Exam 2

- Exam 2 today Thursday April 1st at 6:55pm-8:45pm.

- Honor Code:
  - Open book, open notes, open slides.
  - No use of compilers, no search for answers on the Internet, no communication with others.
  - You must only submit *your own* answers.

- Type into Submitty (like Quizzes).
Topics

- ADTs
  - Benefits of ADT methodology, Specifying ADTs, Rep invariants, Representation exposure, Checking rep invariants, Abstraction functions
ADTs

- Abstract Data Type (ADT): higher-level data abstraction
  - The ADT is operations + state
  - A specification mechanism
  - A way of thinking about programs and design
An ADT Is a Set of Operations

- Operations operate on data representation
- ADT abstracts from organization to meaning of data
- ADT abstracts from structure to use
- Data representation does not matter!

Instead, think of a type as a set of operations:
create, $x()$, $y()$, $r()$, $\theta()$.

Force clients to call operations to access data.
Specifying an ADT

immutable class TypeName

1. overview
2. abstract fields
3. creators
4. observers
5. producers
6. mutators

mutable class TypeName

1. overview
2. abstract fields
3. creators
4. observers
5. producers (rare!)
6. mutators
Connecting Implementation to Specification

- **Representation invariant**: Object $\rightarrow$ boolean
  - Indicates whether data representation is **well-formed**. Only well-formed representations are meaningful
  - Defines the set of **valid** values

- **Abstraction function**: Object $\rightarrow$ abstract value
  - What the data structure really means
    - E.g., array [2, 3, -1] represents $-x^2 + 3x + 2$
  - How the data structure is to be interpreted
Representation Exposure

- Client can get control over rep and break the rep invariant! Consider

```java
IntSet s = new IntSet();
s.add(1);
List<Integer> li = s.getElements();
li.add(1); // Breaks IntSet’s rep invariant!
```

- Representation exposure is external access to the rep. **AVOID!!!**

- If you allow representation exposure, document why and how and feel bad about it
**Representation Exposure**

- Make a copy on the way out:
  ```java
  public List<Integer> getElements() {
      return new ArrayList<Integer>(data);
  }
  ```

- Mutating a copy does not affect IntSet’s rep
  ```java
  IntSet s = new IntSet();
  s.add(1);
  List<Integer> li = s.getElements();
  li.add(1); // mutates new copy, not IntSet’s rep
  ```
Representation Exposure

- Make a copy on the way in too:

  ```java
  public IntSet(ArrayList<Integer> elts) {
    data = new ArrayList<Integer>(elts);
    ...
  }
  ```

- Why?
Abstraction Function

- Abstraction function allows us to reason about correctness of the implementation.

Abstract value

Abstract operation:

Concrete object

Concrete operation (i.e., our implementation of operation):

Abstract value’

Concrete object’

AF:
IntSet Example

Abstract remove(1):
\[ \text{this} = \{ 1 \} \]

Concrete remove(1):
\[ \{ 2, 3 \} \]

Creating concrete object:
- Establish rep invariant
- Establish abstraction function

After every operations:
- Maintains rep invariant
- Maintains abstraction function

Creating concrete object:
\[ \{ 1, 2, 3 \} \]

Concrete remove(1):
\[ [2, 1, 1, 2, 3] \]

After every operations:
- Maintains rep invariant
- Maintains abstraction function

Concrete remove(1):
\[ [2, 2, 3] \]
Topics

- **Testing**
  - Black box heuristics: equivalence partitioning, boundary value analysis, white box heuristics: control-flow graph (CFG), statement coverage, branch coverage, def-use coverage.
Testing Strategies

- **Test case:** specifies
  - Inputs + pre-test state of the software
  - Expected result (outputs and post-test state)

- **Black box testing:**
  - We ignore the code of the program. We look at the specification (roughly, given some input, was the produced output correct according to the spec?)
  - Choose inputs without looking at the code

- **White box (clear box, glass box) testing:**
  - We use knowledge of the code of the program (roughly, we write tests to “cover” internal paths)
  - Choose inputs with knowledge of implementation
Equivalence Partitioning

