CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2016
Doc 9 Decorator, Template, Observer, State
Feb 23, 2016

Decorator Example - Wikipedia

Compute cost of customers drinks

- Normal price
- Happy hour price

```
interface BillingStrategy {
    public double getActPrice(double rawPrice);
}

class NormalStrategy implements BillingStrategy {
    public double getActPrice(double rawPrice) {
        return rawPrice;
    }
}

class HappyHourStrategy implements BillingStrategy {
    public double getActPrice(double rawPrice) {
        return rawPrice*0.5;
    }
}
```

```
class Customer {

    private List<Double> drinks;
    private BillingStrategy strategy;

    public Customer(BillingStrategy strategy) {
        this.drinks = new ArrayList<Double>();
        this.strategy = strategy;
    }

    public void add(double price, int quantity) {
        drinks.add(strategy.getActPrice(price * quantity));
    }
```

```
public void printBill() {
    double sum = 0;
    for (Double i : drinks) {
        sum += i;
    }
    System.out.println("Total due: " + sum);
    drinks.clear();
}

// Set Strategy
public void setStrategy(BillingStrategy strategy) {
    this.strategy = strategy;
}
```

```
Customer a = new Customer(new NormalStrategy());

        // Normal billing
        a.add(1.0, 1);

        // Start Happy Hour
        a.setStrategy(new HappyHourStrategy());
        a.add(1.0, 2);

        // New Customer
        Customer b = new Customer(new HappyHourStrategy());
        b.add(0.8, 1);
        // The Customer pays
        a.printBill();

        // End Happy Hour
        b.setStrategy(new NormalStrategy());
        b.add(1.3, 2);
```

Short & Simple Strategies

Can be replaced with lambda

```
interface BillingStrategy {
    public double getActPrice(double rawPrice);
}

class NormalStrategy implements BillingStrategy {
    public double getActPrice(double rawPrice) {
        return rawPrice;
    }
}

class HappyHourStrategy implements BillingStrategy {
    public double getActPrice(double rawPrice) {
        return rawPrice*0.5;
    }
}
```

```
Function<Double, Double>

price -> price

price -> price * 0.5
```

```
class Customer {

    private List<Double> drinks;
    private Function<Double, Double> strategy;

    public Customer(Function<Double, Double> strategy) {
        this.drinks = new ArrayList<Double>();
        this.strategy = strategy;
    }

    public void add(double price, int quantity) {
        drinks.add(strategy.apply(price * quantity));
    }
```

```
Customer a = new Customer(new NormalStrategy());
    Function<Double, Double> happyHourStrategy = price -> price * 0.5;
    Function<Double, Double> normalStrategy = price -> price;

        // Normal billing
        a.add(1.0, 1);

        // Start Happy Hour
        a.setStrategy(happyHourStrategy);
        a.add(1.0, 2);

        // New Customer
        Customer b = new Customer(happyHourStrategy);
        b.add(0.8, 1);
        // The Customer pays
        a.printBill();

        // End Happy Hour
        b.setStrategy(normalStrategy);
        b.add(1.3, 2);
```

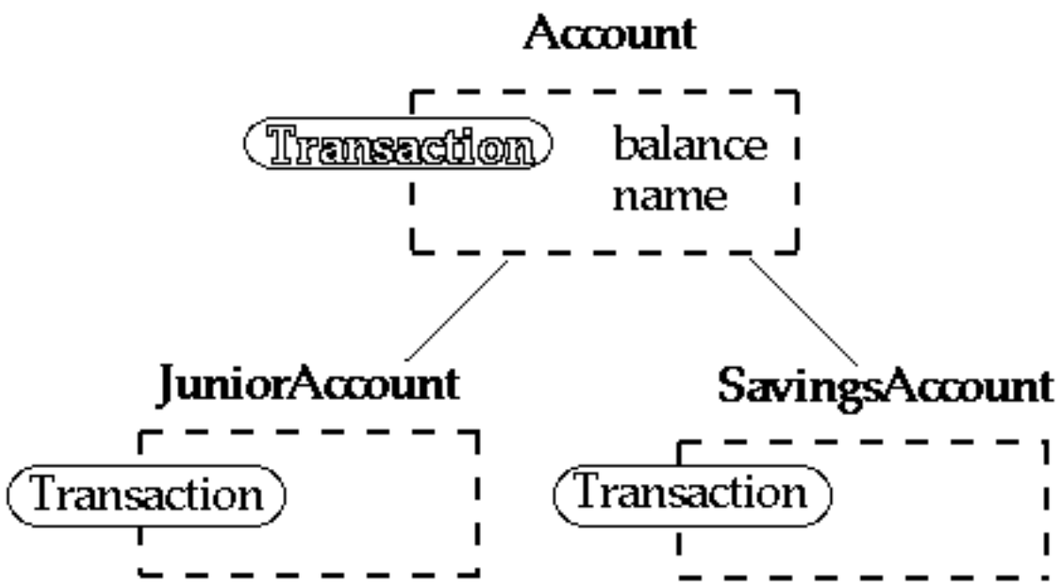
Template Method

Polymorphism

```
class Account {
        public:
                void virtual Transaction(float amount)
                                { balance += amount;}
                Account(char* customerName, float InitialDeposit = 0);
        protected:
                char* name;
                float balance;
}

class JuniorAccount : public Account {
        public:      void Transaction(float amount) {//code here}
}

class SavingsAccount : public Account {
        public:      void Transaction(float amount) {//code here}
}

Account* createNewAccount(){
        // code to query customer and determine what type of
        // account to create
};
```

```
main() {
        Account*  customer;
        customer = createNewAccount();
        customer->Transaction(amount);
}
```

