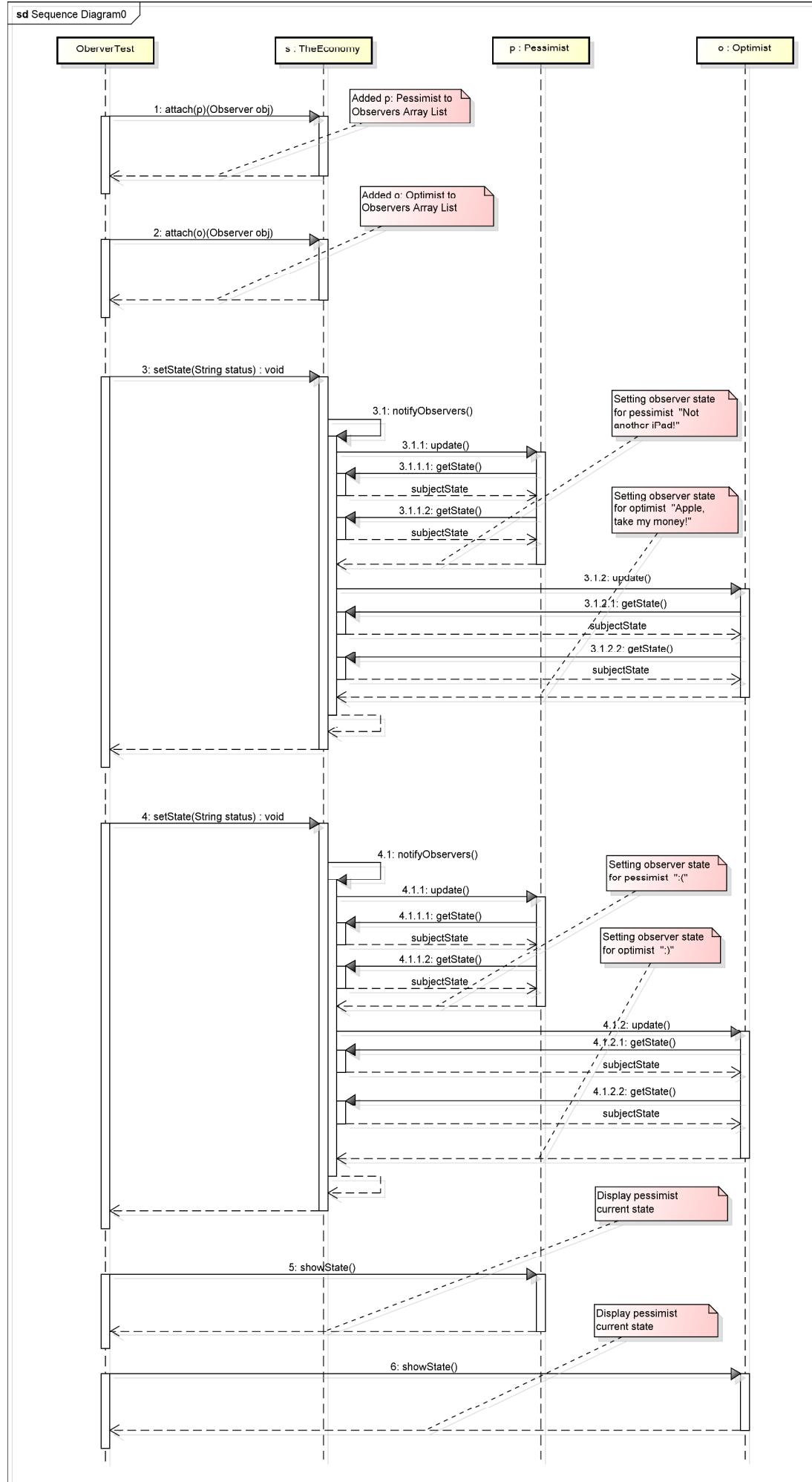


Lab #3 -
Design Patterns Lab
Mukul Ambulgekar

Student Id -009991254



1. Attach pessimist p to the observers Array list

```


    * Called before every test case method.
    */
@Before
public void setUp()
{
}

/**
 * Tears down the test fixture.
 *
 * Called after every test case method.
 */
@After
public void tearDown()
{
}

@Test
public void test1()
{
    observer.TheEconomy s = new observer.TheEconomy();
    observer.Pessimist p = new observer.Pessimist(s);
    observer.Optimist o = new observer.Optimist(s);
    s.attach(p);
    s.attach(o);
    s.setState("The New iPad is out today");
    s.setState("Hey, Its Friday!");

    p.showState();
    o.showState();
}
}


```

Thread "main" stopped at breakpoint.