- Partition the input and/or output domains into equivalence classes
- Write tests with inputs from different equivalence classes in the input domain
- Write tests that produce outputs in different equivalence classes in the output domain
Boundary Value Analysis

- Choose test inputs at the *edges* of input equivalence classes
- Choose test inputs that produce outputs at the edges of output equivalence classes
- Other boundary cases
  - Arithmetic: zero, overflow
  - Objects: null, circular list, aliasing
Control-flow Graph (CFG)

- Assignment \( x = y + z \) => node in CFG: \( x = y + z \)

- If-then-else

  \[
  \text{if (b) S1 else S2} \Rightarrow
  \]

- (b) is a predicate node
  - CFG for S1
  - CFG for S2

end-if
Control-flow Graph (CFG)

- Loop
- While \( (b) \) \( S \) =>

\( (b) \) is a predicate node

CFG for \( S \)
Coverage

- **Statement coverage:** Write a test suite that covers all statements, or in other words, all nodes in the CFG.

- **Branch coverage:** write a test suite that covers all branch edges at predicate nodes.
  - The True and False edge at if-then-else
  - The two branch edges corresponding to the condition of a loop
  - All alternatives in a SWITCH statement
White Box Testing: Dataflow-based Testing

- A definition (def) of $x$ is $x$ at the left-hand-side
  - E.g., $x = y+z$, $x = x+1$, $x = \text{foo}(y)$
- A use of $x$ is when $x$ is at the right-hand side
  - E.g., $z = x+y$, $x = x+y$, $x>y$, $z = \text{foo}(x)$
- A def-use pair of $x$ is a pair of nodes, $k$ and $n$ in the CFG, s.t. $k$ is a def of $x$, $n$ is a use of $x$, and there is a path from $k$ to $n$ free of definition of $x$

\[
\begin{align*}
k: & \quad x = \ldots \\
x: & \quad x = \ldots \\
n: & \quad \ldots = x \ldots
\end{align*}
\]
White Box Testing:
Dataflow-based Testing

- Dataflow-based testing targets: write tests that cover paths between def-use pairs

- Intuition:
  - If code computed a wrong value at a def of x, the more uses of this def of x we “cover”, the higher the possibility that we’ll expose the error
  - If code had erroneous use of x, the more def-use pairs we “cover”, the higher the possibility that we’ll expose the error at the use of x
A Buggy \texttt{gcd}

// requires a,b > 0

\begin{verbatim}
static int gcd(int a, int b) {
    int x=a;
    int y=b;
    while (x != y) {
        if (x > y) {
            x = x - 2y;
        } else {
            y = y - x;
        }
    }
    return x;
}
\end{verbatim}

Let’s test with \texttt{gcd(15,6)} and \texttt{gcd(4,8)}. What’s the statement coverage? Branch?
CFG for Buggy GCD

1. x = a;
   y = b;

2. x != y
   False
   True

3. x > y
   False
   True

4. x = x - 2y;

5. y = y - x;

6. 

7. res = x;

Def-use pairs for x:
   (node 1, node 2)   (4,2)
   (1,3)             (4,3)
   (1,4)             (4,4)
   (1,5)             (4,5)
   (1,7)             (4,7)

Def-use coverage targets: cover paths connecting def-use pairs

“Merge” node
Def-use Coverage Targets

- The All-defs coverage target: for every `def x`, cover at least one path (free of definition of `x`), to at least one `use x`
- The All-uses coverage target: for every `def-use pair of x`, cover at least one path (free of definition of `x`) from the `def x` to the `use x`
- The All-du-paths coverage target: for every `def-use pair of x`, cover every path (free of definition of `x`) from the `def x` to the `use x`
Topics

- Exceptions
  - Preconditions vs. exceptions, throwing and catching, propagation down the call stack, exceptions vs. special values, checked vs. unchecked exceptions
Preconditions vs. Exceptions

- In certain cases, preconditions are a valid choice
  - When checking is expensive. E.g., binarySearch
  - In private methods, usually used in local context

