Real Time Patterns

Carlo U. Nicola, SGI FHNW
With extracts from publications of:
V. Huston, www.vincehuston.org;
A. Holub, www.holub.com;
Topics

1. Preliminary remarks
2. State pattern → Activity Life Cycle
3. Model View Controller pattern
4. Observation pattern → Activity Listener
5. Delegation pattern → Thread safe composition
6. Monitor object pattern → Thread safe composition
What are software patterns?

A pattern is the **abstraction** from a concrete form which keeps **recurring** in very **different** specific non-trivial **contexts**.
Why patterns?

The embedded world consists of reusable components centered around a network: that means distributed applications with a range of QoS needs constructed by integrating components (and frameworks) via different communication mechanisms.

Same pattern of components can solve different problems
State pattern: the abstract problem to solve

The State pattern is a solution to the problem of how to make behaviour depend on a state.

(1) Define a "context" class to present a single interface to the outside world.
(2) Define a State abstract base class.
(3) Represent the different "states" of the state machine as derived classes of the State base class.
(4) Define state-specific behaviour in the appropriate State derived classes.
(5) Maintain a pointer to the current "state" in the "context" class.
(6) Change state means changing the current "state" pointer.
The State pattern does not specify where the state transitions will be defined. The choices are two:

1. The "context" object, or
2. The derived class from each individual State.

The advantage of (2) is the ease of adding new State as derived classes.

The disadvantage of (2) is that each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses. [GoF, p. 308]

A table-driven approach to designing finite state machines does a good job of specifying state transitions, but it is difficult to add actions to accompany the state transitions. The pattern-based approach uses code (instead of data structures) to specify state transitions, but it does a good job of accommodating state transition actions. [GoF, p. 308]
The State pattern is a solution to the problem of how to make behaviour depend on state. Define a "context" class to present a single interface to the outside world. Define a State abstract base class. Represent the different "states" of the state machine as derived classes of the State base class.

A simple ceiling fan:
States: Velocity low, medium, fast

A ceiling fan with deployable wings.
States: deploy, low, medium, fast

A ceiling fan with velocity dependent deployable wings. States: Low, medium, fast and light on.
The class `CeilingFanPullChain` is a wrapper class that delegates to the wrapper class `State` the execution. Concrete classes `Off`, `Low`, etc. implements the real code to do the job.
class CeilingFanPullChain {
    private int currentState;

    public CeilingFanPullChain() {
        currentState = 0;
    }

    public void pull() {
        if (currentState == 0) {
            currentState = 1;
            System.out.println(" low speed");
        } else if (currentState == 1) {
            currentState = 2;
            System.out.println(" medium speed");
        } else if (currentState == 2) {
            currentState = 3;
            System.out.println(" high speed");
        } else {
            currentState = 0;
            System.out.println(" turning off");
        }
    }
}
class CeilingFanAsStatePattern {
    private State currentState;

    public CeilingFanAsStatePattern() {
        currentState = new Off();
    }

    public void setState(State s) {
        currentState = s;
    }

    public void pull() {
        currentState.pull(this);
    }
}
interface State {
    void pull(CeilingFanAsStatePattern wrapper);
}

class Off implements State {
    public void pull(CeilingFanAsStatePattern wrapper) {
        wrapper.setState(new Low());
        System.out.println(" low speed");
    }
}

class Low implements State {
    public void pull(CeilingFanAsStatePattern wrapper) {
        wrapper.setState(new Medium());
        System.out.println(" medium speed");
    }
}

...
The life cycle of an Activity consists of three (four) states:

**Active/Running:** Activity is in foreground; Activity has focus

**Paused:** Still visible, partially overlaid
Focus lost
Completely alive

**Stopped:** Activity is not visible
Retains all state and member information

**Destroyed:** Activity was terminated or killed
MVC pattern: Model-View-Controller

Best with example: An analog clock

Return current time

Ask for current time

Controller:

Repaint when button clicked

Change time when button clicked

Model

View

Model - View - Controller:
**MVC pattern: analog display clock in Java**

**View:** Panel with clock picture

**Controller:** Button panel to control program

**Model:** Clock calculator; knows nothing of graphical user interface
public class ClockModel {
    private int minutes;

    public ClockModel(int m) {
        minutes = m;
    }

    public int getMinutes() {
        return minutes;
    }

    public void setMinutes(int m) {
        minutes = m;
    }

    public int advance() {
        minutes++;
        return minutes;
    }
}
MVC pattern: ClockView

```java
public class ClockView extends JPanel {
    private ClockModel model; // Needs reference to model!

    public ClockView(ClockModel cm) {
        model = cm;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        double minutes = model.getMinutes(); // Uses model
        g.drawOval(100, 0, 100, 100);
        double hourAngle = 2 * Math.PI * (minutes - 3 * 60) / (12 * 60);
        double minuteAngle = 2 * Math.PI * (minutes - 15) / 60;
        g.drawLine(150, 50, 150 + (int) (30 * Math.cos(hourAngle)),
                    50 + (int) (30 * Math.sin(hourAngle)));
        g.drawLine(150, 50, 150 + (int) (45 * Math.cos(minuteAngle)),
                    50 + (int) (45 * Math.sin(minuteAngle)));
    }
}
```
MVC pattern: ClockController (1)

```java
public class ClockController extends JFrame {
    private JLabel hourLabel, minuteLabel;
    private JButton tickButton, resetButton, crazyButton;
    private JPanel buttonHolder; private Container canvas;
    private ClockView drawArea; // Reference to view
    private ClockModel clock; // Reference to model

    public ClockController() {
        canvas = getContentPane();
        canvas.setLayout(new BorderLayout());
        setSize(300, 300);
        setTitle("MVC Clock");
        buttonHolder = new JPanel(); // Create button holder
        canvas.add(buttonHolder, BorderLayout.SOUTH);
        tickButton = new JButton("Tick");
        resetButton = new JButton("Reset");
        hourLabel = new JLabel("12:");
        minuteLabel = new JLabel("00");
        buttonHolder.add(tickButton); buttonHolder.add(resetButton);
        buttonHolder.add(hourLabel); buttonHolder.add(minuteLabel);
        clock = new ClockModel(720); // Creates model object
    }
```
// Creates view object
drawArea = new ClockView(clock);
canvas.add(drawArea, BorderLayout.CENTER);
drawArea.repaint(); // Adds view to canvas, same as buttonHolder

tickButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        clock.advance();
        drawArea.repaint(); // Use model view
        setLabels();
    }
});
resetButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        clock.setMinutes(720);
        drawArea.repaint(); // Use model view
        setLabels();
    }
});
} // End constructor
MVC pattern: ClockController (3)

```java
public void setLabels() {
    // Does not handle midnight correctly
    int hours = clock.getMinutes() / 60;
    int min = clock.getMinutes() - hours * 60;
    hourLabel.setText(hours + "::");
    if (min < 10) // Minutes should be two digits
        minuteLabel.setText("0" + min);
    else
        minuteLabel.setText("" + min);
}

public static void main(String[] args) {
    ClockController application = new ClockController();
    application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    application.setVisible(true);
}
```
Observer pattern: the abstract problem to solve

1. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. [GoF, p. 293]
2. Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
3. The "View" part of Model-View-Controller.
Define an object (the **Subject**) that is the "keeper" of the data model and/or business logic. Delegate all "view" functionality to decoupled and distinct **Observer** objects. Observers register themselves with the **Subject** as they are created. Whenever the **Subject** changes, it broadcasts to all registered Observers that it has changed, and each **Observer** queries the **Subject** for that subset of the Subject's state that it is responsible for monitoring. This allows the number and "type" of "view" objects to be configured dynamically, instead of being statically specified at compile-time.
Observer pattern: Android example (1)

Sieve of Eratosthenes:

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    textN = (EditText) findViewById(R.id.EditTextN);
    textViewResult = (TextView) findViewById(R.id.TextViewRes);