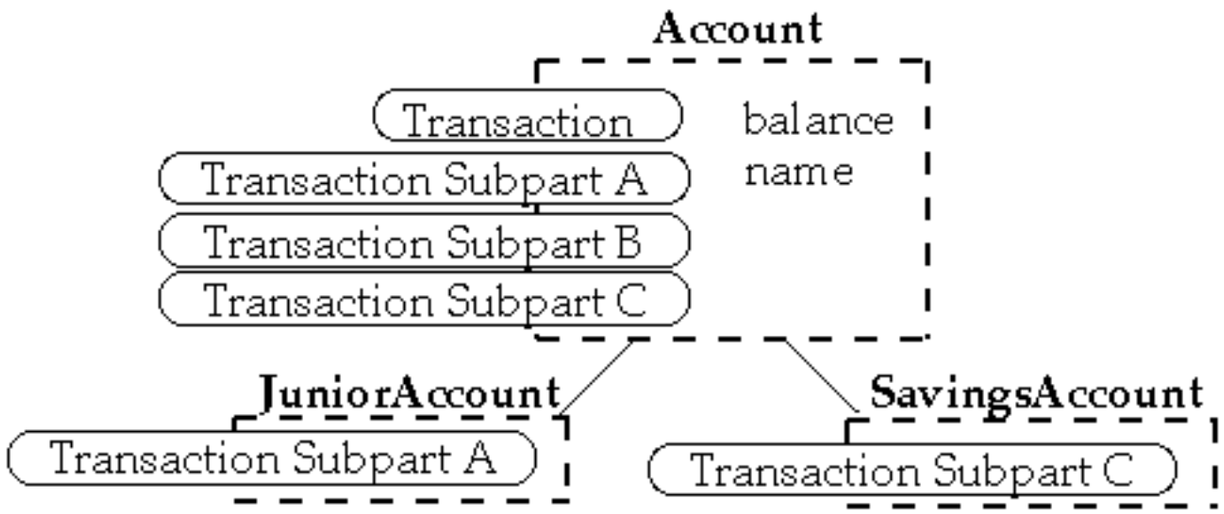
Deferred Methods




```
class Account {
    public:
        void virtual Transaction() = 0;
}

class JuniorAccount : public Account {
    public
        void Transaction() { put code here}
}
```

Template Method

```
class Account {
        public:
                void Transaction(float amount);
        protected:
                void virtual TransactionSubpartA();
                void virtual TransactionSubpartB();
                void virtual TransactionSubpartC();
}

void Account::Transaction(float amount)  {
        TransactionSubpartA();          TransactionSubpartB();
        TransactionSubpartC();          // EvenMoreCode;
}

class JuniorAccount : public Account {
        protected:      void virtual TransactionSubpartA(); }

class SavingsAccount : public Account {
        protected:      void virtual TransactionSubpartC(); }

Account*  customer;
customer = createNewAccount();
customer->Transaction(amount);
```



Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses

Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure

Java Example

```
import  java.awt.*;
class  HelloApplication  extends  Frame
    {
    public  void  paint(  Graphics  display  )
        {
        int startX  =  30;
        int startY  =  40;
        display.drawString(  "Hello World", startX,
startY  );
        }
    }
```

Ruby LinkedList Example

```
class LinkedList
  include Enumerable

  def [](index)
    Code not shown
  end

  def size
    Code not shown
  end

  def each
     Code not shown
  end

  def push(object)
   Code note shown
  end

end
```

```
def testSelect
  list = LinkedList.new
  list.push(3)
  list.push(2)
  list.push(1)

  a = list.select {|x| x.even?}
  assert(a == [2])
end
```

Where does list.select come from?

Methods defined in Enumerable

all?	any?	collect	detect
each_cons	each_slice	each_with_index	entries
enum_cons	enum_slice	enum_with_index	find
find_all	grep	include?	inject
map	max	member?	min
partition	reject	select	sort
sort_by	to_a	to_set	zip

All use "each"

Implement "each" and the above will work

java.util.AbstractCollection

Subclass AbstractCollection

Implement

- iterator
- size
- add

Get

- addAll
- clear
- contains
- containsAll
- isEmpty
- remove
- removeAll
- retainAll
- size
- toArray
- toString

Consequences

This is the most commonly used of the 23 GoF patterns

Important in class libraries

Inverted control structure

- Parent class calls subclass methods

Consequences

Inverted control structure

Java's paint method is a primitive operation called by a parent method

Beginning Java programs don't understand how the following works:

```
import  java.awt.*;
class  HelloApplication  extends  Frame
    {
    public  void  paint(  Graphics  display  )
        {
        int startX  =  30;
        int startY  =  40;
        display.drawString(  "Hello World", startX, startY  );
        }
    }
```

Consequences

Template methods tend to call:

- Concrete operations
- Primitive (abstract) operations
- Factory methods
- Hook operations

Provide default behavior that subclasses can extend

It is important to denote which methods

- Must overridden
- Can be overridden
- Can not be overridden

Refactoring to Template Method

Simple implementation
- Implement all of the code in one method
- The large method you get will become the template method

Break into steps
- Use comments to break the method into logical steps
- One comment per step

