Review

THANKS!

- Professor Barb Cutler and the rest of the Submitty team for the Homework Server!

- Mentors: Victor Zhu, Michael Zemski, Tyler Sontag, Max Wang, Joyce Yom, Ruoyu Chen
- TAs: Smit Pandit, Saswata Paul, Vipula Rawte

Final Exam

- Tuesday, May 9 11:30-2:30 in Sage 3303

- Final exam is cumulative
- Closed notes, closed laptops/phones
- 2 double-sided 8.5x11-sized “cheat” pages

- Review Exam psoted later in the week
- Review Slides and notes
  - http://www.cs.rpi.edu/~thompw4/CSCI-2600/Spring2017/Topics.html
  - http://www.cs.rpi.edu/~thompw4/CSCI-2600/Spring2017/Slides/Review.pdf

- TA office hours this week, Tuesday, Wednesday
- I’ll have office hours Tuesday and Thursday

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!

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

Topics

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

Topics

- <u>Reasoning about code</u>
  - Forward and backward reasoning, logical conditions, Hoare triples, weakest precondition, rules for assignment, sequence, if-then-else, loops, loop invariants, decrementing functions

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 AND y = 2 }

  ```
  z = x + y
  ```

  { x = 1 AND y = 2 AND z = 3 }

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

Backward Reasoning

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

E.g., wp(z=y+1, 2*y < z) = { 2y < y+1 } = {y < 1} // Simplify into { y < 1 }

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

wp(x = 2*y, x < z) = {2*y < z }

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

{ x < z }

- More focused and more useful

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

Exercise. Condition Strength

- Which one is stronger? Assume x and y are ints.

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

$x > 0 \;\&\&\; y = 0$ or $x > 0 \;||\; y = 0$

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

$y \pmod 4 = 2$ or y is even

$x = 10$ or x is even

Hoare Triples

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

Exercises. Hoare Triples

{ x>0 } **`x++`** { x>1 } is true

{ x>0 } **`x++`** { x>-1 } is true

{ x≥0 } **`x++`** { x>1 } is false. Why?

{x>0} **`x++`** {x>0} is ??

{x<0} **`x++`** {x<0} is ??

{x=a} **`if (x < 0) x=-x`** { x = | a | } is ??

{x=y} **`x=x+3`** {x=y} is ??

Exercise

- Let **`P => Q => R`**
- (P is stronger than Q and Q is stronger than R)
- Let **`S => T => U`**
- Let **`{ Q } code { T }`**
- Which of the following are true:

`1. { P } code { T }`

`2. { R } code { T }`

`3. { Q } code { U }`

`4. { Q } code { S }`

Rules for Backward Reasoning: Assignment

// precondition: ??

`x = expression`

// postcondition: Q

Rule: the weakest precondition = Q, with all occurrences of **`x`** in Q replaced by **`expression`**

More formally:

wp(“**`x=expression;`**”,Q) = Q with all occurrences of **`x`** replaced by **`expression`**

Rules for Backward Reasoning: Sequence

// precondition: ??
`S1;` // statement
`S2;` // another statement
// postcondition: Q
Work backwards:
precondition is wp("**`S1;S2;`**", Q) = wp("**`S1;`**",wp("**`S2;`**",Q))
Example:

// precondition: ??
```
x = 0;
y = x+1;
```
// postcondition: y>0

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

Rules for If-then-else

Forward reasoning

{ P }
`if b`
  { P ∧ **`b`** }
  `S1`
  { Q1 }
`else`
  { P ∧ not **`b`** }
  `S2`
  { Q2 }
{ Q1 ∨ Q2 }

Backward reasoning

{ (**`b`**∧wp("**`S1`**",Q))∨(not **`b`**∧wp("**`S2`**",Q)) }
`if b`
  { wp("**`S1`**",Q) }
  `S1`
  { Q }
`else`
  { wp("**`S2`**",Q) }
  `S2`
  { Q }
{ Q }

Exercise

- Compute the weakest precondition:

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

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 invariant must be **relevant**
   - Loop exit condition and loop invariant must imply the desired postcondition
2. Termination
   - (Intuitively) Establish “decrementing function” D. Each iteration decrements D, D = minimum and loop invariant, imply loop exit condition

Example: Reasoning About Loops

$i+z = x$ is the loop invariant

Precondition: x >= 0;

```
i = x;
z = 0;
while (i != 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)

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

D = 0 implies i = 0 (loop exit condition).

Reasoning About Loops

- Loop invariant Inv must be such that

1) P **=>** Inv // Inv holds before loop. Base case
2) { Inv $\wedge$ **b** } **S** { 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
3) (Inv $\wedge$!**b**) **=>** Q // The exit condition **!b** and loop invariant Inv must imply postcondition

- Decrementing function D must be such that

1) D decreases every time we go through the loop
2) D = minimum and Inv must imply loop exit condition **!b**

Exercise

Precondition: $y >= 0$;

```
i = y;
n = 1;
while (i != 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: $\texttt{len} \geq 0 \land \texttt{arr.length} = \texttt{len}$

```
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;
}
```

Our specifications are informal.
Mathematical rigor is not always necessary.
Spec should be precise enough to enable reasoning yet easily readable.

Postcondition: returns `arr[0]+…+arr[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. <u>Any object not listed under this clause is guaranteed untouched</u>
  - 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;
}
```

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));
  }
}
```

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"
    - If C meets the weaker spec (B), it meets the stronger spec (A)
    - The opposite is not necessarily true
- Principle of substitutability:
  - <u>A stronger spec can always be substituted for a weaker one</u>

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

Spec strength, Substitutability and Modularity

Client has contract with **`X`**:

// meets precondition

```
y = x.foo(0);
```

// expects non-zero:

```
z = w/y;
```

Class X

requires: index >= 0

returns: result > 0

```
int foo(int index)
```

Class Y

requires: index >= 1

returns: result >= 0

```
int foo(int index)
```

BAD! **`Y`** surprises the client!

The principle of substitutability tells us that if the specification of **`Y.foo`** **is stronger** than the specification of **`X.foo`**, then it will be ok to use **`Y.foo`**!

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 <u>stronger</u> than spec B if and only if
  for each implementation I, (I satisfies A)=>(I satisfies B)
  which is equivalent to A => B

- A => B means $(P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B)$

Recall from FoCS and/or Intro to Logic: p => q Ξ !p V q

Comparing by Logical Formulas

$(P_A => Q_A) => (P_B => Q_B)$ =
$!(P_A => Q_A) \lor (P_B => Q_B)$ = [due to law p => q = !p ⋁ q]
$!(!P_A \lor Q_A) \lor (!P_B \lor Q_B)$ = [due to p => q = !p ⋁ q]
$(P_A \land !Q_A) \lor (!P_B \lor Q_B)$ = [due to !(p ⋁ q) = !p ∧ !q]
$(!P_B \lor Q_B) \lor (P_A \land !Q_A)$ = [due to commutativity of ⋁]
$(!P_B \lor Q_B \lor P_A) \land (!P_B \lor Q_B \lor !Q_A)$ [distributivity]
$[P_B => (Q_B \lor P_A)] \land [(P_B \land Q_A) => Q_B]$

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

Comparing by Logical Formulas

Constraint on Postconditions

if and only if

$[\ P_B \Rightarrow (Q_B \lor P_A)\ ] \land [\ (\ P_B \land Q_A\ ) \Rightarrow Q_B\ ]$

Constraint on preconsitions

Sometimes we use the simpler test:

Spec A is stronger than Spec B if

$P_B \Rightarrow P_A$ and $Q_A \Rightarrow Q_B$

That is, if A has a weaker precondition and A has a stronger postcondition

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] = value`** [Q_B]

- Specification A:

  requires: **`a`** is non-null [P_A]

  returns: **`i`** such that **`a[i] = value`** if value occurs in **`a`** and **`i = -1`** if value is not in **`a`** [Q_A]

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

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]

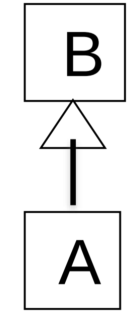
Spec C: requires: true

returns: an int in [2..5]

Spec D: requires: an int in [1..10]

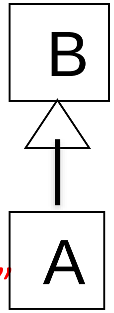
returns: an int in [1..20]

Function Subtyping



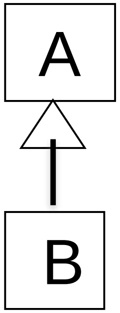
- Method inputs:
  - Parameter types of B.m may be replaced by supertypes in subclass A.m. "contravariance"
    - E.g., B.m(Integer p) and A.m(Number p)
  - This places no extra requirements on the client!
    - E.g., client: `B b; … b.m(q)`. Client knows to provide `q` a Integer or a subtype of Integer. Thus, client code will work fine with `A.m(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.

Function Subtyping



- Method results:
  - Return type of B.m may be replaced by subype 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!

Exercise



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

`Z` is 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);`

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*

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?

