

## Design Patterns, cont.

## Announcements

- HW7, Due Friday November 20
- Grades and feedback for HW0-5 in Homework Server
- Exam1-2, Quiz1-7 in LMS
- Rainbow Grades coming up!
- Quiz 8

2

## Design Patterns So Far

- Creational patterns: **Factories, Prototype, Singleton, Interning**
  - Problem: constructors in Java (and other OO languages) are inflexible
    1. Can't return a subtype of the type they belong to. "Factory" patterns address the issue: Factory method (e.g. `createBicycle()`), Factory class/object, Prototype
    2. Always return a **fresh new object**, can't reuse. "Sharing" patterns address the issue: Singleton, Interning

Fall 15 CSCI 2600, A Milanova

3

## Design Patterns Summary so Far

- Wrappers: **Adapter, Decorator, Proxy**
  - Structural patterns: when we want to change interface or functionality of an existing class, or restrict access to an object
- **Composite**
  - A structural pattern: expresses whole-part structures, gives uniform interface to client

Fall 15 CSCI 2600, A Milanova

4

## Outline of Today's Class

- Behavioral patterns
  - Observer
  - Façade
- Dependences and coupling
- Behavioral patterns for traversing composites
  - Interpreter
  - Procedural
  - Visitor

5

## Observer Pattern

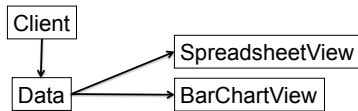
- Question: how to handle an object (model), which has many "observers" (views) that need to be notified and updated when the object changes state
- For example, an interface toolkit with various presentation formats (spreadsheet, bar chart, pie chart). When application data, e.g., **stocks data** (model)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

Fall 15 CSCI 2600, A Milanova

6

## A Naïve Design

- Client stores information in **Data**
- Then **Data** updates the views accordingly



- Problem: to add a view, or change a view, we must change **Data**. Better to insulate **Data** from changes to **Views**!

7

## A Better Design

- 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:

```

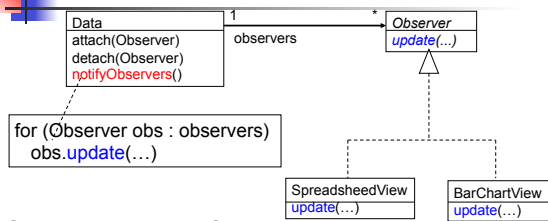
class Data {
  List<Observer> observers;
  void notifyObservers() {
    for (obs : observers)
      obs.update(newData);
  }
}

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

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

8

## Class Diagram



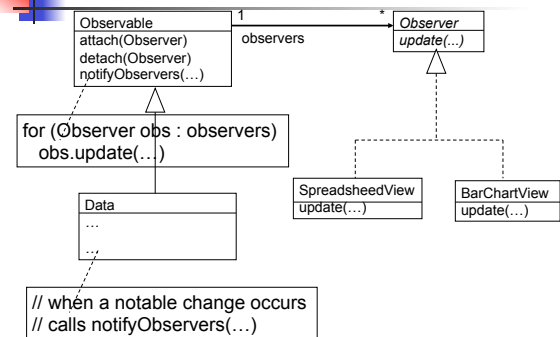
Client is responsible for creation:

```

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

Data keeps list of Views, notifies them when change.  
Data is minimally connected to Views!

## Even Better



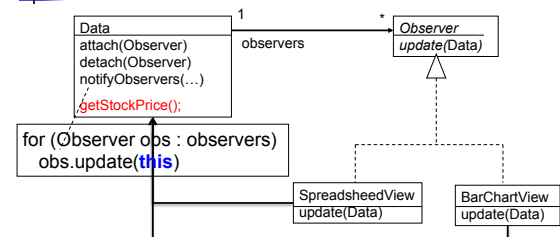
## Push vs. Pull Model

- Question: How does the object (Data in our case) know what info each observer (View) needs?
- Push** model: Object sends the info to Observers
- Pull** model: Object does not send info directly. It gives access to itself to the Observers and lets each Observer extract the data they need

Fall 15 CSCI 2600, A Milanova

11

## Observer Pattern



**Pull** model: observers have access to Data, they can pull the info they need.

## Example of Observer

```
public class SaleItem extends Observable {
    private String name;
    private float price;
    public SaleItem(String name, float price) {
        this.name = name;
        this.price = price;
    }
    public void setName(String name) {
        this.name = name;
        setChanged();
        notifyObservers(name);
    }
    public void setPrice(float price) {
        // analogous to setName
    }
}
```

From JDK

From JDK. Marks that object has changed.