Make step methods
- Implement separate methods for each of the steps

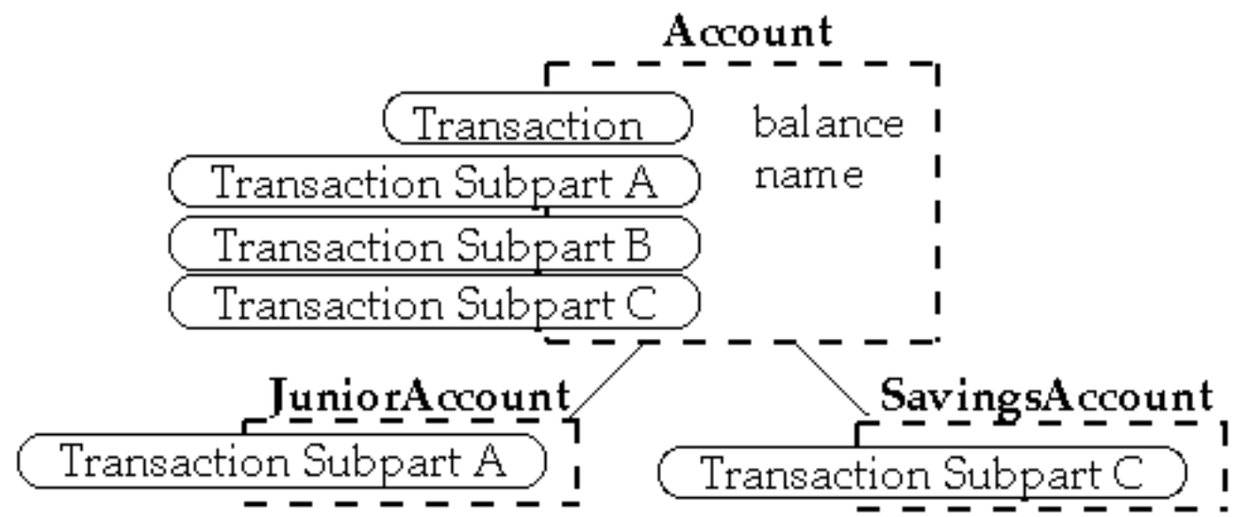
Call the step methods
- Rewrite the template method to call the step methods

Repeat above steps
- Repeat the above steps on each of the step methods
- Continue until:
  - All steps in each method are at the same level of generality
  - All constants are factored into their own methods

Design Patterns Smalltalk Companion pp. 363-364.

Template Method & Functional Programming

```
class Account {
        public:
                void Transaction(float amount);
        protected:
                void virtual TransactionSubpartA();
                void virtual TransactionSubpartB();
                void virtual TransactionSubpartC();
}

void Account::Transaction(float amount)  {
        TransactionSubpartA();          TransactionSubpartB();
        TransactionSubpartC();          // EvenMoreCode;
}

class JuniorAccount : public Account {
        protected:      void virtual TransactionSubpartA(); }

class SavingsAccount : public Account {
        protected:      void virtual TransactionSubpartC(); }

Account*  customer;
customer = createNewAccount();
customer->Transaction(amount);
```



Template Method & Functional Programming

Pass in functions

```
def transaction(defaultPartA, defaultPartB, defaultPartC, amount, account) {
    defaultPartA();
    defaultPartB();
    defaultPartC();
    code code;
}
```

But this adds a lot of arguments

Requires knowing internal workings of transaction

Currying & Partial Evaluation

Pass in functions

```
def defaultTransaction = transaction(defaultPartA, defaultPartB, defaultPartC);
def juniorTransaction = transaction(juniorPartA, defaultPartB, defaultPartC);

defaultTransaction(amount, account);
juniorTransaction(amount, account);
```

But this requires knowing the account type

Multi-methods

```
defmulti transaction(amount, account) (getAccountType)

defmethod transaction(amount, account) (:default) {
    return defaultTransaction(amount, account);
}

defmethod transaction(amount, account) (:junior) {
    return juniorTransaction(amount, account);
}

transaction(amount, account);
```

Now have dynamic dispatch on the type like Java

Template Method vs Functional Solution

	Template Method	Functional
Method Variation	In multiple classes/files	In one place
Add new Variation	Add class/file + method	Add function
Type Logic	In class & parent class	

