

Review

THANKS!

- Professor Barb Cutler for the Homework Server!
- Mentors: Brandon, Will and Leopold
- TAs: Pranshu and Anshul

Fall 15 CSCI 2600, A Milanova

2

Final Exam

- Mon, Dec 21 3-6pm in DCC 318
- Final exam is cumulative
- Closed notes, closed laptops/phones
- 4 “cheat” pages, either four single-sided or two double-sided standard 8.5x11-sized sheets
 - Type or write by hand (which I recommend)

3

How to study

- Review today's slides
- Review Exams 1 and 2
- Back tests, posted on **Announcements page**
- No mentor and no TA office hours next week
- I'll have office hours Friday, Dec 18, 10-2pm and Monday 21, 10-2pm

Fall 15 CSCI 2600, A Milanova

4

Grades

- CHECK YOUR GRADES IN THE LMS!
 - HW7 and HW8 feedback and grades in the Homework server. Let me know if you have questions
 - HW9 will be graded by Friday, Dec 18
- You will have all your grades except for the final exam grade into the LMS by Friday, Dec 18

Fall 15 CSCI 2600, A Milanova

5

PoS is about writing **correct** and **maintainable** software

- Specifications
- Polymorphism, abstraction and modularity
- Design patterns
- Refactoring
- Reasoning about code
- Testing
- Software process
- Tools - Java, Eclipse, Subversion, Junit, EclEmma
 - Principles are far more important than tools!

6

PoS is about writing **correct** and **maintainable** software

- Building correct software is hard!
 - Lots of dependencies
 - Lots of “moving parts”
- Software engineering is primarily about mitigating and managing complexity
 - Specifications, abstraction, design patterns, refactoring, reasoning about code (**invariants** “fix” one part, thus fewer “moving parts” to worry about!), testing
 - All of these mitigate complexity

7

Outline

- Quiz questions in reverse chronological order
- Review of topics in chronological order

Fall 15 CSCI 2600, A Milanova

8

Quiz 10 Questions *

```
JButton b = new JButton("Ouch");
b.addActionListener(new ActionListener() {
    void actionPerformed(ActionEvent e) {
        doSomething();
    }
})
```

Closest **design pattern**:

- a) **Adapter**
- b) **Observer**
- c) **Interpreter**
- d) **Anonymous Class**

Fall 15 CSCI 2600, A Milanova

9

Quiz 10 Questions *

```
JButton b = new JButton("Ouch");
b.addActionListener(new ActionListener() {
    void actionPerformed(ActionEvent e) {
        doSomething();
    }
})
```

The call `b.addActionListener(...)` is a callback

- a) **True**
- b) **False**

Fall 15 CSCI 2600, A Milanova

10

Quiz 10 Questions

```
JButton b = new JButton("Ouch");
b.addActionListener(new ActionListener() {
    void actionPerformed(ActionEvent e) {
        doSomething();
    }
})
```

To repaint in response to button click, we must call `paintComponent(...)` from `doSomething()`

- a) **True**
- b) **False**

Fall 15 CSCI 2600, A Milanova

11

Quiz 9 Questions

A library class accepts and returns values in English units but you need metric units. What **design pattern** allows us to reuse this library class

- a) **Adapter**
- b) **Decorator**
- c) **Proxy**
- d) **Delegation**

A **design pattern** used to restrict access to an object

- a) **Adapter**
- b) **Decorator**
- c) **Proxy**
- d) **Delegation**

12

Quiz 9 Questions

A **design pattern** used to enhance functionality of an object is

- a) Adapter
- b) Decorator
- c) Proxy
- d) Delegation

Design pattern(s) that traverse a hierarchical structure

- a) Procedural
- b) Observer
- c) Interpreter
- d) Visitor

Fall 15 CSCI 2600, A Milanova

13

Quiz 9 Questions

In the Java Collections library class **AbstractCollection** implements methods **contains** and **equals** as these methods work identically across all concrete **Collections**. **contains** and **equals** call **iterator()** which returns an **Iterator** over the collection. However, **AbstractCollection.iterator()** is abstract, thus deferring creation of the **Iterator** to the concrete subclass. What **design pattern(s)** is used here?

- a) Factory Method
- b) Factory Object
- c) Template Method
- d) Inheritance

Fall 15 CSCI 2600, A Milanova

14

A Question I got rid of...

What's wrong with Willy's implementation of interned boxed Integers?

```
class Int {
    private int value;
    private Int (int value) { this.value=value; }
    private static Map<Int,Int> cache =
        new HashMap<>();
    public static Int valueOf(int value) {
        Int tmp = new Int(value);
        if (cache.containsKey(tmp))
            return cache.get(tmp);
        else {
            cache.put(tmp,tmp);
            return tmp;
        }
    }
}
```

15

Quiz 8 Questions

```
Number n;
Integer i;
PositiveInteger pi;
NegativeInteger ni;
PriorityQueue<? extends Integer> pei;
PriorityQueue<? super Integer> psi;

pei = new PriorityQueue<PositiveInteger>();
a) Is legal
b) Is not legal
```

Fall 15 CSCI 2600, A Milanova

16

Quiz 8 Questions *

```
Number n;
Integer i;
PositiveInteger pi;
NegativeInteger ni;
PriorityQueue<? extends Integer> pei;
PriorityQueue<? super Integer> psi;

pei.add(pi);
a) Is legal
b) Is not legal
```

Fall 15 CSCI 2600, A Milanova

17

Quiz 8 Questions

```
Number n;
Integer i;
PositiveInteger pi;
NegativeInteger ni;
PriorityQueue<? extends Integer> pei;
PriorityQueue<? super Integer> psi;

i = pei.poll();
a) Is legal
b) Is not legal
```

Fall 15 CSCI 2600, A Milanova

18

Quiz 8 Questions

```
Number n;
Integer i;
PositiveInteger pi;
NegativeInteger ni;
PriorityQueue<? extends Integer> pei;
PriorityQueue<? super Integer> psi;
```

```
pei = psi;
```

- a) Is legal
b) Is not legal

Fall 15 CSCI 2600, A Milanova

19

Quiz 8 Questions

PriorityQueue<E> constructor:
PriorityQueue(Comparator<? super E> comparator)

```
PriorityQueue<Integer> pqi;
Comparator<Number> cn;
Comparator<Integer> ci;
Comparator<PositiveInteger> cp;
```

Circle the legal instantiations:

- a) `pqi = new PriorityQueue<Integer>(cn);`
b) `pqi = new PriorityQueue<Integer>(ci);`
c) `pqi = new PriorityQueue<Integer>(cp);`

20

Quiz 7 Questions

If there exist non-null references `x`, `y` and `z` such that `x.equals(y)` is false, `y.equals(z)` is true and `x.equals(z)` is true, then `equals` is not transitive.

- a) True
b) False

The consistency property of `hashCode` requires that for every non-null `x` and `y` such that `x.equals(y)` is false, `x.hashCode() != y.hashCode()`.

- a) True
b) False

21

Quiz 7 Questions

`Integer f(String s)` is a function subtype of `Number f(Object o)`.

- a) True
b) False

`Integer f(Object o)` is a function subtype of `Number f(String s)`.

- a) True
b) False

22

Quiz 7 Questions

```
int f(int y) {
    int s = 0;
    int x = 0;
    while (x < y) {
        x = x + 3;
        y = y + 2;
        if (x + y < 10)
            s = s + x + y;
        else
            s = s + x - y;
        // end-if
    } // end-while
    return s;
}
```

Draw the CFG for the function.

Find an argument `a` such that `f(a)` achieves 100% statement coverage.

Is it possible to cover def-use pair `(6:s=s+x-y, 5:s=s+x+y)`?