From JDK. If object has changed, calls `obs.update(this, name)`.

**THE MODEL**

13

## An Observer of Name Changes

```
public class NameObserver implements Observer {
    private String name;

    public void update(Observable obj, Object arg) {
        if (arg instanceof String) {
            name = (String) arg;
            System.out.println("NameObserver:
            Name changed to " + name);
        }
        else
            System.out.println("NameObserver:
            Some other change to observable!");
    }
}
```

From JDK

**THE VIEW**

Implements update from JDK. Results in **callback**!

Fall 15 CSCI 2600, A Milanova

14

## An Observer of Price Changes

```
public class PriceObserver implements Observer {
    private Float price;

    public void update(Observable obj, Object arg) {
        if (arg instanceof Float) {
            price = (Float) arg;
            System.out.println("PriceObserver:
            Price changed to " + price);
        }
        else
            System.out.println("PriceObserver:
            Some other change to observable!");
    }
}
```

We don't care for this now.

**ANOTHER VIEW**

15

## The Client

```
SaleItem si = new SaleItem("Corn Pops", 1.29f);
NameObserver nameObs = new NameObserver();
PriceObserver priceObs = new PriceObserver();

// Now add observers
si.addObserver(nameObs);
si.addObserver(priceObs);

// Make changes to the Subject.
si.setName("Frosted Flakes");
si.setPrice(4.57f);
si.setPrice(9.22f);
si.setName("Sugar Crispies");
```

JDK. Since s is Observable!

**THE CONTROLLER**

16

## Another Example

- An application that computes a path on a map and displays the path. When user requests different path, display changes
- Initially, application displays using a simple text-based UI
  - Therefore, a text-based View (i.e., Observer)
- Later, application will display using a GUI interface
  - A GUI-based View (another Observer)

Fall 15 CSCI 2600, A Milanova

17

## Another Example of Observer

```
// Represents sign-up sheet of students
public class SignupSheet extends Observable {
    private List<String> students =
        new ArrayList<String>();
    public void addStrudent(String student) {
        students.add(student);
        notifyObservers();
    }
    public int size() {
        return students.size();
    }
}
```

**THE MODEL**

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

18

## Example of Observer

```
public class SignupObserver extends Observer {
    // called from notifyObservers, which
    // was called when SignupSheet changed
    public void update(Observable o,
                      Object arg) {
        System.out.println("Signup count: "
                          + ((SignupSheet)o).size());
    }
}
```

THE VIEW

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

19

The SignupSheet observable  
is sent when notifyObservers  
called update(this,...)

We don't care for  
arg now.

## The Client

```
SignupSheet s = new SignupSheet();
s.addStudent("Ana");
// nothing visible happens. Why?

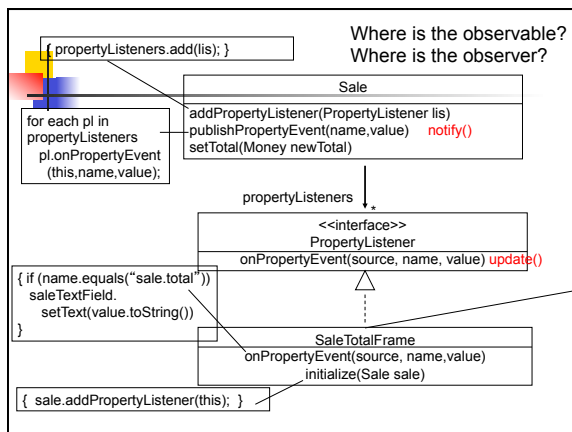
s.addObserver(new SignupObserver());
s.addStudent("Katarina");
// what happens now?
```

What model's used here? Push model or pull model?

THE CONTROLLER

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

20



## Model-view Principle

- Observer pattern known as **Model-view** or **Model-view-controller**
- "Model" objects (e.g., Sale, SignupSheet) should not know about concrete "view" objects (e.g., SaleTotalFrame, SignupObserver)
- Domain layer should be minimally connected with presentation layer
  - Open/closed principle: if user decides to change/upgrade interface, the change shall trigger no modification to domain layer

22

## Façade Pattern

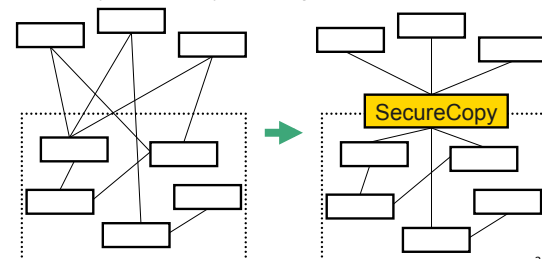
- Question: how to handle the case, when we need a subset of the functionality of a powerful, extensive and complex library
- Example: We want to perform secure file copies to a server. There is a powerful and complex general purpose security library. What is the best way to interact with this library?