Observer

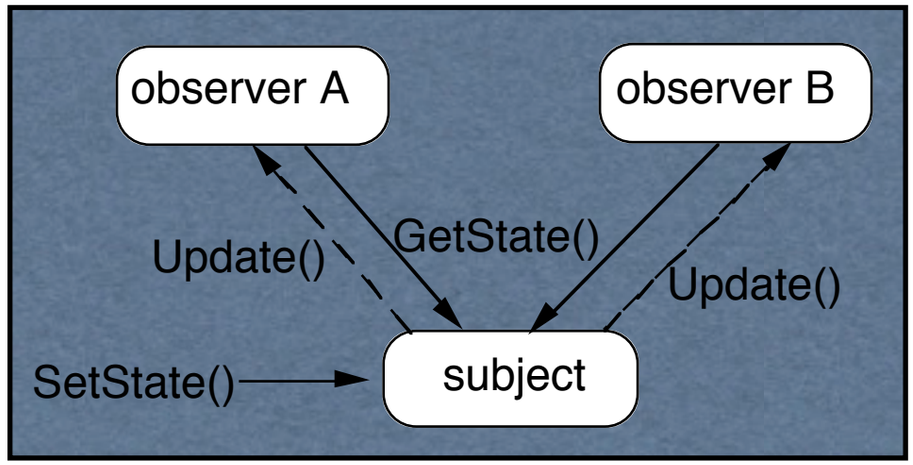
Observer

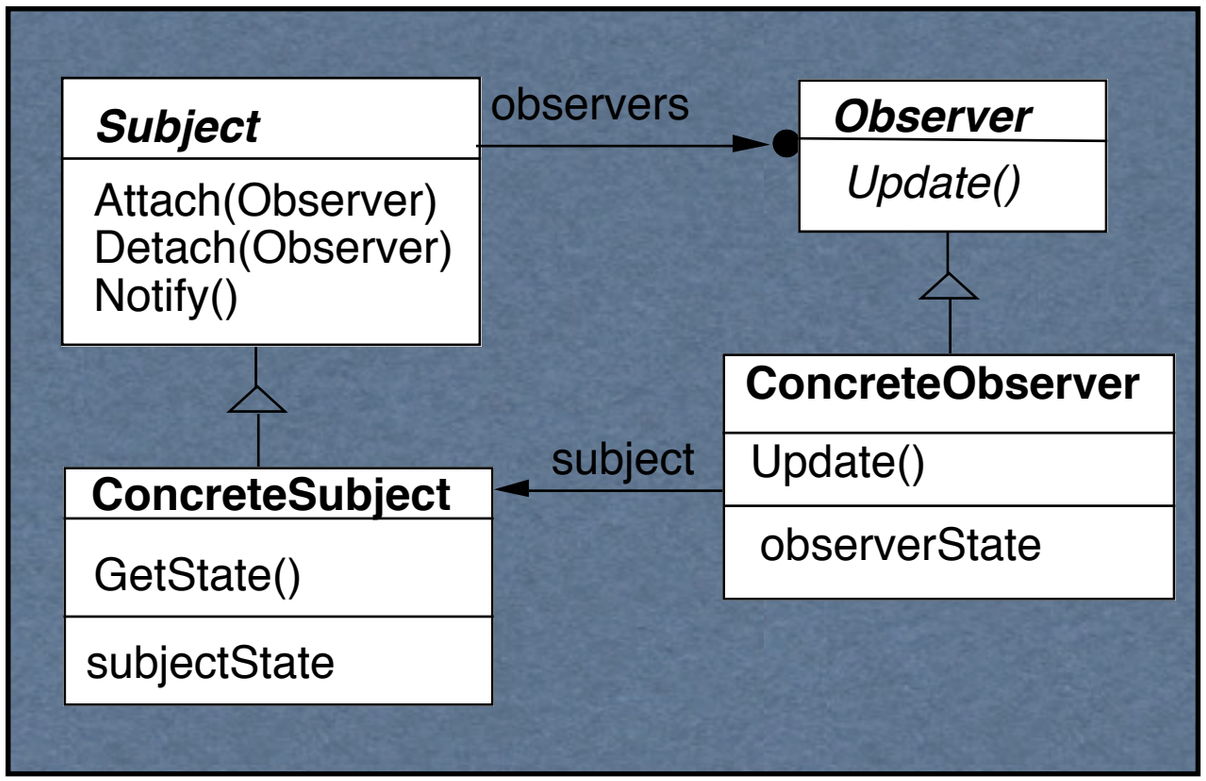
One-to-many dependency between objects

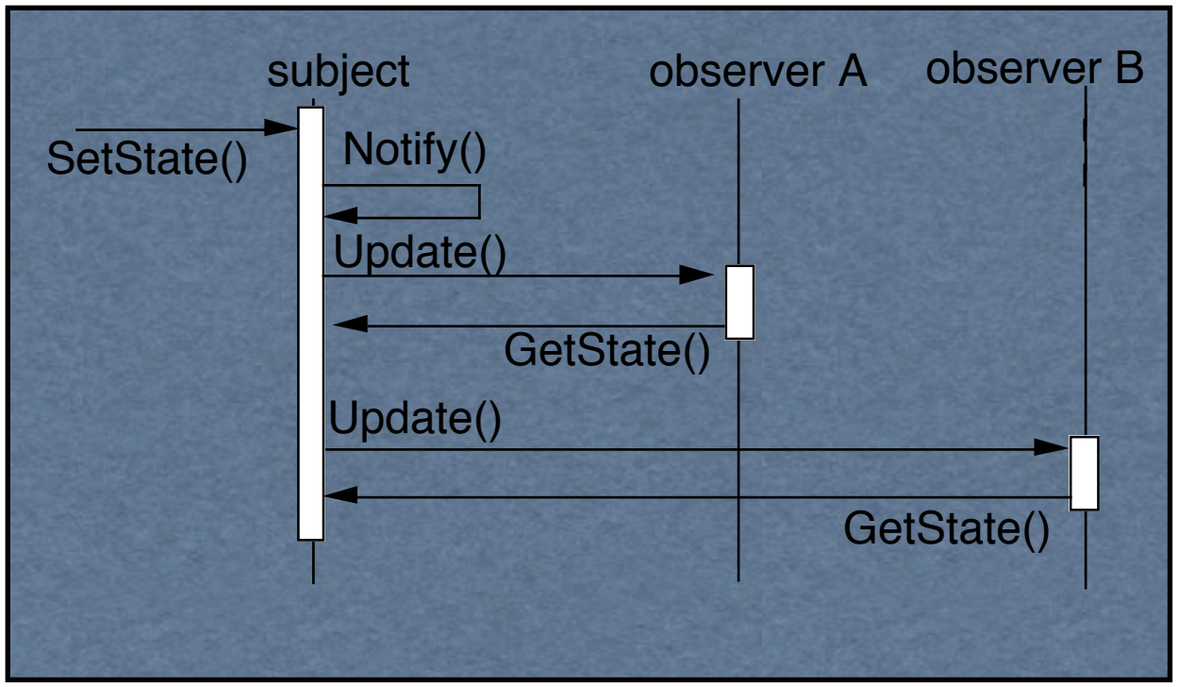
When one object changes state,

all its dependents are notified and updated automatically

Structure


observer A
observer B
Update()
GetState()
Update()
SetState()
subject


Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
Observer
Update()
ConcreteObserver
Update()
observerState
subject
ConcreteSubject
GetState()
subjectState


subject
observer A
observer B
SetState()
Notify()
Update()
GetState()
Update()
GetState()

Common Java Example - Listeners

Java Interface

View.OnClickListener

abstract void onClick(View v)

Called when a view has been clicked.