- Whenever possible, remove preconditions from public methods and specify behavior
  - Usually, this entails throwing an Exception
  - Stronger spec, easier to use by client
Throwing and Catching

- Java maintains a call stack of methods that are currently executing.
- When an exception is thrown, control transfers to the nearest method with a matching `catch` block.
  - If none found, top-level handler.
- Exceptions allow for non-local error handling.
  - A method far down the call stack can handle a deep error!
Informing the Client of a Problem

- Special value
  - `null - Map.get(x)`
  - `-1 - List.indexOf(x)`
  - `NaN - sqrt` of negative number

- Problems with using special value
  - Hard to distinguish from real values
  - Error-prone: programmer forgets to check result?
    The value is illegal and will cause problems later
  - Ugly

- Exceptions are generally a better way to inform of a problem
Two Distinct Uses of Exceptions

- Failures
  - Unexpected by your code
  - Usually unrecoverable. If condition is left unchecked, exception propagates down the stack

- Special results
  - Expected by your code
  - Unknowable for the client of your code
  - Always check and handle locally. Take special action and continue computing
Java Exceptions: Checked vs. Unchecked Exceptions

- **Checked exceptions.** For special results
  - Library: **must declare** in signature
  - Client: **must either catch or declare** in signature
  - It is guaranteed there is a dynamically enclosing catch

- **Unchecked exceptions.** For failures
  - Library: no need to declare
  - Client: no need to catch
  - `RuntimeException` and `Error`
Topics

- **Equality**
  - Properties of equality, reference vs. value equality, equality and inheritance, `equals` and `hashCode`, equality and mutation
In Java, `==` tests for reference equality. This is the strongest form of equality.

Usually we need a weaker form of equality, value equality.

In our `Point` example, we want `x` to be “equal” to `y` because the `x` and `y` objects hold the same value.

Need to override `Object.equals`
Properties of Equality

- Equality is an equivalence relation
  - Reflexive \( a \text{.equals}(a) \)
  - Symmetric \( a \text{.equals}(b) \iff b \text{.equals}(a) \)
  - Transitive \( a \text{.equals}(b) \land b \text{.equals}(c) \implies a \text{.equals}(c) \)
Equality and Inheritance

- Let B extend A
- “Natural” definition of B.equals(Object) may lose symmetry
- “Fix” may render equals() non-transitive

One can avoid these issues by defining equality for exact classes (has pitfalls too)

```java
if (!o.getClass().equals(getClass()))
    return false;
```
equals and hashCode

- **hashCode** computes an index for the object (to be used in hashtables)
- **Javadoc for Object.hashCode()**:  
  - “Returns a hash code value of the object. This method is supported for the benefit of hashtables such as those provided by HashMap.”
  - Self-consistent: \( o \).hashCode() == \( o \).hashCode()  
    ... as long as \( o \) does not change between the calls
  - **Consistent with equals() method**: \( a \).equals(\( b \))  
    => \( a \).hashCode() == \( b \).hashCode()
Equality, mutation and time

- If two objects are equal now, will they always be equal?
  - In mathematics, the answer is “yes”
  - In Java, the answer is “you choose”
  - The Object spec does not specify this

- For immutable objects
  - Abstract value never changes, equality is eternal

- For mutable objects
  - We can either compare abstract values now, or
  - be eternal (can’t have both since value can change)
Equality and Mutation

- Client may violate rep invariant of a Set container (rep invariant: there are no duplicates in set) by mutating elements after insertion

```java
Set<Date> s = new HashSet<Date>();
Date d1 = new Date(0);
Date d2 = new Date(1);
s.add(d1);
s.add(d2);
s.add(d2);  // mutation after d2 already in the Set!
d2.setTime(0);  // mutation after d2 already in the Set!
for (Date d : s) { System.out.println(d); }
```
Topics

- **Subtyping vs. subclassing**
  - Subtype polymorphism, true subtypes and the LSP, specification strength and comparing specifications (again), Function subtyping
Subtype Polymorphism

- **Subtype polymorphism** – the ability to use a subclass where a superclass is expected
  - Thus, *dynamic method binding*
    - `class A { void m() { ... } }
    - `class B extends A { void m() { ... } }
    - `class C extends A { void m() { ... } }
    - Client: `A a; ... a.m();` // Call `a.m()` can bind to any of `A.m`, `B.m` or `C.m` at runtime!