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?
   }
}
```

Using Wildcards

```
class NaturalNumber {

    private int i;

    public NaturalNumber(int i) { this.i = i; }
    // ...
}

class EvenNumber extends NaturalNumber {

    public EvenNumber(int i) { super(i); }
    // ...
}

List<EvenNumber> le = new ArrayList<>();
List<? extends NaturalNumber> ln = le;
ln.add(new NaturalNumber(35));  // compile-time error
```

Using Wildcards

- Because List<EvenNumber> is a subtype of List<? extends NaturalNumber>, you can assign le to ln.
- But you cannot use ln to add a natural number to a list of even numbers.

- The following operations on the list are possible:
  - You can add null.
  - You can invoke clear.
  - You can get the iterator and invoke remove.

- You can see that the list defined by List<? extends NaturalNumber> is not read-only in the strictest sense of the word, but you might think of it that way because you cannot store a new element or change an existing element in the list.

Legal Operations on Wildcards

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

List<? extends Integer> lei;
```

First, which of these is legal?

~~`lei = new ArrayList<Object>();`~~

~~`lei = new ArrayList<Number>();`~~

`lei = new ArrayList<Integer>();`

`lei = new ArrayList<PositiveInteger>();`

`lei = new ArrayList<NegativeInteger>();`

Which of these is legal?

~~`lei.add(o);`~~

~~`lei.add(n);`~~

~~`lei.add(i);`~~

~~`lei.add(p);`~~

`lei.add(null);`

`o = lei.get(0);`

`n = lei.get(0);`

`i = lei.get(0);`

~~`p = lei.get(0);`~~

Legal Operations on 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);`~~

Topics

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

ADTs

- Abstract Data Type (ADT): higher-level data abstraction
  - The ADT is <u>operations</u> + <u>object</u>
  - 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!

```
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

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

Connecting Implementation to Specification

- Representation invariant: Object → boolean
  - Indicates whether data representation is well-formed. Only well-formed representations are meaningful
  - Defines the set of valid values
- Abstraction function: Object → 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

- 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

Checking Rep Invariant

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

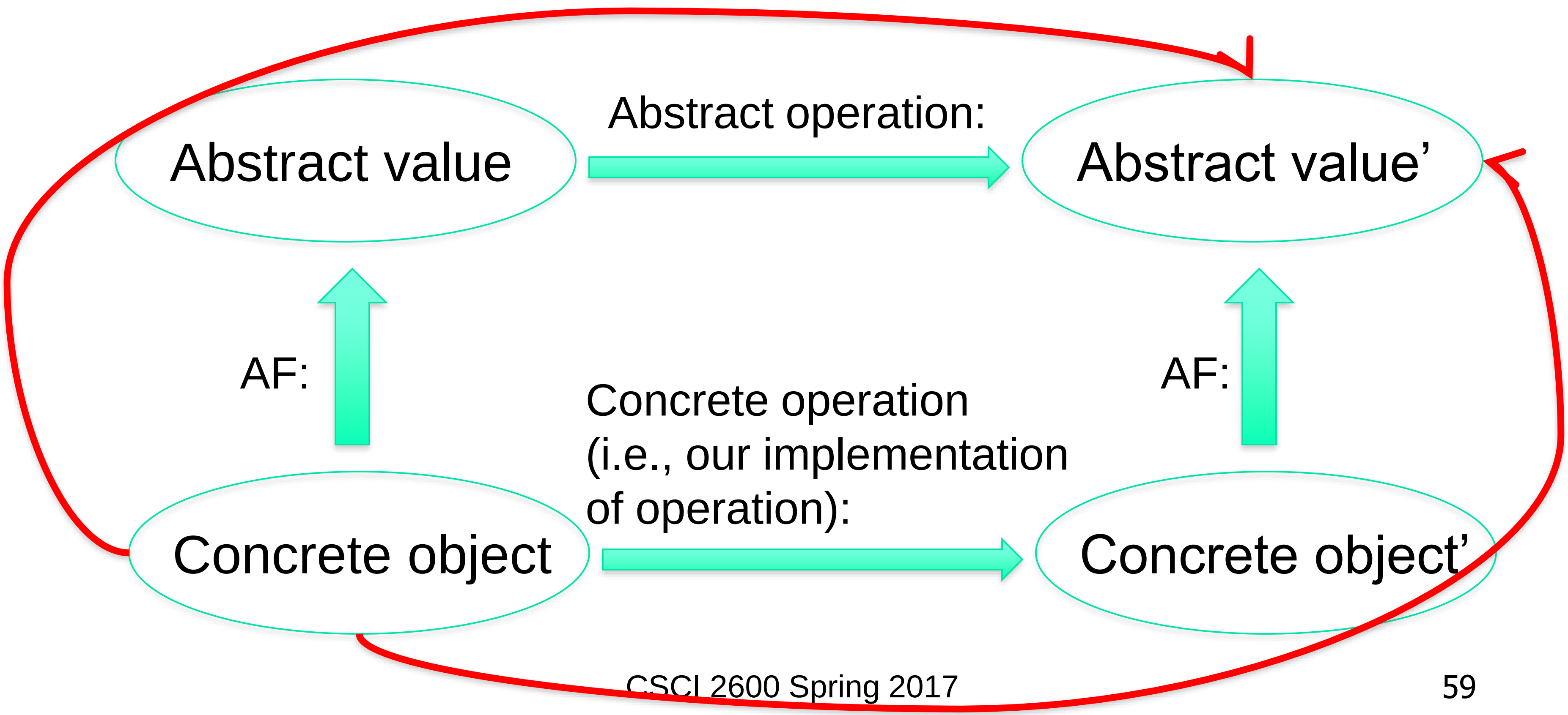
```
private void checkRep() {
  for (int i=0; i<data.size; i++)
    if (data.indexOf(data.elementAt(i)) != i)
      throw RuntimeException("duplicates");
}
```

Abstraction Function: mapping rep to abstract value

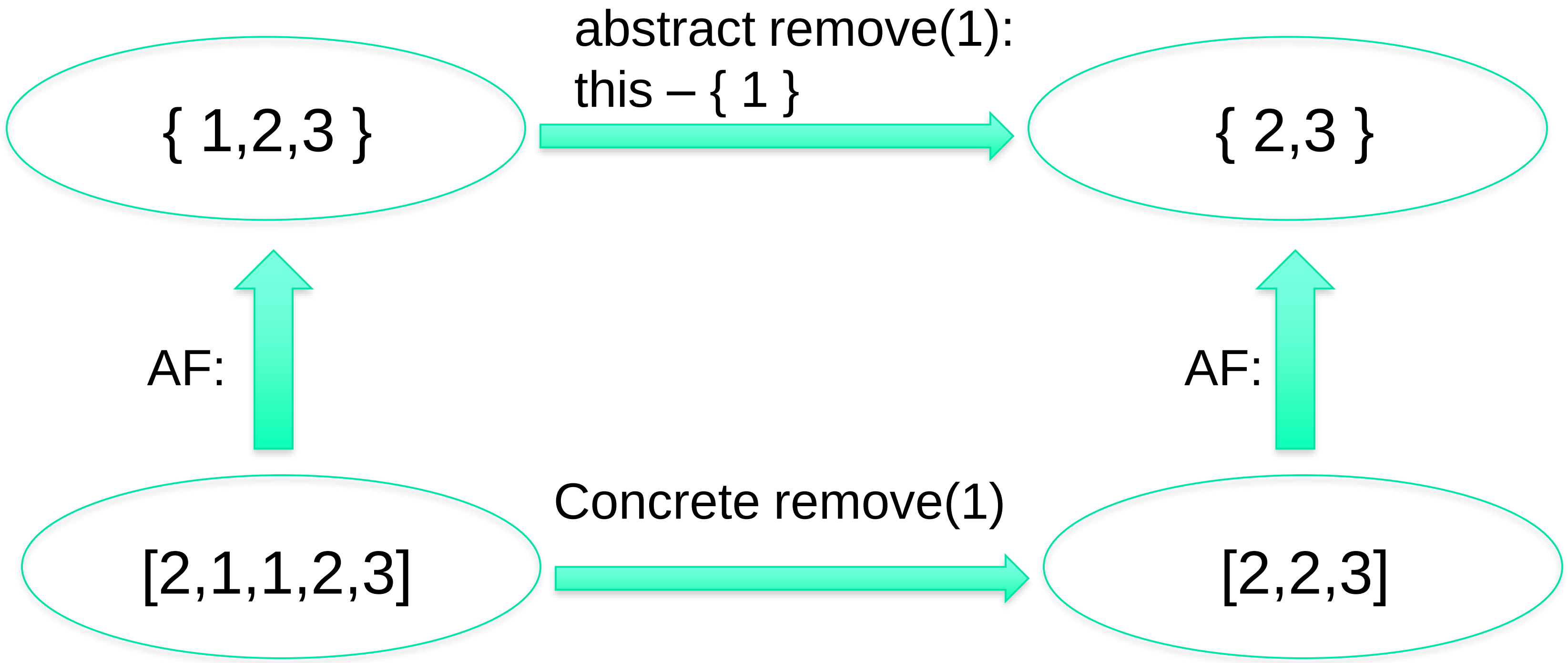
- Abstraction function: Object → abstract value
  - I.e., the object's rep maps to abstract value
    - IntSet e.g.: list [2, 3, 1] → { 1, 2, 3 }
  - Many objects map to the same abstract value
    - IntSet e.g.: [2, 3, 1] → { 1, 2, 3 } and [3, 1, 2] → { 1, 2, 3 } and [1, 2, 3] → { 1, 2, 3 }
- Not a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

Correctness

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



IntSet Example



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

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

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

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;
   }
}
```

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

Exercise: Willy's **`IntStack`**

- What if Willy added this method:

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

- Does the proof still hold?