Java Example

```
public class CreateUIInCodeActivity extends Activity implements View.OnClickListener{
    Button test;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    test = (Button) this.findViewById(R.id.test);
    test.setOnClickListener(this);
  }

  public void onClick(View source) {
    Toast.makeText(this, "Hello World", Toast.LENGTH_SHORT).show();
  }
}
```

Pseudo Java Example

```
public class Subject {
   Window display;
   public void someMethod() {
      this.modifyMyStateSomeHow();
      display.addText( this.text() );
   }
}
```

Abstract coupling - Subject & Observer

Broadcast communication

Updates can take too long

```
public class Subject {
   ArrayList observers = new ArrayList();

   public void someMethod() {
      this.modifyMyStateSomeHow();
      changed();
   }

   private void changed() {
      Iterator needsUpdate = observers.iterator();
      while (needsUpdate.hasNext() )
         needsUpdate.next().update( this );
   }
}

public class SampleWindow {
   public void update(Object subject) {
      text = ((Subject) subject).getText();
      Thread.sleep(10000).
   }
}
```

Some Language Support

Smalltalk	Java	Ruby	Clojure	Observer Pattern
Object	Observer		function	Abstract Observer class
Object & Model	Observable	Observable	watches on data	Subject class

Smalltalk Implementation

Object implements methods for both Observer and Subject.

Actual Subjects should subclass Model

Java's Observer

Class java.util.Observable

```
void  addObserver(Observer o)
void  clearChanged()
int           countObservers()
void  deleteObserver(Observer o)
void  deleteObservers()
boolean    hasChanged()
void  notifyObservers()
void  notifyObservers(Object arg)
void  setChanged()
```

Java	Observer Pattern
Interface Observer	Abstract Observer class
Observable class	Subject class

Observable object may have any number of Observers

Whenever the Observable instance changes,
it notifies all of its observers

Notification is done by calling the update() method on all observers.

Interface java.util.Observer

Allows all classes to be observable by instances of class Observer

Java Example

```
class Counter extends Observable  {
        public static final String INCREASE = "increase";
        public static final String DECREASE = "decrease";
        private int count = 0;
        private String label;

        public Counter( String label )    {       this.label = label; }

        public String label()                 {  return label; }
        public int value()                    {  return count; }
        public String toString()              {  return String.valueOf( count );}

        public void increase() {
                count++;
                setChanged();
                notifyObservers( INCREASE );
        }

        public void decrease()  {
                count--;
                setChanged();
                notifyObservers( DECREASE );
                }
        }
```

Java Observer

```
class IncreaseDetector implements Observer {
        public void update( java.util.Observable whatChanged,
                                                java.lang.Object message) {
                if ( message.equals( Counter.INCREASE) )  {
                        Counter increased = (Counter) whatChanged;
                        System.out.println( increased.label() + " changed to " +
                                                                increased.value());
                }
        }

        public static void main(String[] args) {
                Counter test = new Counter();
                IncreaseDetector adding = new IncreaseDetector();
                test.addObserver(adding);
                test.increase();
}
```

Ruby Example

```
require'observer'

class Counter
    include Observable

    attr_reader :count

    def initialize
        @count = 0
    end
    def increase
        @count += 1
        changed
        notify_observers(:INCREASE)
    end

    def decrease
        @count -= 1
        changed
        notify_observers(:DECREASE)
    end
end
```

```
class IncreaseDetector

    def update(type)
        if type == :INCREASE
            puts('Increase')
        end
    end
end

count = Counter.new()
puts count.count
count.add_observer(IncreaseDetector.new)
count.increase
count.increase
puts count.count
```

Implementation Issues

Mapping subjects(Observables) to observers

Use list in subject

Use hash table

```
public class Observable {
    private boolean changed = false;
    private Vector obs;

public Observable() {
        obs = new Vector();
    }

public synchronized void addObserver(Observer o) {
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }
```

Observing more than one subject

If an observer has more than one subject how does it know which one changed?

Pass information in the update method

Deleting Subjects

In C++ the subject may no longer exist

Java/Smalltalk observer may prevent subject from garbage collection

Who Triggers the update?

Have methods that change the state trigger update

```
class Counter extends Observable  {              // some code removed
        public void increase()  {
                count++;
                setChanged();
                notifyObservers( INCREASE );
        }
}
```

Have clients call Notify at the right time

```
class Counter extends Observable  {        // some code removed
                public void increase()  {    count++;   }
}

Counter pageHits = new Counter();
pageHits.increase();
pageHits.increase();
pageHits.increase();
pageHits.notifyObservers();
```

Subject is self-consistent before Notification

```
class ComplexObservable extends Observable {
        Widget frontPart = new Widget();
        Gadget internalPart = new Gadget();

        public void trickyChange() {
                frontPart.widgetChange();
                internalpart.anotherChange();
                setChanged();
                notifyObservers( );
        }
}

class MySubclass extends ComplexObservable {
        Gear backEnd = new Gear();

        public void trickyChange() {
                super.trickyChange();
                backEnd.yetAnotherChange();
                setChanged();
                notifyObservers( );
        }
}
```

Adding information about the change

push models - add parameters in the update method

```
class IncreaseDetector extends Counter implements Observer { // stuff not shown

        public void update( Observable whatChanged, Object message) {
                if ( message.equals( INCREASE) )
                        increase();
        }
}

class Counter extends Observable {          // some code removed
        public void increase()  {
                count++;
                setChanged();
                notifyObservers( INCREASE );
        }
}
```

Adding information about the change

pull model - observer asks Subject what happened

```
class IncreaseDetector extends Counter implements Observer {
        public void update( Observable whatChanged ) {
                if ( whatChanged.didYouIncrease() )
                        increase();
        }
}

class Counter extends Observable {          // some code removed
        public void increase() {
                count++;
                setChanged();
                notifyObservers( );
        }
}
```

Scaling the Pattern

Java Event Model

AWT/Swing components broadcast events to Listeners

JDK1.0 AWT components broadcast an event to all its listeners

A listener normally not interested all events

Broadcasting to all listeners was too slow with many listeners

Java 1.1+ Event Model

Each component supports different types of events:

Component supports

ComponentEvent	FocusEvent
KeyEvent	MouseEvent

Each event type supports one or more listener types:

MouseEvent

MouseListener	MouseMotionListener

Each listener interface replaces update with multiple methods

MouseListener

mouseClicked()	mouseEntered()
mousePressed()	mouseReleased()

