CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2013
Doc 9 Pattern Intro, Observer
Feb 25, 2014

References

A Pattern Language, Christopher Alexander, 1977

Patterns for Classroom Education, Dana Anthony, pp. 391-406, Pattern Languages of Program Design 2, Addison Wesley, 1996

Smalltalk Best Practice Patterns, Kent Beck, 1997

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995

Pattern Beginnings

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution"

A Pattern Language, Christopher Alexander, 1977

A Place To Wait

The process of waiting has inherent conflicts in it.

Waiting for doctor, airplane etc. requires spending time hanging around doing nothing

Cannot enjoy the time since you do not know when you must leave

Classic "waiting room"

Dreary little room
People staring at each other
Reading a few old magazines
Offers no solution

Fundamental problem

How to spend time "wholeheartedly" and
Still be on hand when doctor, airplane etc arrive

Fuse the waiting with other activity that keeps them in earshot

- Playground beside Pediatrics Clinic
- Horseshoe pit next to terrace where people waited

Allow the person to become still meditative

- A window seat that looks down on a street
- A protected seat in a garden
- A dark place and a glass of beer
- A private seat by a fish tank

A Place To Wait

Therefore:

"In places where people end up waiting create a situation which makes the waiting positive. Fuse the waiting with some other activity - newspaper, coffee, pool tables, horseshoes; something which draws people in who are not simple waiting. And also the opposite: make a place which can draw a person waiting into a reverie; quiet; a positive silence"

Chicken And Egg

Problem

Two concepts are each a prerequisite of the other
To understand A one must understand B
To understand B one must understand A
A "chicken and egg" situation

Constraints and Forces

First explain A then B

Everyone would be confused by the end

Simplify each concept to the point of incorrectness to explain the other one

People don't like being lied to

Solution

Explain A & B correctly by superficially

Iterate your explanations with more detail in each iteration

Patterns for Classroom Education, Dana Anthony, pp. 391-406, Pattern Languages of Program Design 2, Addison Wesley, 1996

Design Principle 1

Program to an interface, not an implementation

Use abstract classes (and/or interfaces in Java) to define common interfaces for a set of classes

Declare variables to be instances of the abstract class not instances of particular classes