2. Added pessimist p to observers Array list

```


package observer;

import java.util.ArrayList;

public class ConcreteSubject implements Subject {
    private String subjectState;
    private ArrayList<Observer> observers = new ArrayList<Observer>();

    public String getState() {
        return subjectState;
    }

    public void setState(String status) {
        subjectState = status;
        notifyObservers();
    }

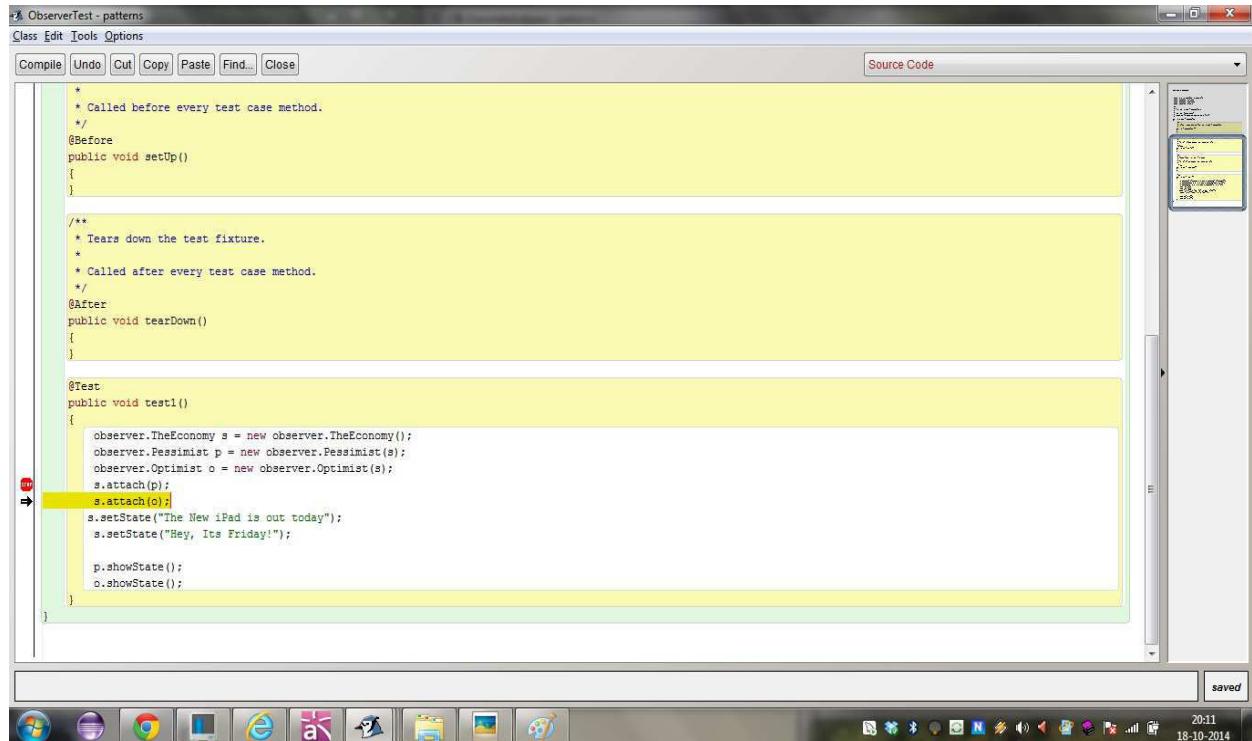
    public void attach(Observer obj) {
        observers.add(obj);
    }

    public void detach(Observer obj) {
        observers.remove(obj);
    }

    public void notifyObservers() {
        for (Observer obj : observers) {
            obj.update();
        }
    }
}


```

3. Attach optimist o to the observers Array list



```


    * Called before every test case method.
    */
@Before
public void setUp()
{
}

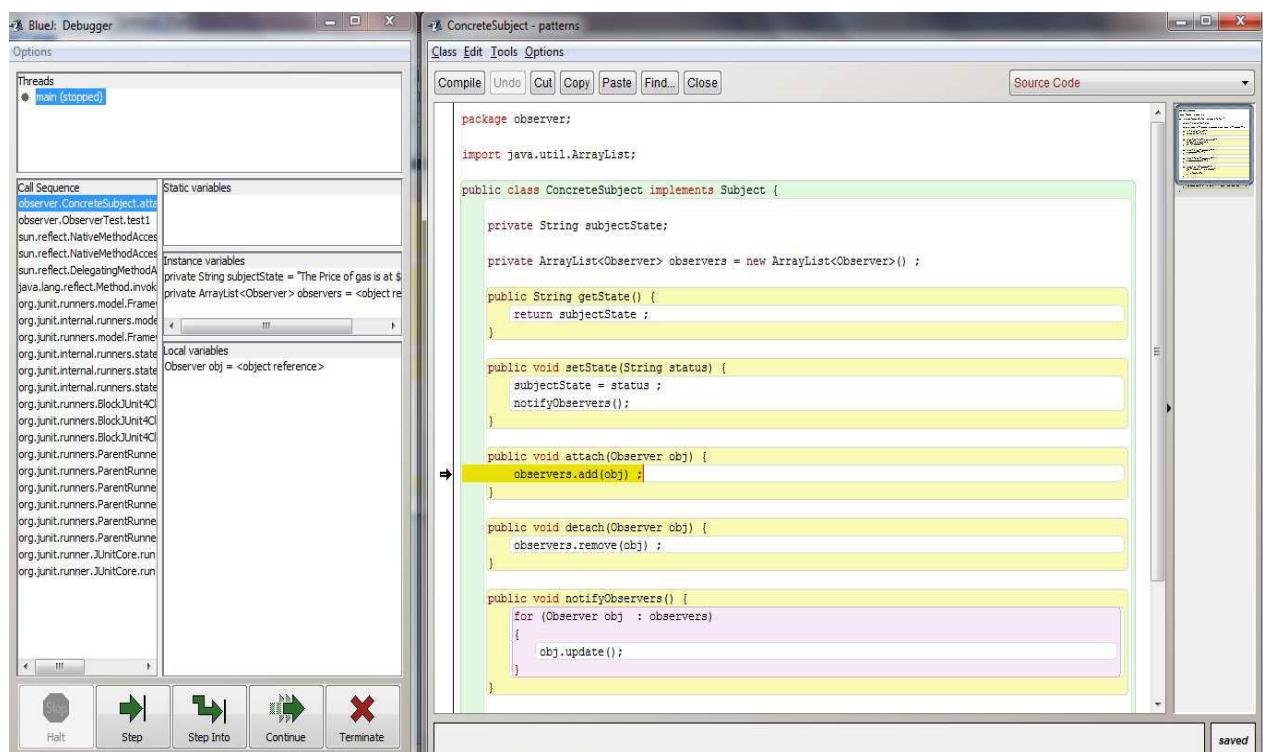
/**
 * Tears down the test fixture.
 *
 * Called after every test case method.
 */
@After
public void tearDown()
{
}

@Test
public void test1()
{
    observer.TheEconomy s = new observer.TheEconomy();
    observer.Pessimist p = new observer.Pessimist(s);
    observer.Optimist o = new observer.Optimist(s);
    s.attach(p);
    s.attach(o);
    s.setState("The New iPad is out today");
    s.setState("Hey, Its Friday!");

    p.showState();
    o.showState();
}
}