- Subtype polymorphism is a language feature
  --- essential object-oriented language feature
  - **Java subtype**: B extends A or B implements I
  - A Java subtype is not necessarily a true subtype!
Benefits of Subtype Polymorphism

- “Science” of software design teaches Design Patterns

- Design patterns promote design for extensibility and reuse

- Nearly all design patterns make use of subtype polymorphism
Subtypes are Substitutable

- Subtypes are **substitutable** for supertypes
  - Instances of subtype won’t surprise client by expecting more than the supertype
  - Instances of subtypes won’t surprise client by failing to satisfy supertype postcondition
- **B** is a **true subtype** (or “behavioral” subtype) of **A** if **B** has stronger specification than **A**
  - Not the same as **Java subtype**!
  - Java subtypes that are not true subtypes are confusing and dangerous
Liskov Substitution Principle (LSP)

- Due to Barbara Liskov, Turing Award 2008
- LSP: A subclass B should be substitutable for its superclass A. I.e., B is a true subtype of A

To ensure that B is substitutable:
- B does not remove methods from A
- For each B.m that “replaces” A.m, B.m’s specification is stronger than A.m’s specification
  - Client: A a; ... a.m(int x, int y); Call a.m can bind to B’s m. B’s m should not surprise client
Function Subtyping

- In programming languages function subtyping deals with substitutability of functions

  - Question: under what conditions on the parameter and return types $A, B, C$ and $D$, is function $A \ f(B)$ substitutable for $C \ f(D)$

  - Reasons at the level of the type signature

  - Rule: $A \ f(B)$ is a function subtype of $C \ f(D)$ if $A$ is a subtype of $C$ and $B$ is a supertype of $D$

  - Guarantees substitutability
Type Signature of Substituting Method is Stronger

- Method parameters (inputs):
  - Parameter types of \texttt{A.m} may be replaced by supertypes in subclass \texttt{B.m}. "contravariance"
    - E.g., \texttt{A.m(String p)} and \texttt{B.m(Object p)}
  - \texttt{B.m} places no extra requirements on the client!
    - E.g., client: \texttt{A a; \ldots a.m(q)}. Client knows to provide \texttt{q} a String. Thus, client code will work fine with \texttt{B.m(Object p)}, which asks for less: an Object, and clearly, every String is an Object
  - Java does not allow change of parameter types in an overriding method. More on Java overriding shortly
Type Signature of Substituting Method is Stronger

- Method returns (results):
  - Return type of A.m may be replaced by subtype in subclass B.m. "covariance"
    - E.g., Object A.m() and String B.m()
  - B.m does not violate expectations of the client!
    - E.g., client: A a; ... Object o = a.m(). Client expects an Object. Thus, String will work fine
  - No new exceptions unless B.m has weaker preconditions. Exceptions subtypes are ok.
  - Java does allow a subtype return type in an overriding method!
Reasoning about Specs

- **Function subtyping** reasons with type signatures
- Remember, type signature is a specification
  - Precondition: requires arguments of given type
  - Postcondition: promises result of given type
- Compiler checks **function subtyping**
- **Specifications** add reasoning about behavior and effects
  - Precondition: stated by **requires** clause
  - Postcondition: stated by **modifies, effects, returns** and **throws** clauses
Reason about Specs

- "Behavioral" subtyping generalizes function subtyping
- B.m is a true subtype (behavioral subtype) of A.m
  - B.m has weaker precondition than A.m
    - Generalizes "B.m’s parameter is a supertype of A.m’s parameter" premise of function subtyping rule
    - Contravariance
  - B.m has stronger postcondition than A.m
    - Generalizes "B.m’s return is a subtype of A.m’s return"
    - Covariance
- These 2 conditions guarantee B.m’s spec is stronger than A.m’s spec, and B.m is substitutable for A.m