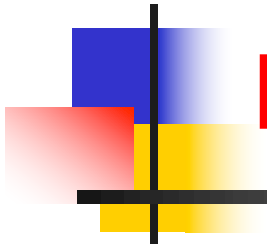
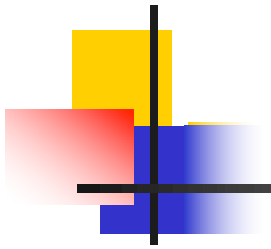


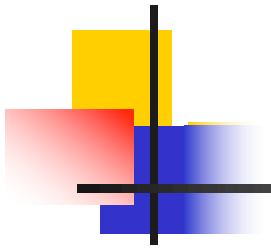
Review

Significant portions of this course material are based on (and taken with permission from) Michael Ernst's course on Software Design and Implementation at the U. of Washington.



Final Exam

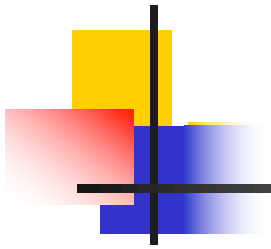
- Final Exam on Thursday May 6th at 6:55pm-9:45pm.
  - Final exam is cumulative. Any topic covered in this course can be on the final exam.
- Honor Code:
  - Open book, open notes, open slides.
  - No use of compilers, no search for answers on the Internet, no communication with others.
  - You must only submit ***your own*** answers.
- Type into Submitty (like Quizzes).



Study

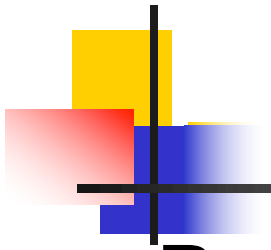
- Review slides
- Review Exams 1 and 2
- Review Quizzes. Solutions are published.
- Work in small groups, discuss problems with your peers, mentors, TAs. Ask questions in the Submitty forum and/or attend office hours to verify your solutions or to get help if you are stuck.

PSoft is about writing correct and maintainable software

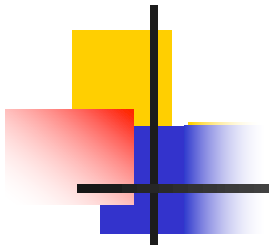


- Specifications
- Polymorphism, abstraction and modularity
- Design patterns
- Refactoring
- Reasoning about code
- Testing
- Tools - Java, Eclipse, Git, Junit, EclEmma

PSoft is about writing correct and maintainable software

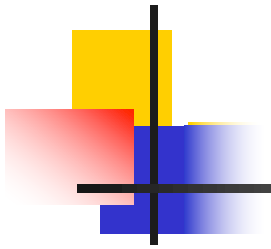


- Building correct software is hard!
  - Lots of dependencies
  - Lots of “moving parts”
- Software engineering is primarily about mitigating and managing complexity
  - Specifications, abstraction, design patterns, refactoring, reasoning about code (invariants “fix” one part, thus fewer “moving parts”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



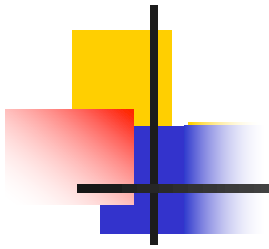
Outline

- Review of topics in chronological order



Topics

- Reasoning about code
- Specifications
- ADTs, rep invariants and abs. functions
- Testing
- Subtyping vs. subclassing
- Parametric polymorphism (Generics)
- Equality
- Design patterns and refactoring



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

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, some of 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., { 2y < y+1 } // Simplify into { y < 1 }

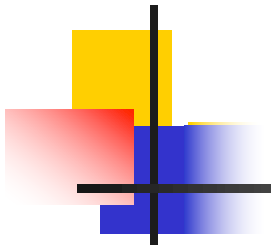
```
z = y + 1  // Substitute y+1 for z in 2y < z
```

{ 2*y < z }

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

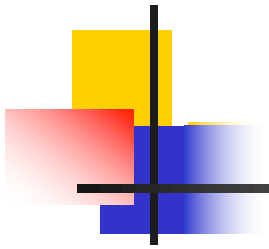
{ x < z }

- More focused and more useful



Condition Strength

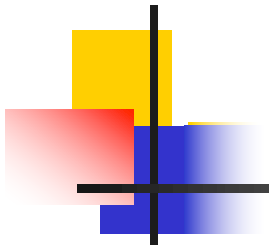
- "P is stronger than Q" means "P implies Q"
  - Notice it is reflexive (since P => P).
- "P is stronger than Q" means "P guarantees no less than Q"
  - E.g., x>0 is stronger than x>-1
- No more values satisfy P than Q
  - E.g., fewer values satisfy x>0 than x>-1
- Stronger means more specific
- Weaker means more general



Exercise. Condition Strength

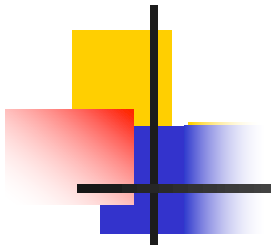
- Which one is stronger?

$x > -10$	or	$x > 0$
$x > 0 \;\&\&\; y = 0$	or	$x > 0 \;\|\|\; y = 0$
$0 \le x \le 10$	or	$5 \le x \le 11$
$y \equiv 2 \pmod 4$	or	y is even
$y \equiv 1 \pmod 3$	or	y is odd
$x = 10$	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`** and it terminates, 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

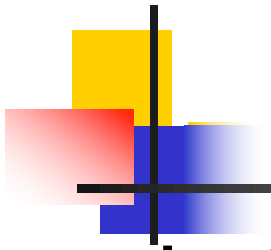
{ x≥0 } **`x++`** { x>1 } is false. Why?

{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

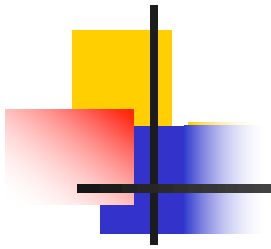
- Let `P => Q => R`

(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. `{ P } code { U }`
4. `{ P } code { S }`

Rules for Backward Reasoning: Assignment



// precondition: ??

```
x = expression
```

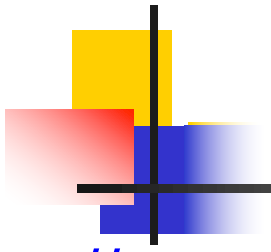
// postcondition: Q

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

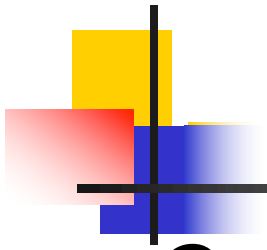
Rules for If-then-else

Forward reasoning

{ P }
if b
{ P ∧ **b** }
S1
{ Q1 }
else
{ P ∧ ¬**b** }
S2
{ Q2 }
{ Q1 ∨ Q2 }

Backward reasoning

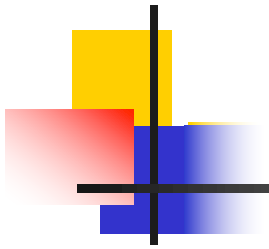
{ (**b**∧wp("**S1**",Q))∨(**b**∧wp("**S2**",Q)) }
if b
{ wp("**S1**",Q) }
S1
{ Q }
else
{ wp("**S2**",Q) }
S2
{ Q }
{ Q }



Exercise

- Compute the weakest precondition:

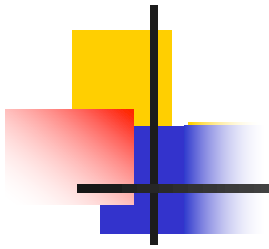
```
if (x < 0) {
    y = -x;
}
else {
    y = x;
}
{ y = |x| }
```



Exercise

- Find the postcondition:

```
{ p² + q² = r }
r = r/p


q = q*q/p
```

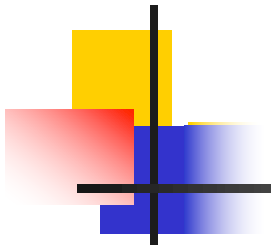


Exercise

- 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
   - Find 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.
   1. D stays in the range of natural numbers, D>=0
   2. Each iteration decrements D
   3. D = 0 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>=0, $D_{before} > D_{after}$. D = 0 implies i = 0 (loop exit condition).

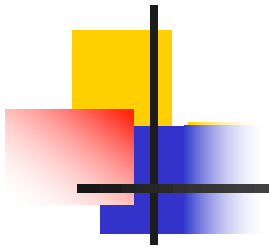
Reasoning About Loops

- Loop invariant Inv must be such that

1) P => Inv // Inv holds before loop. Base case
2) { Inv ∧ `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 ∧ !`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 stays in the natural numbers
3) D = 0 and Inv must imply loop exit condition `!b`



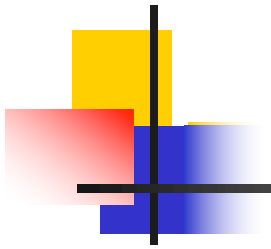
Exercise

Precondition: y >= 0;

```
i = y;
n = 1;
while (i != 0) {
   n = n*x;
   i = i-1;
}
```

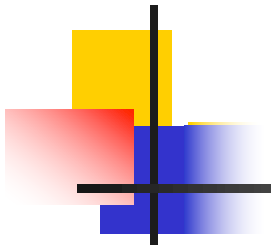
Postcondition: $n = x^y$;

Prove partial correctness and termination



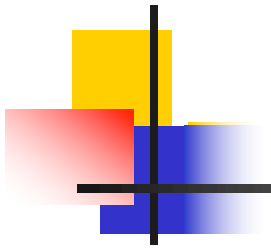
Topics

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



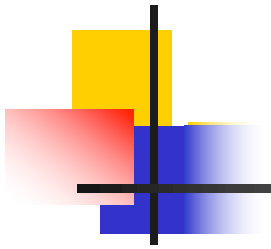
Specifications

- 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!)



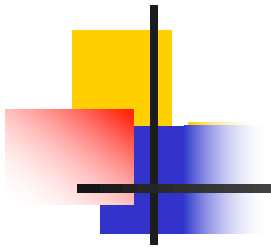
Specifications

- 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



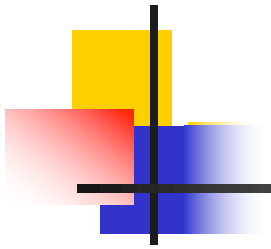
Example Specification

Precondition: `len ≥ 1 && a.length = len`

Postcondition: `result = a[0]+…+a[a.length-1]`

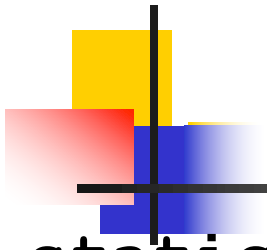
```
int sum(int[] a, int len) {
    int sum = a[0];
    int i = 1;
    while (i < len) {
        sum = sum + a[i];
        i = i+1;
    }
    return sum;
}
```

For our purposes, we will be writing specifications that are a bit less formal than this example. Mathematical rigor is welcome, but not always necessary.



PSoft 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)
```

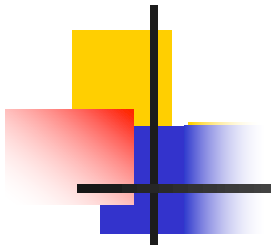
requires: lst1 is non-null and lst2 is non-null

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;
}
```

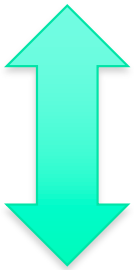


Specification Strength

- "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

Specification Strength and Modularity

Client => Library L1



Library L2

L2 must be stronger than L1

Spec strength, Substitutability and Modularity

Client has contract with **X**:

```
// meets precondition
y = x.foo(0);
// expects non-zero:
z = w/y;
```

Class X

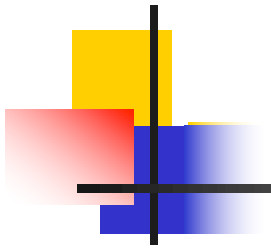
requires: index >= 0
returns: result > 0
`int foo(int index)`

Class Y

requires: index >= 1
returns: result >= 0
`int foo(int index)`

BAD! **Y** surprises the client!
Principle of substitutability tells us that if the specification of **Y.foo is stronger** than the specification of **X.foo**, then it is safe to use **Y.foo** where **X.foo** is expected.