Fall 15 CSCI 2600, A Milanova

23

## Façade Pattern

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



24

## Façade Pattern

- Façade reduces interactions between client and the complex library
- Façade hides (mostly irrelevant) complexity of the library
- If library changes, we'll only need to change the Façade, the client remains insulated
  - Open/closed principle: when change happens, the change has minimal impact

Fall 15 CSCI 2600, A Milanova

25

## Outline of Today's Class

- Behavioral patterns
  - Observer
  - Façade
  - Dependences and coupling
- Behavioral patterns for traversing composites
  - Interpreter
  - Procedural
  - Visitor

26

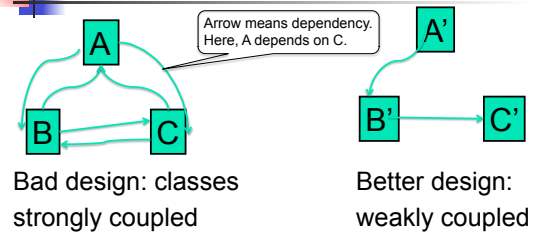
## Interactions Between Modules

- Interactions between modules (in our designs, module = class) cause complexity
- To simplify, split design into parts that don't interact much
- **Coupling** is the amount of interaction among classes
  - Roughly, if class A calls methods/uses fields of class B, then there is **coupling** from A to B
- In design, we strive towards **low (weak) coupling**, i.e., minimal, necessary interactions

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

27

## Low Coupling



Fall 15 CSCI 2600, A Milanova

28

## Coupling is the Path to the Dark Side

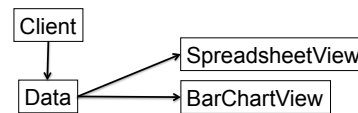
- Coupling leads to complexity
- Complexity leads to confusion
- Confusion leads to suffering
- If once you start down the dark path, forever will it dominate your destiny, consume you it will

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

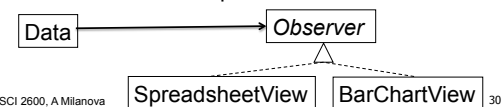
29

## Observer promotes low coupling

- Bad. Data does not need depend on Views



- Better: Weaken dependency of Data on Views
  - Introduce a weaker spec in the form of interface

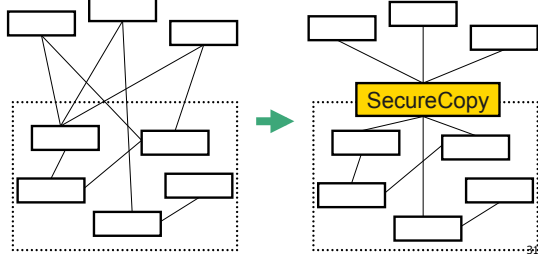


Fall 15 CSCI 2600, A Milanova

30

## Façade promotes low coupling

Façade weakens the dependency between Client and library.  
Introduce Façade object: reduce #dependences from 3\*5 to 3+5!



## Outline of Today's Class

- Behavioral patterns
  - Observer
  - Façade
- Dependences and coupling
- Behavioral patterns for traversing composites
  - Interpreter
  - Procedural
  - Visitor

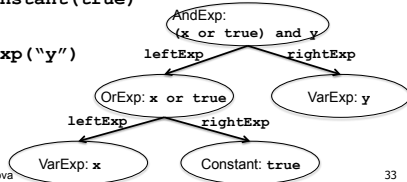
32

## Composite Objects

- Expression (x or true) and y

```
new AndExp(
    new OrExp(
        new VarExp("x"),
        new Constant(true)
    ),
    new VarExp("y")
)
```

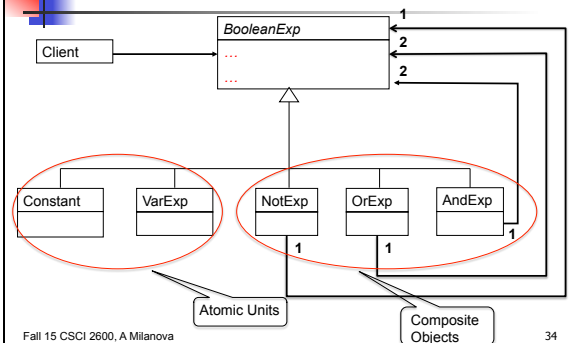
We have a **hierarchical structure**:  
AndExp is top, OrExp and VarExp are below in the hierarchy, etc.



