exposure 7
- 3.7 Options For Avoiding Representation Exposure 7

4 **Testing** 8
- 4.1 Terminology 8
- 4.2 How to Test 8
- 4.3 Test Suite Subdomains 8
- 4.4 Unit Testing 8
- 4.5 Integration Testing 8
- 4.6 Boundary / Special Case Testing 9
- 4.7 Black Box / Specification Testing 9
- 4.8 Clear Box / Implementation Testing 9
- 4.9 Regression Testing 9

5 **Modular Design** 10
- 5.1 Terminology 10

6 **Equality** 11
- 6.1 Properties of Equality 11
- 6.2 Reference Equality 11
- 6.3 equals() Specification 11
- 6.4 hashCode() Specification 11

7 Exception Handling **12**
7.1 Assertions 12
7.2 Preconditions 12
7.3 Special Values 12
7.4 Exception 13

8 Polymorphism **14**
8.1 Subtype 14
8.2 Subclass 14

9 Abstraction **15**
9.1 Abstraction 15
9.2 Varieties of Abstraction 15
9.3 Generics Bounds 15
9.4 Generics Type Erasure 16

1 Hoare Logic

1.1 Hoare Triple

A Hoare triple is two assertions and one piece of code

```
{{P}} S {{Q}}
```

- `P` is the precondition
- `S` is the code
- `Q` is the postcondition

1.2 Comparing Assertions

If `P1` ⇒ `P2`, then `P1` is stronger than `P2`

- Whenever `P1` holds, `P2` also holds
- `P1` puts more constraints on program states
- `P1` is more specific about the program state

1.3 Modifying Hoare Triples

Suppose that `{{P}} S {{Q}}` is valid

- The precondition `P` can be strengthened
- The postcondition `Q` can be weakened

The Hoare triple remains valid if any of these changes are made

1.4 Forward Reasoning

1. Start with the given precondition
2. Fill in the strongest postcondition for each step downwards

1.5 Backward Reasoning

1. Start with the required postcondition
2. Fill in the weakest precondition for each step upwards

1.6 Combining Forward and Backward Reasoning

1. Reason forward from precondition
2. Reason backward from postcondition
3. Meet in the middle (top assertion must be stronger than bottom one)

1.7 IF Statements

```
// Forward reasoning
{{P}}
if (cond)
    {{P and cond}}
    S1
    {{Q1}}
else
    {{P and !cond}}
    S2
    {{Q2}}
{{Q1 or Q2}}

// Backward reasoning
{{(P1 and cond) or (P2 and !cond)}}
if (cond)
    {{P1}}
    S1
    {{Q}}
else
    {{P2}}
    S2
    {{Q}}
{{Q}}
```

1.8 Loop Invariants

A loop invariant is an assertion that holds at the top of a loop

```
{{Inv: I}}
while (cond)
    {{I and cond}}
    S
    {{I}}
{{I and !cond}}
```

- It holds when we first get to the loop
- It holds each time we execute `S` and come back to the top

2 Specifications

2.1 Javadoc

```
/**
 * @param          description of parameters
 * @return         describes return value
 * @throws         lists possible exceptions and the conditions
 * @spec.requires lists all obligations on client arguments
 * @spec.modifies lists objects that may be affected by method
 * @spec.effects  gives guarantees on final state of affected
 *                 objects
 *                 under which they are thrown
 */
```

2.2 Comparing Specifications

If `S1` is stronger than `S2`

- Precondition of `S1` is weaker than that of `S2`
- Postcondition of `S1` is stronger than that of `S2`

If `S1` is weaker than `S2`

- Precondition of `S1` is stronger than that of `S2`
- Postcondition of `S1` is weaker than that of `S2`

2.3 Modifying Specifications

Strengthening specifications

- Should not break clients
- Could break implementation

Weakening specifications

- Could break clients
- Should not break implementation

3 Abstract Data Types

3.1 Abstract Data Types

An ADT abstracts from the organization to the meaning of data

- Details of data structures are hidden from the client
- Client can only see the operations that are provided

Benefits:

- Allow us to change how data is stored
- Improve performance
- Change algorithms
- Delay decisions on how ADTs are implemented (modular)