- a) Yes
b) No

Specification tests is just another name for black-box tests

- a) True
b) False

23

Topics

- Reasoning about code
- Specifications
- ADTs, rep invariants and abs. functions
- Testing
- Subtyping vs. subclassing
- Equality
- Design patterns and refactoring
- Usability, Software process, Requirements

Fall 15 CSCI 2600, A Milanova

24

Topics

Reasoning about code

- Forward and backward reasoning, logical conditions, Hoare triples, weakest precondition, rules for assignment, sequence, if-then-else, loops, loop invariants, decrementing functions

Fall 15 CSCI 2600, A Milanova

25

Forward Reasoning

- Forward reasoning simulates the execution of the code. Introduces facts as it goes along

E.g., $\{x = 1\}$

$y = 2 * x$

$\{x = 1 \text{ AND } y = 2\}$

$z = x + y$

$\{x = 1 \text{ AND } y = 2 \text{ AND } z = 3\}$

- Collects all facts, often those facts are irrelevant to the goal

Fall 15 CSCI 2600, A Milanova

26

Backward Reasoning

- Backward reasoning “goes backwards”. Starting from a postcondition, finds the weakest precondition that ensures the given postcondition

E.g., $\{2y < y+1\}$ // Simplify into $\{y < 1\}$

$z = y + 1$ // Substitute $y+1$ for z in $2y < z$

$\{2*y < z\}$

$x = 2*y$ // Substitute rhs $2*y$ for x in $x < z$

$\{x < z\}$

- More focused and more useful

27

Condition Strength

- “P is stronger than Q” means “P implies Q”
- “P is stronger than Q” means “P guarantees more than Q”
 - E.g., $x > 0$ is stronger than $x > -1$
- Fewer values satisfy P than Q
 - E.g., fewer values satisfy $x > 0$ than $x > -1$
- Stronger means more specific
- Weaker means more general

28

Exercise. Condition Strength

- Which one is stronger?

$x > -10$ or $x > 0$

$x > 0 \text{ \&\& } y = 0$ or $x > 0 \text{ || } y = 0$

$0 \leq x \leq 10$ or $5 \leq x \leq 11$

$y \equiv 2 \pmod{4}$ or y is even

$y \equiv 1 \pmod{3}$ or y is odd

$x = 10$ or x is even

29

Hoare Triples

- A Hoare Triple: $\{P\} \text{ code } \{Q\}$
 - P and Q are logical conditions (statements) about program values, and `code` is program code (in our case, Java code)
- “ $\{P\} \text{ code } \{Q\}$ ” means “if P is true and we execute `code`, then Q is true afterwards”
 - “ $\{P\} \text{ code } \{Q\}$ ” is a logical formula, just like “ $0 \leq \text{index}$ ”

Fall 15 CSCI 2600, A Milanova (modified from slide by Michael Ernst, UW)

30

Exercises. Hoare Triples

$\{x > 0\} x++ \{x > 1\}$ is true
 $\{x > 0\} x++ \{x > -1\}$ is true
 $\{x \geq 0\} x++ \{x > 1\}$ is false. Why?
 $\{x > 0\} x++ \{x > 0\}$ is ??
 $\{x < 0\} x++ \{x < 0\}$ is ??
 $\{x = a\} \text{if } (x < 0) \ x = -x \{x = |a|\}$ is ??
 $\{x = y\} x = x + 3 \{x = y\}$ is ??

Fall 15 CSCI 2600, A Milanova

31

Exercise

- Let $P \Rightarrow Q \Rightarrow R$
(P is stronger than Q and Q is stronger than R)
- Let $S \Rightarrow T \Rightarrow U$
- Let $\{Q\} \text{code} \{T\}$
- Which of the following are true:
 - $\{P\} \text{code} \{T\}$
 - $\{R\} \text{code} \{T\}$
 - $\{P\} \text{code} \{U\}$
 - $\{P\} \text{code} \{S\}$

32

Rules for Backward Reasoning: Assignment

// precondition: ??
 $x = \text{expression}$
// postcondition: Q
Rule: the **weakest precondition** = Q, with all occurrences of **x** in Q replaced by **expression**
More formally:
 $\text{wp}("x = \text{expression};", Q) = Q$ with all occurrences of **x replaced by **expression****

33

Rules for Backward Reasoning: Sequence

// precondition: ??
 $s1;$ // statement
 $s2;$ // another statement
// postcondition: Q
Work backwards:
 precondition is $\text{wp}("s1; s2;", Q) = \text{wp}("s1;", \text{wp}("s2;", Q))$
Example:

// precondition: ??	// precondition: ??
$x = 0;$	$x = 0;$
$y = x + 1;$	// postcondition for $x = 0$; same as
// postcondition: $y > 0$	// precondition for $y = x + 1$;
	$y = x + 1;$
	// postcondition $y > 0$

34

Rules for If-then-else

Forward reasoning	Backward reasoning
$\{P\}$	$\{(b \wedge \text{wp}("s1", Q)) \vee (\neg b \wedge \text{wp}("s2", Q))\}$
if b	if b
$\{P \wedge b\}$	$\{\text{wp}("s1", Q)\}$
s1	s1
$\{Q1\}$	$\{Q\}$
else	else
$\{P \wedge \neg b\}$	$\{\text{wp}("s2", Q)\}$
s2	s2
$\{Q2\}$	$\{Q\}$
$\{Q1 \vee Q2\}$	$\{Q\}$

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

35

Exercise

- Compute the weakest precondition:

```

if (x < 0) {
    y = -x;
}
else {
    y = x;
}
{ y = |x| }
    
```

36

Exercise

- Find the strongest postcondition:

$\{ p^2 + q^2 = r \}$

$r = r/p$

$q = q*q/p$

Exercise

- Find the weakest precondition

$y = x + 4;$

if ($x > 0$) {

$y = x*x - 1;$

}

else {

$y = y + x;$

}

{ $y = 0$ }

Reasoning About Loops by Induction

1. Partial correctness

- Guess and prove **loop invariant** using computation induction
- Loop exit condition** and **loop invariant** must imply the desired postcondition

2. Termination

- (Intuitively) Establish "decrementing function" D . Each iteration decrements D , $D = 0$ and loop invariant, imply loop exit condition

Example: Reasoning About Loops

$i+z = x$ is the loop invariant

Precondition: $x \geq 0;$

$i = x;$

$z = 0;$

while ($i \neq 0$) {
 $z = z+1;$
 $i = i-1;$
 }

Postcondition: $x = z;$

Need to prove:

1. $x = z$ holds after the loop (**partial correctness**)

2. Loop terminates (**termination**)

- $i=x$ and $z=0$ give us that $i+z = x$ holds at 0th iteration of loop // **Base case**
- Assuming that $i+z = x$ holds after k th iteration, we show it holds after $(k+1)$ st iteration // **Induction**
 $z_{\text{new}} = z + 1$ and $i_{\text{new}} = i - 1$ thus
 $z_{\text{new}} + i_{\text{new}} = z + 1 + i - 1 = z + i = x$
- If loop terminated, we know $i = 0$. Since $z+i = x$ holds, we have $x = z$
- Loop terminates. D is i . $D_{\text{before}} > D_{\text{after}}$. $D = 0$ implies $i = 0$ (loop exit condition).

Reasoning About Loops

- Loop invariant **Inv** must be such that

- $P \Rightarrow \text{Inv}$ // **Inv** holds before loop. **Base case**
- $\{ \text{Inv} \wedge b \} S \{ \text{Inv} \}$ // Assuming **Inv** held after k th iteration and execution took a $(k+1)$ st iteration, then **Inv** holds after $(k+1)$ st iteration. **Induction**
- $(\text{Inv} \wedge !b) \Rightarrow Q$ // The exit condition $!b$ and loop invariant **Inv** must imply postcondition

- Decrementing function **D** must be such that

- D** decreases every time we go through the loop
- D = 0** and **Inv** must imply loop exit condition $!b$

Exercise

Precondition: $y \geq 0;$

$i = y;$

$n = 1;$

while ($i \neq 0$) {
 $n = n*x;$
 $i = i-1;$
 }

Postcondition: $n = x^y;$

Prove partial correctness and termination

Topics

■ Specifications

- Benefits of specifications, PoS specification convention, specification style, specification strength (stronger vs. weaker specifications), comparing specifications via logical formulas, converting PoS specifications into logical formulas

Specifications

- A specification consists of a **precondition** and a **postcondition**
 - Precondition: conditions that hold before method executes
 - Postcondition: conditions that hold after method finished execution (if precondition held!)