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

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

- Specification B:

  requires: **`a`** is non-null and **`value`** occurs in **`a`** **[P_B]**

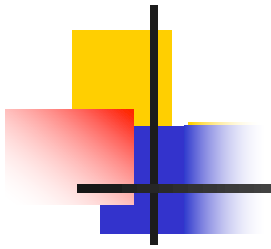
  returns: **`i`** such that **`a[i] = value`** **[Q_B]**

- 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, Q_A => Q_B. Q_B can be rewritten as "if **`value`** occurs in **`a`**" since P_B must hold. (Q_A includes Q_B and one more condition)



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]

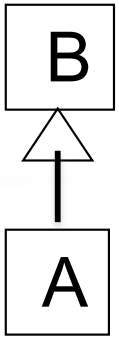
Spec C: requires: true

returns: an int in [2..5]

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

returns: an int in [1..20]

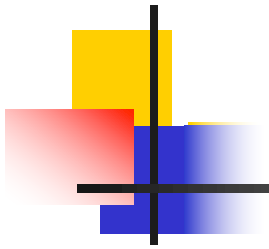
Function Subtyping



- 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` an Integer or a subtype of Integer. Thus, client code will work fine with `A.m(Number p)`, which asks for less: a Number or a subtype of Number
  - Java does not allow change of parameter types in an overriding method.

Function Subtyping

- Results (Outputs):
  - 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 except un-ckecked ones under weaker preconditions. Existing exceptions can be replaced by subtypes
  - Java does allow a subtype return type in an overriding method!



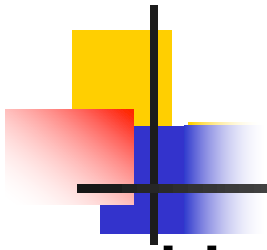
Exercise

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

Let **Z** be subtype of **Y**, **Y** be 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

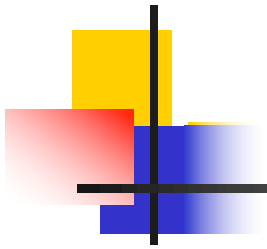


- Use **`<? extends T>`** when you *get* (read) values from a *producer (? is return)*
- Use **`<? super T>`** when you *add* (write) values into a *consumer (? is parameter)*
- E.g.:

```
<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*

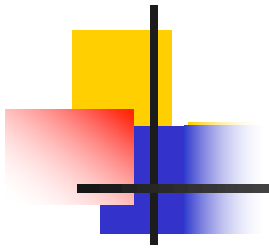
Using Wildcards



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?
   }
}
```

Legal Operations on Wildcards

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

List<? extends Integer> lei;
```

First, which of these is legal?

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

Which of these is legal?

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

Legal Operations on Wildcards

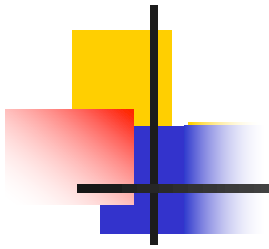
```
Object o;
Number n;
Integer i;
PositiveInteger p;

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>();
```

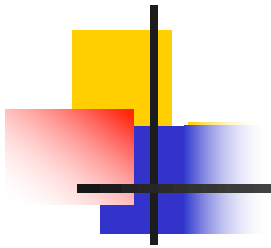
Which of these is legal?

```
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);
```



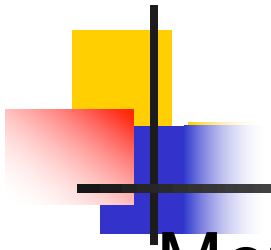
Topics

- 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



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



An ADT Is a Set of Operations

- Methods 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

`class TypeName`

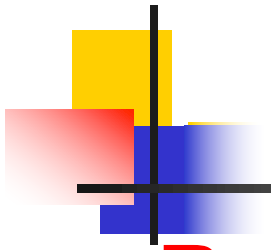
1. overview
2. abstract fields
3. creators
4. observers
5. producers
6. ~~mutators~~

mutable

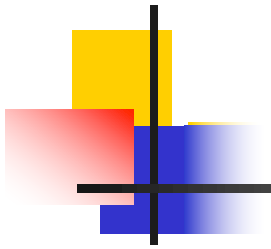
`class TypeName`

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

Connecting Implementation to Specification



- Representation invariant: Object → 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

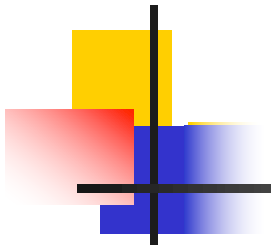


Representation Exposure

- 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

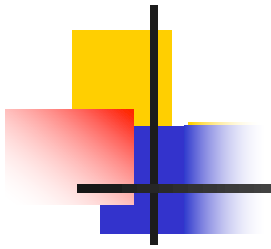


Checking Rep Invariant

- 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");
}
```

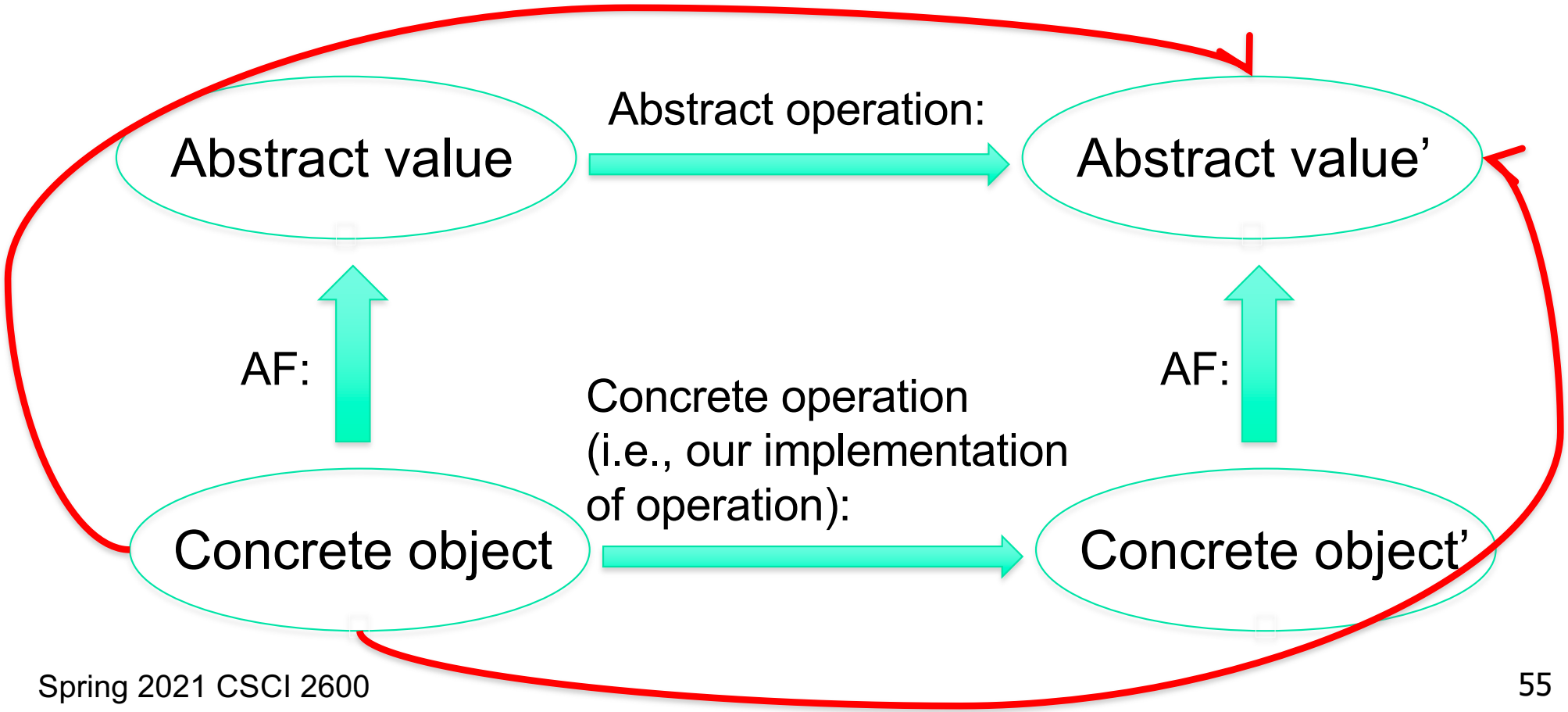
Abstraction Function: mapping rep to abstract value



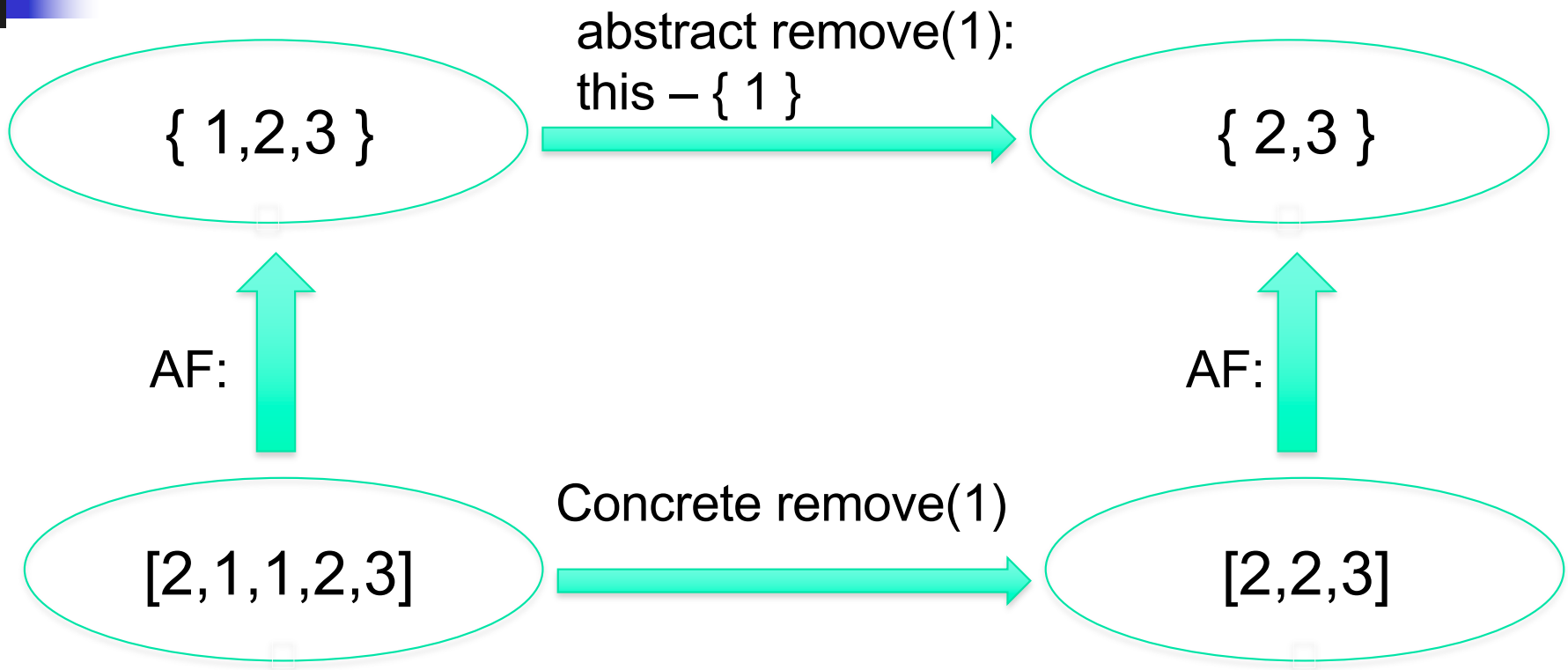
- Abstraction function: Object → abstract value
  - I.e., the object's rep maps to abstract value
    - IntSet e.g.: list [2, 3, 1] → { 1, 2, 3 }
  - Many objects map to the same abstract value
    - IntSet e.g.: [2, 3, 1] → { 1, 2, 3 } and
      [3, 1, 2] → { 1, 2, 3 } and [1, 2, 3] → { 1, 2, 3 }
- 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