Listeners

- Only register for events of interest
- Don't need case statements to determine what happened

Small Models

Often an object has a number of fields(aspects) of interest to observers

Rather than make the object a subject make the individual fields subjects

- Simplifies the main object
- Observers can register for only the data they are interested in

VisualWorks ValueHolder

Subject for one value

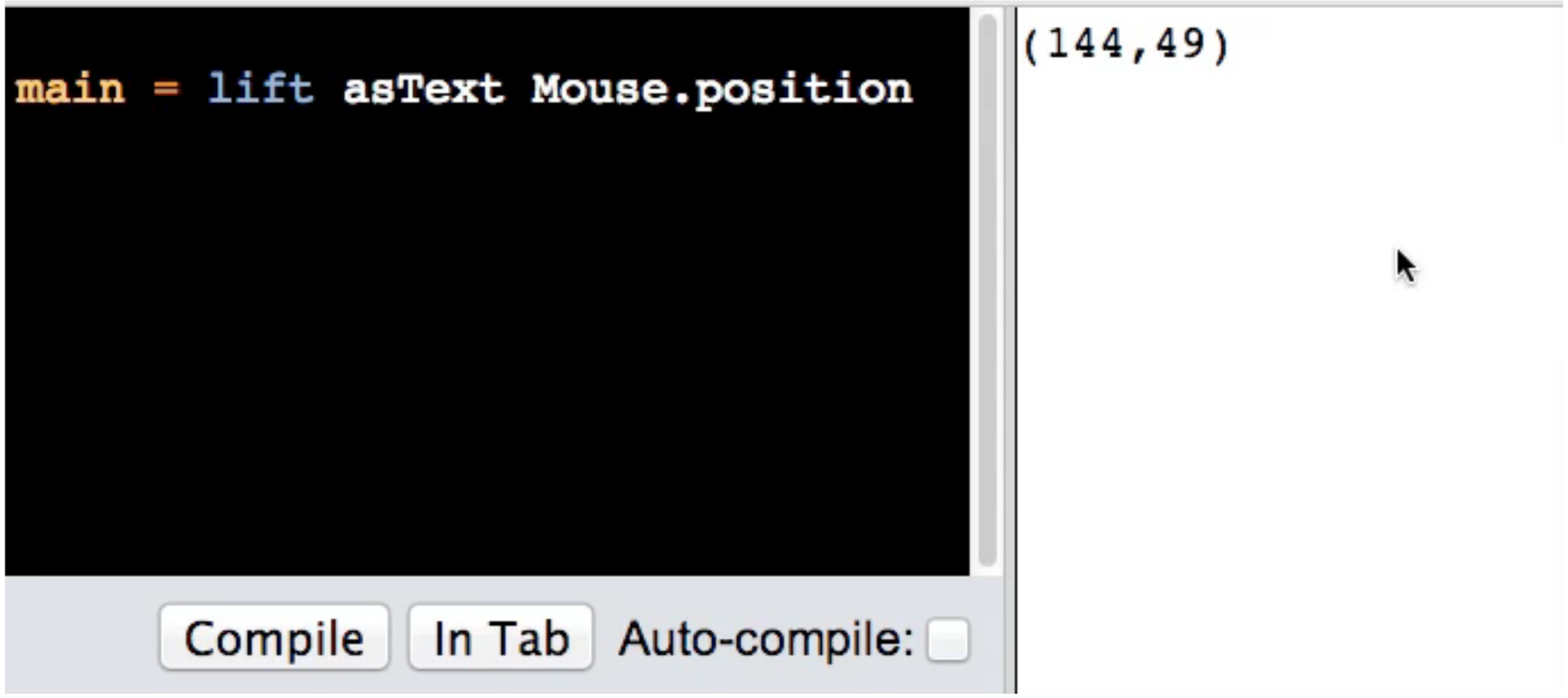
ValueHolder allows you to:

- Set/get the value
  - Setting the value notifies the observers of the change
- Add/Remove dependents

Reactive Programming

datatypes that represent a value 'over time'

Spreadsheets
Elm
Meteor.js



http://stackoverflow.com/questions/1028250/what-is-functional-reactive-programming
http://elm-lang.org/
http://en.wikipedia.org/wiki/Reactive_programming