3.2 Components of an ADT

Immutable ADTs

- Overview
- Abstract state
- Creators
- Observers/Getters
- Producers

Mutable ADTs

- Overview
- Abstract state
- Creators
- Observers/Getters
- Mutators

3.3 Abstraction State

The abstract state is an abstract description of what the object represents

- Contains no implementation details
- Leaves us free to change concrete representation in the future
- Described in mathematical concepts

3.4 Abstraction Function

Translates the object state into the abstract state

- Says what the data structure means in terms of their abstract state
- Describes how the implementation fields are used to describe the abstract state
- Allows us to check that the abstract state satisfies the postcondition

3.5 Representation Invariant

Guarantees a certain relationship among implementation fields

- Must hold at all times
- No instance of the object should ever violate the representation invariant

3.6 Representation Exposure

Representation exposure is external access to internal fields

- Almost always bad
- Can cause bugs that will be very hard to detect
- Should be clearly documented if allowed

How to avoid representation exposure

- Copy in parameters that become part of the implementation
- Copy out results that are part of the implementation
- Alternatively make them unmodifiable

3.7 Options For Avoiding Representation Exposure

- If $O(n)$ time is acceptable, return a copy of the object
  - Safest option
  - Allows changeability
- If $O(1)$ time is required, return an unmodifiable object
  - Prevents breaking the representation invariant
- Document whether return value is a fresh mutable object (copied) or a read-only access (unmodifiable)

4 Testing

4.1 Terminology

- High code coverage
  - Ensure test suite covers all of the program
  - Assess quality of the test suite with % coverage
  - 100% coverage may not be a reasonable target

4.2 How to Test

Write the test

1. Choose input/configuration
2. Define the expected outcome

Run the test

1. Run with input and record the actual outcome
2. Compare actual outcome to expected outcome

4.3 Test Suite Subdomains

A subdomain is a subset of possible test inputs

- A subdomain is revealing for error E if either
  - Every input in that subdomain triggers error E
  - No input in that subdomain triggers error E
- Need to test at least one input from each revealing subdomain to check for bugs
  - If you test one input from every revealing subdomain, you are guaranteed to find the bug
- Need to guess revealing subdomains
  - Make educated guesses where the bugs might be

4.4 Unit Testing

Unit testing focuses on one module (i.e. class or method)

- Tests a single unit in isolation from others

4.5 Integration Testing

Integration testing verifies that the modules fit together properly

- Usually done after the modules are well tested
- Usually done after unit testing

4.6 Boundary / Special Case Testing

Test for cases at the edge of subdomains to check for:

- Off-by-one bugs
- Overflow errors
- Object aliasing
- Smallest/largest values
- Zero/null values

4.7 Black Box / Specification Testing

Only the specification of the procedure is known, the internals and implementation is unknown

- Avoids psychological biases
- Allows tests to be written first
- Tests do not need to be changed if procedure implementation is changed

4.8 Clear Box / Implementation Testing

Only the internals and implementation of the procedure is known, the specification is unknown

- Control-flow details
- Performance optimizations
- Alternate algorithms for different cases

4.9 Regression Testing

When a bug is found in production, add the input that elicited the bug to the test suite

- Ensures that your fix solves the problem
- Protects against future revisions that reintroduce the bug

5 Modular Design

5.1 Terminology

- Coupling
  - How much dependency there is between modules
  - Inter-module
  - Less coupling is better
    - Can understand each module without much understanding of others
- Cohesion
  - How well parts of a module fit an work together
  - Intra-module
  - More cohesion is better
    - Forms something that is self-contained, independent and with a well-defined purpose
- Completeness
  - Objects should be fully initialized at the end of constructors
- Consistency
  - Names, parameter ordering and behavior should be consistent

6 Equality

6.1 Properties of Equality

- Reflexive

  `a.equals(a) == true`

- Symmetric

  `a.equals(b)` $\Leftrightarrow$ `b.equals(a)`

- Transitive

  `a.equals(b) && b.equals(c)` $\Leftrightarrow$ `a.equals(c)`

A relation that is reflexive, transitive and symmetric is called an equivalence relation

6.2 Reference Equality

Reference equality means an object is equal only to itself

- `a == b` if and only if `a` and `b` point to the same object
- Is the smallest/strongest equivalence relation on objects