Specifications

- A specification is a **contract** between a method and its caller
 - Obligations of the method (**implementation of specification**): agrees to provide postcondition if precondition held!
 - Obligations of the caller (**user of specification**): agrees to meet the precondition and not expect more than promised postcondition

Benefits of Specifications

- Document method behavior
 - Imagine if you had to read the code of the Java libraries to figure what they do!
 - An abstraction – abstracts away unnecessary detail
- Promotes **modularity**
- Enables reasoning about correctness
 - Through testing and/or verification

Example Specification

Precondition: $\text{len} \geq 0 \wedge \text{arr.length} = \text{len}$

```
double sum(int[] arr, int len) {  
    double sum = 0.0;  
    int i = 0;  
    while (i < len) {  
        sum = sum + arr[i];  
        i = i+1;  
    }  
    return sum;  
}
```

Postcondition: returns $\text{arr}[0] + \dots + \text{arr}[\text{arr.length}-1]$

PoS Specifications

- Specification convention due to Michael Ernst
- The precondition
 - **requires**: clause spells out constraints on client
- The postcondition
 - **modifies**: lists objects (typically parameters) that may be modified by the method. Any object not listed under this clause is guaranteed untouched
 - **throws**: lists possible exceptions
 - **effects**: describes final state of modified objects
 - **returns**: describes return value

Exercise

```
static List<Integer> listAdd(List<Integer> lst1,
                             List<Integer> lst2)
```

requires: lst1 is non-null and lst2 is non-null

modifies:

effects:

returns:

```
static List<Integer> listAdd(List<Integer> lst1,
                             List<Integer> lst2) {
    List<Integer> res = new ArrayList<Integer>();
    for (int i = 0; i < lst1.size(); i++)
        res.add(lst1.get(i) + lst2.get(i));
    return res;
}
```

Fall 15 CSCI 2600, A Milanova (due to Michael Ernst)

49

Exercise

```
static void listAdd2(List<Integer> lst1,
                    List<Integer> lst2)
```

requires: lst1 non-null and lst2 non-null

modifies:

effects:

returns:

```
static void listAdd(List<Integer> lst1,
                    List<Integer> lst2) {
    for (int i = 0; i < lst1.size(); i++) {
        lst1.set(i, lst1.get(i) + lst2.get(i));
    }
}
```

Fall 15 CSCI 2600, A Milanova (due to Michael Ernst)

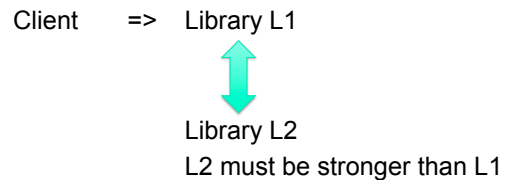
50

Specification Strength

- "A is stronger than B" means
 - For every implementation I
 - "I satisfies A" implies "I satisfies B"
 - The opposite is not necessarily true
 - For every client C
 - "C meets the obligations of B" implies "C meets the obligations of A"
 - The opposite is not necessarily true
- Principle of substitutability:
 - A stronger spec can always be substituted for a weaker one

51

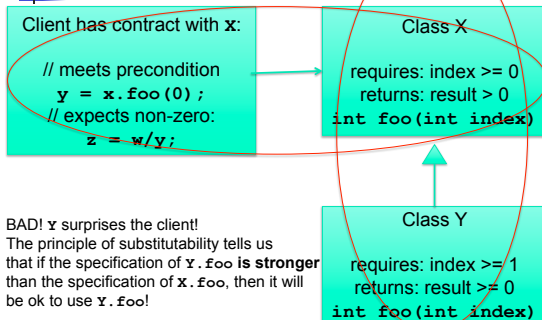
Specification Strength and Modularity



Spring 15 CSCI 2600, A Milanova

52

Spec strength, Substitutability and Modularity



Strengthening and Weakening Specification

- Strengthen a specification
 - Require less of client: fewer conditions in **requires** clause AND/OR
 - Promise more to client: **effects**, **modifies**, **returns**
 - Effects/modifies affect fewer objects
- Weaken a specification
 - Require more of client: add conditions to **requires** AND/OR
 - Promise less to client: **effects**, **modifies**, **returns** clauses are weaker, thus easier to satisfy in code

Comparing Specifications by Logical Formulas

- Specification A is a logical formula: $P_A \Rightarrow Q_A$
(meaning, precondition of A implies postcondition of A)
- Spec A is **stronger** than spec B if and only if
for each implementation I, (I satisfies A) \Rightarrow (I satisfies B)
which is equivalent to $A \Rightarrow B$
- $A \Rightarrow B$ means $(P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B)$

Recall from FoCS and/or Intro to Logic: $p \Rightarrow q \equiv !p \vee q$

Fall 15 CSCI 2600, A Milanova

55

Comparing by Logical Formulas

$(P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B) =$
 $!(P_A \Rightarrow Q_A) \vee (P_B \Rightarrow Q_B) =$ [due to law $p \Rightarrow q = !p \vee q$]
 $!(P_A \vee Q_A) \vee (!P_B \vee Q_B) =$ [due to $p \Rightarrow q = !p \vee q$]
 $(P_A \wedge !Q_A) \vee (!P_B \vee Q_B) =$ [due to $!(p \vee q) = !p \wedge !q$]
 $(!P_B \vee Q_B) \vee (P_A \wedge !Q_A) =$ [due to commutativity of \vee]
 $(!P_B \vee Q_B \vee P_A) \wedge (!P_B \vee Q_B \vee !Q_A)$ [distributivity]
 $[P_B \Rightarrow (Q_B \vee P_A)] \wedge [(P_B \wedge Q_A) \Rightarrow Q_B]$
 Translation: A is stronger than B if and only if
 P_B implies Q_B or P_A AND
 Q_A together with P_B implies Q_B

Fall 15 CSCI 2600, A Milanova

56

Comparing by Logical Formulas

if and only if

$$[P_B \Rightarrow (Q_B \vee P_A)] \wedge [(P_B \wedge Q_A) \Rightarrow Q_B]$$

CONSTRAINT ON
PRECONDITIONS

CONSTRAINT ON
POSTCONDITIONS

Sometimes we use the simpler test:
Spec A is stronger than Spec B if
 $P_B \Rightarrow P_A$ and $Q_A \Rightarrow Q_B$

Fall 15 CSCI 2600, A Milanova

57

Example:

`int find(int[] a, int value)`

- Specification B:
requires: a is non-null and value occurs in a $[P_B]$
returns: i such that $a[i] = \text{value}$ $[Q_B]$
- Specification A:
requires: a is non-null $[P_A]$
returns: i such that $a[i] = \text{value}$ if value occurs in a
and i = -1 if value is not in a $[Q_A]$

Clearly, $P_B \Rightarrow P_A$ (P_B includes P_A and one more condition)
Also, $P_B \wedge Q_A \Rightarrow Q_B$. P_B says that "value occurs in a" and
 Q_A says "value occurs in a \Rightarrow returns i such that
 $a[i] = \text{value}$ ". Thus, "returns i such that $a[i] = \text{value}$ ",
which is exactly Q_B , holds.

58

Exercise: Order by Strength

- Spec A: requires: a non-negative int argument
returns: an int in $[1..10]$
- Spec B: requires: int argument
returns: an int in $[2..5]$
- Spec C: requires: true
returns: an int in $[2..5]$
- Spec D: requires: an int in $[1..10]$
returns: an int in $[1..20]$

Fall 15 CSCI 2600, A Milanova

59

Function Subtyping



- Method inputs:
 - Parameter types of $B.m$ may be replaced by supertypes in subclass $A.m$. "contravariance"
 - E.g., $B.m(\text{Integer } p)$ and $A.m(\text{Number } p)$
 - This places no extra requirements on the client!
 - E.g., client: $B \ b; \dots \ b.m(q)$. Client knows to provide q a Integer or a subtype of Integer. Thus, client code will work fine with $A.m(\text{Number } p)$, which asks for less: an Number or a subtype of Number
 - Java does not allow change of parameter types in an overriding method. More on Java overriding shortly.