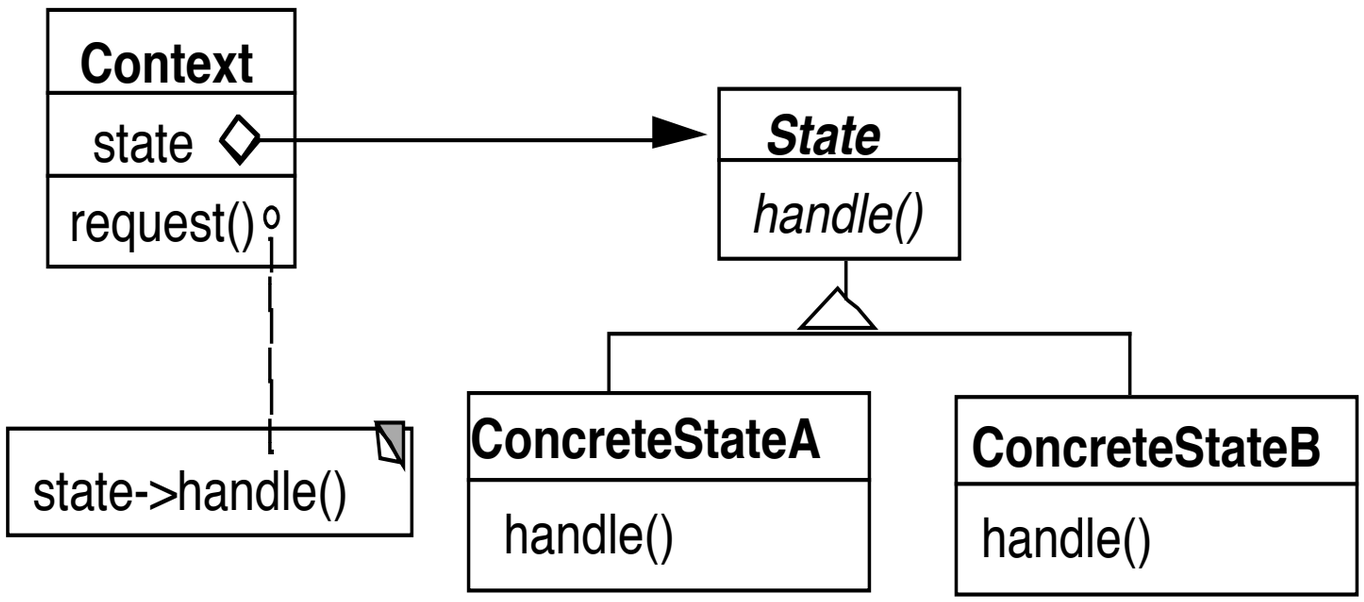
State

State Pattern

Allow an object to alter its behavior when its internal state changes

The object will appear to change it class

Structure


Context
state
request()
State
handle()
state->handle()
ConcreteStateA
handle()
ConcreteStateB
handle()

Grade Program

Operations

View assignment dates

Log in

View grades

Post grades

States

Not logged in

Valid operations

View dates, Log in

Invalid operations

View & post grades

Logged in - student

Valid operations

View dates & grades

Invalid operations

Post grades, log in

Logged in - instructor

Valid operations

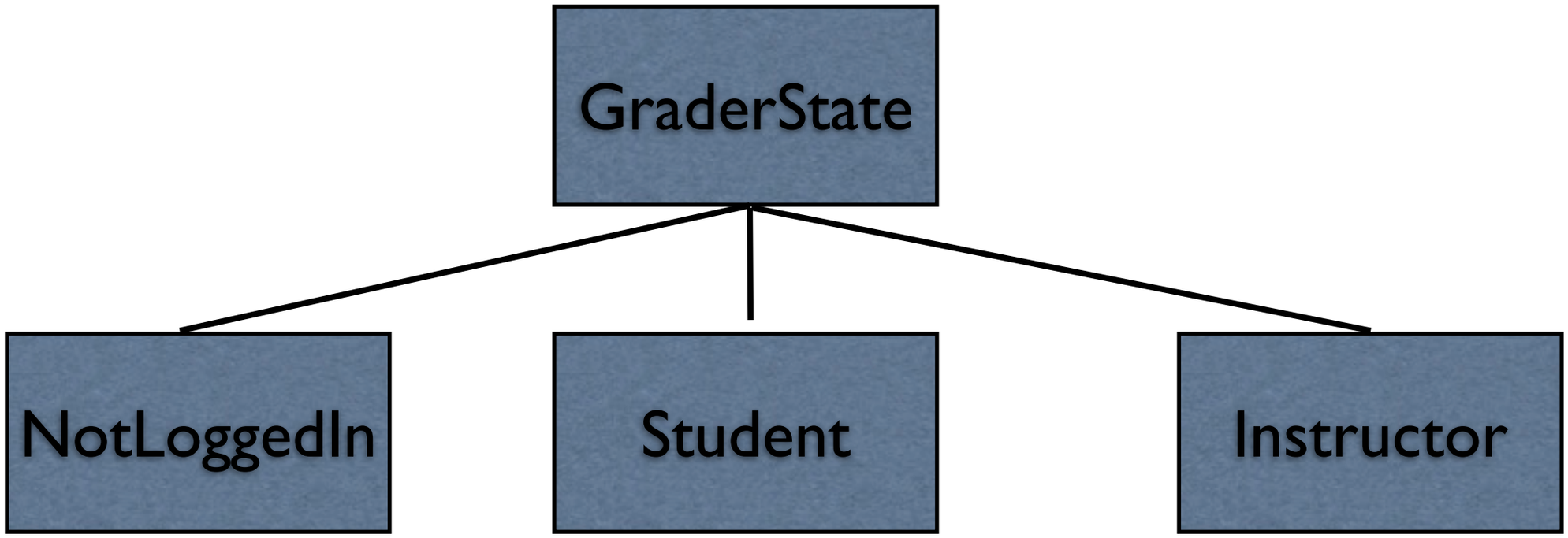
View dates & grades, post grades

Invalid operations

log in

```
public class Grader {
    static final int NOT_LOGGED_IN = 0;
    static final int STUDENT = 1;
    static final int INSTRUCTOR = 2;
    int state = NOT_LOGGED_IN;

    public viewGrades() {
        if (state == NOT_LOGGED_IN)
            redirectToLogin();
        if (state == STUDENT)
            showStudentGrade();
        if (state == INSTRUCTOR)
            showAllGrades();
    }
}
```

```
public postGrades() {
    if (state == NOT_LOGGED_IN)
        redirectToLogin();
    if (state == STUDENT)
        showError();
    if (state == INSTRUCTOR)
        getGradeFile();
}
```




```
public class GraderState {
    public GraderState login() {...}
    public GraderState viewGrades() {}
    public GraderState postGrades() {}
}
```