```

4. Added pessimist p to observers Array list



```


package observer;

import java.util.ArrayList;

public class ConcreteSubject implements Subject {

    private String subjectState;

    private ArrayList<Observer> observers = new ArrayList<Observer>();

    public String getState() {
        return subjectState;
    }

    public void setState(String status) {
        subjectState = status;
        notifyObservers();
    }

    public void attach(Observer obj) {
        observers.add(obj);
    }

    public void detach(Observer obj) {
        observers.remove(obj);
    }

    public void notifyObservers() {
        for (Observer obj : observers) {
            obj.update();
        }
    }
}


```

5. setState (String Status)

```

/*
 * Called before every test case method.
 */
@Before
public void setUp()
{
}

/**
 * Tears down the test fixture.
 *
 * Called after every test case method.
 */
@After
public void tearDown()
{
}

@Test
public void test1()
{
    observer.TheEconomy s = new observer.TheEconomy();
    observer.Pessimist p = new observer.Pessimist(s);
    observer.Optimist o = new observer.Optimist(s);
    s.attach(p);
    s.attach(o);
    s.setState("The New iPad is out today");
    s.setState("Hey, Its Friday!");

    p.showState();
    o.showState();
}
}

```

6. Setting subject State

```

package observer;

import java.util.ArrayList;

public class ConcreteSubject implements Subject {
    private String subjectState;

    private ArrayList<Observer> observers = new ArrayList<Observer>();

    public String getState() {
        return subjectState;
    }

    public void setState(String status) {
        subjectState = status;
        notifyObservers();
    }

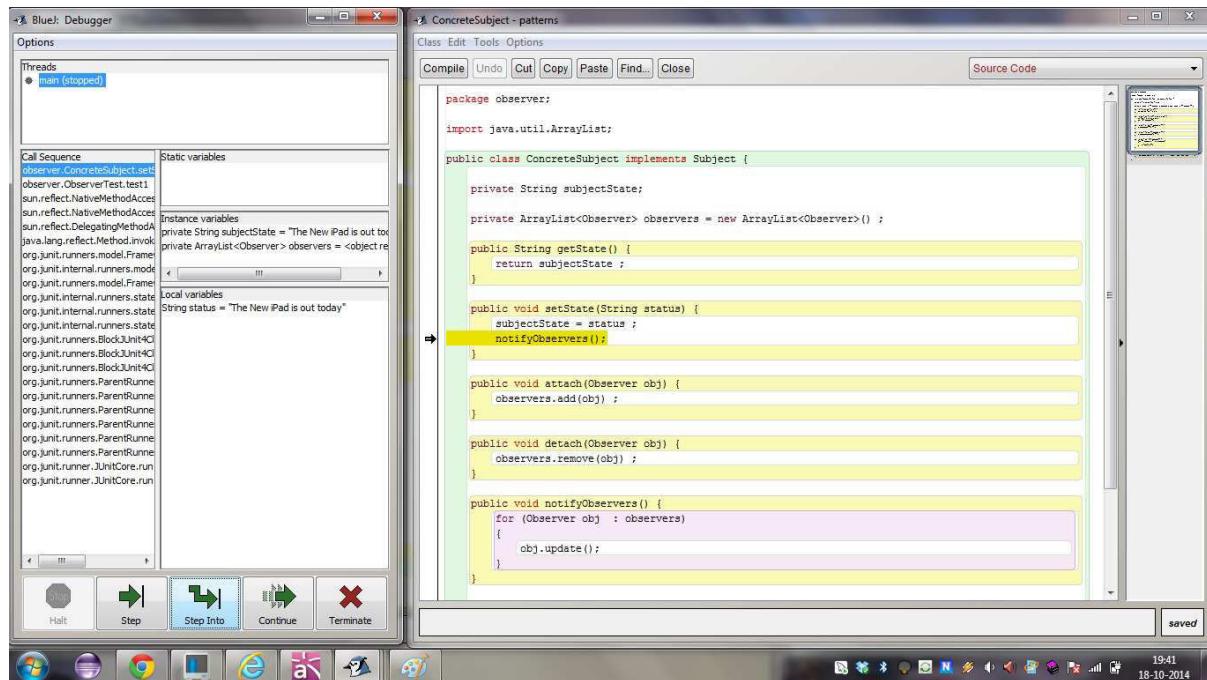
    public void attach(Observer obj) {
        observers.add(obj);
    }

    public void detach(Observer obj) {
        observers.remove(obj);
    }

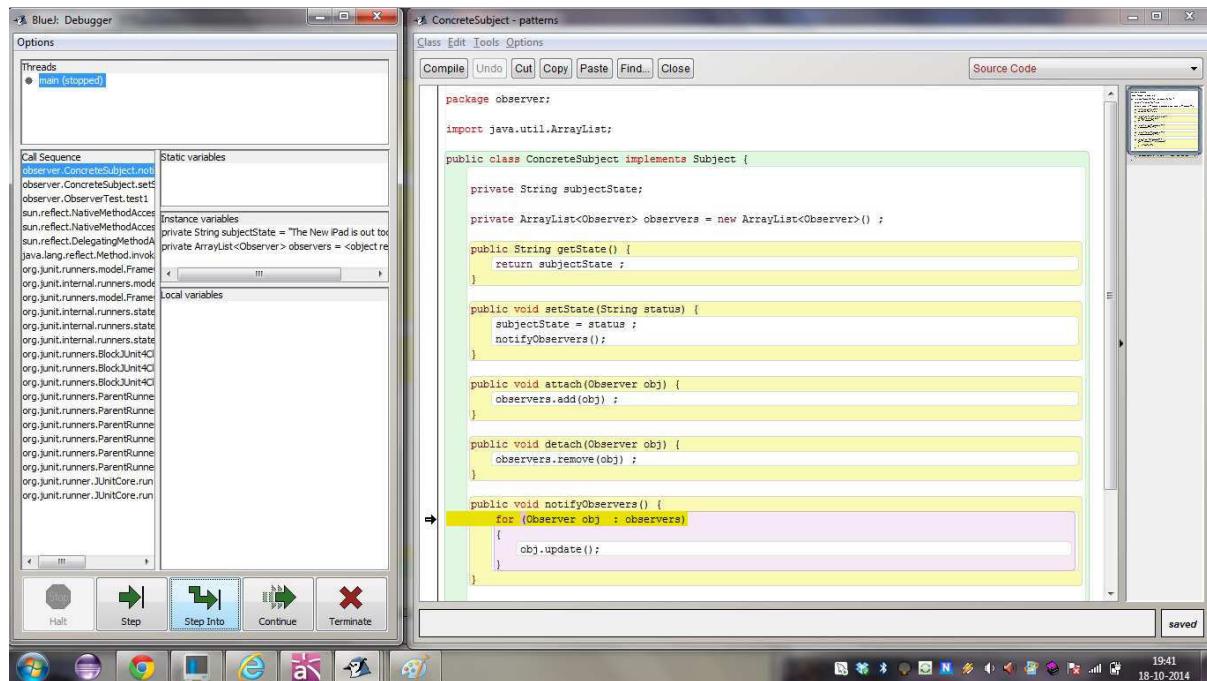
    public void notifyObservers() {
        for (Observer obj : observers) {
            obj.update();
        }
    }
}

```

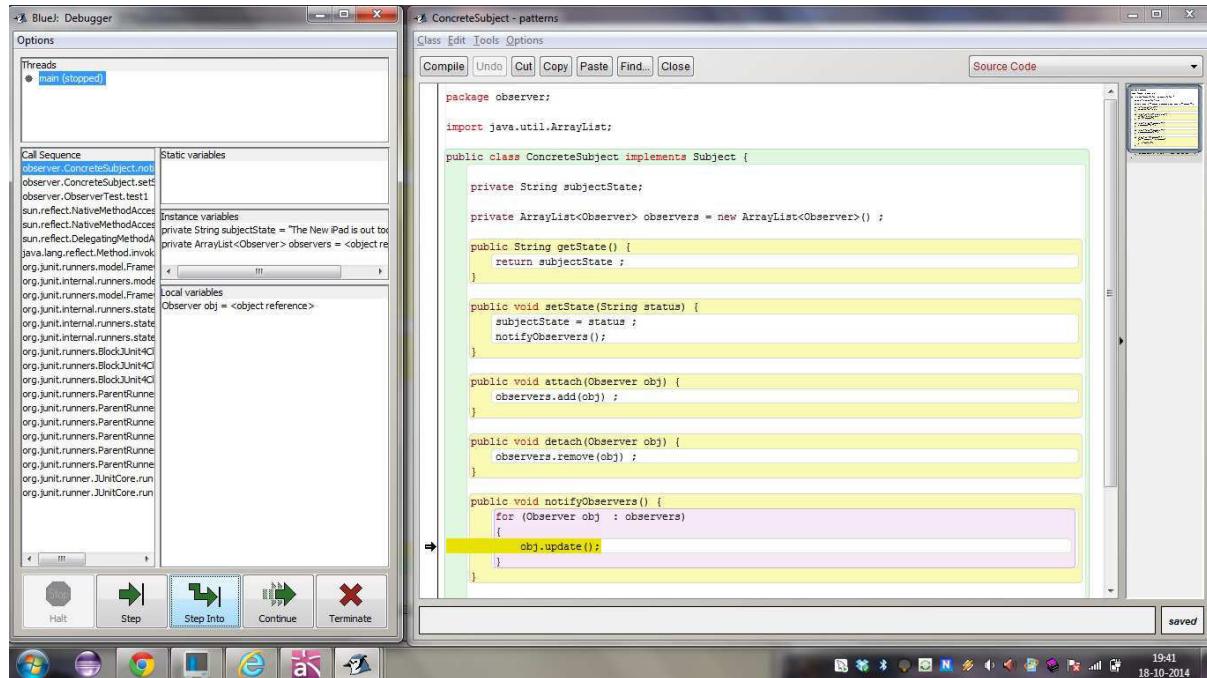
7. Calling notify observers.