60

Function Subtyping

Method results:

- Return type of **B.m** may be replaced by subtype in subclass **A.m**. “**covariance**”
 - E.g., **Number B.m()** and **Integer A.m()**
- This does not violate expectations of the client!
 - E.g., client: **B b; ... Number n = b.m()**. Client expects a **Number**. Thus, **Integer** will work fine
- No new exceptions. Existing exceptions can be replaced by subtypes
- Java **does** allow a subtype return type in an overriding method!



61

Exercise

A's m: **X m(X y, String s);**

Let **z** be a subtype of **y**, **y** is subtype of **x**, which **m** is function subtype of **A's m**?

B's m:

Y m(Object y, Object s);

Z m(Y y, String s);

62

How to Use Wildcards

- Use **<? extends T>** when you *get* (read) values from a *producer* (*? is return*)
- Use **<? super T>** when you *add* (write) values into a *consumer* (*? is parameter*)
- E.g.:


```
<T> void copy(List<? super T> dst, List<? extends T> src)
```
- PECS: Producer Extends, Consumer Super
- Use neither, just **<T>**, if both *add* and *get*

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

63

Using Wildcards

Any collection of subtypes of **E** is fine

```

class HashSet<E> implements Set<E> {
    void addAll(Collection<? extends E> c) {
        // What does this give us about c?
        // i.e., what can code assume about c?
        // What operations can code invoke on c?
    }
}
    
```

- There is also **<? super E>**
- Intuitively, why **<? extends E>** makes sense here?

Fall 15 CSCI 2600, A Milanova (based on slide due to Michael Ernst)

64

Using Wildcards

```

class PriorityQueue<E> extends AbstractQueue<E> {
    PriorityQueue(int capacity, Comparator<? super E> c) {
        // What does this give us about c?
        // i.e., what can code assume about c?
        // What operations can code invoke on c?
    }
}
    
```

Fall 15 CSCI 2600, A Milanova

65

Legal Operations on Wildcards

Which of these is legal?

```

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>();
    
```

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

66

Legal Operations on Wildcards

```
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>();

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);
```

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

67

Topics

- ADTs, representation invariants and abstraction functions
 - Benefits of ADT methodology, Specifying ADTs
 - Rep invariant, abstraction function, representation exposure, checkRep, properties of abstraction function, benevolent side effects, proving rep invariants

Fall 15 CSCI 2600, A Milanova

68

ADTs

- Abstract Data Type (ADT):** higher-level data abstraction
 - The ADT is operations + object
 - A specification mechanism
 - A way of thinking about programs and design

Fall 15 CSCI 2600, A Milanova (slide based on slide by Michael Ernst)

69

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!

```
class Point {
    float x, y;
}
```

```
class Point {
    float r, theta;
}
```

- Instead, think of a type as a **set of operations**: create, **x()**, **y()**, **r()**, **theta()**.
- Force clients to call operations to access data

Fall 15 CSCI 2600, A Milanova (slide modified from material by Michael Ernst)

70

Specifying an ADT

immutable	mutable
class TypeName	class TypeName
1. overview	1. overview
2. abstract fields	2. abstract fields
3. creators	3. creators
4. observers	4. observers
5. producers	5. producers (rare!)
6. mutators	6. mutators

Fall 15 CSCI 2600, A Milanova (slide by Michael Ernst)

71

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

Fall 15 CSCI 2600, A Milanova (based on slides by Michael Ernst)

72

Representation Exposure

- Suppose we add this method to IntSet:


```
public List<Integer> getElements() {
    return data;
}
```
- Now client has direct access to the rep `data`, can modify rep and break rep invariant
- Representation exposure** is external access to the rep. **AVOID!!!**
- Better: make a copy on the way out; make a copy on the way in

Fall 15 CSCI 2600, A Milanova

73

Checking Rep Invariant

- Always check if rep invariant holds when debugging
- Leave checks anyway, if they are inexpensive
- Checking rep invariant of IntSet


```
private void checkRep() {
    for (int i=0; i<data.size; i++)
        if (data.indexOf(data.elementAt(i)) != i)
            throw RuntimeException("duplicates");
}
```

Fall 15 CSCI 2600, A Milanova

74

Abstraction Function: mapping rep to abstract value

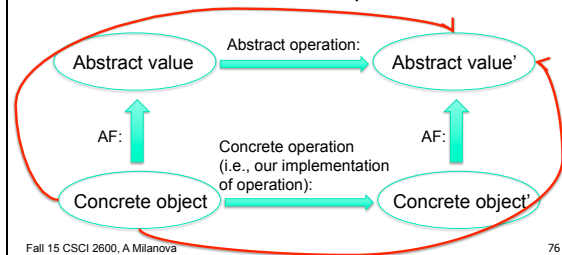
- Abstraction function: Object \rightarrow abstract value
 - I.e., the object's rep maps to abstract value
 - IntSet e.g.: list [2, 3, 1] \rightarrow { 1, 2, 3 }
 - Many objects map to the same abstract value
 - IntSet e.g.: [2, 3, 1] \rightarrow { 1, 2, 3 } and [3, 1, 2] \rightarrow { 1, 2, 3 } and [1, 2, 3] \rightarrow { 1, 2, 3 }
- Not a function in the opposite direction
 - One abstract value maps to many objects

Fall 15 CSCI 2600, A Milanova

75

Correctness

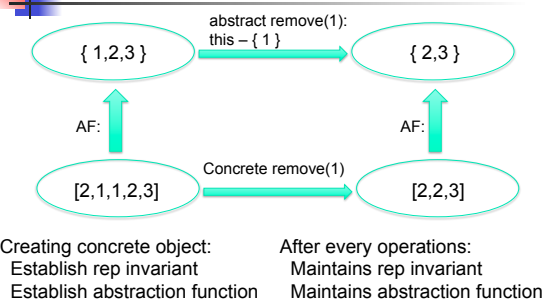
- Abstraction function allows us to reason about correctness of the implementation



Fall 15 CSCI 2600, A Milanova

76

IntSet Example



Fall 15 CSCI 2600, A Milanova

77

Proving rep invariants by induction

- Proving facts about infinitely many objects
- Base step
 - Prove rep invariant holds on exit of constructor
- Inductive step
 - Assume rep invariant holds **on entry** of method
 - Then **prove** rep invariant holds **on exit**
- Intuitively: there is no way to make an object, for which the rep invariant does not hold
- Remember, our proofs are informal

Fall 15 CSCI 2600, A Milanova (based on a slide by Michael Ernst)

78

Exercise: Willy's IntStack

Prove rep invariant holds

```
class IntStack {
    // Rep invariant: |theRep| = size
    // and theRep.keySet = {i | 1 ≤ i ≤ size}
    private IntMap theRep = new IntMap();
    private int size = 0;

    public void push(int val) {
        size = size+1;
        theRep.put(size, val);
    }
    public int pop() {
        int val = theRep.remove(size);
        size = size-1;
        return val;
    }
}
```

79

Exercise: Willy's IntStack

- Base case
 - Prove rep invariant holds on exit of constructor
- Inductive step
 - Prove that if rep invariant holds on entry of method, it holds on exit of method
 - push
 - Pop
- For brevity, ignore popping an empty stack

Fall 15 CSCI 2600, A Milano

80

Exercise: Willy's IntStack

- What if Willy added this method:

```
public IntMap getMap() {
    return theRep;
}
```

- Does the proof still hold?