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

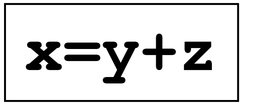
- Partition the input and/or output domains into equivalence classes
  - E.g., spec of sqrt(double x):
    - returns: square root of x if x >= 0
    - throws: IllegalArgumentException if x < 0
- Partition the input domain
  - E.g., test x < 0, test x = 0, test x >= 0
- Partition the output domain too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

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

Control-flow Graph (CFG)

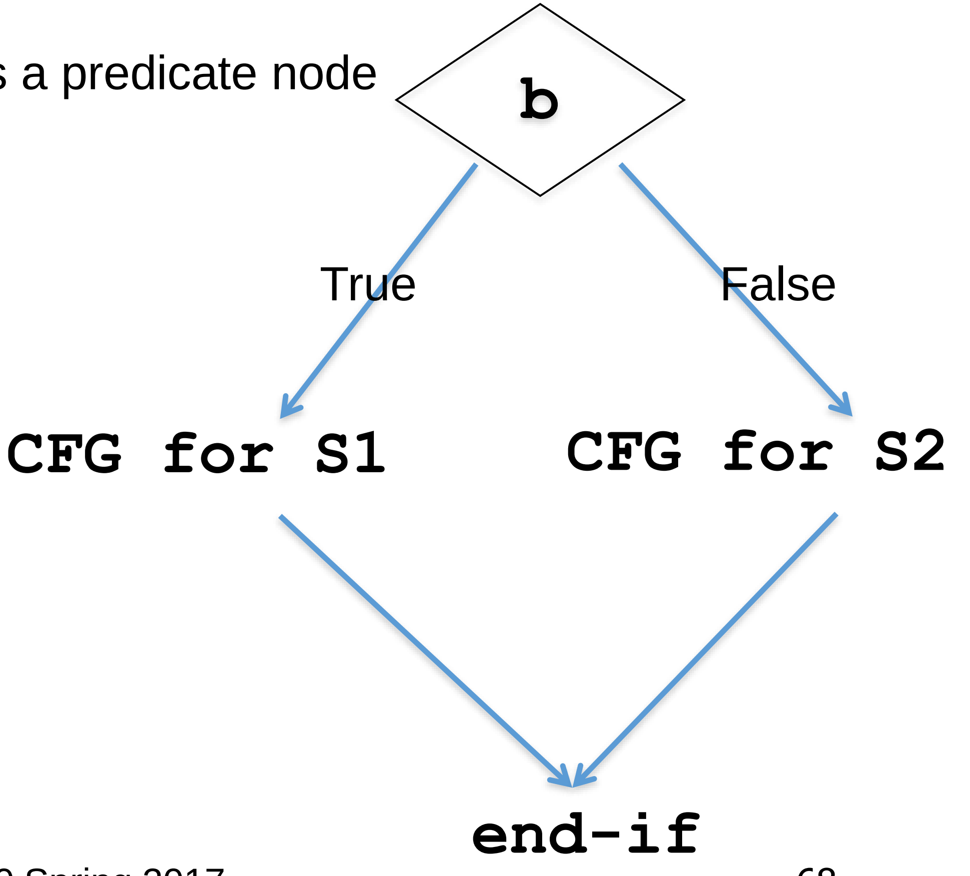
- Assignment **`x=y+z`** => node in CFG:



- If-then-else

`if (b) S1 else S2` =>

`(b)` is a predicate node

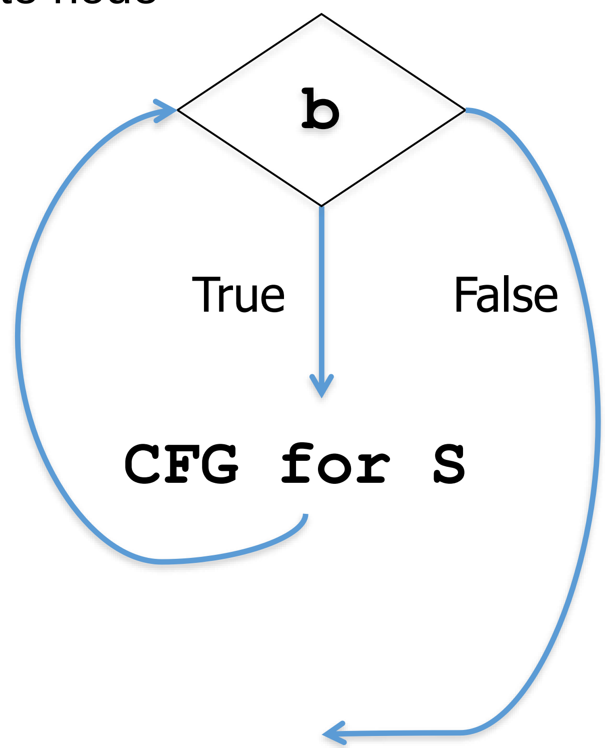


Control-flow Graph (CFG)

- Loop

`while (b) S  =>`

`(b)` is a predicate node



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

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)**?

Topics

- <u>Subtyping vs. subclassing</u>
  - Subtype polymorphism, true subtypes and the LSP, specification strength and function subtyping, Java subtypes (overriding and overloading)

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!

override `A.m`

- 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!

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

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

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

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

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);
 }
}
```

```
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 {
…
}
```

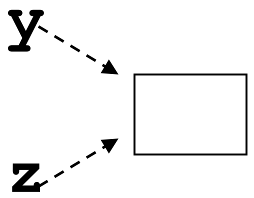
Why not move `void accept(Visitor v)` up into superclass `BooleanExp`?

Topics

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

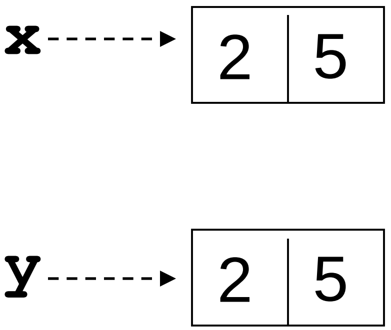
Equality: `==` and `equals()`

- 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



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)

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;
```

`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 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)

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); }
```

How does a Method Call Execute?

- For example, x.foo(5);
- Compile time
  - Determine what class to look in
  - Determine the method signature (method family)
  - Find all methods in the class with the right name
    - Includes *inherited* methods
    - Keep only methods that are accessible
      - E.g. a private method is not accessible to calls from outside the class
    - Keep only methods that are applicable
      - The types of the actual arguments (e.g. 5 has type int above) must be subtypes of the corresponding formal parameter type
    - Select the most specific method
      - m1 is more specific than m2 if each argument of m1 is a subtype of the corresponding argument of m2
    - Keep track of the method's signature (argument types) for run-time

How does a Method Call Execute?

- Run time
  - Determine the run-time type of the receiver
    - x in this case
    - Look at the object in the heap to find out what its run-time type is
  - Locate the method to invoke
    - Starting at the run-time type, look for a method with the right name and argument types that are identical to those in the method found statically
    - If it is found in the run-time type, invoke it.
    - Otherwise, continue the search in the superclass of the run-time type
    - This procedure will always find a method to invoke, due to the checks done during static type checking

Exercise: Remember **Duration**

Two method families.

```
class Object {
   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 choses family equals(Duration d)
```

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 choses equals(Object o)
// At runtime: Duration.equals(Object o)
```

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 choses equals(Object o)
// At runtime: Duration.equals(Object o)
```

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 choses equals(Object o)
// At runtime: Duration.equals(Object o)
```

Exercise

```
class Y extends X { … }

class A {
   X m(Object o) { … }
}
class B extends A {
   X m(Z z) { … }
}
class C extends B {
   Y m(Z z) { … }
}
```

```
A a = new B();
Object o = new Object();
// Which m is called?
X x = a.m(o);
```

```
A a = new C();
Object o = new Z();
// Which m is called?
X x = a.m(o);
```

Exercise

```
class Y extends X { … }
class W extends Z { … }
class A {
   X m(Z z) { … }
}
class B extends A {
   X m(W w) { … }
}
class C extends B {
   Y m(W w) { … }
}
```

```
A a = new B();
W w = new W();
// Which m is called?
X x = a.m(w);
```

```
B b = new C();
W w = new W();
// Which m is called?
X x = b.m(w);
```

Topics

- <u>Design Patterns</u>
  - 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

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
- What problems are the design patterns trying to solve?
- Can you give an example where it would be used?
- Important Patterns
  - Factory
  - Interning
  - Observer
  - Visitor
  - Singleton
  - Wrapper (adapter, decorator, proxy)
  - Composite

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?

Exercises (creational patterns)

- Can interning be applied to mutable types?

- Can a mutable class be a Singleton?