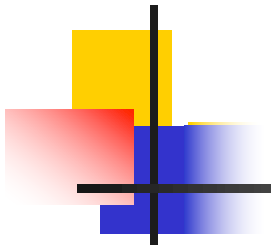
IntSet Example



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

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

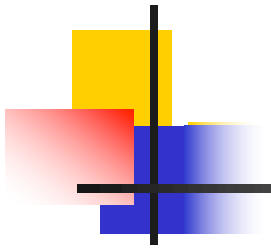
Proving rep invariants by induction



- Proving facts about infinitely many objects
- Basis step
  - Prove rep invariant holds on exit of constructor, producers
- 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
- Our proofs are informal

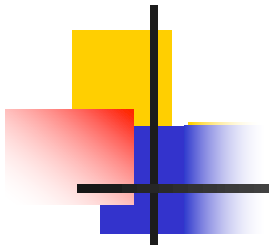
Exercise: Willy's `IntStack` Prove rep invariant holds

```
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

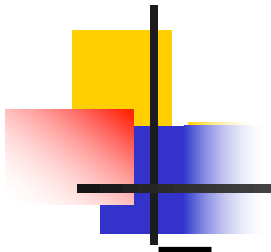


Exercise: Willy's `IntStack`

- What if Willy added this method:

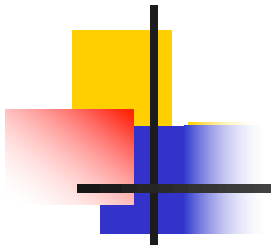
```
public IntMap getMap() {
    return theRep;
}
```

- 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

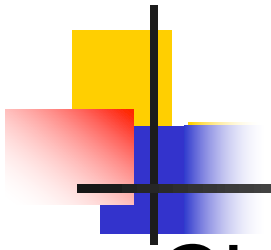


Equivalence Partitioning

- 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

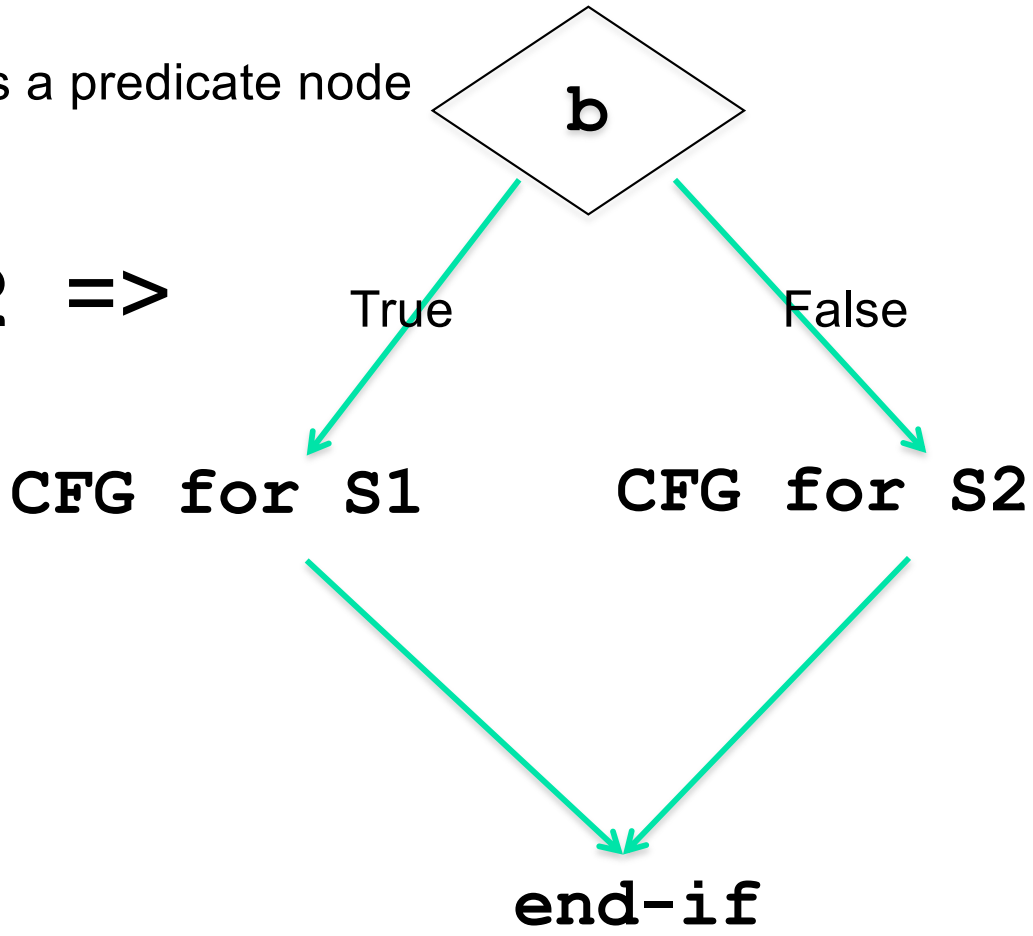
- Choose test inputs at the edges of the input equivalence classes
  - Sqrt example: test with 0,
- Choose test inputs that produce outputs at the edges of output equivalence classes
- Other boundary cases
  - Arithmetic: zero, overflow
  - Objects: null, circular list, aliasing

Control-flow Graph (CFG)

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

- If-then-else

`if (b) S1 else S2 =>`

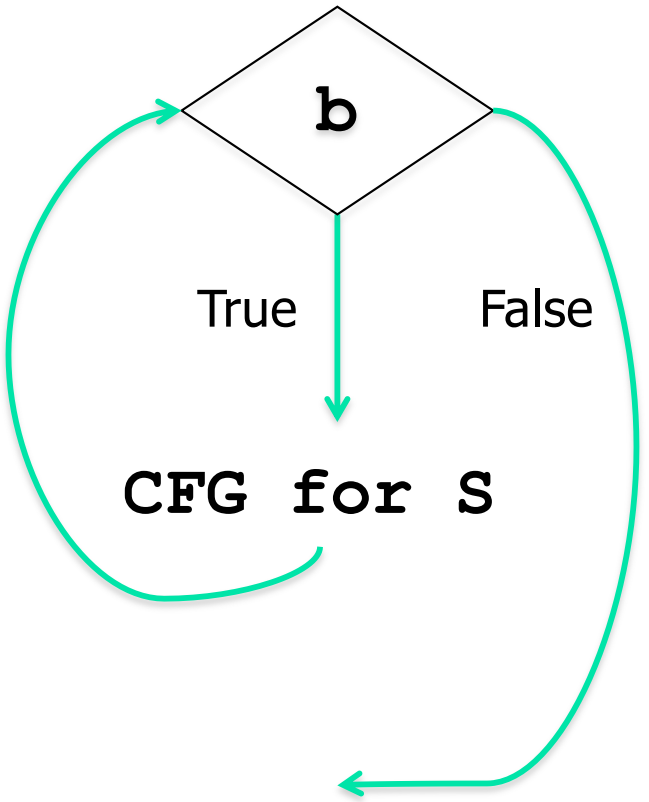


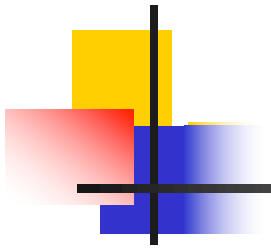
Control-flow Graph (CFG)

- Loop

```
while (b) S   =>
```

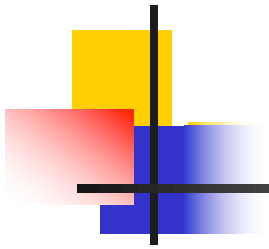
`(b)` is a predicate node





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



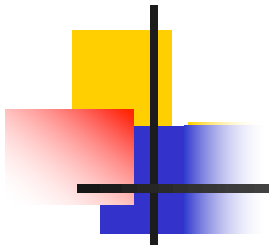
Exercise

- 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)**?

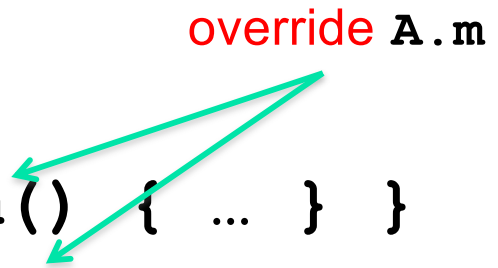


Topics

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

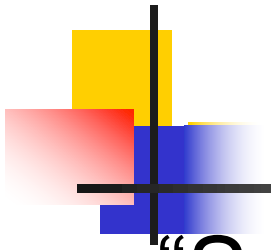
Subtype Polymorphism

- 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!

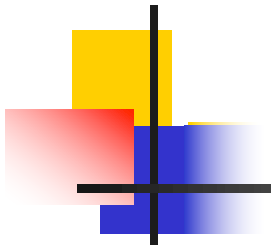


- Subtype polymorphism is a language feature --- essential object-oriented language feature
  - Java subtype: B extends A or B implements I
  - A Java subtype is not necessarily a true subtype!

Benefits of Subtype Polymorphism

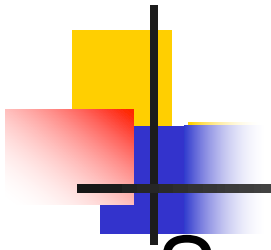


- "Science" of software design teaches Design Patterns
- Design patterns promote design for extensibility and reuse
- Nearly all design patterns make use of subtype polymorphism!



What is True Subtyping?

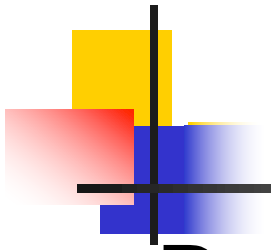
- Also called behavioral subtyping
  - A true subtype is not only a Java subtype but a "behavioral subtype"
- B is subtype of A means every B is an A
- B shall "behave" as an A
  - B shall require no more than A
  - B shall promise at least as much as A
  - In other words, B will do fine where an A is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



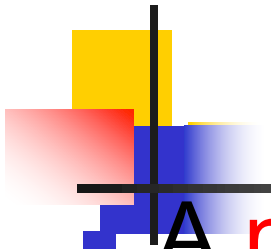
Subtypes are Substitutable

- 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

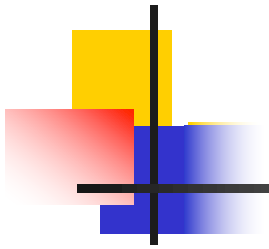
Exercise

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);
 }
}
```

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

```
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 {
...
}
```