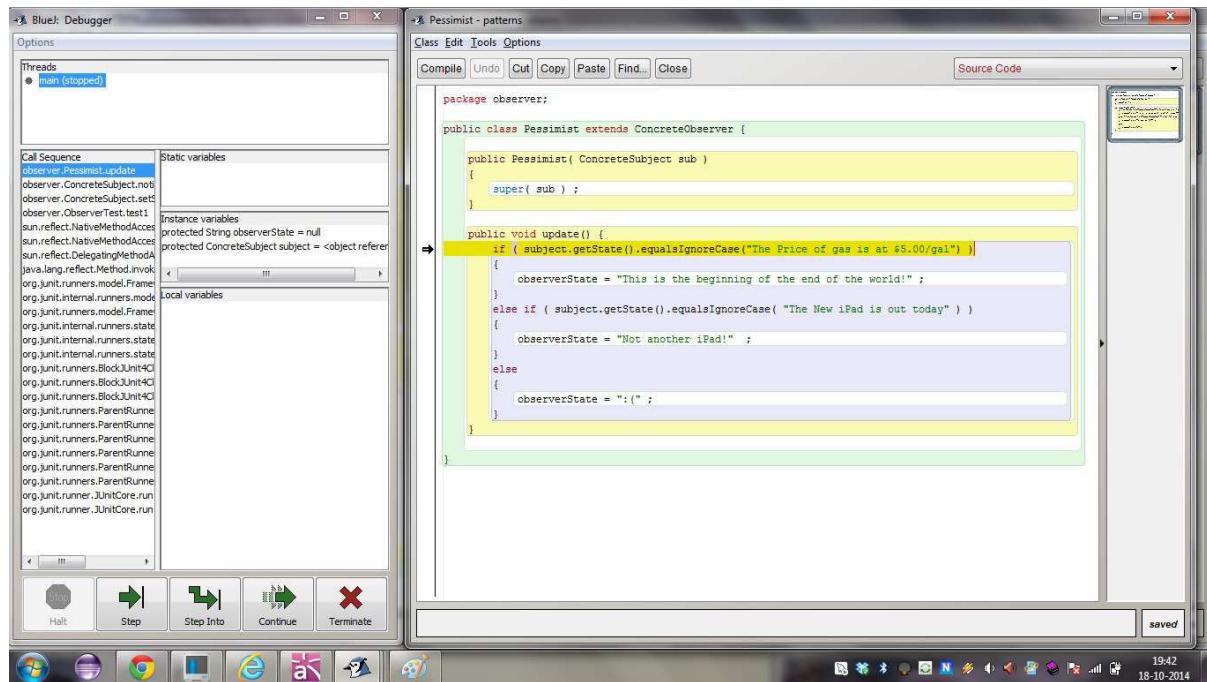
8. Notifying all observers



9. Updating observer's state.



10. Getting subject state.



11. Returning subject state

The screenshot shows the BlueJ IDE interface. On the left is the 'ConcreteSubject' class, which implements the 'Subject' interface. The code defines a private string 'subjectState' and a private ArrayList of observers. It includes methods for getting the state, setting the state, attaching observers, detaching observers, and notifying all observers. On the right is the 'ConcreteSubject - patterns' window, showing the concrete implementation of the pattern.

```

package observer;

import java.util.ArrayList;

public class ConcreteSubject implements Subject {
    private String subjectState;
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    public String getState() {
        return subjectState;
    }
    public void setState(String status) {
        subjectState = status;
        notifyObservers();
    }
    public void attach(Observer obj) {
        observers.add(obj);
    }
    public void detach(Observer obj) {
        observers.remove(obj);
    }
    public void notifyObservers() {
        for (Observer obj : observers) {
            obj.update();
        }
    }
}

```

12. Getting subject state.

The screenshot shows the BlueJ IDE interface. On the left is the 'Pessimist' class, which extends 'ConcreteObserver'. It overrides the 'update' method to check if the subject's state equals 'The New iPad is out today'. If it does, it sets the observer state to 'Not another iPad!'. Otherwise, it sets it to a colon. On the right is the 'Pessimist - patterns' window, showing the concrete implementation of the pattern.

```

package observer;

public class Pessimist extends ConcreteObserver {
    public Pessimist(ConcreteSubject sub) {
        super(sub);
    }
    public void update() {
        if (subject.getState().equalsIgnoreCase("The New iPad is out today")) {
            observerState = "Not another iPad!";
        } else {
            observerState = ":" ;
        }
    }
}