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

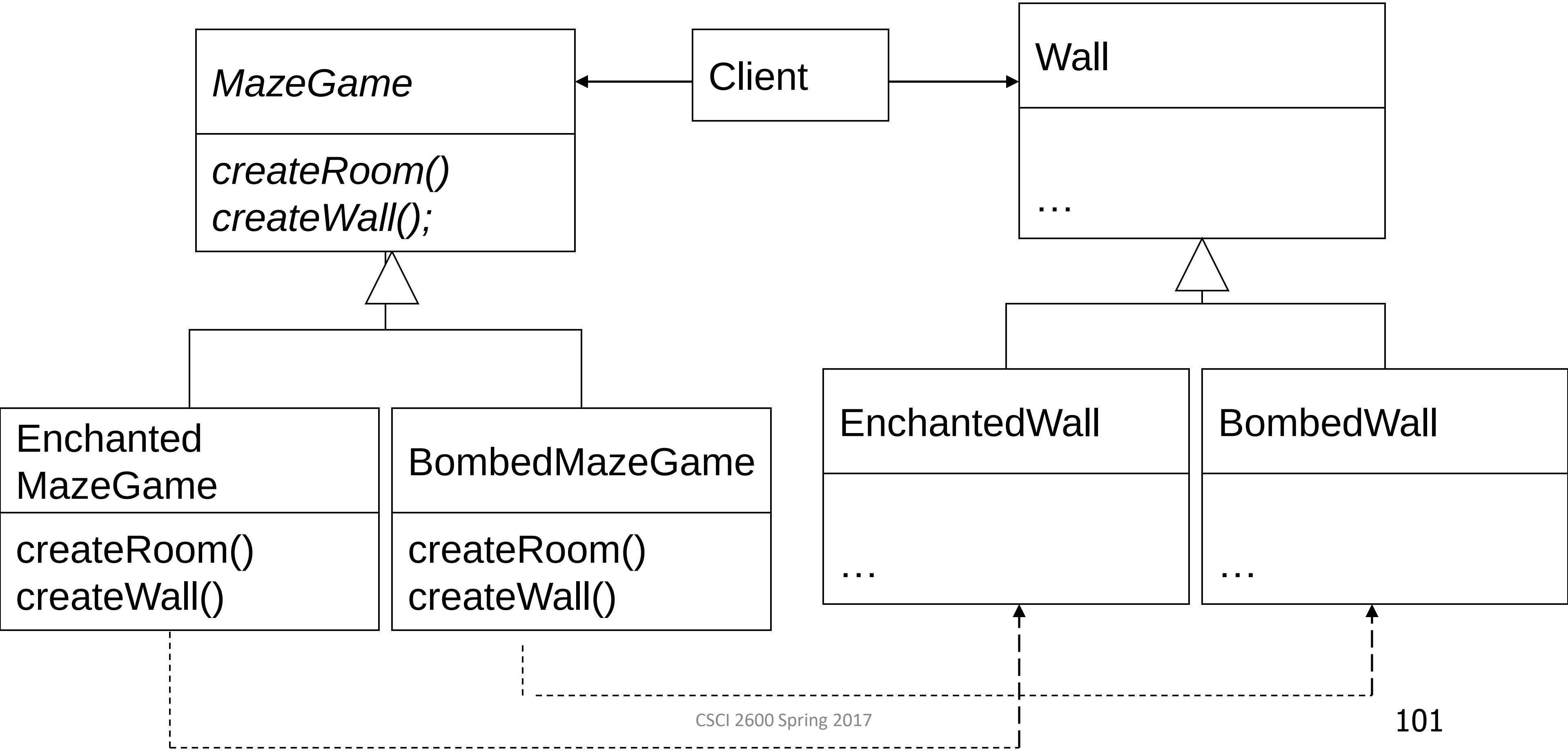
Factory Method

- 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 = craeteRoom(); Room r2 = …
   Wall w1 = createWall(r1,r2); … createDoor(w1); …
 }
}
```

Factory Method Class Diagram

- MazeGame and Products Hierarchies



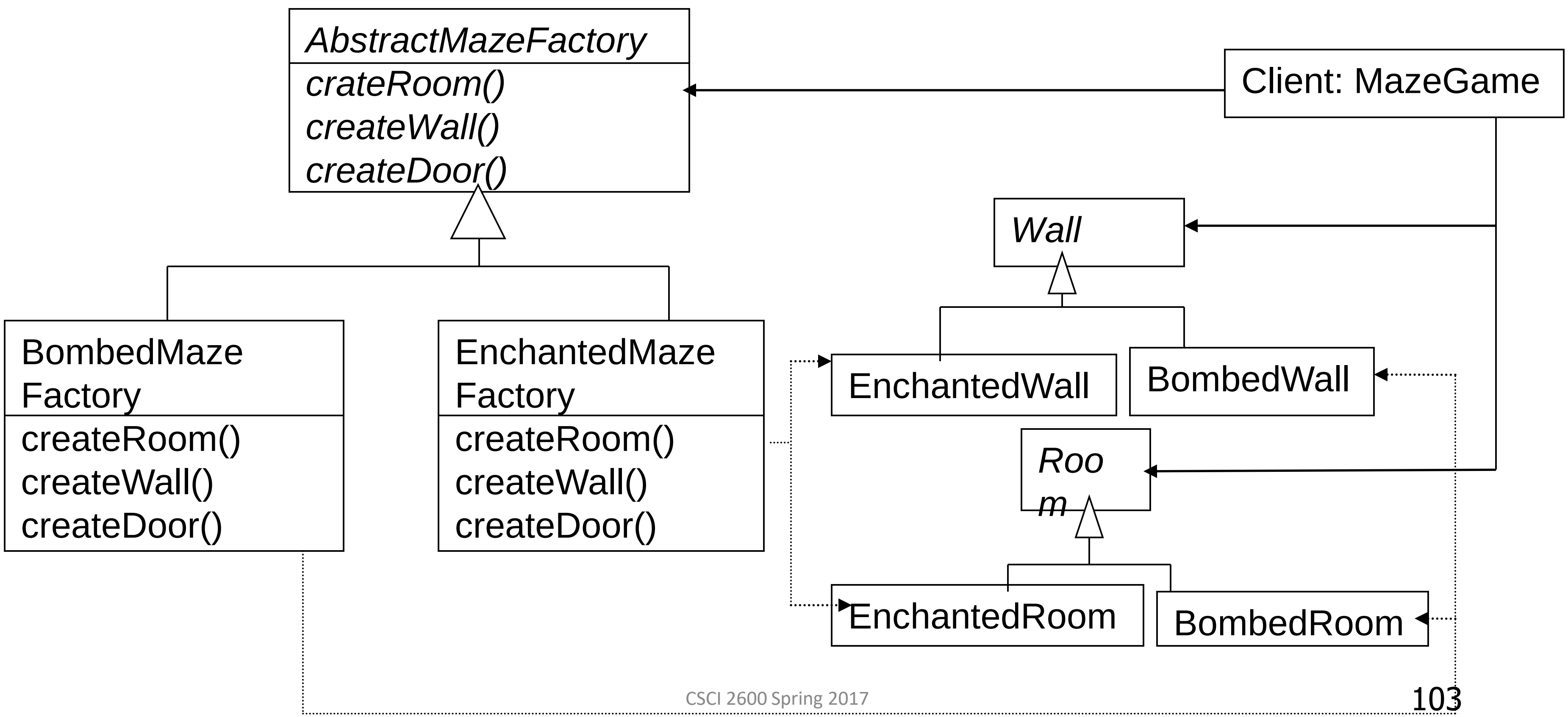
Factory Class/Object

- 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.craeteRoom(); Room r2 = …
   Wall w1 = mfactory.createWall(r1,r2);
   Door d1 = mfactory.createDoor(w1); …
 }
}
```

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

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



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() { … }
}
```

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(); …
 }
}
```