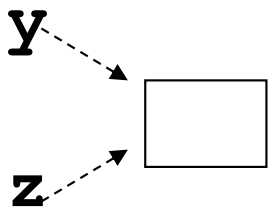
Topics

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

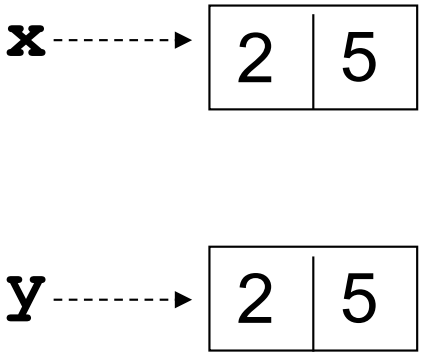
Equality: == and `equals()`

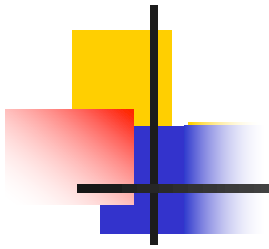
- In Java, `==` tests for reference equality. This is the strongest form of equality

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



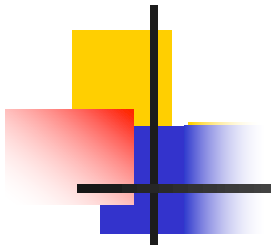
- In our `Point` example, we want `x` to be "equal" to `y` because the `x` and `y` objects hold the same value
  - Need to override Object.equals





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)

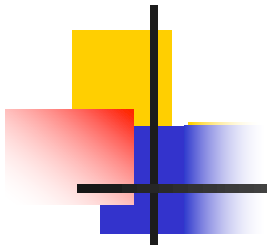


Equality and Inheritance

- 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;
```

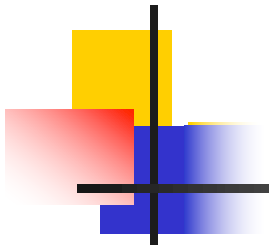


`equals` and `hashCode`

- **`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

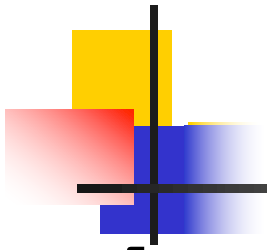
- Consistent with **`equals()`** method: **`a.equals(b) => a.hashCode() == b.hashCode()`**



Equality and Mutation

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

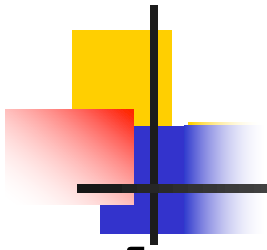
```
Set<Date> s = new HashSet<Date>();
Date d1 = new Date(0);
Date d2 = new Date(1);
s.add(d1);
s.add(d2);
d2.setTime(0); // mutation after d2 already in the Set!
for (Date d : s) { System.out.println(d); }
```



Exercise: Remember `Duration`

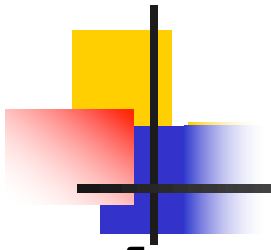
Two method families.

```
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)
```



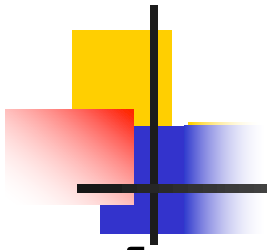
Exercise: Remember `Duration`

```
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)
```



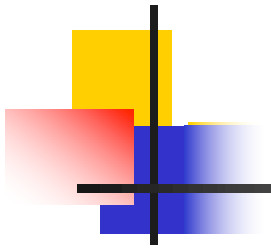
Exercise: Remember `Duration`

```
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)
```

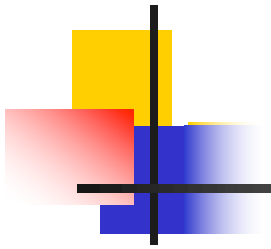


Exercise

```
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);
```

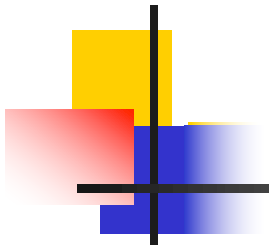


Exercise

```
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) { … }
}
```

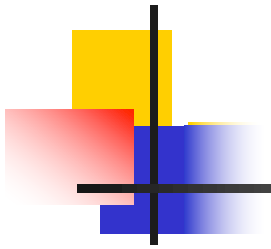
```
A a = new B();
W w = new W();
// Which m is called?
X x = a.m(w);
```

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



Topics

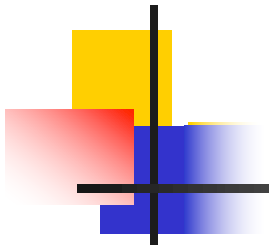
- Design Patterns
  - Creational patterns: Factory method, Factory class, Prototype, Singleton, Interning
  - Structural patterns:
    - Wrappers: Adapter, Decorator, Proxy
    - Composite
    - Façade
  - Behavioral patterns:
    - Interpreter, Procedural, Visitor
    - Observer
    - State, Strategy,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



Design Patterns

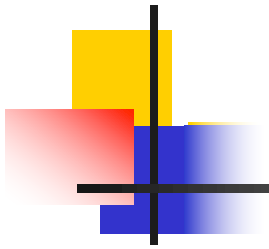
- A design pattern is a solution to a design problem that occurs over and over again
- Design patterns promote extensibility and reuse
  - Open/Closed Principle:

  Help build software that is open to extension but closed to modification
- Majority of design patterns make use of subtype polymorphism



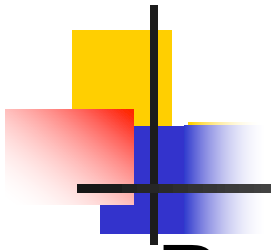
Exercises (creational patterns)

- 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

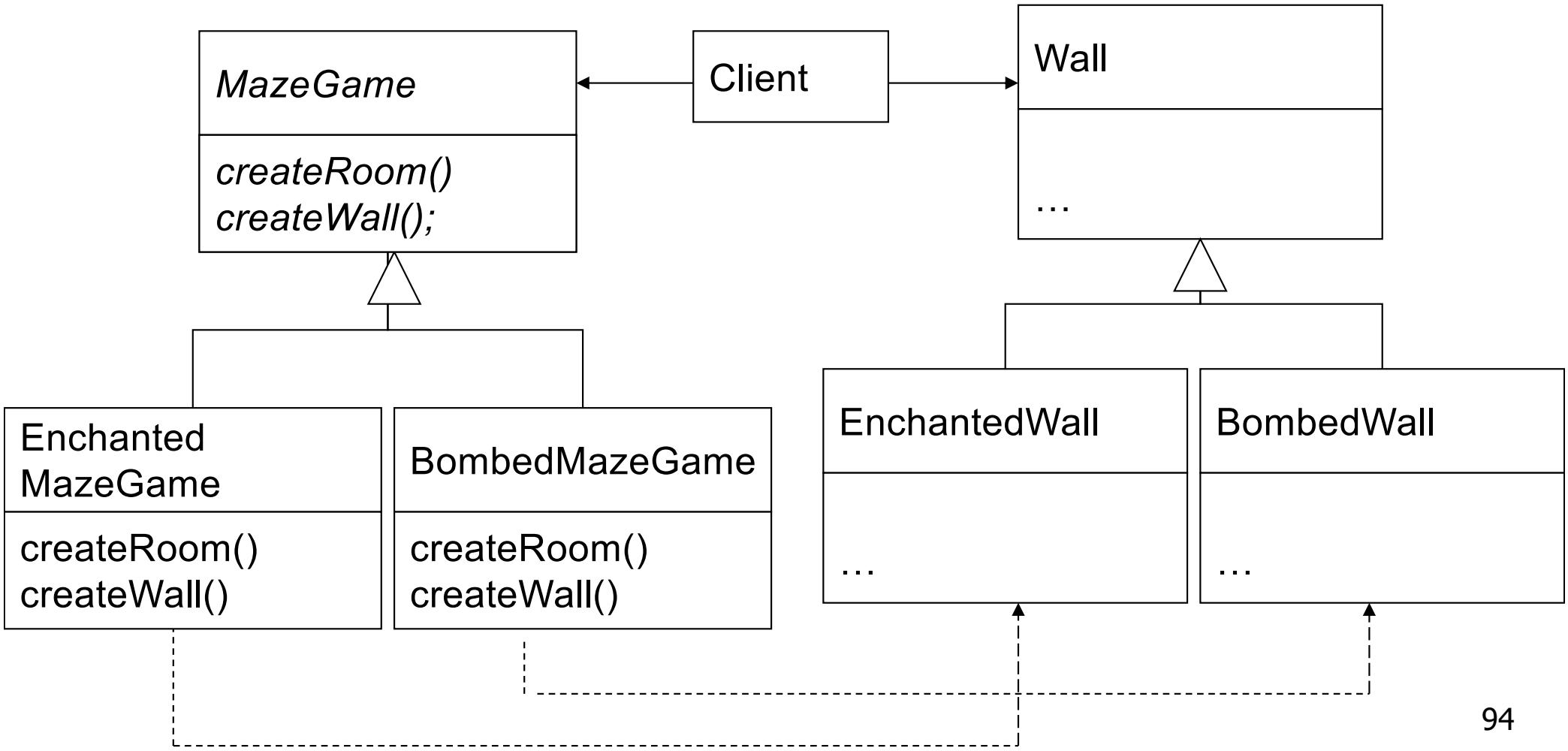
Factory Method

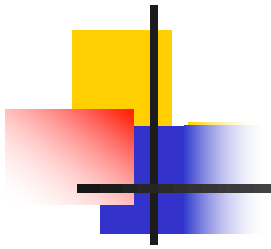
- MazeGames are created the same way. Each MazeGame (Enchanted, Bomed) works with its own Room, Wall and Door products
- Factory method allows each MazeGame to create its own products (MazeGame defers creation)