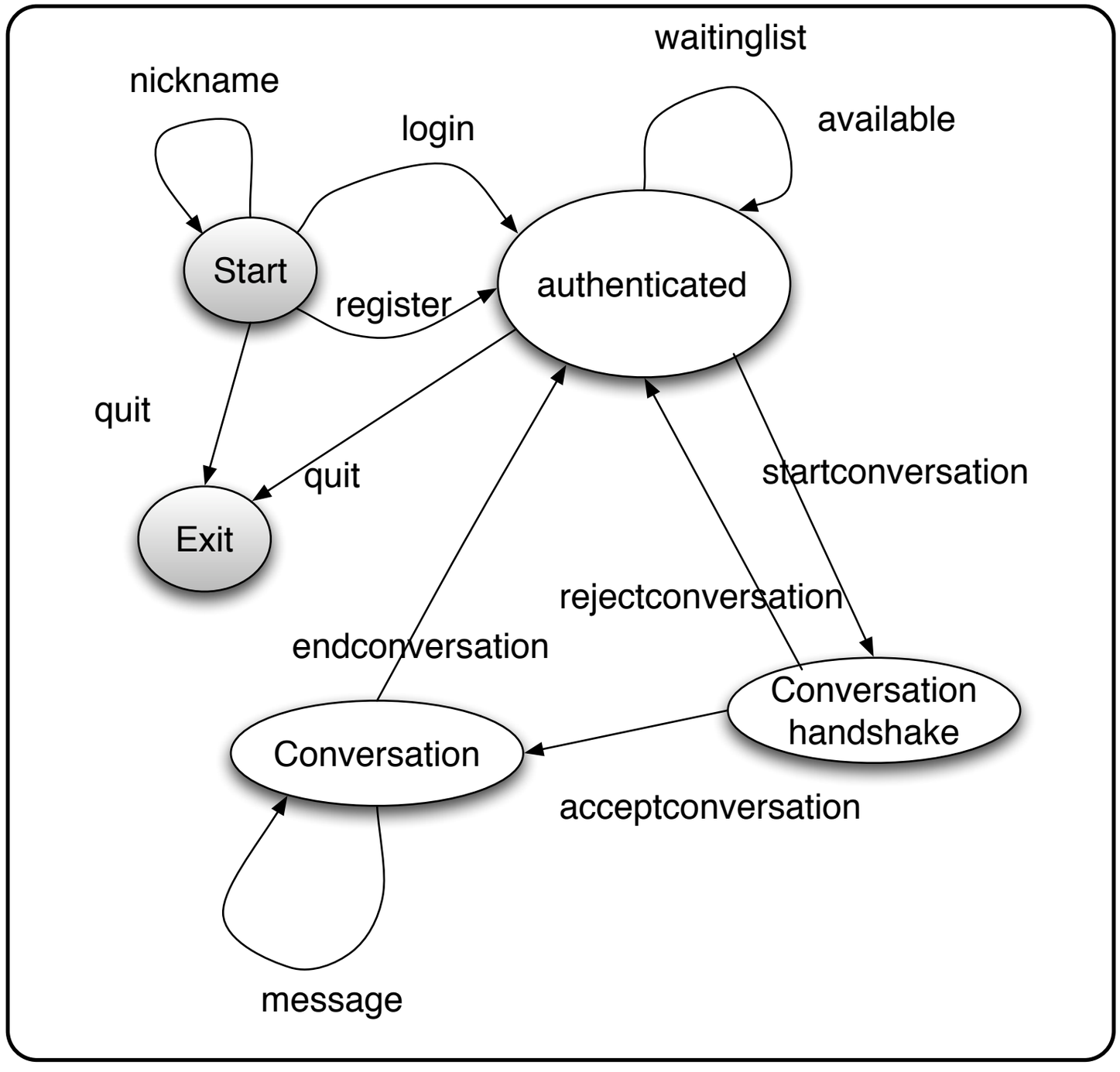
```
public class Grader {
    GraderState state = new NotLoggedIn();

    public void login() {
        state = state.login();
    }

    public viewGrades() {
        state = state.viewGrades();
    }
}
```

Example: SDChat Server

Commands

"available"
"login"
"register"
"nickname"
"startconversation"
"quit"
"waitinglist"
"acceptconversation"
"message"
"rejectconnection"
"endconversation"

Server States



Without States

```
public class SDChatServer {

    String handleNickname(String data) {
        if (state != START)
            return someErrorMessage();
        handle the main case
    }

    String handleLogin(String data) {
        if (state != START)
            return someErrorMessage();
        handle main case
    }

    String handleWaitinglist(String data) {
        if (state != AUTHENTICATED)
            return someErrorMessage();
        handle main case
    }
```

Who defines state Transitions - Context

```
class Context {
    private AbstractState state = new StartState();

    public Bar foo(int x) {
        int result = state.foo(x);
        if (someConditionHolds() )
            state = nextState();
        return result;
        }
```

Who defines state Transitions - States

```
class Context {
    private AbstractState state = new StartState();

    public void foo(int x) {
        state = state.foo(x);
        }
```

What if foo returns a value?

Who defines state Transitions - States

```
class Context {
    private AbstractState state = new StartState();

    public int foo(int x) {
        return state.foo(x, this);
        }

    protected void setState(AbstractState newState) {
        state = newState;
    }
```

Sharing State Objects

Stateless state

- State objects without fields
- Can be shared by multiple contexts

Can store date in context and pass as arguments

Large number of state transitions can be expensive

Only create state once & reuse same object

Changing Class - No Need for Context

Language Dependent Feature

Smalltalk & Lisp

```
class Truthful extends Oracle {

    public boolean foo(int x) {
        int result = state.foo(x);
        this.changeClassTo(Random);
        return result;
    }
```

State Verses Strategy

Rate of Change

Strategy

Context usually contains just one strategy object

State

Context often changes state objects

State Verses Strategy

Exposure of Change

Strategy

Strategies all do the same thing

Client do not see change in behavior of Context

State

States act differently

Client see the change in behavior

Multiple Dispatch & State Pattern

```
(defmulti view-grades (fn [user] (:state user))

(defmethod view-grades :not-logged-in
    [user]
    (go-to-log-in-page user))

(defmethod view-grades :student
    [user]
    (student-grade user))

(defmethod view-grades :instructor
    [user]
    (all-course-grades user)))
```