    // Capture our button from layout
    Button buttonStart = (Button) findViewById(R.id.ButtonStart);
    // Register the onClick listener with the implementation above
    buttonStart.setOnClickListener(startListener);
}
```
private OnClickListener startListener = new OnClickListener() {
    public void onClick(View v) {
        try {
            N = Integer.parseInt(textN.getText().toString());
            textViewResult.setText("Running (N = " + N + ")");
            isPrime = new boolean[N];
            nbPrimes = 0;
            for (int i = 0; i < N; i++) {
                isPrime[i] = true;
            }
            for (int i = 2; i < N; i++) {
                if (isPrime[i] == true) {
                    int j = i + i;
                    nbPrimes++;
                    while (j < N) {
                        isPrime[j] = false;
                        j += i;
                    }
                }
            }
            textViewResult.setText("Number of prime number(s) from 1 to "
                                  + N + ": " + nbPrimes);
        } catch (NumberFormatException e) {
            textViewResult.setText("Please enter an integer value");
        }
    }
};
Delegation pattern: the abstract problem to solve

Delegation is a way to make objects’ *composition* as powerful for reuse as *inheritance*. In delegation, two objects are involved in handling a request: a receiving object delegates operations to its delegate. This is analogous to subclasses deferring requests to parent classes. But with inheritance, an inherited operation can always refer to the receiving object through the `this` member reference (in Java). With delegation the receiver achieves the same effect by passing itself to the delegate and so let the delegated operation refer to the receiver.

Several design patterns use delegation: *State*, *Strategy* and *Visitor* (see references).
Delegation pattern: simple example

```
interface I {
    void f();
    void g();
}

class A implements I {
    public void f() {
        System.out.println("A: doing f()");
    }
    public void g() {
        System.out.println("A: doing g()");
    }
}

class B implements I {
    public void f() {
        System.out.println("B: doing f()");
    }
    public void g() {
        System.out.println("B: doing g()");
    }
}
```
Delegation pattern: simple example

class C implements I {
   // delegation
   private I i = new A();

   public void f() {
      i.f();
   }

   public void g() {
      i.g();
   }

   // normal attributes
   void toA() {
      i = new A();
   }

   void toB() {
      i = new B();
   }
}
Delegation pattern: simple example test