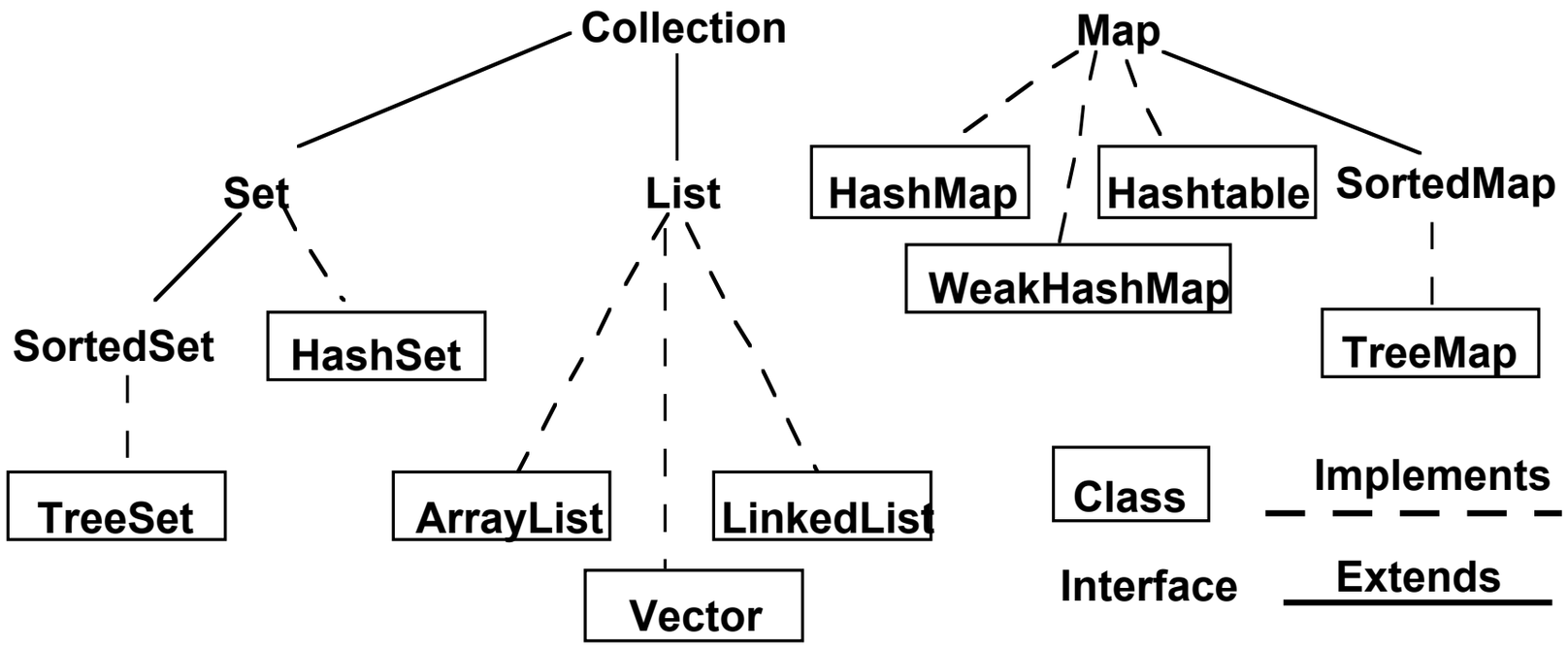
Benefits of programming to an interface

Client classes/objects remain unaware of the classes of objects they use, as long as the objects adhere to the interface the client expects

Client classes/objects remain unaware of the classes that implement these objects.

Clients only know about the abstract classes (or interfaces) that define the interface.

Programming to an Interface



Collection students = new XXX;
students.add(aStudent);

students can be any collection type

We can change our mind on what type to use

Interface & Duck Typing

In dynamically typed languages programming to an interface is the norm

Dynamically typed languages tend to lack a way to declare an interface

Design Principle 2

Favor object composition over class inheritance

Composition

- Allows behavior changes at run time
- Helps keep classes encapsulated and focused on one task
- Reduce implementation dependencies

Inheritance

```
class A {
      Foo x
      public int complexOperation() {
      blah }
}

class B extends A {
      public void bar() { blah}
}
```

Composition

```
class B {
      A myA;
      public int complexOperation() {
            return myA.complexOperation()
      }

      public void bar() { blah}
}
```

Designing for Change

Creating an object by specifying a class explicitly

Abstract factory, Factory Method, Prototype

Dependence on hardware and software platforms

Abstract factory, Bridge

Dependence on object representations or implementations

Abstract factory, Bridge, Memento, Proxy

Algorithmic dependencies

Builder, Iterator, Strategy, Template Method, Visitor

Tight Coupling

Abstract factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

Extending functionality by subclassing

Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

Dependence on specific operations

Chain of Responsibility, Command

Inability to alter classes conveniently

Adapter, Decorator, Visitor

Kent Beck's Rules for Good Style

One and only once

In a program written in good style, everything is said once and only once

Methods with the same logic
Objects with same methods
Systems with similar objects

rule is not satisfied

Lots of little Pieces

"Good code invariably has small methods and small objects"

Small pieces are needed to satisfy "once and only once"

Make sure you communicate the big picture or you get a mess

Rates of change

Don't put two rates of change together

An object should not have a field that changes every second & a field that change once a month

A collection should not have some elements that are added/removed every second and some that are add/removed once a month

An object should not have code that has to change for each piece of hardware and code that has to change for each operating system

Replacing Objects

Good style leads to easily replaceable objects

"When you can extend a system solely by adding new objects without modifying any existing objects, then you have a system that is flexible and cheap to maintain"

Moving Objects

"Another property of systems with good style is that their objects can be easily moved to new contexts"