Singleton Pattern

- Guarantees there is a single instance of the class

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

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

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;
   }
}
```

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?

Exercises (structural patterns)

- What pattern helps restrict access to an object?

- What is the difference between an object adapter and a class adapter?
- What pattern hides a large and complex library and promotes low coupling between the library and the client?

Wrappers

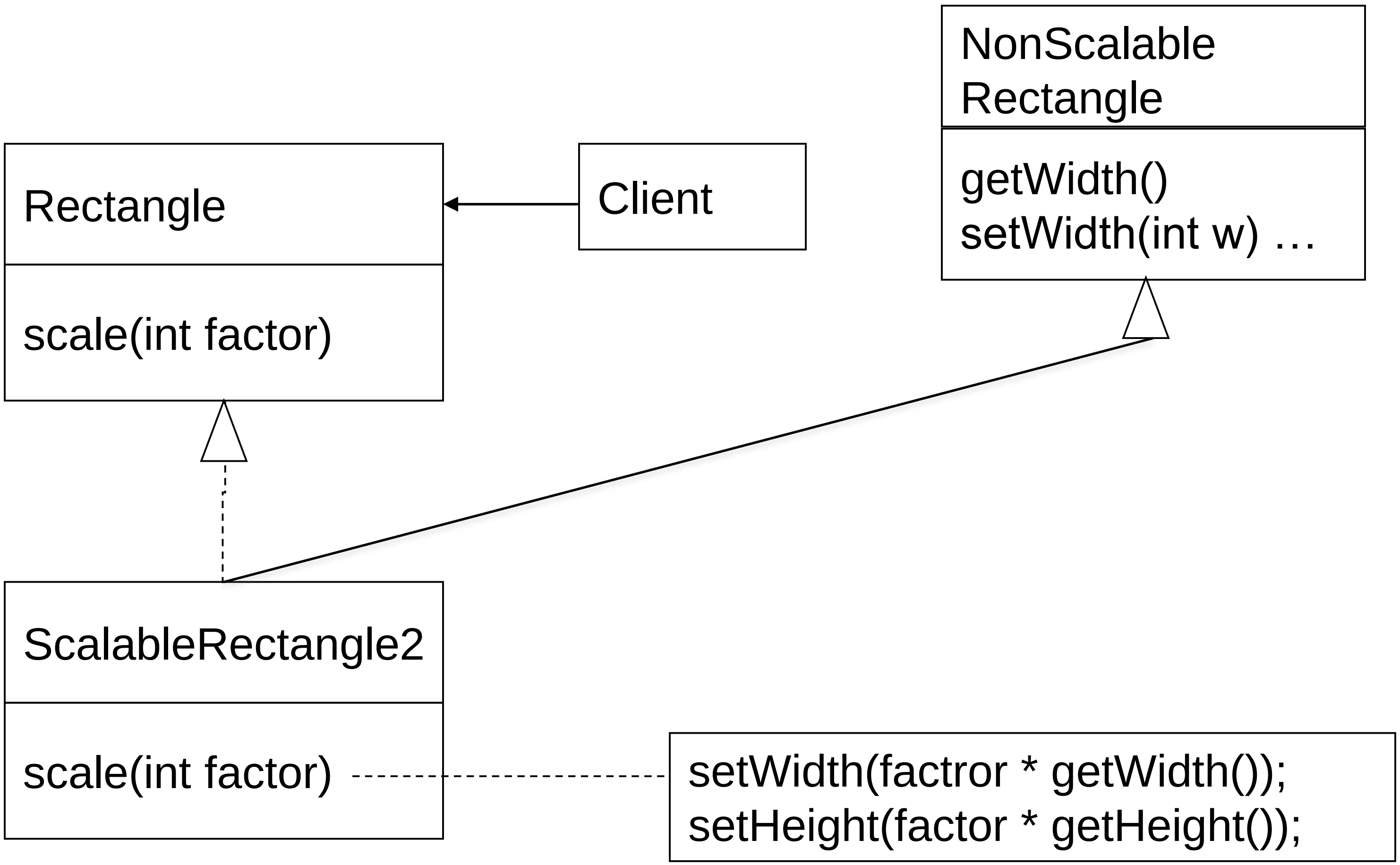
- A wrapper pattern uses composition/delegation
- Wrappers are a thin layer over an encapsulated class
  - Modify the interface
  - Extend behavior
  - Restrict access
- The encapsulated object (delegate) does most work
- Adapter: modifies interface, same functionality
- Decorator: same interface, extends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

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

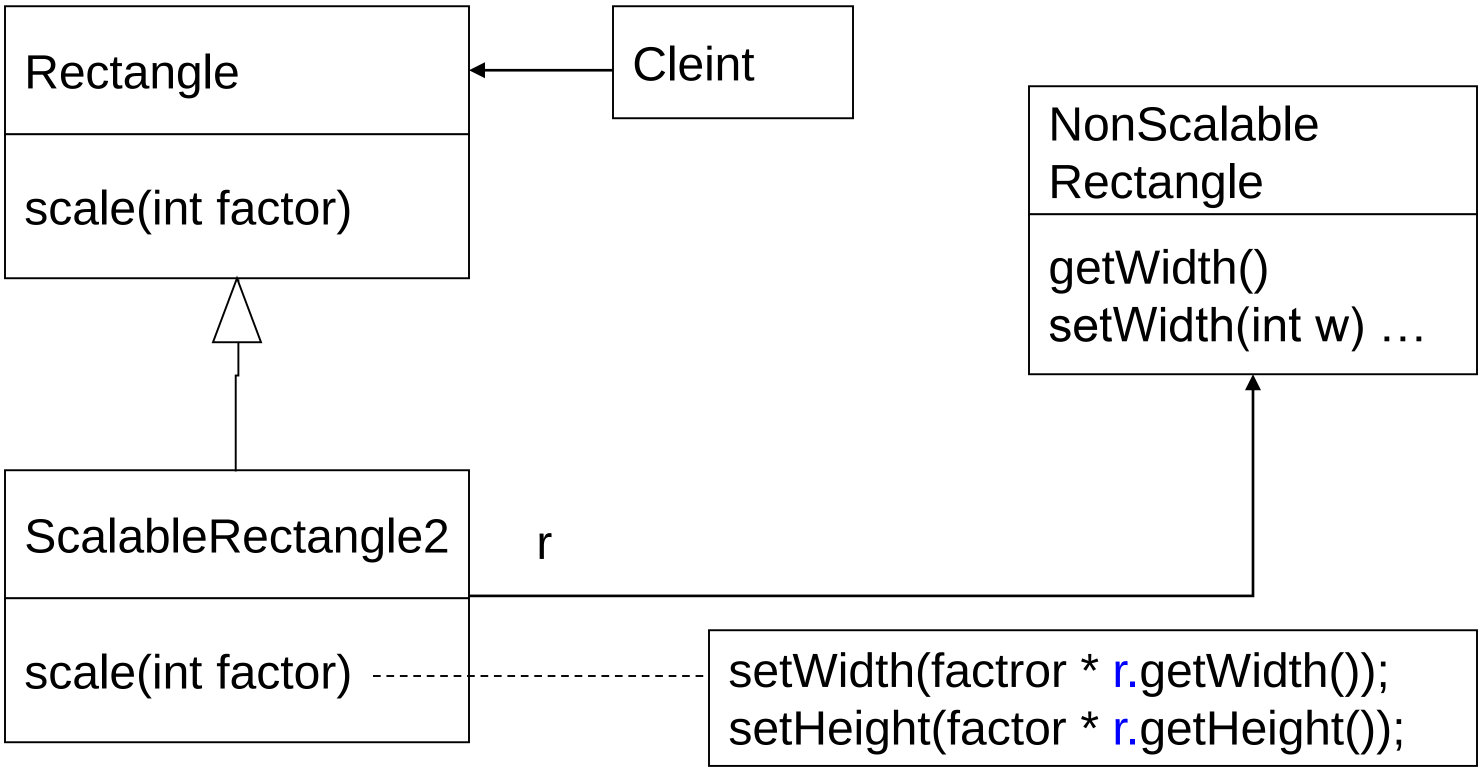
Class Adapter

- Adapts through subclassing



Object Adapter

- Adapts through delegation:



Adapter Example: Scaling Rectangles

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

Class Adapter

- Adapting via subclassing

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

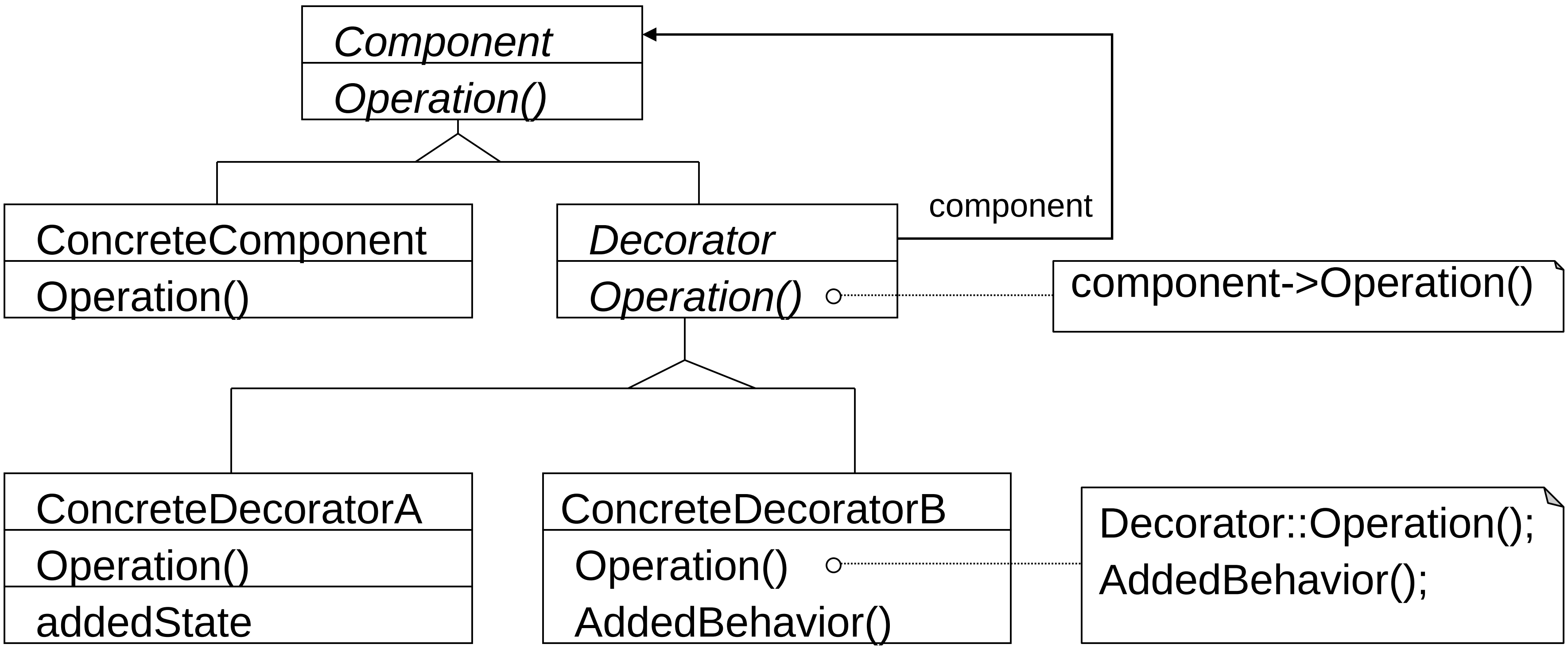
Object Adapter

- 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(); }
   …
}
```