```java
public class TestDelegationPattern {
    public static void main(String[] args) {
        C c = new C();
        c.f(); // output: A: doing f()
        c.g(); // output: A: doing g()
        c.toB();
        c.f(); // output: B: doing f()
        c.g(); // output: B: doing g()
        c.toA();
        c.f();
        c.g();
    }
}
```
A small UML vocabulary edited from a paper of A. Hollub
A small UML vocabulary (1)

**Use-Case** diagram:

If *follower* requires *leader*, then *leader* must be completed before you can execute the *follower* use case. (You must create an account before you can place an order.)

When *extension* extends *base*, all the activities of the *base* use case are also performed in the *extension* use case, but the *extension* use case adds additional activities to—or slightly modifies existing activities of—the *base* use case. (To place a recurring order, you must perform all the activities of placing an order plus set up the recurrence). If a set of activities occur in several use cases, it's reasonable to "normalize" these common activities out into a *base* use case, and then extend it as necessary.

A sub-case. If *case* includes a *sub-case*, then the activities of *sub-case* are performed one or more times in the course of performing *case*. (An "Authenticate" sub-case may be included in several larger use cases, for example.) The sub-case is usually represented in the using use case as a single box marked with the sub-case name and the stereotype «use case».
A small UML vocabulary (2)

The symbol for a class.

«…..» identify stereotypes. E.g.: «utility», «abstract» «interface».
Can use a graphic instead of word.(«interface» often represented as small circle)
Access privileges can precede name:
+ public
# protected
~ package
– private
-- implementation visibility (inaccessible to other objects)
(+) forced public. Override of an interface method that should be treated as private, even if it’s declared public.

Use italics for abstract-class and interface names.
A small UML vocabulary (3)

The symbol for an association: Relationships between classes.

Associated classes are connected by lines. The relationship is identified, if necessary, with a < or > to indicate direction (or use solid arrowheads). The role that a class plays in the relationship is identified on that class's side of the line. Stereotypes (like «friend») are appropriate. Unidirectional message flow can be indicated by an arrow (but is implicit in situations where there is only one role):

Cardinality:

- 1 (usually omitted if 1:1)
- n (unknown at compile time, but bound)
- 0..1 (1..2 1..n)
- 1..* (1 or more)
- * (0 or more)
A small UML vocabulary (4)

The symbol for *inheritance*:

The derived class is the base class, but with additional (or modified) properties. Derived (sub) class is a specialization of (extends) the base (super) class.
A small UML vocabulary (5)

The symbol for `interface`.

A contract that specifies a set of methods that must be implemented by the derived class. In C++, an `interface` is a class containing nothing but pure virtual methods. Java supports them directly. (c.f. "abstract class," which can contain method and field definitions in addition to the abstract declarations.)

Interfaces contain no attributes, so the "attributes" compartment is always empty.

The "inheritance" relationship line is dashed if the base class is an interface.

The "ball and socket" notation at left is new in UML 2.0. Classes that consume (require) an interface display a "socket" labelled with the interface name (A at left). Classes that provide (implement) an interface display a "ball" labelled with the interface name (B at left). Combining the two is a compact way to say that the Consumer talks to the provider via the named interface.
A small UML vocabulary (6)

The symbol for **aggregation**:

![Diagram](image)

An "aggregate" represents a whole that comprises various parts; so, a Committee is an aggregate of its Members. A Meeting is an aggregate of an Agenda, a Room, and the Attendees. At implementation time, this relationship is not containment. (A meeting does not contain a room.) Similarly, the parts of the aggregate might be doing other things elsewhere in the program, so they might be referenced by several objects. In other words, There's no implementation-level difference between aggregation and a simple "uses" relationship (an "association" line with no diamonds on it at all). In both cases an object has references to other objects. Though there's no implementation difference, it's definitely worth capturing the relationship in the UML, both because it helps you understand the domain model better, and because there are subtle implementation issues. I might allow tighter coupling relationships in an aggregation than I would with a simple "uses," for example.

Notice: Composition vs. Aggregation: Neither "aggregation" nor "composition" really have direct analogs in many languages (Java, for example).
Composition involves even tighter coupling than aggregation, and definitely involves containment. The basic requirement is that, if a class of objects (call it a "container") is composed of other objects (call them the "elements"), then the elements will come into existence and also be destroyed as a side effect of creating or destroying the container. It would be rare for a element not to be declared as private. An example might be an Customer's name and address. A Customer without a name or address is a worthless thing. By the same token, when the Customer is destroyed, there's no point in keeping the name and address around. (Compare this situation with aggregation, where destroying the Committee should not cause the members to be destroyed---they may be members of other Committees).

In terms of implementation, the elements in a composition relationship are typically created by the constructor or an initializer in a field declaration, but Java doesn't have a destructor, so there's no way to guarantee that the elements are destroyed along with the container. In C++, the element would be an object (not a reference or pointer) that’s declared as a field in another object, so creation and destruction of the element would be automatic. Java has no such mechanism. It's nonetheless important to specify a containment relationship in the UML, because this relationship tells the implementation/testing folks that your intent is for the element to become garbage collectable (i.e. there should be no references to it) when the container is destroyed.
Callback, recursion: Sequential UML diagrams

At left, the sender sends an asynchronous message to the active-object receiver for background processing, passing it the object to notify when the operation is complete (in this case, itself). The receiver calls its own msg(...) method to process the request, and that method issues the callback() call when it's done.

Note that:

- The callback() message is running on the receiver object's message-processing thread.
- The callback() method is, however, a member of the sender object's class, so has access to all the fields of the sender object.

Since the original thread (from which the original request() was issued) is also running, you must synchronize access to all fields shared by both callback() and other methods of the sender object.