Observer

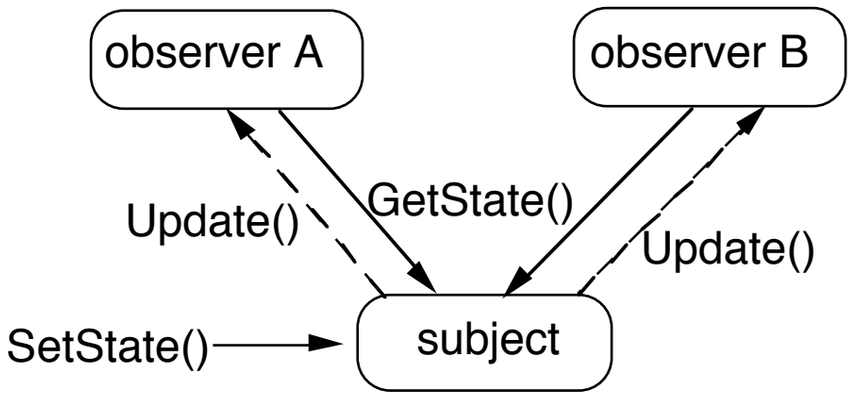
Observer

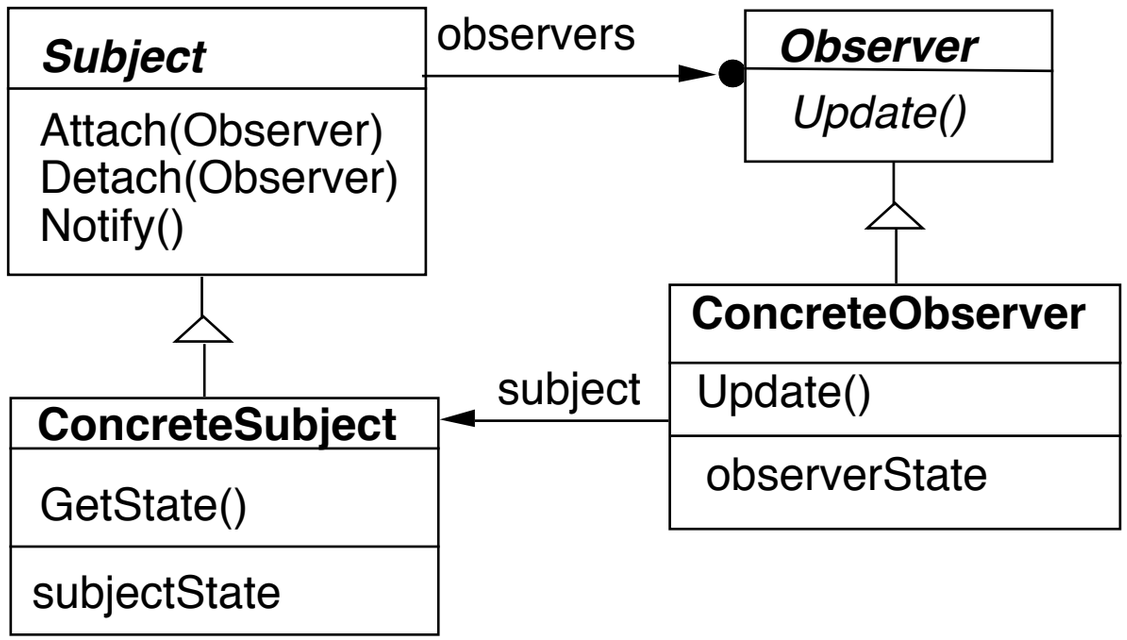
One-to-many dependency between objects

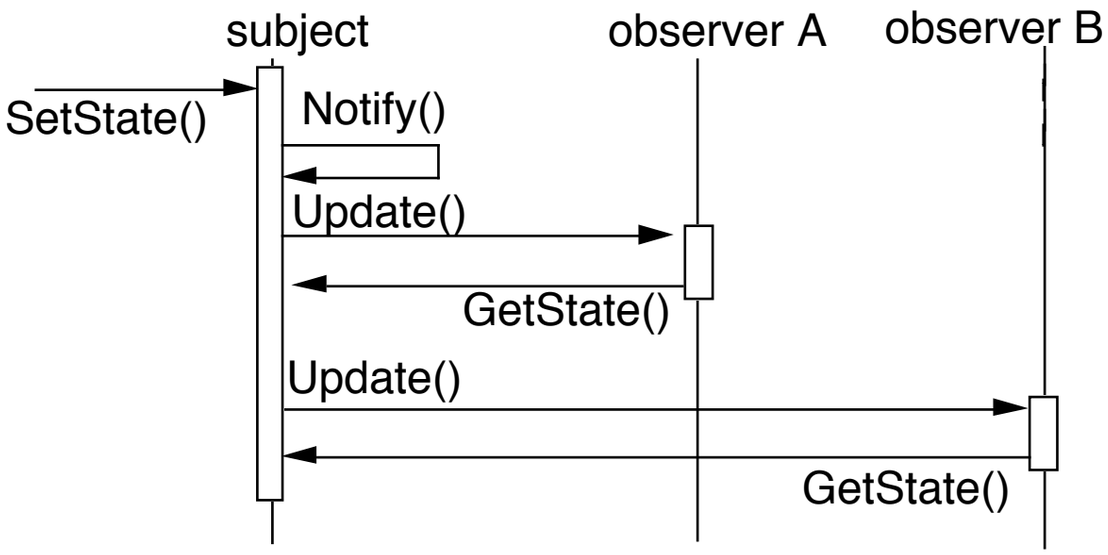
When one object changes state,

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	Observer Pattern
Object	Observer		Abstract Observer class
Object & Model	Observable	Observable	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