6.3 equals() Specification

Implementations of the `equals()` class must be

- Reflexive
- Symmetric
- Transitive
- Consistent
  - `x.equals(y)` should consistently return true or false provided that neither is mutated
- For any `x` that is non-null, `x.equals(null)` should return false

6.4 hashCode() Specification

If two objects are equal, they must have the same hash code

- `a.equals(b)` $\Rightarrow$ `a.hashCode() == b.hashCode()`
- No guarantees that `a` and `b` have different hash codes if `!a.equals(b)`

7 Exception Handling

7.1 Assertions

```
assert condition;
assert condition : "message";
```

Used for internal consistency checks that should never fail

- Throws an AssertionError if condition is false
- Must be enabled or disabled at runtime
  - `java -ea` runs code with assertions enabled
  - `java` runs code with assertions disabled (default)
- Usually disabled in production code

7.2 Preconditions

```
/**
 * @spec.requires lists all obligations on client arguments
 */
```

Used in contexts in which calls can be checked via reasoning

- Where checking for preconditions at runtime would be prohibitive
- Not enforced by code, only by specifications

7.3 Special Values

```
public int method() {
    ...
    if (specialCondition) {
        return -1;
    }
    ...
}
```

Used in common cases that are illegal but likely to occur

- Where clients are likely to remember to check for special values
- Special values should not occur during normal operation

7.4 Exception

```java
    public void method() throws NameOfException {
        ...
        throw new NameOfException();
        ...
    }
```

Used in unpredictable contexts and rare or exceptional cases

- There are two kinds of exceptions
  - Checked exceptions (standard exceptions)
    - For special cases that may occur
    - Must be declared in method signature
    - Must be caught by a method in the runtime stack
  - Unchecked exceptions (runtime exceptions)
    - For unexpected cases that should never occur
    - No need to be declared
    - No need to be caught
- Halts program execution if not caught

8 Polymorphism

8.1 Subtype

```
// composition implementation of a subtype
class SubType {
    private SuperType mySuperType;

    // SubType's method accesses SuperType's public
    // methods to 'inherit' functionality
    public ... {
        ...
    }
}
```

A subtype is a strengthening of a supertype

- Every object that satisfies the rules for a subtype also satisfies the rules for the supertype
  - Subtype may have a weaker precondition
  - Subtype may have a stronger postcondition
- A subtype can substitute the supertype
- A subtype inherits the supertype's functionality and expands upon it
- The term subtype refers to its specification

For method inputs:

- A subtype argument can be replaced with a supertype argument
- Places no extra demand on clients
- Will overload the method

For method outputs:

- A supertype return value can be replaced with a subtype return value
- No new exceptions

8.2 Subclass

```
// inheritance implementation of a subclass
class SubClass extends SuperClass {

    // additional functionality added
    public ... {
        ...
    }
}
```

- A subclass may not satisfy the rules for the superclass
- A subclass may not be able to substitute the superclass
- A subclass inherits the superclass' functionality and expands upon it
- The term subclass refers to its implementation

9 Abstraction

9.1 Abstraction

- Hides details from the client
  - Increases readability
  - Increases changeability
- Allows meaningful naming of concepts
  - i.e. object identifiers
- Allows reuse of code in new contexts
  - i.e. classes, methods, ADTs

9.2 Varieties of Abstraction

- Abstraction over computation
  - Abstracts the computation from the client
  - i.e. procedures, methods
- Abstraction over data
  - Abstracts the storage of data from the client
  - i.e. ADTs, classes, interfaces
- Abstraction over types
  - Abstracts the data type from the client
  - i.e. generics

9.3 Generics Bounds

- `<TypeVar extends SuperType>`
  - an upper bound
  - accepts given supertype or any of its subtypes
- `<TypeVar extends Interface>`
  - an upper bound
  - accepts a data type that implements given interface
- `<TypeVar extends ClassA & InterfaceB & InterfaceC & ...>`
  - multiple upper bounds
- `<? super SubType>`
  - a lower bound
  - accepts given subtype or any of its supertypes

9.4 Generics Type Erasure

All generics are treated as type `Object` by the Java compiler

- Compiler cannot distinguish between generics
- Casting generics will result in a warning
- Cannot use `instanceof Class<E>`
  - Can only use `instanceof Class<?>`
  - Checks that it is an instance of `Class`, but does not guarantee it uses the same generics