Fall 15 CSCI 2600, A. Milanova

33

## Composite Pattern: Class diagram



Fall 15 CSCI 2600, A. Milanova

34

## Exercise: Object Structure

- Draw the object structure (a tree!) for expression x and true and (y or z)

Fall 15 CSCI 2600, A. Milanova

35

## Traversing Composites

- Question: How to perform operations on composite objects (on all parts of the component)?
- The Interpreter, Procedural and Visitor patterns address this question

Fall 15 CSCI 2600, A. Milanova

36

## Operations on Boolean Expressions

- Need to write code for each Operation/Object pair

	Objects				
	VarExp	Constant	AndExp	OrExp	NotExp
Operations					
evaluate					
pretty-print					

- Question: do we group together (in a class) the code for a particular object or the code for a particular operation?

Fall 15 CSCI 2600, A. Milanova

37

## Interpreter and Procedural Patterns

- Interpreter: groups code for similar **objects**, spreads apart code for similar operations
- Procedural: groups code for similar **operations**, spreads apart code for similar objects

	VarExp	AndExp
Operations		
evaluate		
pretty-print		

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

38

## Interpreter Pattern

```

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

class VarExp extends BooleanExp {
    ...
    boolean eval(Context c) {
        ...
    }
    String prettyPrint() {
        ...
    }
}

class AndExp extends BooleanExp {
    boolean eval(Context c) {
        ...
    }
    String prettyPrint() {
        ...
    }
}

```

Add a method to each class for each supported operation

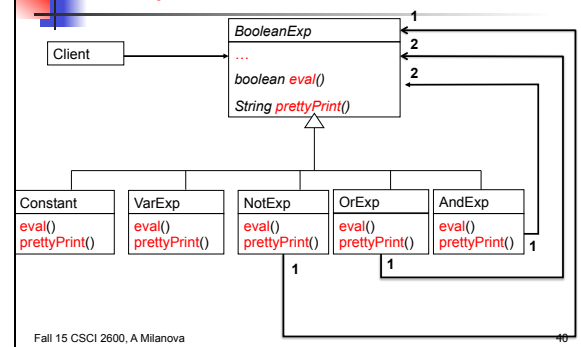
Dynamic dispatch chooses right implementation at call

BooleanExp myExpr = ...  
myExpr.eval(c);

Fall 15 CSCI 2600, A. Milanova

39

## Interpreter Pattern



Fall 15 CSCI 2600, A. Milanova

40

## Interpreter Pattern

```

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);
    }
}

```

Fall 15 CSCI 2600, A. Milanova

41

## Interpreter Pattern

```

class AndExp extends BooleanExp {
    private BooleanExp leftExp;
    private BooleanExp rightExp;
    AndExp(BooleanExp left, BooleanExp right) {
        leftExp = left;
        rightExp = right;
    }
    boolean eval(Context c) {
        return leftExp.eval(c) && rightExp.eval(c);
    }
}

// analogous definitions for OrExp and NotExp

```

Fall 15 CSCI 2600, A. Milanova

42

## Procedural Pattern

	VarExp	AndExp
evaluate		
pretty-print		

// Classes for expressions don't have eval!

```

class Evaluate {
    boolean evalConstExp(Constant c) {
        c.value(); // returns value of constant
    }
    boolean evalAndExp(AndExp e) {
        BooleanExp leftExp = e.leftExp();
        BooleanExp rightExp = e.rightExp();

        //Problem: How to invoke the right
        //implementation for leftExp and rightExp?
    }
    // also, evalVarExp, evalOrExp, evalNotExp
}

```

43

## Procedural Pattern

	VarExp	AndExp
evaluate		
pretty-print		

```

class Evaluate {
    Context c;
    ...
    boolean evalExp(BooleanExp e) {
        if (e instanceof VarExp)
            return evalVarExp((VarExp) e);
        else if (e instanceof Constant)
            return evalConstExp((Constant) e);
        else if (e instanceof OrExp)
            return evalOrExp((OrExp) e);
        else ...
    }
}

```

What is the problem with this code?