```
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





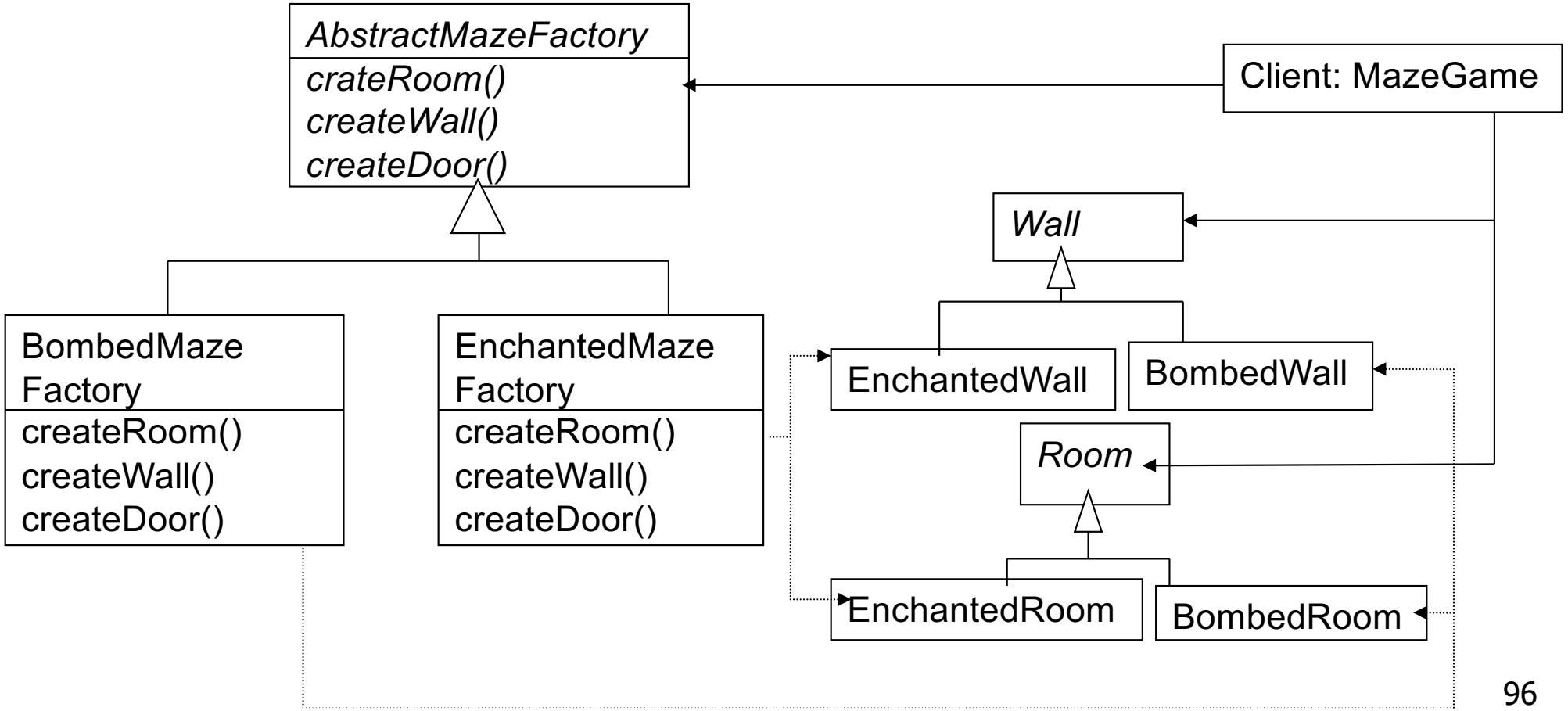
Factory Class/Object

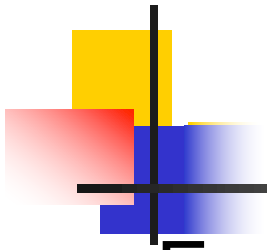
- Encapsulate factory methods in a factory object
- MazeGame gives control of creation to factory object

```
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

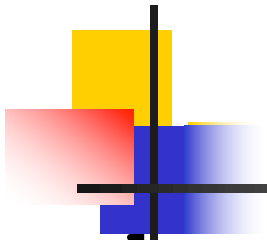




The Prototype Pattern

- Every object itself is a factory
- Each class contains a **`clone`** method and returns a copy of the receiver object
- (Be careful when using **`clone`**. Could be better off using a factory method.)

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



Using Prototypes

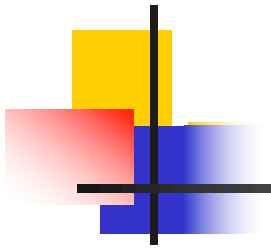
```
class MazeGame {
 Room rproto;
 Wall wproto;
 Door dproto
 MazeGame(Room r, Wall w, Door d) {
    rproto = r; wproto = w; dproto = d;
 }
 Maze createMaze() {
    …
    Room r1 = rproto.clone(); Room r2 = …
    Wall w1 = wproto.clone();
    Door d1 = dproto.clone(); …
 }
}
```

Singleton Pattern

- Guarantees there is a single instance of the class. Most popular implementation:

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

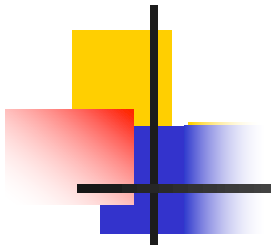
Factory method --- it produces the instance of the class. Private constructor.



Interning Pattern

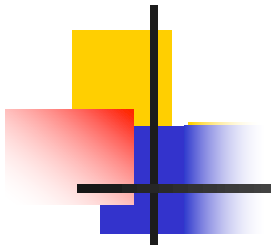
- Reuse existing objects with same value
  - To save space, to improve performance
- Permitted for immutable types only
- Maintain a collection of all names. If an object already exists return that object:

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



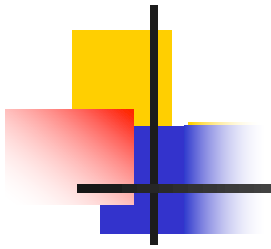
Exercises (structural patterns)

- 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?



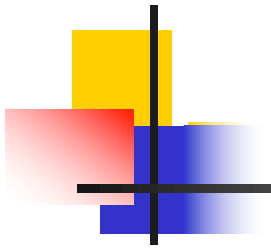
Exercises (structural patterns)

- What pattern helps restrict access to an object?
- What is the difference between an object adapter and a class adapter? Which one is more efficient?
- 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

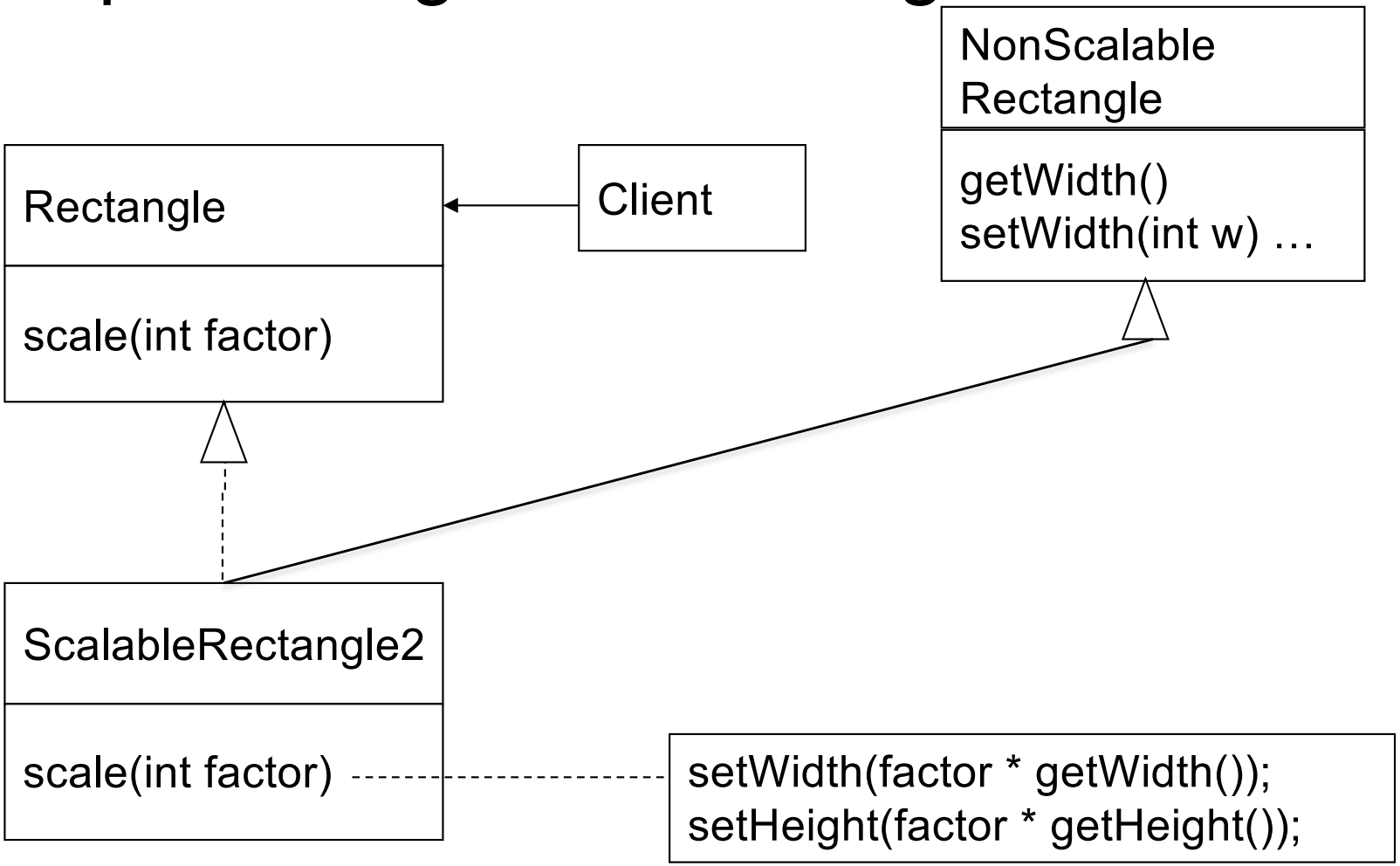


Adapter Pattern

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

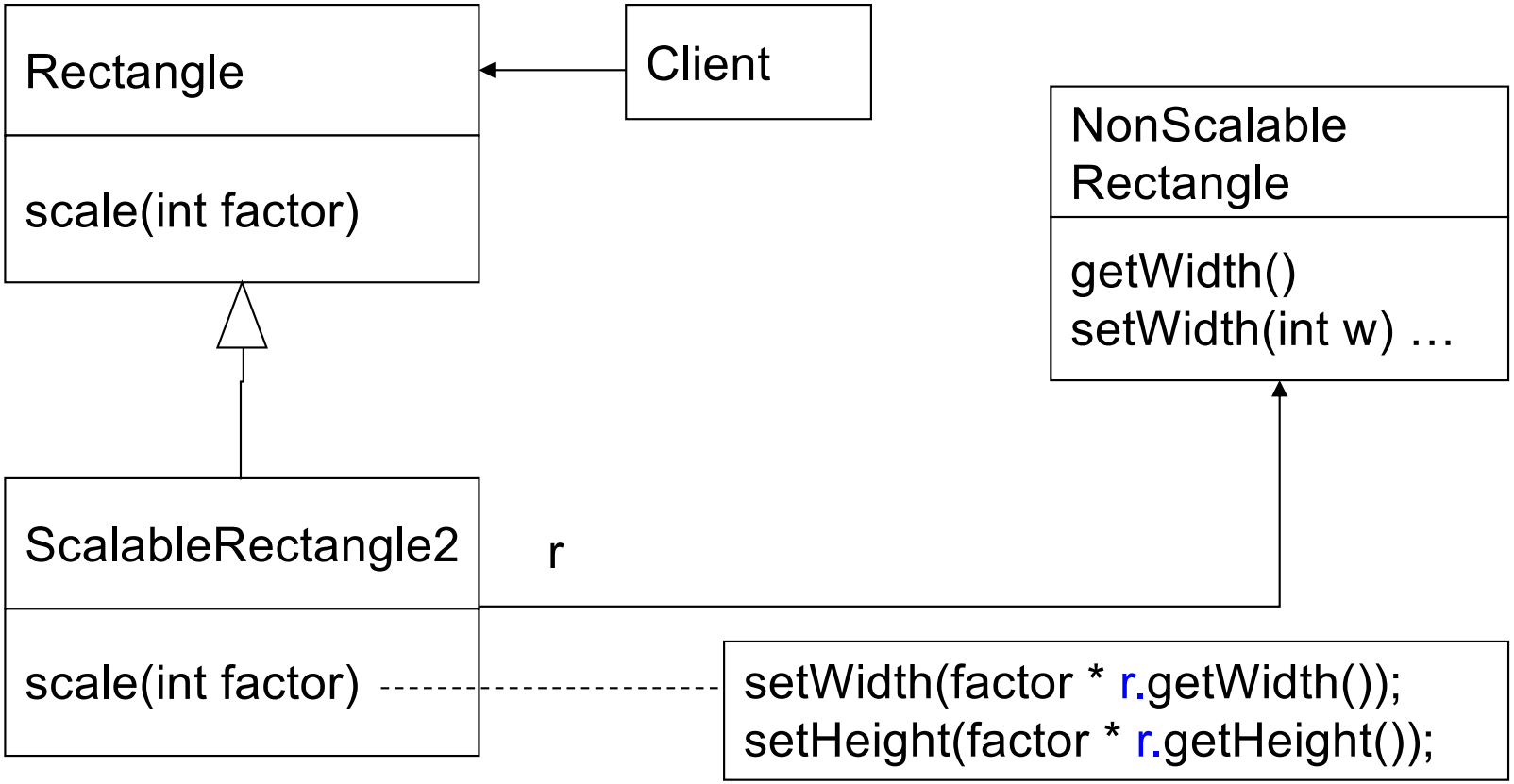
Class Adapter

- Adapts through subclassing

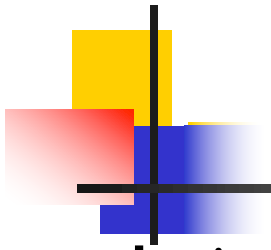


Object Adapter

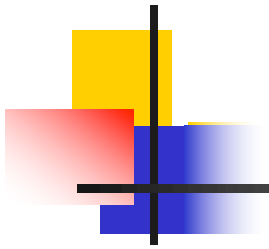
- Adapts through delegation:



Adapter Example: Scaling Rectangles



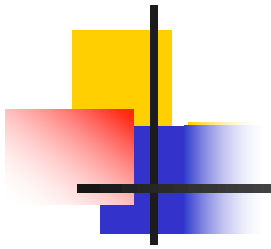
```
interface Rectangle {
  void scale(int factor); //grow or shrink by factor
  …
  float getWidth();
  flaot area();
}
class Client {
  void clientMethod(Rectangle r) {
    … r.scale(2);
  }
}
class NonScalableRectangle {
  void setWidth(); …
  // no scale method!
}
```



Class Adapter

- Adapting via subclassing

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



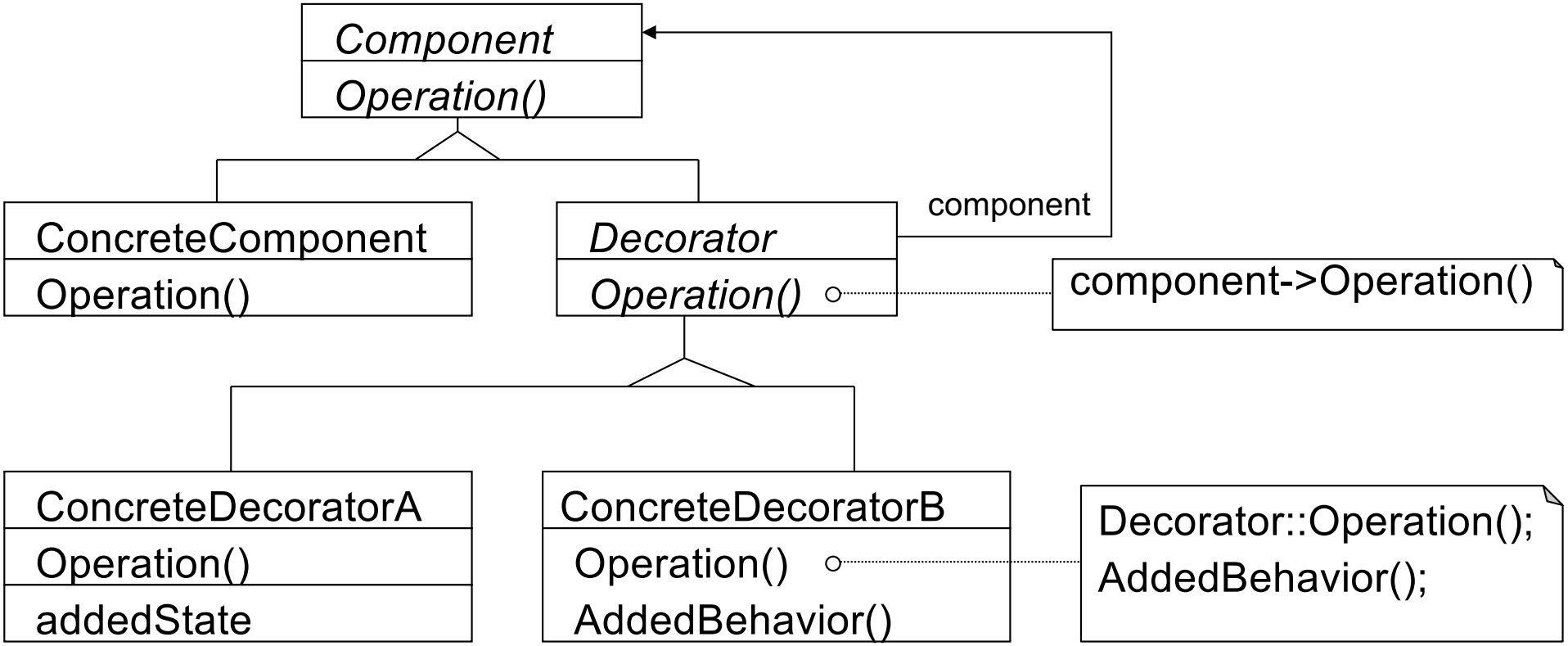
Object Adapter

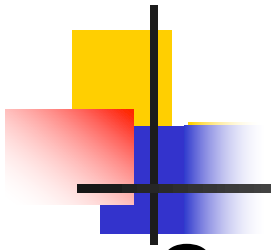
- Adapting via delegation: forward to delegate