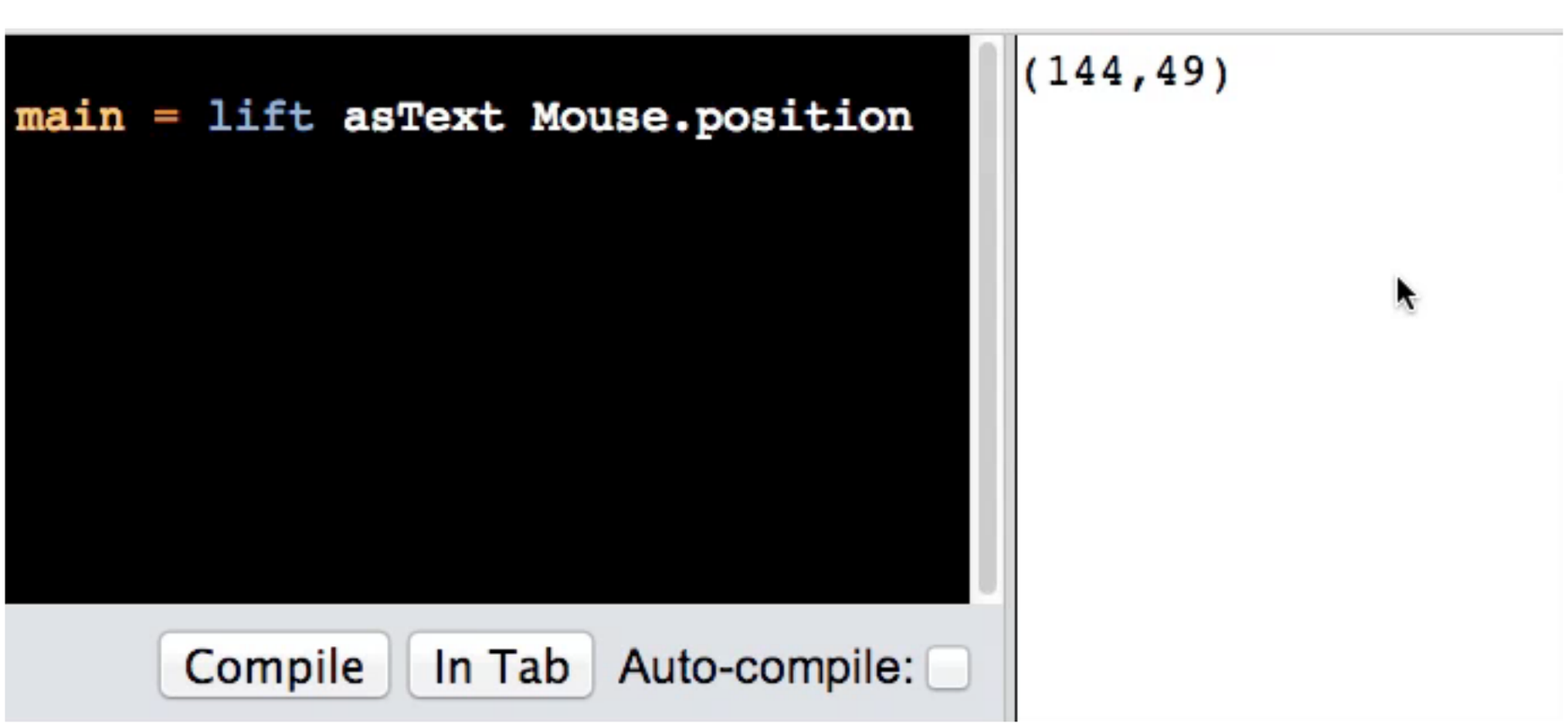
ValueHolder allows you to:

- 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