```

13. Setting Pessimist observer's state

The screenshot shows the BlueJ IDE interface. On the left, the 'Threads' window shows a single thread named 'main (stopped)'. On the right, the 'Pessimist - patterns' window displays the following Java code:

```

package observer;

public class Pessimist extends ConcreteObserver {
    public Pessimist( ConcreteSubject sub )
    {
        super( sub );
    }

    public void update()
    {
        if ( subject.getState().equalsIgnoreCase("The Price of gas is at $5.00/gal") )
        {
            observerState = "This is the beginning of the end of the world! ";
        }
        else if ( subject.getState().equalsIgnoreCase( "The New iPad is out today" ) )
        {
            observerState = "Not another iPad!";
        }
        else
        {
            observerState = ":(" ;
        }
    }
}

```

14. Setting Optimist observer's state

The screenshot shows the BlueJ IDE interface. On the left, the 'Threads' window shows a single thread named 'main (stopped)'. On the right, the 'Optimist - patterns' window displays the following Java code:

```

package observer;

public class Optimist extends ConcreteObserver {
    public Optimist( ConcreteSubject sub )
    {
        super( sub );
    }

    public void update()
    {
        if ( subject.getState().equalsIgnoreCase("The Price of gas is at $5.00/gal") )
        {
            observerState = "Great! It's time to go green. ";
        }
        else if ( subject.getState().equalsIgnoreCase( "The New iPad is out today" ) )
        {
            observerState = "Apple, take my money!";
        }
        else
        {
            observerState = ":)";
        }
    }
}

```

15. setState (String Status)

```


/*
 * Called before every test case method.
 */
@Before
public void setUp()
{
}

/**
 * Tears down the test fixture.
 *
 * Called after every test case method.
 */
@After
public void tearDown()
{
}

@Test
public void test1()
{
    observer.TheEconomy s = new observer.TheEconomy();
    observer.Pessimist p = new observer.Pessimist(s);
    observer.Optimist o = new observer.Optimist(s);
    s.attach(p);
    s.attach(o);
    s.setState("The New iPad is out today");
    s.setState("Hey, Its Friday!");

    p.showState();
    o.showState();
}
}


```

The screenshot shows a Java IDE window titled "ObserverTest - patterns". The code editor displays a test class with a single test method, `test1()`. Inside the method, there is a sequence of state transitions on an object `s` of type `TheEconomy`. The first transition sets the state to "The New iPad is out today", and the second transition sets it to "Hey, Its Friday!". Both transitions are highlighted with yellow boxes. A red arrow points from the first transition to the second. The code editor has tabs for "Compile", "Undo", "Cut", "Copy", "Paste", "Find...", and "Close". A "Source Code" tab is also visible. On the right side of the IDE, there is a "Project Explorer" view showing the file structure. The status bar at the bottom indicates the date and time as "18-10-2014 20:12".