```
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

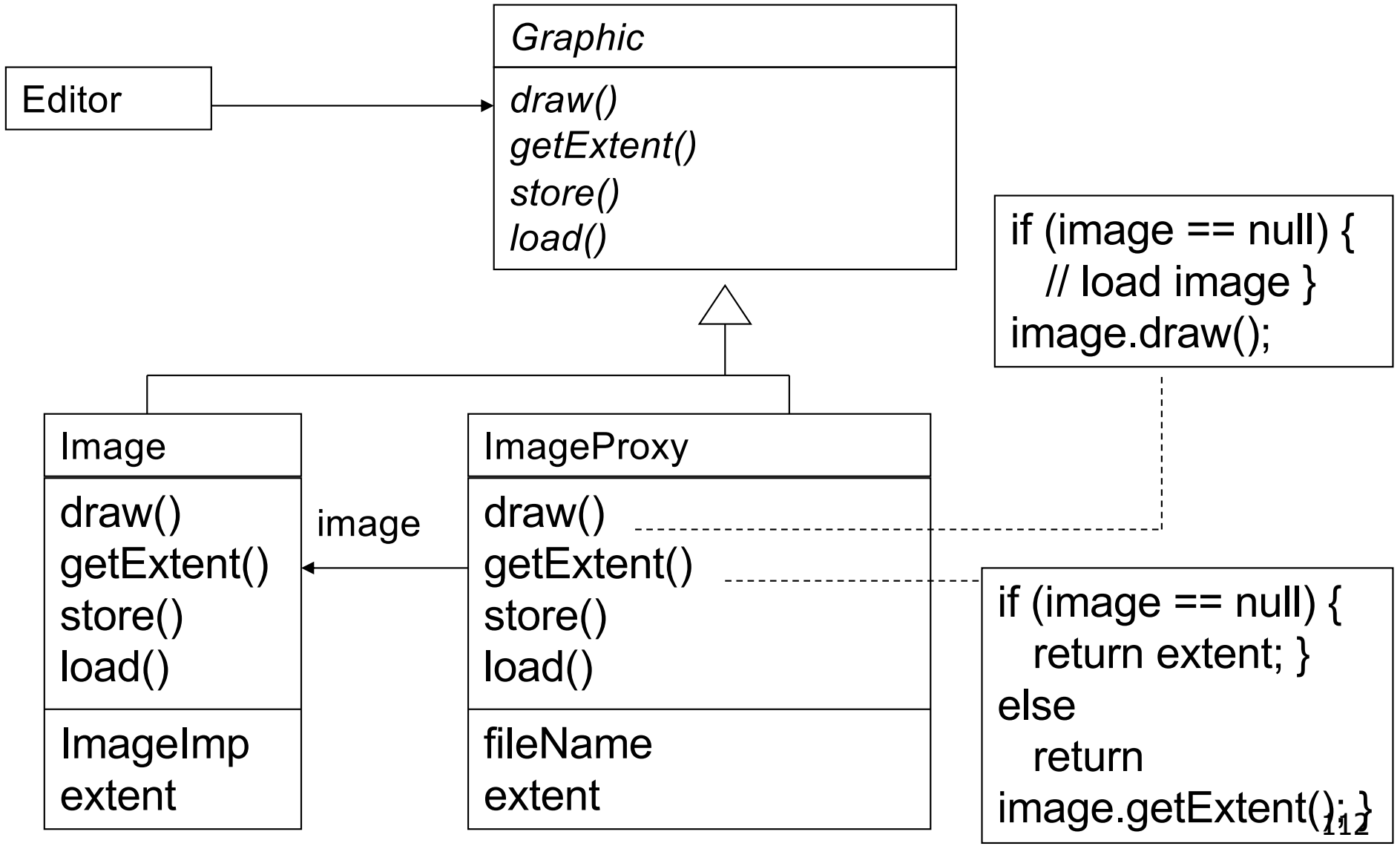


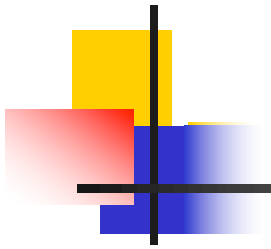


Proxy Pattern

- 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



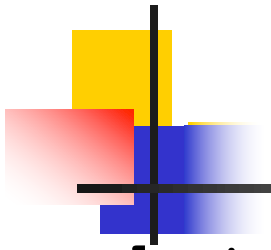


Composite Pattern

- Good for part-whole relationships
  - Can represent arbitrarily complex objects

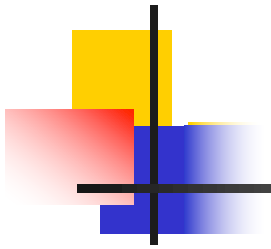
- Client treats a composite object (a collection of units) the same as a simple object (an atomic unit)

Using Composite to represent boolean expressions



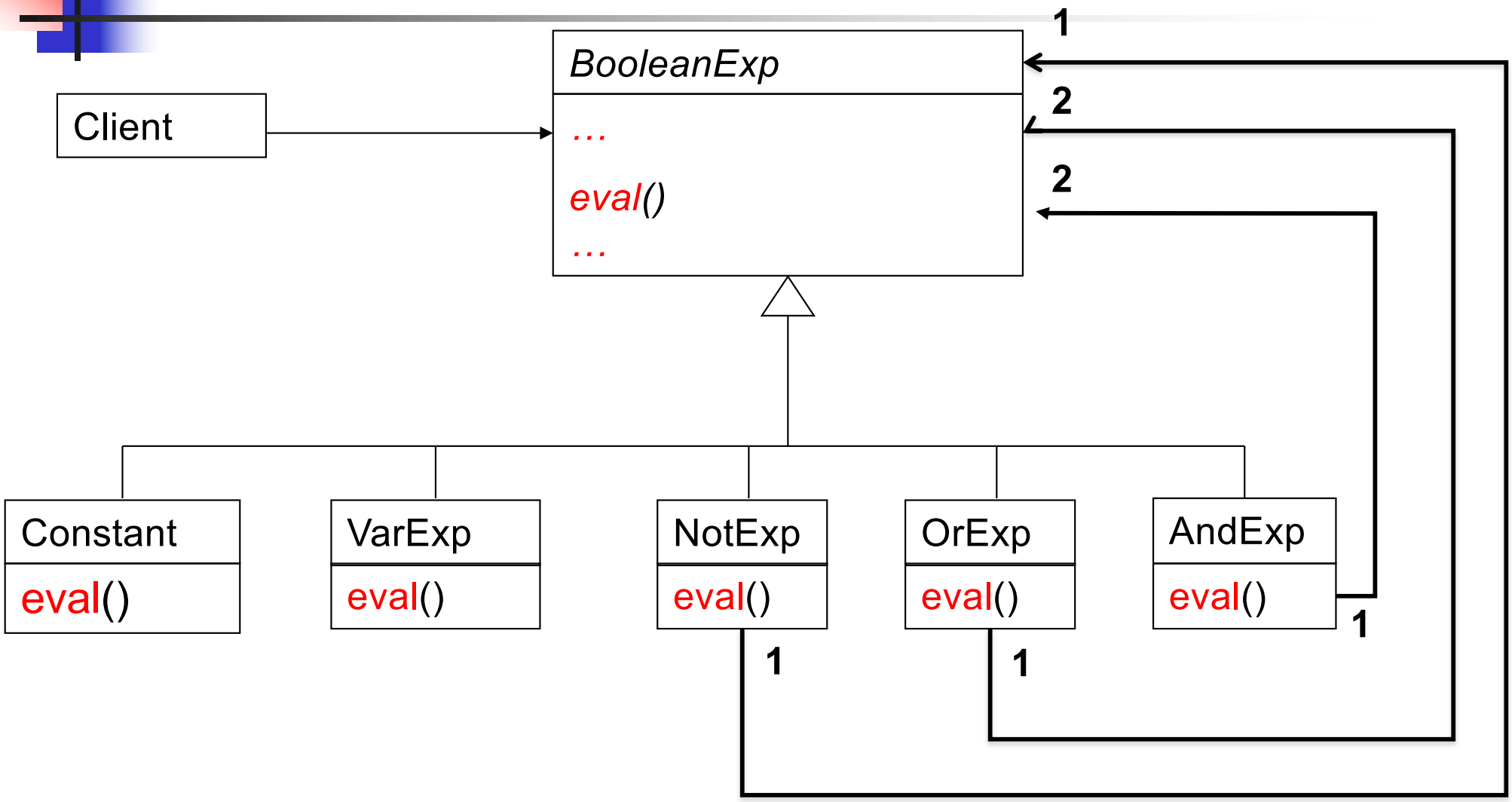
```
abstract class BooleanExp {
  boolean eval(Context c);
}
class Constant extends BooleanExp {
  private boolean const;
  Constant(boolean const) { this.const=const; }
  boolean eval(Context c) { return const; }
}

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

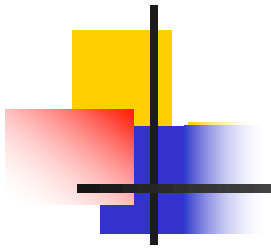
Using Composite to represent boolean expressions