Fall 15 CSCI 2600, A Milanova

44

## Visitor Pattern, a Variant of the Procedural Pattern

- Visitor helps traverse a hierarchical structure
- Nodes (objects in the hierarchy) **accept** visitors
- Visitors **visit** nodes (objects)

```

class SomeBooleanExp extends BooleanExp {
    void accept(Visitor v) {
        for each child of this node {
            child.accept(v);
        }
        v.visit(this);
    }
}

```

n.accept(v) traverses the structure rooted at n, performing v's operation on every element

```

class Visitor {
    void visit(SomeBooleanExp e) { do work on e }
}

```

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

45

## 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);
    }
}

```

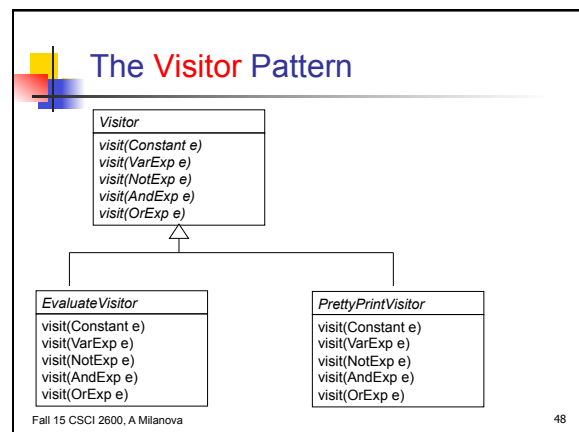
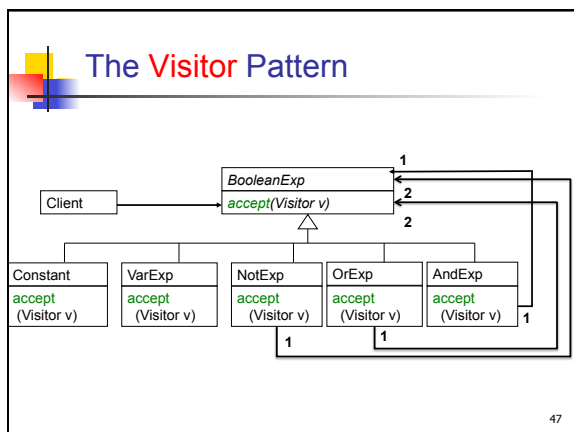
```

class Evaluate
    implements Visitor {
    // state, needed to
    // evaluate
    void visit(VarExp e) {
        //evaluate Var exp
    }
    void visit(AndExp e) {
        //evaluate And exp
    }
}
class PrettyPrint
    implements Visitor {
    ...
}

```

Fall 15 CSCI 2600, A Milanova

46





## Question

```
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 CounterVisitor implements Visitor {
    int count = 0;
    void visit(VarExp e) {
        //??
    }
    void visit(Constant e) {
        //??
    }
    void visit(AndExp e) {
        //??
    }
    ...
}
```

49

## Exercise: Write Count Visitor which counts #subexpressions in a BooleanExp object

```
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 CounterVisitor implements Visitor {
    int count = 0;
    void visit(VarExp e) {
        //??
    }
    void visit(Constant e) {
        //??
    }
    void visit(AndExp e) {
        //??
    }
    ...
}
```

50

## Starting the Visitor

```
BooleanExp myExp =
    new AndExp(
        new OrExp(new VarExp("x"), new VarExp("y")),
        new VarExp("z")
    );
Visitor v = new CounterVisitor();

myExp.accept(v);
```

Spring 15 CSCI 2600, A Milanova

51

## Exercise: Write Evaluate Visitor which evaluates a BooleanExp object

```
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 EvaluateVisitor implements Visitor {
    // ??
    void visit(VarExp e) {
        // ??
    }
    void visit(Constant e) {
        // ??
    }
    void visit(AndExp e) {
        // ??
    }
    ...
}
```

52

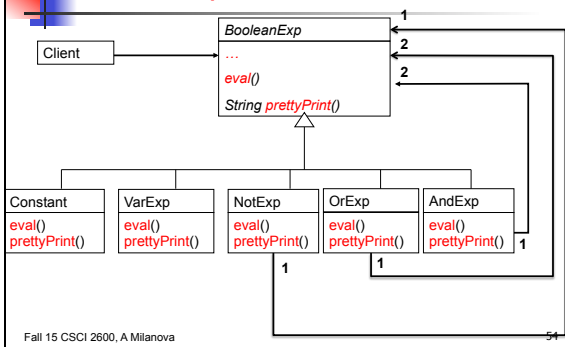
## Exercise: Write a Visitor that Computes the Cost of a Bicycle Component (Note: Cost of a composite is sum of costs of components + assembly cost)