Structure of Decorator

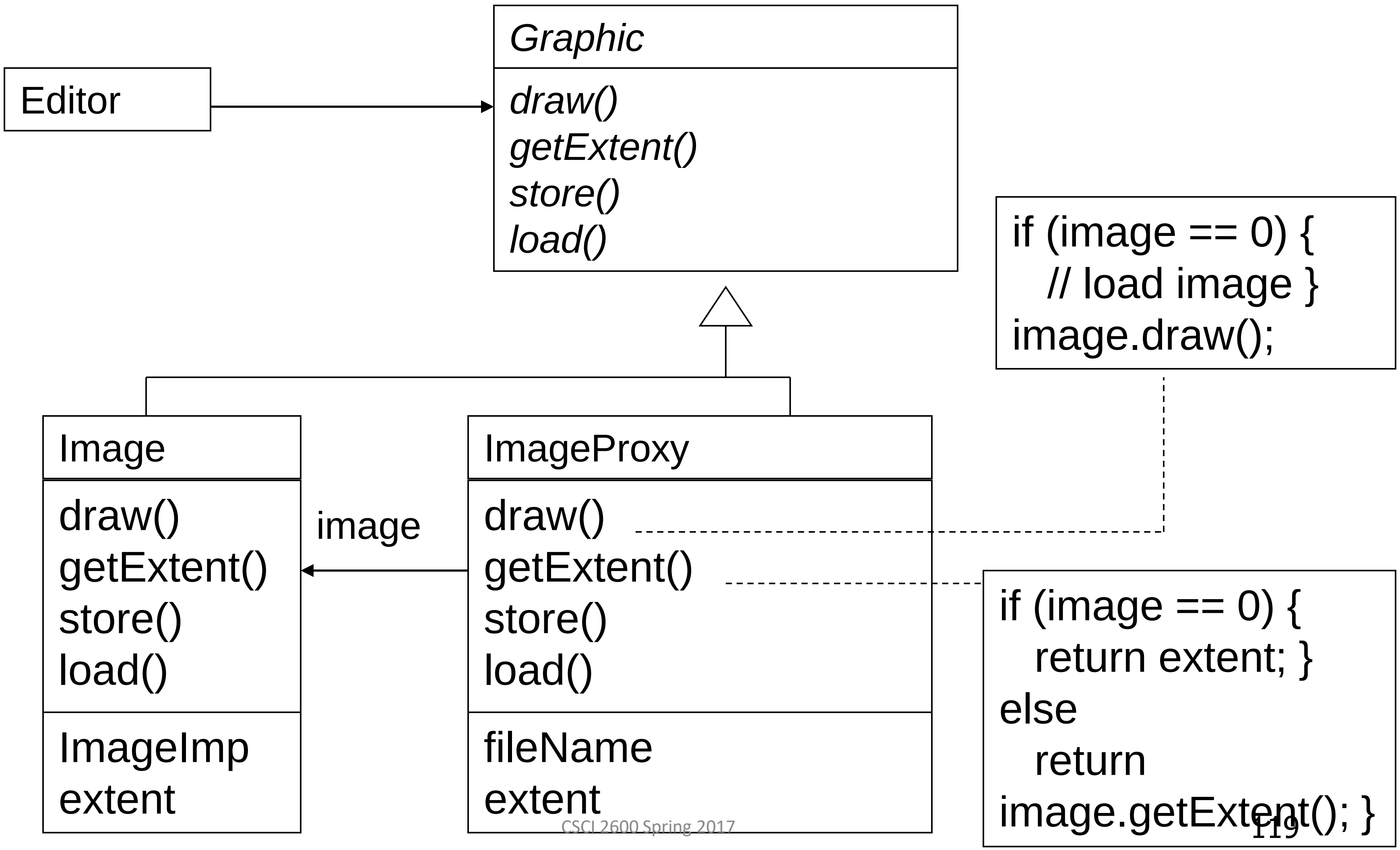
- Motivation: add small chunks of functionality without changing the interface



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

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)

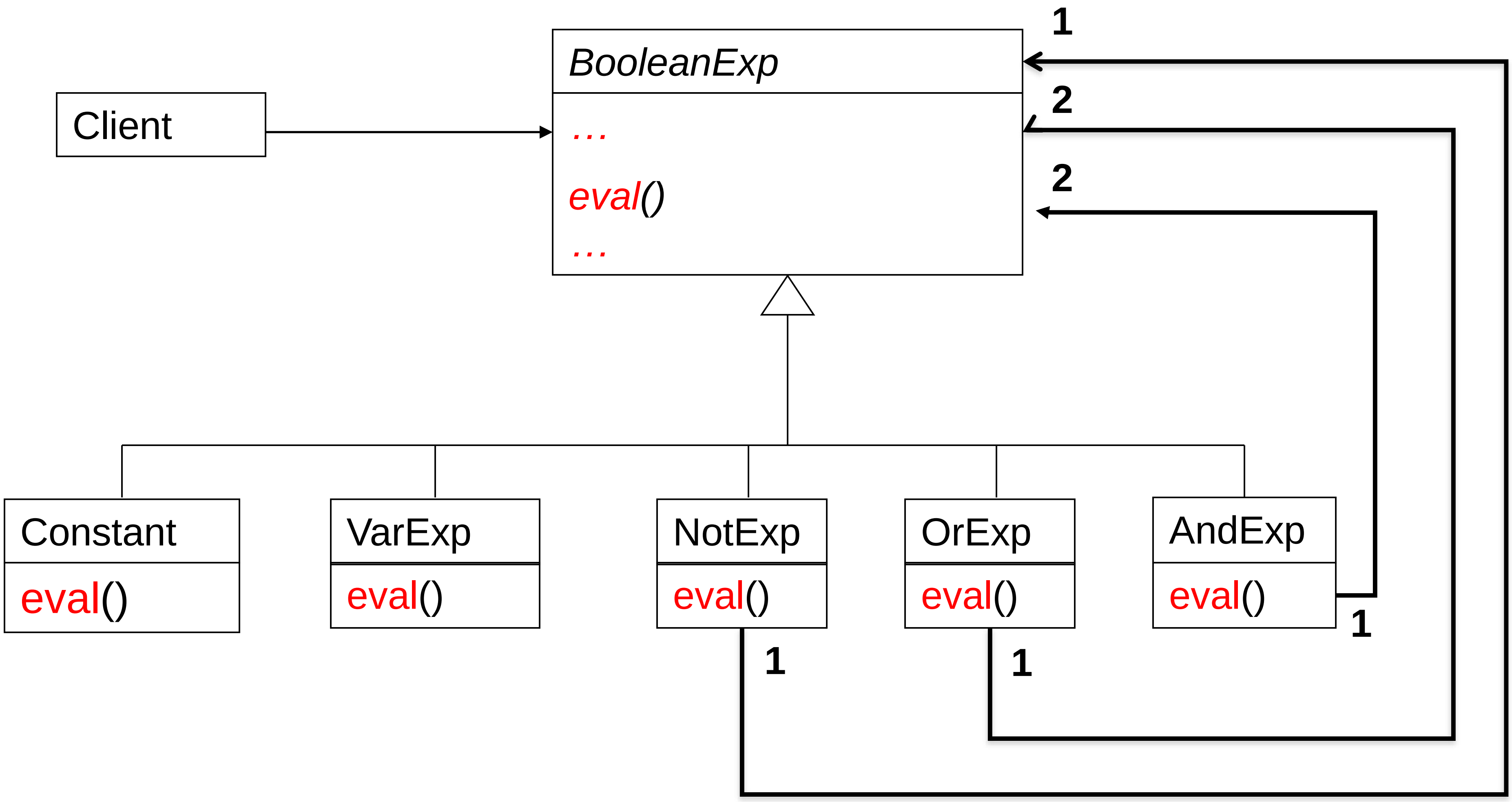
Using Composite to represent boolean expressions

```
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);
  }
}
```