```
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
```

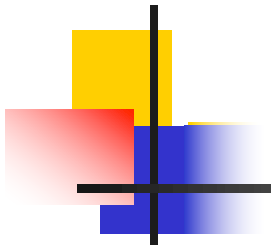
Object Structure vs. Class Diagram





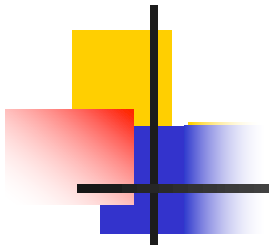
Exercises (Behavioral Patterns)

- 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?



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?

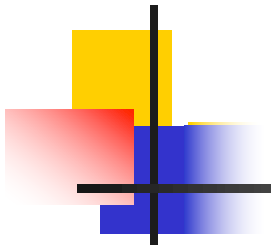


Exercises

- What pattern allows for an object to maintain multiple views that must be updated when the object changes?

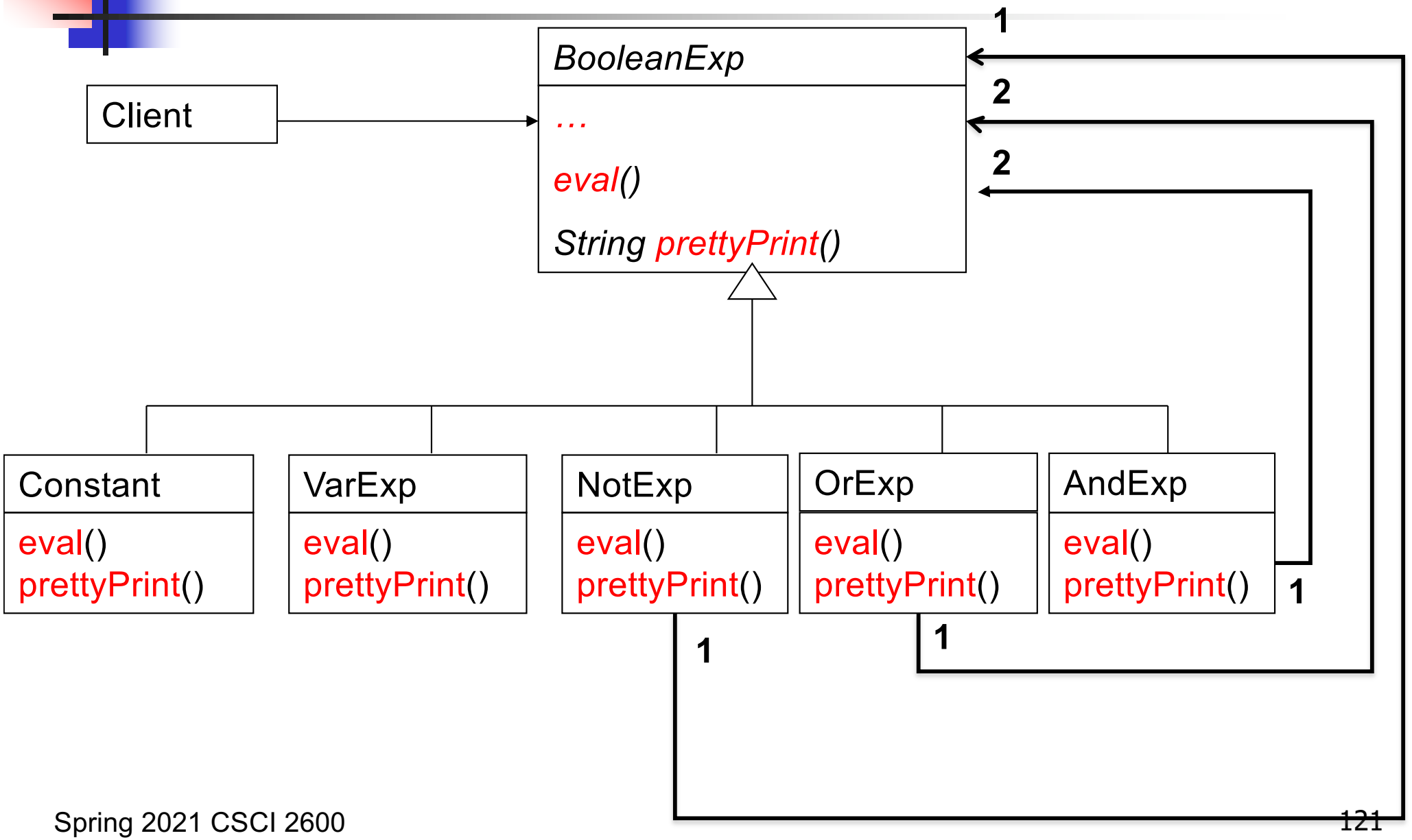
- Give an example of usage of the Composite pattern in the Java standard library
- Given an example of usage of the Observer pattern in the Java standard library

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

Interpreter Pattern

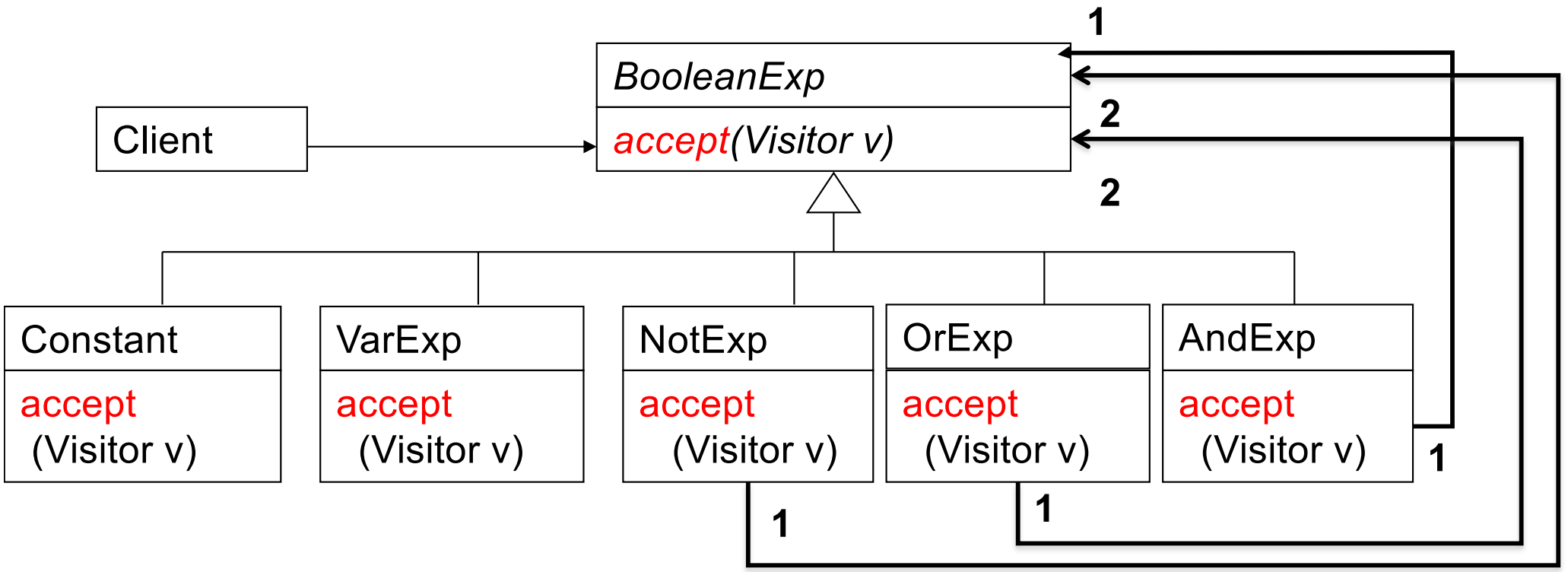


Visitor Pattern

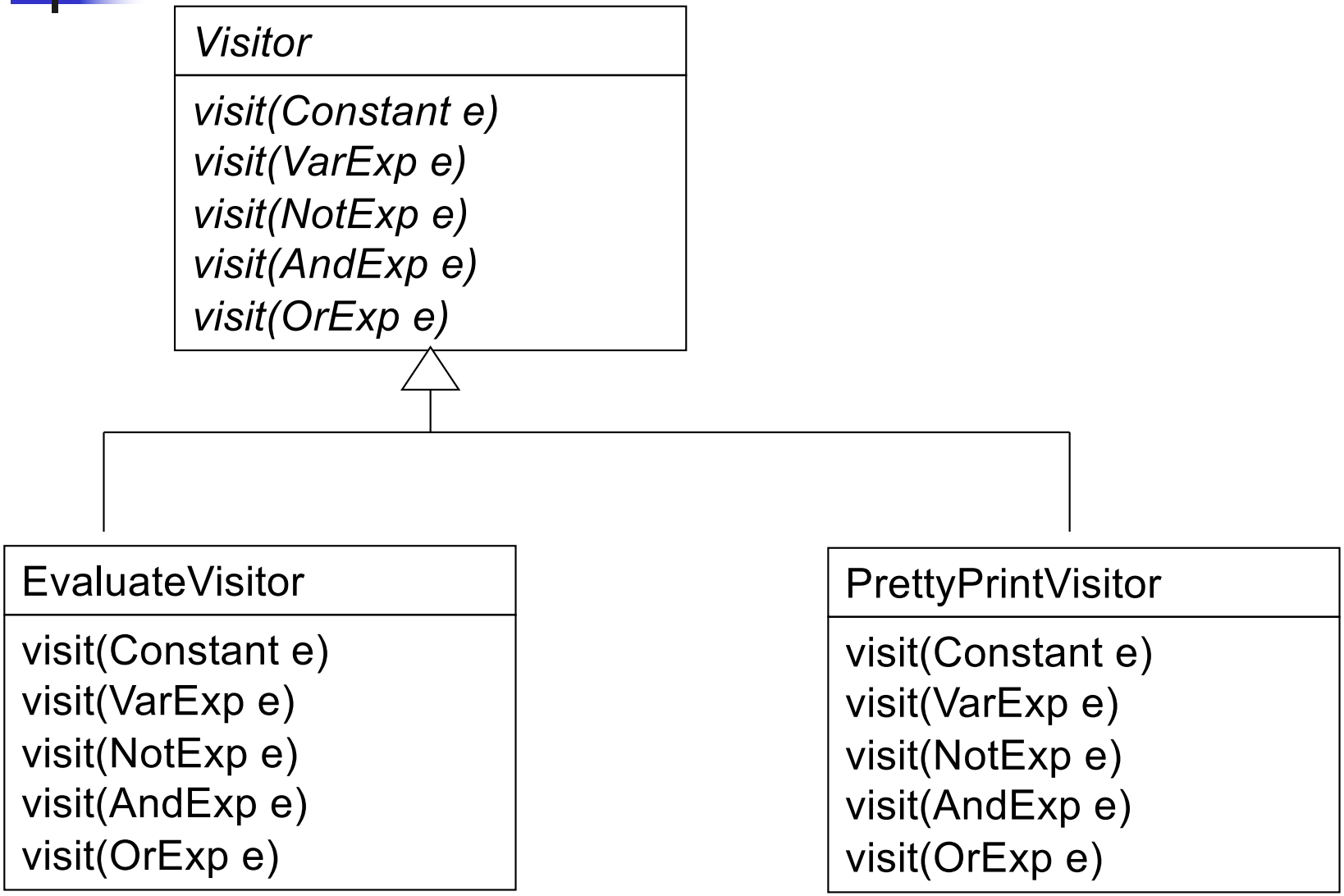
```
class VarExp extends
            BooleanExp {
 void accept(Visitor v) {
  v.visit(this);
 }
}
class AndExp extends
            BooleanExp {
 BooleanExp leftExp;
 BooleanExp rightExp;
 void accept(Visitor v) {
  leftExp.accept(v);
  rightExp.accept(v);
  v.visit(this);
 }
}
```

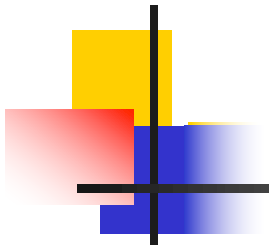
```
class Evaluate
  implements Visitor {
  // keeps state
void visit(VarExp e)
 {
  //evaluate var exp
 }
 void visit(AndExp e)
 {
  //evaluate And exp
 }
}
class PrettyPrint
  implements Visitor {
…
}
```