16. Setting subject state.

The screenshot shows a Java IDE window with two panes. The left pane is a "BlueJ Debugger" window titled "ConcreteSubject - patterns". It displays a "Threads" tab with one thread named "main (stopped)". Below it are sections for "Call Sequence", "Static variables", "Instance variables", and "Local variables". The "Local variables" section shows a variable `subjectState` with the value "The New iPad is out today". The right pane is a code editor titled "ConcreteSubject - patterns" with the following code:

```


package observer;

import java.util.ArrayList;

public class ConcreteSubject implements Subject {
    private String subjectState;
    private ArrayList<Observer> observers = new ArrayList<Observer>();

    public String getState() {
        return subjectState;
    }

    public void setState(String status) {
        subjectState = status;
        notifyObservers();
    }

    public void attach(Observer obj) {
        observers.add(obj);
    }

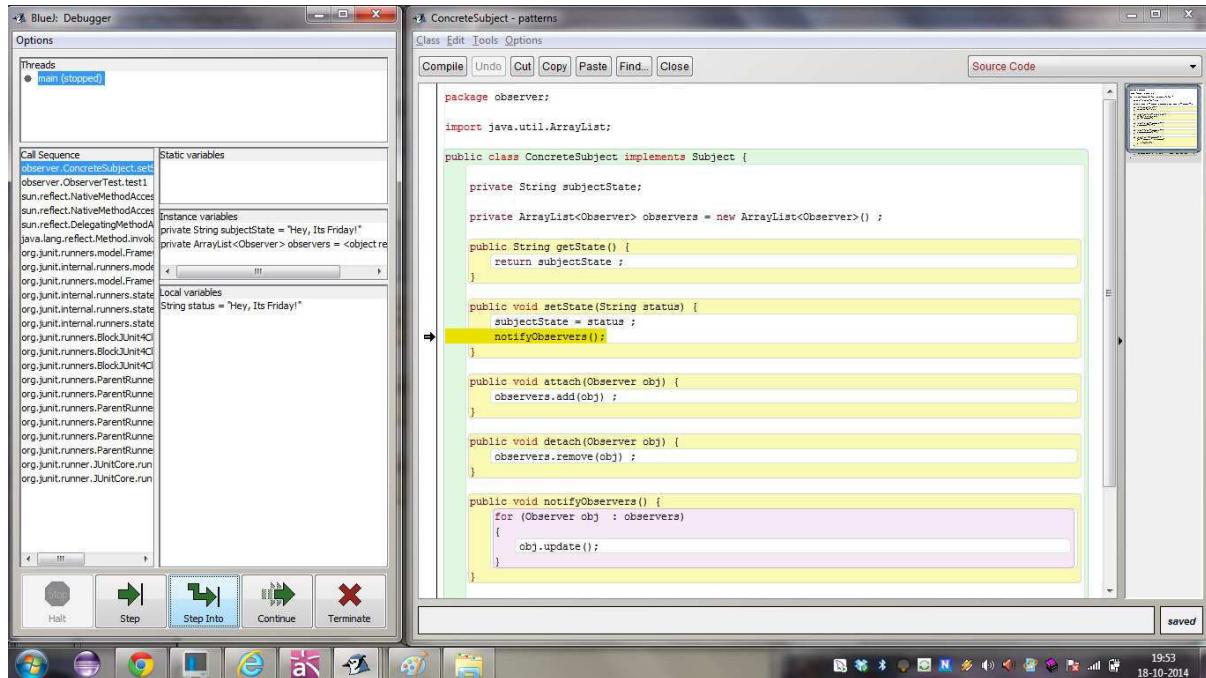
    public void detach(Observer obj) {
        observers.remove(obj);
    }

    public void notifyObservers() {
        for (Observer obj : observers) {
            obj.update();
        }
    }
}

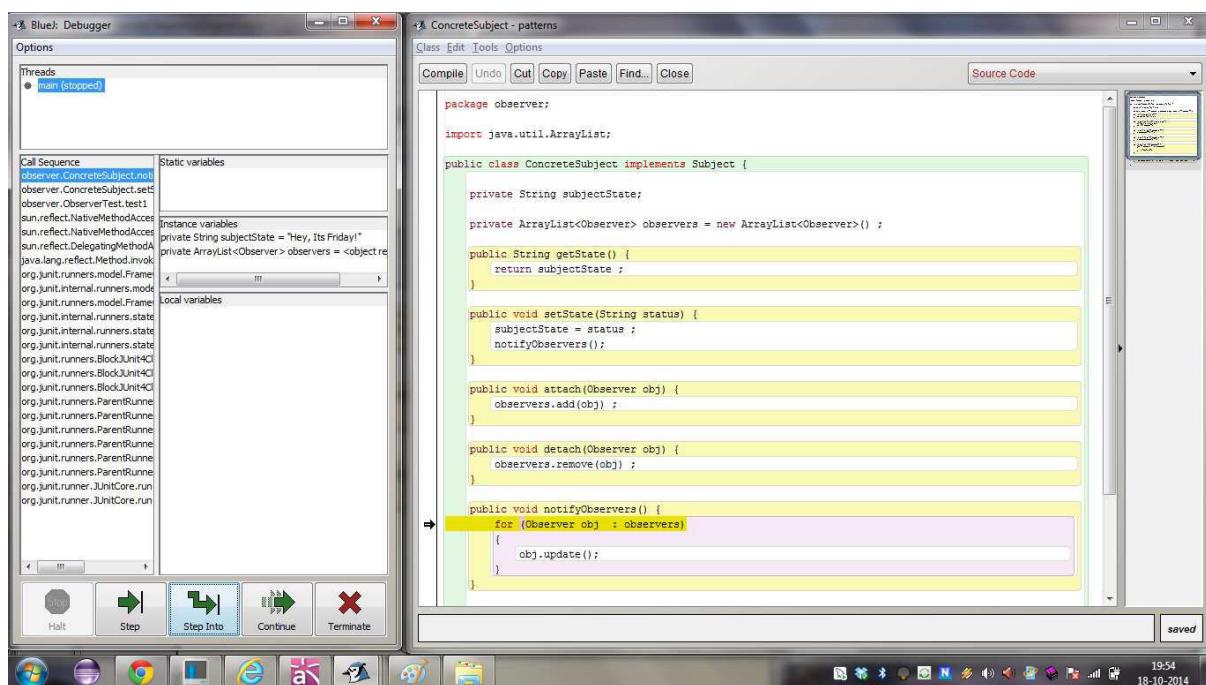

```

The code editor has tabs for "Compile", "Undo", "Cut", "Copy", "Paste", "Find...", and "Close". A "Source Code" tab is also visible. On the right side of the IDE, there is a "Project Explorer" view showing the file structure. The status bar at the bottom indicates the date and time as "18-10-2014 19:53".

17. Calling notify observers



18. Notifying all observer's.



19. Calling update observer's state.

The screenshot shows the BlueJ IDE interface. On the left is the 'Threads' window, which is currently stopped at the main thread. On the right is the 'ConcreteSubject - patterns' window, showing the source code for the ConcreteSubject class. The 'notifyObservers()' method is highlighted with a yellow background. The code is as follows:

```

package observer;

import java.util.ArrayList;

public class ConcreteSubject implements Subject {
    private String subjectState;
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    public String getState() {
        return subjectState;
    }
    public void setState(String status) {
        subjectState = status;
        notifyObservers();
    }
    public void attach(Observer obj) {
        observers.add(obj);
    }
    public void detach(Observer obj) {
        observers.remove(obj);
    }
    public void notifyObservers() {
        for (Observer obj : observers) {
            obj.update();
        }
    }
}

```

20. Getting subject state

The screenshot shows the BlueJ IDE interface. On the left is the 'Threads' window, which is currently stopped at the main thread. On the right is the 'Pessimist - patterns' window, showing the source code for the Pessimist class. The 'update()' method is highlighted with a yellow background. The code is as follows:

```

package observer;

public class Pessimist extends ConcreteObserver {
    public Pessimist(ConcreteSubject sub) {
        super(sub);
    }
    public void update() {
        if (subject.getState().equalsIgnoreCase("The Price of gas is an $5.00/gal")) {
            observerState = "This is the beginning of the end of the world!";
        } else if (subject.getState().equalsIgnoreCase("The New iPad is out today")) {
            observerState = "Not another iPad!";
        } else {
            observerState = ":"( ";
        }
    }
}