Fall 15 CSCI 2600, A Milano

81

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

Fall 15 CSCI 2600, A Milano

82

Equivalence Partitioning

- Partition the input and/or output domains into equivalence classes
 - E.g., spec of sqrt(double x):
returns: square root of x if $x \geq 0$
throws: IllegalArgumentException if $x < 0$
- Partition the input domain
 - E.g., test $x < 0$, test $x = 0$, test $x \geq 0$
- Partition the output domain too
 - E.g., test $x < 1$, $x = 1$, $x > 1$ (something interesting happens at 1)

83

Boundary Value Analysis

- Choose test inputs at the edges of the input equivalence classes
 - Sqrt example: test with 0,
- Choose test inputs that produce outputs at the edges of output equivalence classes
- Other boundary cases
 - Arithmetic: zero, overflow
 - Objects: null, circular list, aliasing

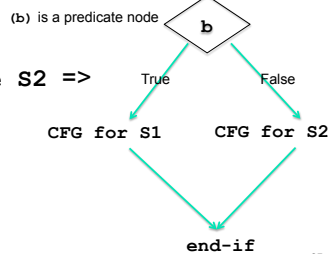
84

Control-flow Graph (CFG)

- Assignment $x=y+z \Rightarrow$ node in CFG: $x=y+z$

If-then-else

if (b) S1 else S2 \Rightarrow



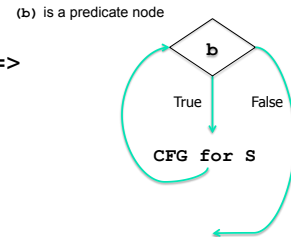
Fall 15 CSCI 2600, A Milanova

85

Control-flow Graph (CFG)

Loop

while (b) S \Rightarrow



Fall 15 CSCI 2600, A Milanova

86

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
- Def-use coverage**

Fall 15 CSCI 2600, A Milanova

87

Exercise

Draw the CFG for
 // requires: positive integers a,b
 static int gcd(int a, int b) {
 while (a != b) {
 if (a > b) {
 a = a - 2b;
 } else {
 b = b - a;
 }
 }
 return a;
 }

What is %branch coverage for gcd(15, 6)?

88

Topics

Subtyping vs. subclassing

- Subtype polymorphism, true subtypes and the LSP, specification strength and function subtyping, Java subtypes (overriding and overloading)

Fall 15 CSCI 2600, A Milanova

89

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!

Fall 15 CSCI 2600, A Milanova

91

What is True Subtyping?

- Also called **behavioral subtyping**
 - A true subtype is not only a Java subtype but a “behavioral subtype”
- B is subtype of A means every B is an A
- B shall “behave” as an A
 - B shall require no more than A
 - B shall promise at least as much as A
 - In other words, B will do fine where an A is expected

Fall 15 CSCI 2600, A Milanova

92

Subtypes are Substitutable

- Subtypes are **substitutable** for supertypes
 - Instances of subtypes won’t surprise client by requiring more than the supertype’s specification
 - Instances of subtypes won’t surprise client by failing to satisfy supertype specification
- 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 substitutable are **confusing** and **dangerous**

93

Liskov Substitution Principle (LSP)

- Due to Barbara Liskov, Turing Award 2008
- LSP: A subclass B of A should be substitutable for A, i.e., B should be a true subtype of A
- Reasoning about substitutability of B for A
 - B should not remove methods from A
 - For each B.m, which “substitutes” 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 and B’s m should not surprise client

94

Overloading vs. Overriding

- A **method family** contains multiple implementations of same **name + parameter types** (but not return type!)
- Which **method family** is determined at **compile time** based on **compile-time** types
 - E.g., family put(Object key, Object value) or family put(String key, String value)
- Which **implementation** from the **method family** runs, is determined at **runtime** based on the type of the receiver

Fall 15 CSCI 2600, A Milanova (based on slides by Mike Ernst)

95

Exercise

At compile-time call resolves method family **visit(VarExp)**

```
class VarExp extends BooleanExp {
    void accept(Visitor v) {
        v.visit(this);
    }
}
class Constant extends BooleanExp {
    void accept(Visitor v) {
        v.visit(this);
    }
}
```

Why not move void accept(Visitor v) up into superclass BooleanExp?

Fall 15 CSCI 2600, A Milanova

```
class Evaluate implements Visitor {
    // state, needed to
    // evaluate
    void visit(VarExp e) {
        //evaluate Var exp
    }
    void visit(Constant e) {
        //evaluate And exp
    } //visit for all exps
}
class PrettyPrint implements Visitor {
    ...
}
```

96

Topics

Equality

- Properties of equality, reference vs. value equality, equality and inheritance, equals() and hashCode(), equality and mutation

Fall 15 CSCI 2600, A Milanova

97

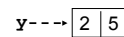
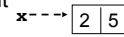
Equality: == and equals()

- 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

98

Properties of Equality

- Equality is an **equivalence relation**

- Reflexive** `a.equals(a)`
- Symmetric** `a.equals(b) ⇔ b.equals(a)`
- Transitive** `a.equals(b) ∧ b.equals(c) ⇒ a.equals(c)`

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

99

Equality and Inheritance

- Let B extend A
- “Natural” definition of **B.equals** is not symmetric
- Fix renders **equals** non transitive

- One can avoid these issues by allowing equality for exact classes:

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

Fall 15 CSCI 2600, A Milanova

100

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()`

101

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 chose”
 - 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)

Fall 15 CSCI 2600, A Milanova (slide by Michael Ernst)

102

Equality and Mutation

- Mutation can **violate rep invariant** of a Set container (rep invariant: there are no duplicates in set) by **mutating after insertion**

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

103

Exercise: Remember Duration

```
class Object {
    // Two method families.
    public boolean equals(Object o);
}
class Duration {
    public boolean equals(Object o) //override
    public boolean equals(Duration d);
}
Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// Compiler chooses family equals(Duration d)
```

Fall 15 CSCI 2600, A Milanova

104

Exercise: Remember Duration

```
class Object {
    public boolean equals(Object o);
}
class Duration {
    public boolean equals(Object o);
    public boolean equals(Duration d);
}
Object d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// Compiler chooses equals(Object o)
// At runtime: Duration.equals(Object o)
```

105

Exercise: Remember Duration

```
class Object {
    public boolean equals(Object o);
}
class Duration {
    public boolean equals(Object o);
    public boolean equals(Duration d);
}
Object d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// Compiler chooses equals(Object o)
// At runtime: Duration.equals(Object o)
```

106

Exercise: Remember Duration

```
class Object {
    public boolean equals(Object o);
}
class Duration {
    public boolean equals(Object o);
    public boolean equals(Duration d);
}
Duration d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// Compiler chooses equals(Object o)
// At runtime: Duration.equals(Object o)
```

107

Exercise

```
class Y extends X { ... } A a = new B();
                          Object o = new Object();
class A {
    X m(Object o) { ... } // Which m is called?
}
class B extends A {
    X m(Z z) { ... }
}
class C extends B {
    Y m(Z z) { ... }
}
A a = new C();
Object o = new Z();
// Which m is called?
X x = a.m(o);
```

Fall 15 CSCI 2600, A Milanova

108

Exercise

```
class Y extends X { ... } A a = new B();
class W extends Z { ... } W w = new W();
class A {
    X m(Z z) { ... }           // Which m is called?
    X x = a.m(w);
}
class B extends A {
    X m(W w) { ... }           B b = new C();
}
class C extends B {
    Y m(W w) { ... }           W w = new W();
    X x = b.m(w);
}
```

Fall 15 CSCI 2600, A Milanova

109

Topics

■ Design Patterns

- Creational patterns: Factory method, Factory class, Prototype, Singleton, Interning
- Structural patterns:
 - Wrappers: Adapter, Decorator, Proxy
 - Composite
 - Façade
- Behavioral patterns:
 - Interpreter, Procedural, Visitor
 - Observer
 - State, Strategy, Template Method

110

Design Patterns

- A **design pattern** is a solution to a design problem that occurs over and over again
- Design patterns promote extensibility and reuse
 - Open/Closed Principle:
Help build software that is **open to extension but closed to modification**
- Majority of design patterns make use of subtype polymorphism

Fall 15 CSCI 2600, A Milanova

111

Exercises (creational patterns)

- What pattern forces a class to have a single instance?
- What patterns allow for creation of objects that are subtypes of a given type?
- What pattern helps reuse existing objects?