The Visitor Pattern



The Visitor Pattern

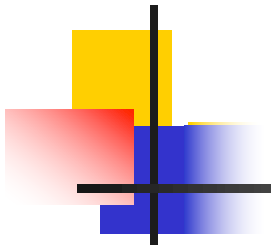




Exercise

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

- Write **`EvaluateVisitor implements Visitor`**
  - Evaluates boolean expression



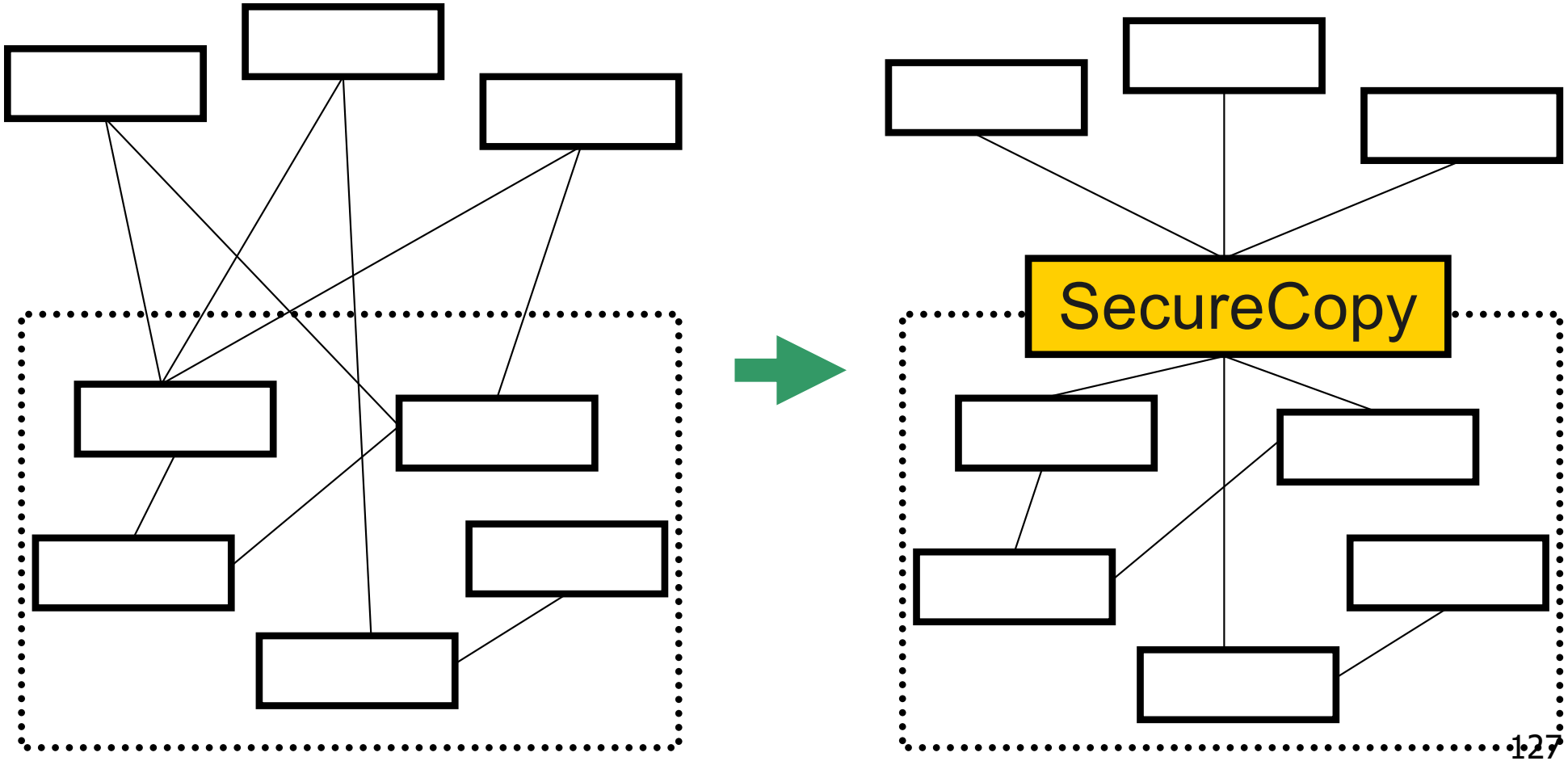
Façade Pattern

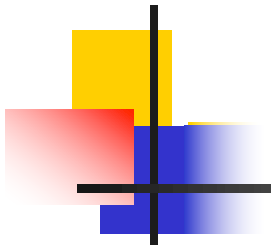
- Question: how to handle the case, when we need a subset of the functionality of a powerful, extensive and complex library

- 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.

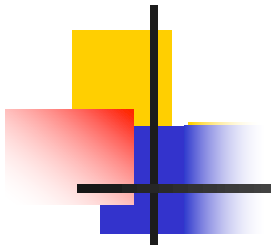




Observer Pattern

- 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 update Views when it changes

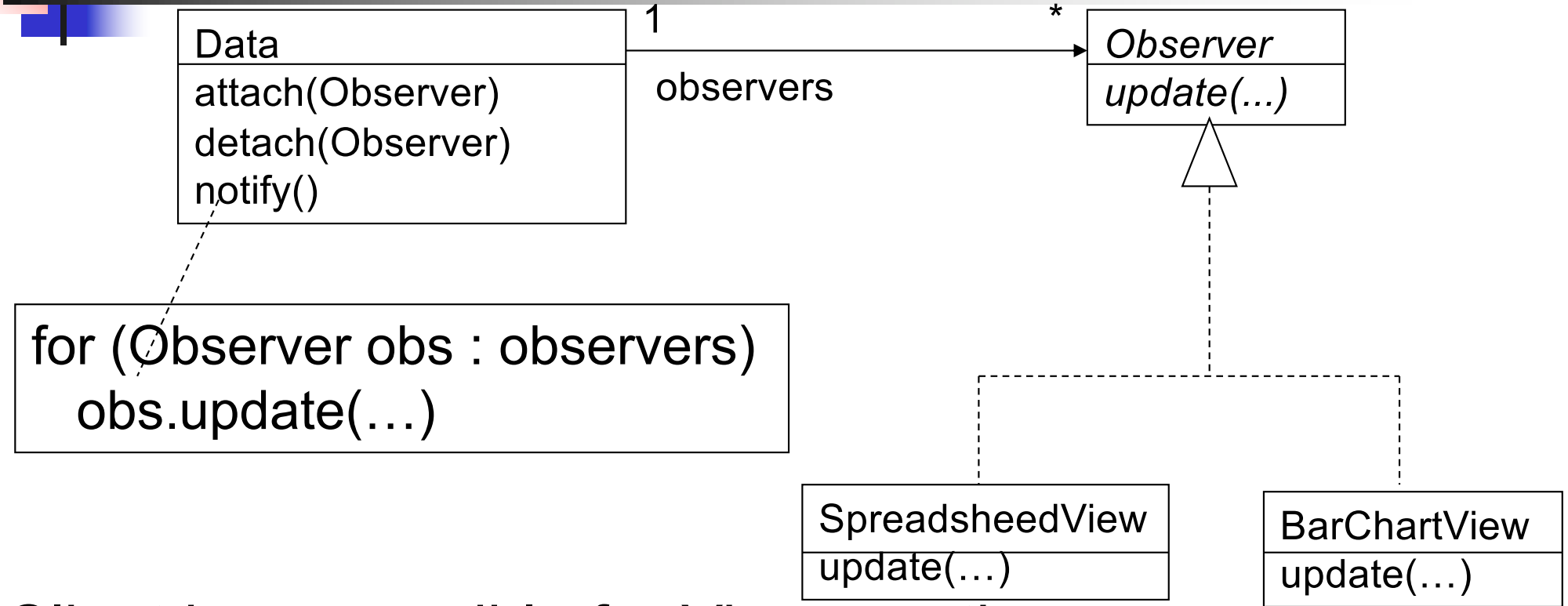
Old, naive design:

```
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(…);
}
```

Class Diagram

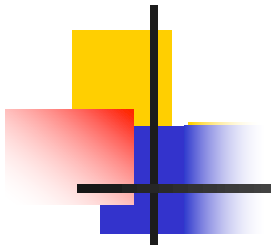


Client is responsible for View creation:

```
data = new Data();
data.attach(new BarChartView());
```

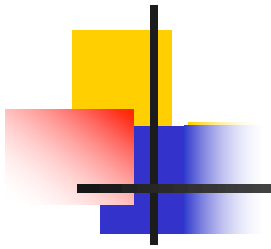
Data keeps list of Views, notifies them when change.

Data is minimally connected to Views!



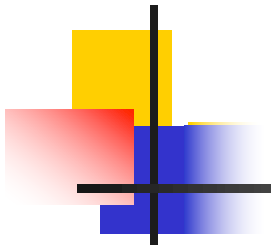
Push vs. Pull Model

- 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



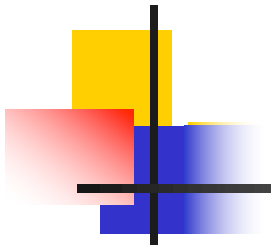
Refactoring

- 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
- **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
- Remember: small change + tests