```
class Skewer extends BicycleComponent {
    void accept(Visitor v) {
        v.visit(this);
    }
}
class Wheel extends BicycleComponent {
    BicycleComponent skewer;
    BicycleComponent hub;
    ...
    void accept(Visitor v) {
        skewer.accept(v);
        hub.accept(v);
        v.visit(this);
    }
}
```

```
class CostVisitor implements Visitor {
    ...
    void visit(Skewer e) {
        //??
    }
    void visit(Hub e) {
        //??
    }
    void visit(Wheel e) {
        //??
    }
    ...
}
```

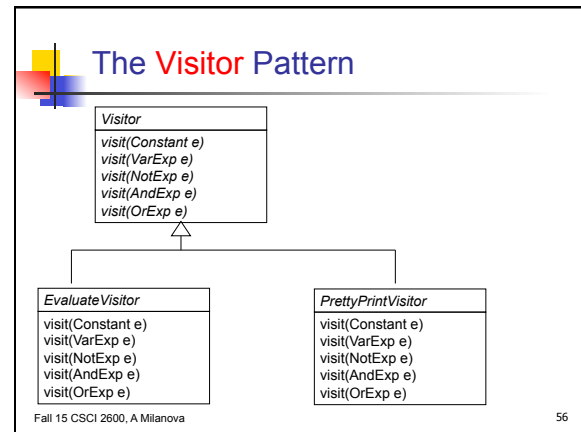
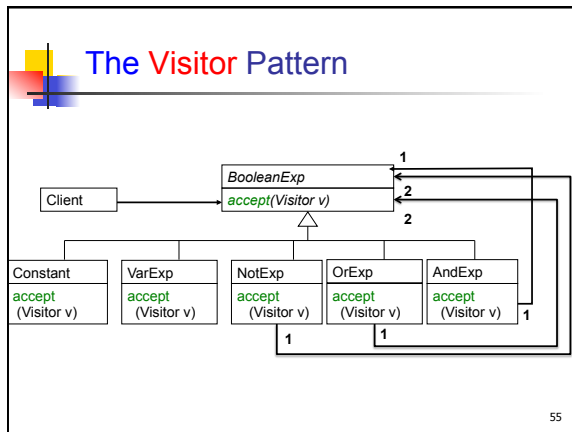
53

## The Interpreter Pattern



Fall 15 CSCI 2600, A Milanova

54



## Visitor Pattern

- Must add definitions of **visit** in Visitor hierarchy and **accept** in Object hierarchy
- visit** may do many different things: evaluate, count nodes, pretty print, etc.
- It is easy to add operations (just add a new Visitor class!), but it is hard to add nodes (must modify entire hierarchy of Visitors!)
- Visitor pattern similar to Iterator but different because it has knowledge of structure, not just sequence

57

## Visitor Pattern's Double Dispatch

	VarExp	AndExp
evaluate		
pretty-print		

**myExp.accept(v):** we want to choose the right operation  
**myExp.accept(v)** // dynamically dispatch the right  
 // implementation of **accept**, e.g., **AndExp.accept**

```

class AndExp {
    void accept(Visitor v) {
        ...
        v.visit(this); // at compile-time, chooses the
    } // method family: visit(AndExp). At
    // runtime, dispatches the right implementation of
    // visit(AndExp), e.g.,
    // EvaluateVisitor.visit(AndExp)
  }
  
```

58

## Design Patterns Summary so Far

- Factory method, Factory object, Prototype**
  - Creational patterns: address problem that constructors can't return subtypes
- Singleton, Interning**
  - Creational patterns: address problem that constructors always return a new instance of class
- Wrappers: Adapter, Decorator, Proxy**
  - Structural patterns: when we want to change interface or functionality of an existing class, or restrict access to an object

59

## Design Patterns Summary so Far

- Composite**
  - A structural pattern: expresses whole-part structures, gives uniform interface to client
- Interpreter, Procedural, Visitor**
  - Behavioral patterns: address the problem of how to traverse composite structures

60