Fall 15 CSCI 2600, A Milanova

112

Exercises (creational patterns)

- Can interning be applied to mutable types?
- Can a mutable class be a Singleton?

Fall 15 CSCI 2600, A Milanova

113

Creational Patterns

- Problem: constructors in Java (and other OO languages) are inflexible
 1. Can't return a subtype of the type they belong to
 2. Always return a fresh new object, can't reuse
- "Factory" creational patterns present a solution to the first problem
 - Factory method, Factory object, Prototype
- "Sharing" creational patterns present a solution to the second problem
 - Singleton, Interning

114

Factory Method

- MazeGames are created the same way. Each MazeGame (Enchanted, Bombed) works with its own Room, Wall and Door products
- Factory method allows each MazeGame to create its own products (MazeGame defers creation)

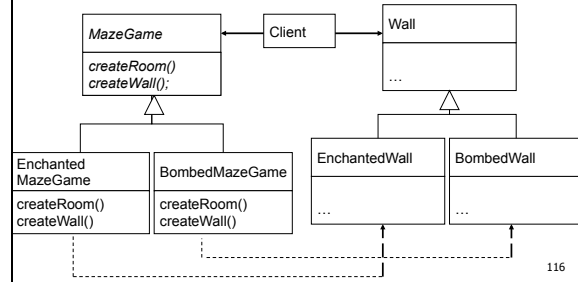
```
abstract class MazeGame {
    abstract Room createRoom();
    abstract Wall createWall();
    abstract Door createDoor();
    Maze createMaze() {
        ...
        Room r1 = createRoom(); Room r2 = ...
        Wall w1 = createWall(r1,r2); ... createDoor(w1); ...
    }
}
```

Fall 15 CSCI 2600, A Milanova

115

Factory Method Class Diagram

- MazeGame and Products Hierarchies



116

Factory Class/Object

- Encapsulate factory methods in a factory object
- MazeGame gives control of creation to factory object

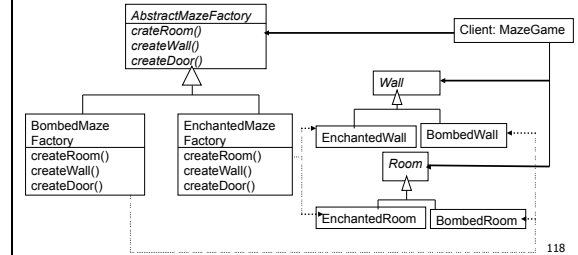
```
class MazeGame {
    AbstractMazeFactory mfactory;
    MazeGame(AbstractMazeFactory mfactory) {
        this.mfactory = mfactory;
    }
    Maze createMaze() {
        ...
        Room r1 = mfactory.createRoom(); Room r2 = ...
        Wall w1 = mfactory.createWall(r1,r2);
        Door d1 = mfactory.createDoor(w1); ...
    }
}
```

Fall 15 CSCI 2600, A Milanova

117

Factory Class/Object Pattern (also known as Abstract Factory)

- Motivation: Encapsulate the factory methods into one class. Separate control over creation



118

The Prototype Pattern

- Every object itself is a factory
- Each class contains a clone method and returns a copy of the receiver object

```
class Room {
    Room clone() { ... }
}
```

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

119

Using Prototypes

```
class MazeGame {
    Room rproto;
    Wall wproto;
    Door dproto
    MazeGame(Room r, Wall w, Door d) {
        rproto = r; wproto = w; dproto = d;
    }
    Maze createMaze() {
        ...
        Room r1 = rproto.clone(); Room r2 = ...
        Wall w1 = wproto.clone();
        Door d1 = dproto.clone(); ...
    }
}
```

Fall 15 CSCI 2600, A Milanova

120

Singleton Pattern

- Guarantees there is a single instance of the class

```
class Bank {  
    private Bank() { ... }  
    private static Bank instance;  
    public static Bank getInstance() {  
        if (instance == null)  
            instance = new Bank();  
        return instance;  
    }  
}
```

Factory method --- it produces the instance of the class

Fall 15 CSCI 2600, A Milanova

121

Interning Pattern

- Reuse existing objects with same value
 - To save space, to improve performance
- Permitted for immutable types only
- Maintain a collection of all names. If an object already exists return that object:

```
HashMap<String,String> names;  
String canonicalName(String n) {  
    if (names.containsKey(n))  
        return names.get(n);  
    else {  
        names.put(n,n);  
        return n;  
    }  
}
```

Fall 15 CSCI 2600, A Milanova

122

Exercises (structural patterns)

- What design pattern represents complex whole-part objects?
- What design pattern changes the interface of a class without changing its functionality?
- What design pattern adds small pieces of functionality without changing the interface?

Fall 15 CSCI 2600, A Milanova

123

Exercises (structural patterns)

- What pattern helps restrict access to an object?
- What is the difference between an object adapter and a class adapter? Which one is more efficient?
- What pattern hides a large and complex library and promotes low coupling between the library and the client?

Fall 15 CSCI 2600, A Milanova

124

Wrappers

- A wrapper pattern uses composition/delegation
- Wrappers are a thin layer over an encapsulated class
 - Modify the interface
 - Extend behavior
 - Restrict access
- The encapsulated object (delegate) does most work
- Adapter**: modifies interface, same functionality
- Decorator**: same interface, extends functionality
- Proxy**: same interface, same functionality

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

125

Adapter Pattern

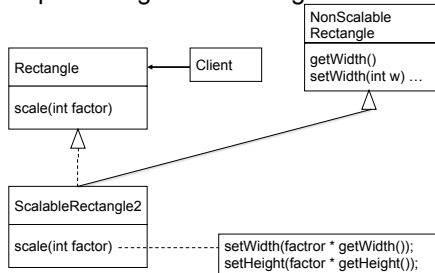
- Change an interface without changing functionality of the encapsulated class. Reuse functionality
 - Rename methods
 - Convert units
 - Implement a method in terms of another

Fall 15 CSCI 2600, A Milanova

126

Class Adapter

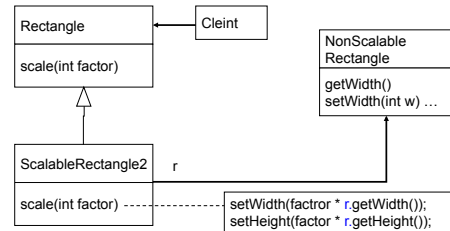
- Adapts through subclassing



127

Object Adapter

- Adapts through delegation:



128

Adapter Example: Scaling Rectangles

```

interface Rectangle {
    void scale(int factor); //grow or shrink by factor
    ...
    float getWidth();
    float area();
}
class Client {
    void clientMethod(Rectangle r) {
        ... r.scale(2);
    }
}
class NonScalableRectangle {
    void setWidth(); ...
    // no scale method!
}
    
```

Fall 15 CSCI 2600, A. Milanova

129

Class Adapter

- Adapting via subclassing

```

class ScalableRectangle1
    extends NonScalableRectangle
    implements Rectangle {
    void scale(int factor) {
        setWidth(factor*getWidth());
        setHeight(factor*getHeight());
    }
}
    
```

Fall 15 CSCI 2600, A. Milanova

130

Object Adapter

- Adapting via delegation: forward to delegate

```

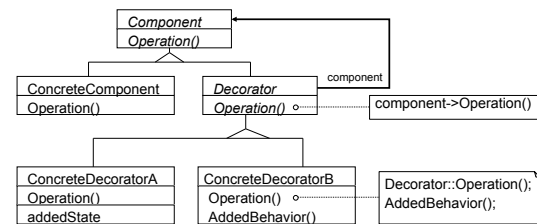
class ScalableRectangle2 implements Rectangle {
    NonScalableRectangle r; // delegate
    ScalableRectangle2(NonScalableRectangle r) {
        this.r = r;
    }
    void scale(int factor) {
        setWidth(factor * r.getWidth());
        setHeight(factor * r.getHeight());
    }
    float getWidth() { return r.getWidth(); }
    ...
}
    
```