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
```

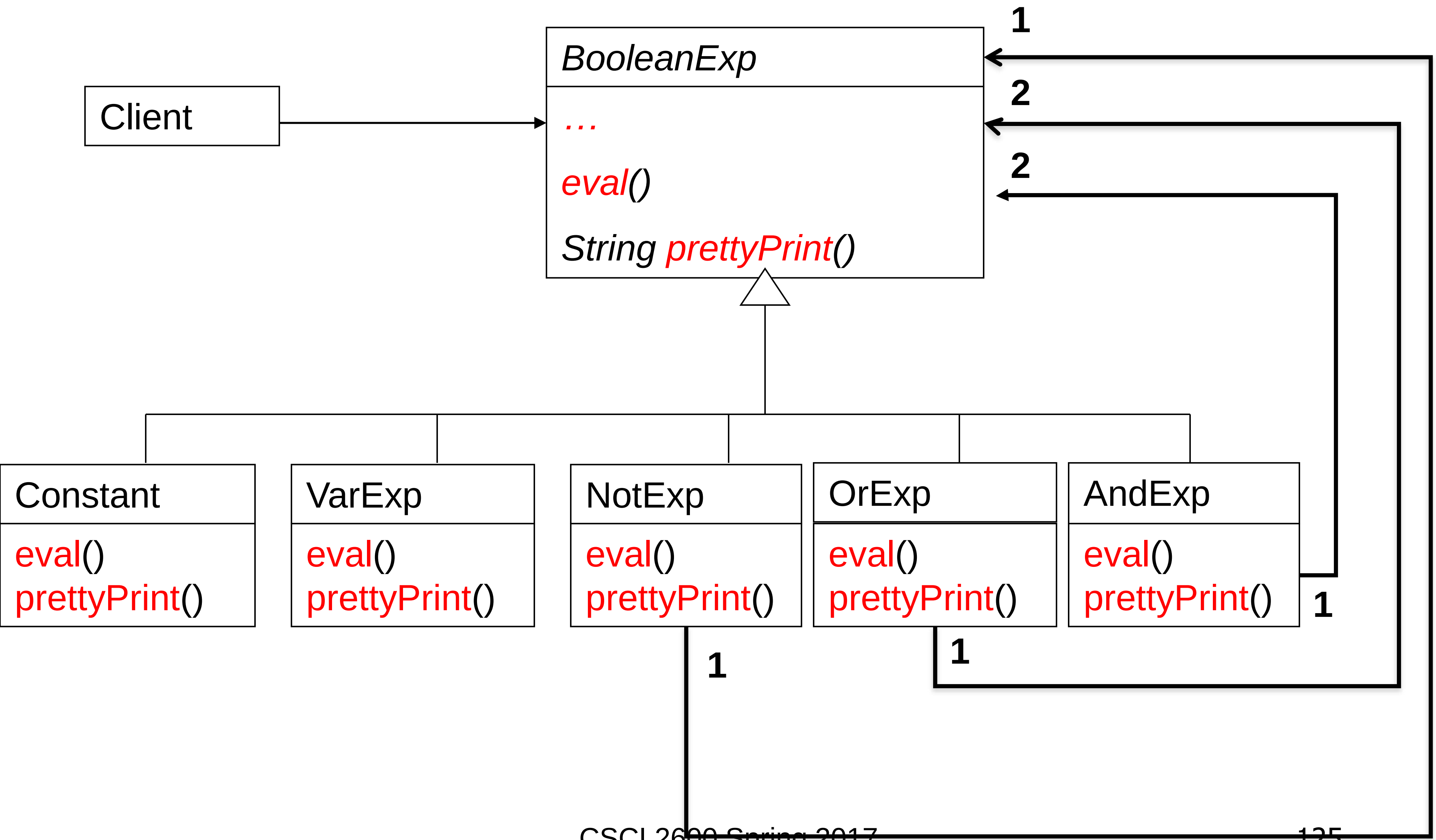
Object Structure vs. Class Diagram



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 patter – 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

Interpreter Pattern

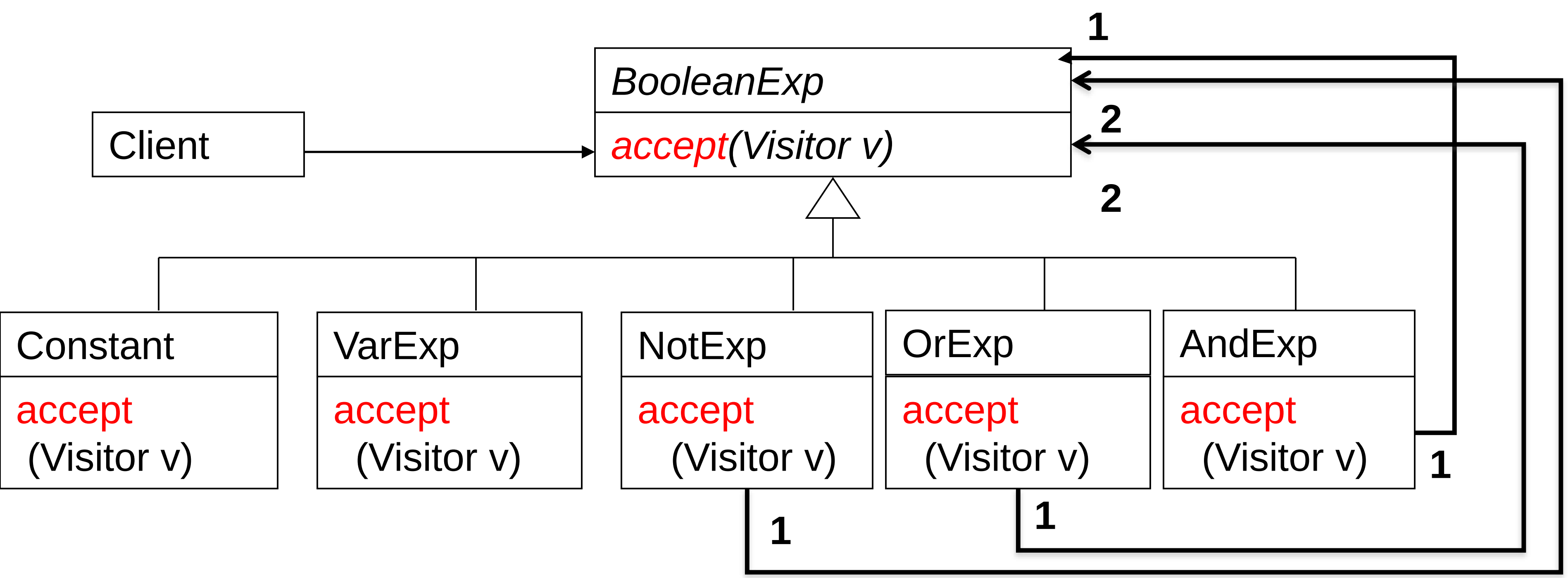


Visitor Pattern

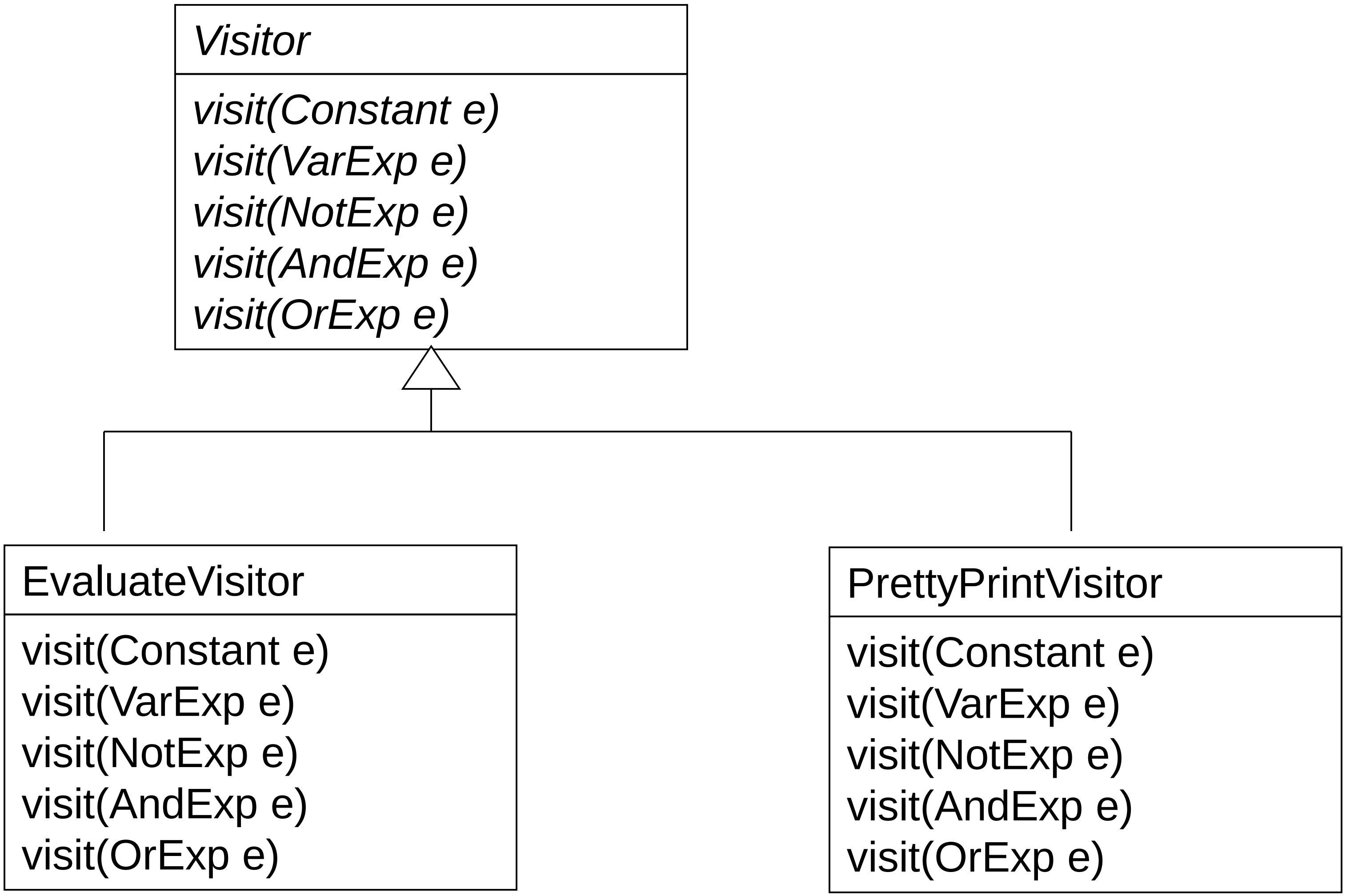
```
class VarExp extends
          BooleanExp {
 void accept(Visitor v) {
  v.visit(this);
 }
}
class AndExp extends
           BooleanExp {
 BooleanExp leftExp;
 BooleanExp rightExp;
 void accept(Visitor v) {
  leftExp.accept(v);
  rightExp.accept(v);
  v.visit(this);
 }
}
```