```

21. Returning subject state

```

package observer;

import java.util.ArrayList;

public class ConcreteSubject implements Subject {
    private String subjectState;
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    public String getState() {
        return subjectState;
    }
    public void setState(String status) {
        subjectState = status;
        notifyObservers();
    }
    public void attach(Observer obj) {
        observers.add(obj);
    }
    public void detach(Observer obj) {
        observers.remove(obj);
    }
    public void notifyObservers() {
        for (Observer obj : observers) {
            obj.update();
        }
    }
}

```

22. Getting subject state

```

package observer;

public class Pessimist extends ConcreteObserver {
    public Pessimist(ConcreteSubject sub) {
        super(sub);
    }
    public void update() {
        if (subject.getState().equalsIgnoreCase("The Price of gas is at $5.00/gal")) {
            observerState = "This is the beginning of the end of the world!";
        } else if (subject.getState().equalsIgnoreCase("The New iPad is out today")) {
            observerState = "Not another iPad!";
        } else {
            observerState = ":"( ";
        }
    }
}

```

23. Setting pessimist observers state

The screenshot shows the BlueJ IDE interface. On the left is the 'BlueJ: Debugger' window, which displays the call stack, static variables, instance variables, and local variables for the current thread. The 'Threads' tab shows one thread named 'main (stopped)'. The 'Call Sequence' tab shows the method call stack: 'observer.Pessimist.update'. The 'Source Code' tab contains the code for the Pessimist class:

```

package observer;
public class Pessimist extends ConcreteObserver {
    public Pessimist( ConcreteSubject sub )
    {
        super( sub );
    }

    public void update()
    {
        if ( subject.getState().equalsIgnoreCase("The Price of gas is at $5.00/gal") )
        {
            observerState = "This is the beginning of the end of the world!";
        }
        else if ( subject.getState().equalsIgnoreCase( "The New iPad is out today" ) )
        {
            observerState = "Not another iPad!";
        }
        else
        {
            observerState = ":(" ;
        }
    }
}

```

The code editor has syntax highlighting and a code completion dropdown visible. The status bar at the bottom right shows the time as 19:56 and the date as 18-10-2014.

24. Setting optimist observer state

The screenshot shows the BlueJ IDE interface again. The 'BlueJ: Debugger' window is identical to the previous one, showing the call stack and variable states for the 'main' thread. The 'Source Code' tab now contains the code for the Optimist class:

```

package observer;
public class Optimist extends ConcreteObserver {
    public Optimist( ConcreteSubject sub )
    {
        super( sub );
    }

    public void update()
    {
        if ( subject.getState().equalsIgnoreCase("The Price of gas is at $5.00/gal") )
        {
            observerState = "Great! It's time to go green.";
        }
        else if ( subject.getState().equalsIgnoreCase( "The New iPad is out today" ) )
        {
            observerState = "Apple, take my money!";
        }
        else
        {
            observerState = ":)" ;
        }
    }
}

```

The code editor shows the same syntax highlighting and code completion feature as before. The status bar at the bottom right shows the time as 19:59 and the date as 18-10-2014.

25. Calling show state for Pessimist observer

```

/*
 * Called before every test case method.
 */
@Before
public void setUp()
{
}

/**
 * Tears down the test fixture.
 *
 * Called after every test case method.
 */
@After
public void tearDown()
{
}

@Test
public void test1()
{
    observer.TheEconomy s = new observer.TheEconomy();
    observer.Pessimist p = new observer.Pessimist(s);
    observer.Optimist o = new observer.Optimist(s);
    s.attach(p);
    s.attach(o);
    s.setState("The New iPad is out today");
    s.setState("Hey, Its Friday!");

    p.showState();
    o.showState();
}
}

```

26. Displaying current state of pessimist

```

package observer;

public class ConcreteObserver implements Observer {
    protected String observerState;
    protected ConcreteSubject subject;

    public ConcreteObserver( ConcreteSubject theSubject ) {
        this.subject = theSubject ;
    }

    public void update() {
        // do nothing
    }

    public void showState() {
        System.out.println( "Observer: " + this.getClass().getName() + " = " + observerState );
    }
}

```

27. Calling show state for Optimist observer

```


/*
 * Called before every test case method.
 */
@Before
public void setUp()
{
}

/**
 * Tears down the test fixture.
 *
 * Called after every test case method.
 */
@After
public void tearDown()
{
}

@Test
public void test1()
{
    observer.TheEconomy s = new observer.TheEconomy();
    observer.Pessimist p = new observer.Pessimist(s);
    observer.Optimist o = new observer.Optimist(s);
    s.attach(p);
    s.attach(o);
    s.setState("The New iPad is out today");
    s.setState("Hey, Its Friday!");

    p.showState();
    o.showState();
}
}


```

28. Displaying current state of pessimist

```


package observer;

public class ConcreteObserver implements Observer {

    protected String observerState;
    protected ConcreteSubject subject;

    public ConcreteObserver( ConcreteSubject theSubject )
    {
        this.subject = theSubject ;
    }

    public void update()
    {
        // do nothing
    }

    public void showState()
    {
        System.out.println( "Observer: " + this.getClass().getName() + " = " + observerState );
    }
}


```