Fall 15 CSCI 2600, A. Milanova

131

Structure of Decorator

- Motivation: add small chunks of functionality without changing the interface



Fall 15 CSCI 2600, A. Milanova

132

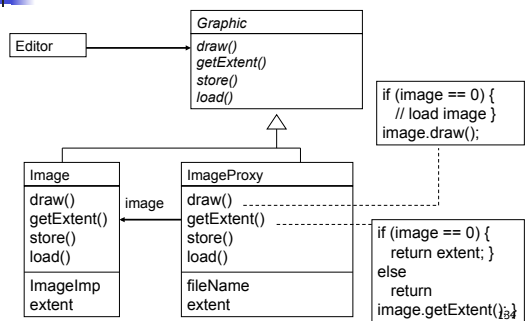
Proxy Pattern

- Same interface and functionality as the enclosed class
- Control access to other object
 - Communication: manage network details when using a remote object
 - Locking: serialize access by multiple clients
 - Security: permit access only if proper credentials
 - Creation: object might not yet exist (creation is expensive). Hide latency when creating object. Avoid work if object never used

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

133

Proxy Example: manage creation of expensive object



Composite Pattern

- Good for part-whole relationships
 - Can represent arbitrarily complex objects
- Client treats a **composite** object (a **collection** of units) the **same** as a simple object (an **atomic** unit)

Fall 15 CSCI 2600, A Milanova

135

Using Composite to represent boolean expressions

```
abstract class BooleanExp {
    boolean eval(Context c);
}

class Constant extends BooleanExp {
    private boolean const;
    Constant(boolean const) { this.const=const; }
    boolean eval(Context c) { return const; }
}

class VarExp extends BooleanExp {
    String varname;
    VarExp(String var) { varname = var; }
    boolean eval(Context c) {
        return c.lookup(varname);
    }
}
```

136

Using Composite to represent boolean expressions

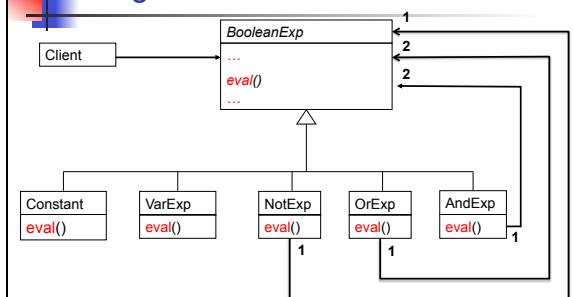
```
class AndExp extends BooleanExp {
    private BooleanExp leftExp;
    private BooleanExp rightExp;
    boolean eval(Context c) {
        return leftExp.eval(c) && rightExp.eval(c);
    }
}

// analogous definitions for OrExp and NotExp
```

Fall 15 CSCI 2600, A Milanova

137

Object Structure vs. Class Diagram



Fall 15 CSCI 2600, A Milanova

138

Exercises (Behavioral Patterns)

- What pattern(s) help traverse composite objects?
- What pattern(s) groups unrelated traversal operations into classes in the composite hierarchy?
- What pattern(s) group all related traversal operations into separate classes?

Fall 15 CSCI 2600, A Milanova

139

Exercises

- If you anticipate the composite hierarchy to change and the set of operations to stay constant, what pattern would you rather use, **Interpreter** or **Visitor**?
- Conversely, if you anticipate no changes in the composite hierarchy (e.g., BooleanExp doesn't change), but you expect addition of traversal operations, what pattern would you use, **Interpreter** or **Visitor**?

Fall 15 CSCI 2600, A Milanova

140

Exercises

- What pattern allows for an object to maintain multiple views that must be updated when the object changes?
- Give an example of usage of the Composite pattern in the Java GUI library
- Given an example of usage of the Observer pattern in the Java GUI library

Fall 15 CSCI 2600, A Milanova

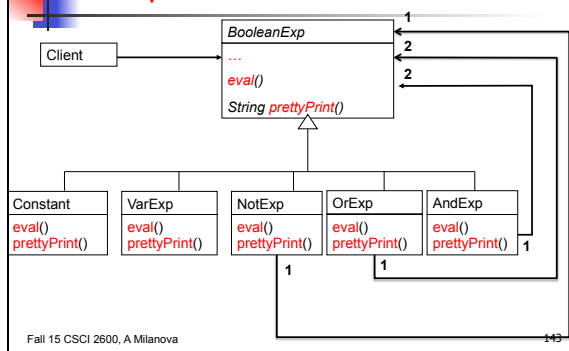
141

Patterns for Traversing Composites

- Interpreter** pattern
 - Groups operations per class. Each class implements operations: **eval**, **prettyPrint**, etc.
 - Easy to add a class to the Composite hierarchy, hard to add a new operation
- Procedural** pattern
 - Groups similar operations together
- Visitor** pattern – a variation of Procedural
 - Groups operations together. Classes in composite hierarchy implement **accept(Visitor)**
 - Easy to add a class with operations in Visitor hierarchy, harder to add a new class in Composite hierarchy

142

Interpreter Pattern



Fall 15 CSCI 2600, A Milanova

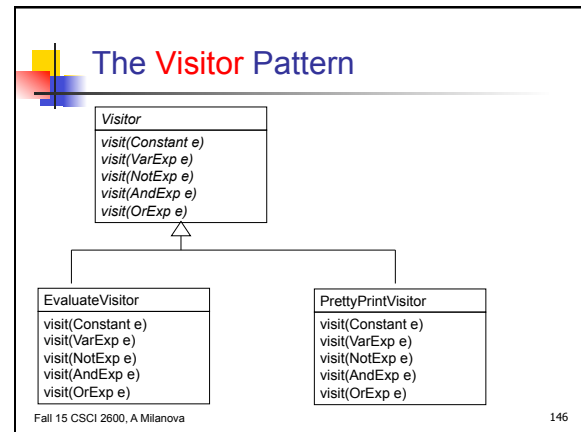
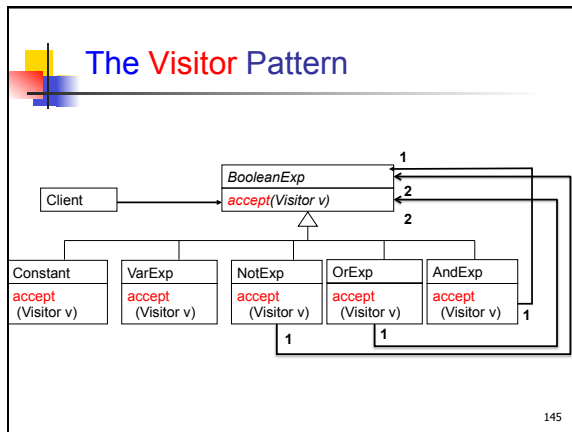
143

Visitor Pattern

```

class VarExp extends BooleanExp {
    implements Visitor {
        // keeps state
        void visit(VarExp e) {
            //evaluate var exp
        }
        void visit(AndExp e) {
            //evaluate And exp
        }
    }
}
class AndExp extends BooleanExp {
    BooleanExp leftExp;
    BooleanExp rightExp;
    void accept(Visitor v) {
        leftExp.accept(v);
        rightExp.accept(v);
        v.visit(this);
    }
}
class PrettyPrint implements Visitor {
    ...
}
    
```

144



Exercise

- Write **Count** implements **Visitor**
 - Counts # subexpressions in a boolean expression
- Write **EvaluateVisitor** implements **Visitor**
 - Evaluates boolean expression

Fall 15 CSCI 2600, A Milanova

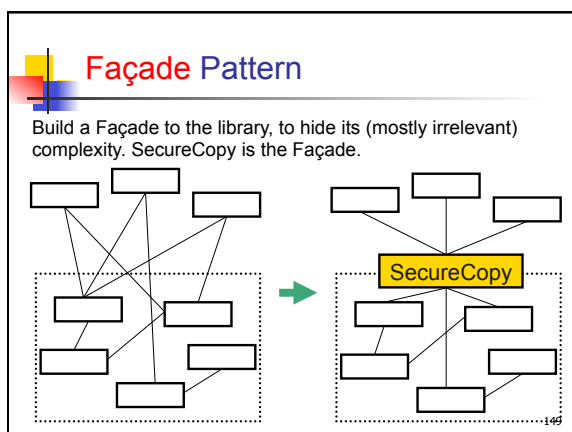
147

Façade Pattern

- Question: how to handle the case, when we need a subset of the functionality of a powerful, extensive and complex library
- Example: We want to perform secure file copies to a server. There is a powerful and complex general purpose library. What is the best way to interact with this library?