```
class Evaluate
  implements Visitor {
  // keeps state
void visit(VarExp e)
 {
  //evaluate var exp
 }
 void visit(AndExp e)
 {
  //evaluate And exp
 }
}
class PrettyPrint
  implements Visitor {
…
}
```

The Visitor Pattern



The Visitor Pattern



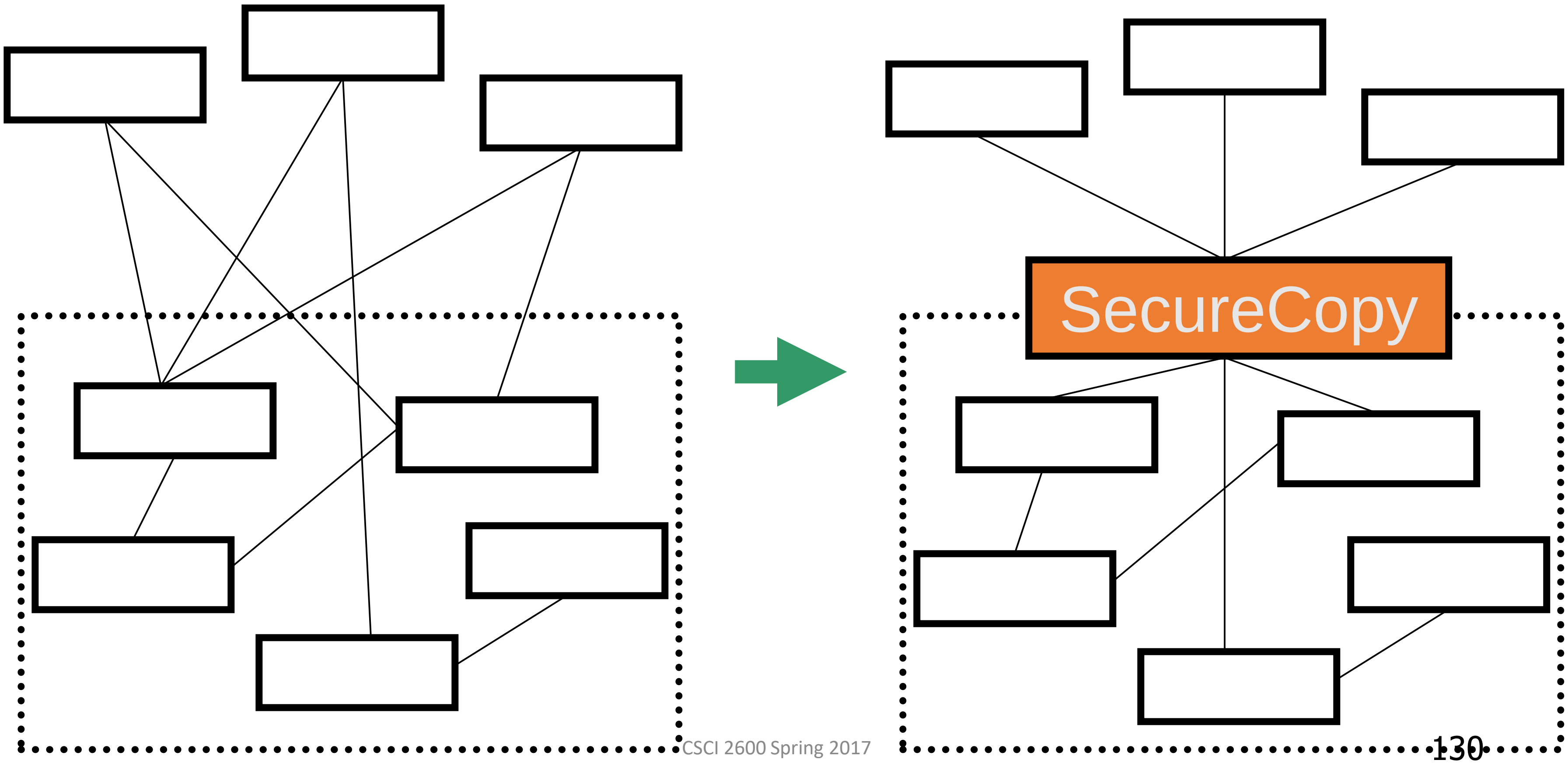
Façade Pattern

- 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?

Façade Pattern

Build a Façade to the library, to hide its (mostly irrelevant) complexity. SecureCopy is the Façade.



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

A Better Design: The Observer

- Data class has minimal interaction with Views
  - Only needs to <u>update</u> Views when it changes

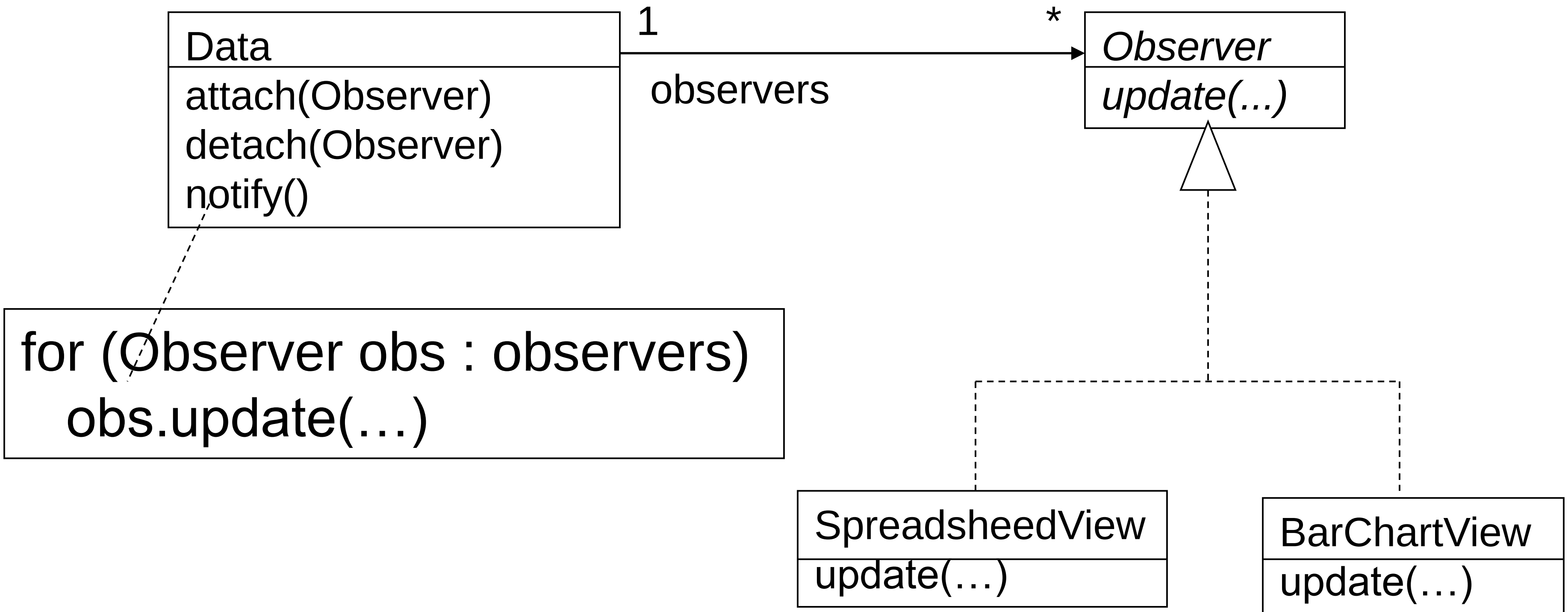
Old, naive design:

```
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(…);
}
```

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

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

Refactoring

- Refactorings attack code smells/antipatterns
- **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…

Refactorings

- Extract Method, Move Method, Remplace 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

Topics

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

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?

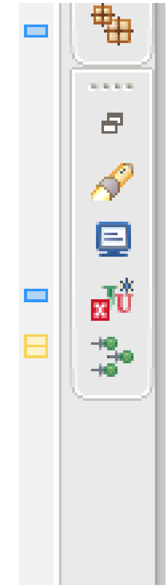
Usability

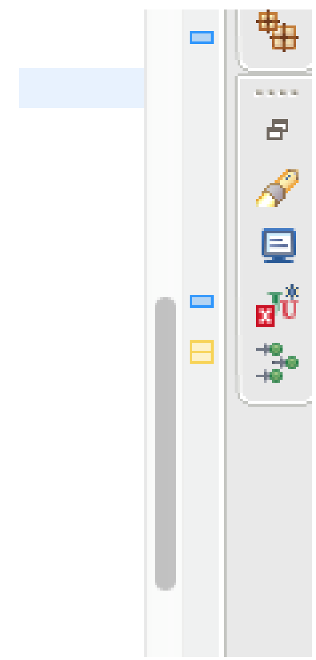
- Design principles for learnability
  - Consistency: internal, external, metaphorical
  - Use simple words, not tech jargon
  - Recognition, not recall
- Design principles for visibility
  - Make system state visible
  - Give prompt feedback

- Simplicity!

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





Topics

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

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?

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

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

Classification of Requirements

- FURPS+ model
- The FURPS:
  - Functionality, Usability, Reliability, Performance, Supportability
- The +:
  - Design constraints, implementation requirements (e.g., must use Java), other

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: usability, reliability, performance, supportability)

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!

Example Use Case

Main Success Scenario

Actors

- 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

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

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