Fall 15 CSCI 2600, A Milanova

148



Observer Pattern

- Question: how to handle an object (model), which has many "observers" (views) that need to be notified and updated when the object changes state
- For example, an interface toolkit with various presentation formats (spreadsheet, bar chart, pie chart). When application data, e.g., stocks data (model) changes, all presentations (views) should change accordingly

Fall 15 CSCI 2600, A Milanova

150

A Better Design: The Observer

- Data class has minimal interaction with Views
 - Only needs to update Views when it changes

Old, naive design:

```
class Data {
    ...
    void updateViews() {
        spreadsheet.update(newData);
        barChart.update(newData);
        // Edit this method when
        // different views are added.
        // Bad!
    }
}
```

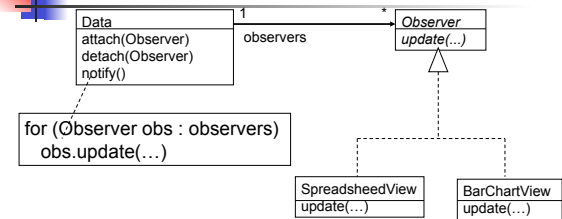
Better design:

```
class Data {
    List<Observer> observers;
    void notifyObservers() {
        for (obs : observers)
            obs.update(newData);
    }
}

interface Observer {
    void update(...);
}
```

151

Class Diagram



Client is responsible for View creation:

```
data = new Data();
data.attach(new BarChartView());
// Data keeps list of Views, notifies them when change.
// Data is minimally connected to Views!
```

Push vs. Pull Model

- Question: How does the object (Data in our case) know what info each observer (View) needs?
- A **push** model sends all the info to Views
- A **pull** model does not send info directly. It gives access to the Data object to all Views and lets each View extract the data they need

Fall 15 CSCI 2600, A. Milanova

153

Refactoring

- Premise: we have written complex (ugly) code that works. Can we simplify this code?
- Refactoring: structured, disciplined methodology for rewriting code
 - Small step** behavior-preserving transformations
 - Followed by **running test cases**

Fall 15 CSCI 2600, A. Milanova

154

Refactoring

- Refactorings attack **code smells**
- Code smells** – bad coding practices
 - E.g., big method
 - An oversized “God” class
 - Similar subclasses
 - Little or no use of interfaces and polymorphism
 - High coupling between objects,
 - Duplicate code
 - And more...

Fall 15 CSCI 2600, A. Milanova

155

Refactorings

- Extract Method, Move Method, Replace Temp with Query, Replace Type Code with State/Strategy, Replace Conditional with Polymorphism
- Goal: achieve code that is short, tight, **clear** and without duplication
- Did I say this already: **small change + tests**

Fall 15 CSCI 2600, A. Milanova

156

Topics

- Usability
 - Definition of usability, dimensions of usability, design principles for learnability, visibility, efficiency and safety, Fitts's law, Steering law

Fall 15 CSCI 2600, A. Milanova 157

Usability

- Usability: how well users can use the system's functionality
- Dimensions of usability
 - Learnability: is it easy to learn?
 - Efficiency: once learned, is it fast to use?
 - Safety: are errors few and recoverable?
 - Memorability: is it easy to remember what you learned?
 - Satisfaction: is it enjoyable to use?

Fall 15 CSCI 2600, A. Milanova. Slide from Michael Ernst 158

Usability

- Design principles for **learnability**
 - Consistency: internal, external, metaphorical
 - Use simple words, not tech jargon
 - Recognition, not recall
- Design principles for **visibility**
 - Make system state visible
 - Give prompt feedback
- **Simplicity!**

Fall 15 CSCI 2600, A. Milanova 159

Usability

- Design principles for **efficiency**
 - Human motor processor, Fitts's law and Steering law:
 - Make important targets big and nearby
 - Avoid steering tasks
 - Provide shortcuts
- Design principles for **safety (error handling)**
 - Avoid mode errors
 - Use confirmation windows sparingly

Fall 15 CSCI 2600, A. Milanova. Slide from Michael Ernst 160

Topics

- Software Process
 - Software lifecycle, activities (requirements, design, implementation, testing) and their artifacts, requirements analysis, software processes

Fall 15 CSCI 2600, A. Milanova 161

Software Process

- **Software lifecycle** activities:
 - Requirements analysis
 - **Design**
 - **Implementation**
 - Integration + **Testing and verification**
 - Deployment and maintenance
 - Maintenance is costly. **The later a problem is found, the costlier it is to fix**
- **Software process** puts these together
 - How do we combine these activities?
 - In what order?

Fall 15 CSCI 2600, A. Milanova 162

Activities and Their Artifacts

- Requirements analysis produces “requirements documents”
 - Use-case model, supplementary specifications
- Design produces “design models”
 - **Class diagrams**, interaction diagrams, ADT specs, other
- Implementation produces, well, ... obviously code
 - + specs for classes and individual methods, AFs and RIs
 - **Readability** of code is crucial!
- Testing produces
 - Test suites

Fall 15 CSCI 2600, A Milanova

163

Requirements Analysis is Hard

- Requirements analysis determines the functional and non-functional requirements of the system
- Requirements are a major causes of project failure
 - Poor user input
 - Incomplete requirements
 - Changing requirements

Fall 15 CSCI 2600, A Milanova

164

Classification of Requirements

- **FURPS+** model
- The **FURPS**:
 - **F**unctionality, **U**sability, **R**eliability, **P**erformance, **S**upportability
- The **+**:
 - Design constraints, implementation requirements (e.g., must use Java), other

Fall 15 CSCI 2600, A Milanova

165

Requirements Analysis Artifacts

- Requirements analysis produces:
 - **Use-case model**
 - A set of use cases
 - Specifies the **functional requirements** (behavior, features) of the system
 - **Supplementary specification**
 - Specifies non-functional requirements (-ilities: **u**sability, **r**eliability, **p**erformance, **s**upportability)

Fall 15 CSCI 2600, A Milanova

166

Use Cases

- Describe the interaction of the user with the system as TEXT stories
- The most widely used approach to requirements analysis in modern software practice
 - Requirements are discovered and recorded through use cases
 - All other activities influenced by use cases!

Fall 15 CSCI 2600, A Milanova

167

Example Use Case

- Point-of-sale (POS) system
- **Process Sale**: A **customer** arrives at checkout with items to buy. The **cashier** uses the **POS system** to record each purchased item. The **system** presents a running total and line-item details. The **customer** enters payment information, which the **system** validates and records. The **system** updates inventory. The **customer** receives a receipt.
- The use case is a collection of scenarios: **main success scenario** + **scenario variations**

Fall 15 CSCI 2600, A Milanova

168

Software Process

- **Software lifecycle** activities:
 - Requirements analysis
 - Design
 - Implementation
 - Testing
 - Deployment and maintenance
- **Software process** puts these activities together
- **Software process** forces attention to these activities and their artifacts

169

Some Software Processes

- **Code-and-fix** (ad-hoc): write some code, make up some inputs, debug
- **Waterfall**: 1st: requirements analysis, 2nd: design, 3rd: implementation, 4th: testing
- **Iterative** (Unified process, Agile, Scrum) repeat activities: (a small chunk of requirements, design, implementation, testing)⁺
- **Other**

Fall 15 CSCI 2600, A. Milanova

170