ISSN 1342-2812

Applying DominoJ to GoF Design Patterns

YungYu Zhuang and Shigeru Chiba

December 2011, C–277

Applying DominoJ to GoF Design Patterns

YungYu Zhuang and Shigeru Chiba

Graduate School of Information Science and Engineering, Tokyo Institute of Technology, Japan

Technical Report

Abstract

In this report we study practical effectiveness of *method slot*, which is a new language construct integrating events and methods. We use its language implementation, *DominoJ*, to implement design patterns in the "GoF" book, and found that 16 out of 23 patterns can benefit from it. Here we list down the sample code in Java and DominoJ for the 16 patterns, and discuss the benefits.

1 Introduction

Events have been known as useful programming concepts. In the languages without events support, some techniques such as Observer pattern are developed. On the other hand, in some languages such as EScala [3], events are introduced by language constructs which are separate from methods. We proposed a single mechanism named *method slot*, which supports both events and methods, and *DominoJ*[1], which extends Java to support method slots. In order to study its practical effectiveness, we implemented design patterns in the "GoF" book [2] in DominoJ and Java, and compared them with respect to the number of lines of code and modularity criteria borrowed from [4] as shown in Table 1. We classify the patterns into five categories according to the benefit brought by method slots. In the following sections we discuss what modularity properties can be improved and why method slots can be applied.

Table 1: The benefit of applying DominoJ to design patterns

	Modularity Properties					#Lines of Sample Code	
Pattern Name	Locality	Reusability	Composition	Unpluggability	Non-inheritance	in Java	in DominoJ
Chain of Responsibility	✓	✓	✓	✓	✓⋆	38	28
Composite		✓	✓	✓	✓⋆	41	16
Decorator	✓		✓	✓		26	20
Observer	✓	✓	✓	✓	✓⋆	71	32
Proxy				✓	✓⋆	47	61
State	✓					66	69
Strategy			✓	✓	✓⋆	36	28
Visitor	✓		✓	✓		63	69
Abstract Factory	✓⋆				✓⋆	41	58
Bridge	✓⋆				✓⋆	58	64
Builder	✓⋆				✓⋆	55	69
Factory Method	✓⋆				✓⋆	67	97
Template Method	✓				✓⋆	31	45
Adapter	✓					51	48
Facade	✓⋆					34	53
Mediator	✓			✓		68	49
Command Flyweight Interpreter Iterator Memento Prototype Singleton	Same implementation for Java and DominoJ						

The ✓ mark means that DominoJ has better modularity than Java when implementing the pattern.
The ⋆ mark means that AspectJ does not provide such modularity when implementing the pattern, while DominoJ does.

[1]The DominoJ compiler is available on the project page: http://www.csg.is.titech.ac.jp/projects/dominoj/

2 Event propagation

2.1 Observer Pattern

Intent

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."—From the GoF book. What this pattern does is event mechanism. Once observers register themselves to the subject, the subject will notify them when something happens. The dependent list, registration, and notification are built in DominoJ, so the only thing we have to do is binding observers to the subject.

Sample Code in Java

```java
// here we define Observer and Subject as interfaces due to single inheritance limitation
// pattern code will be put into the classes which implement them
public interface IObserver {
    public void updateBySubject(ISubject s);
}

public interface ISubject {
    public void attachObserver(IObserver o);
    public void detachObserver(IObserver o);
    public void notifyObservers();
}

public class Timer {
      :
}

// the subject
public class ClockTimer extends Timer implements ISubject {
    private ArrayList<IObserver> observers;

    public ClockTimer() {
        observers = new ArrayList<IObserver>();
    }

    public void attachObserver(IObserver o) {
        observers.add(o);
    }

    public void detachObserver(IObserver o) {
        observers.remove(o);
    }

    public void notifyObservers() {
        Iterator<IObserver> iter = observers.iterator();
        while(iter.hasNext()) {
            iter.next().updateBySubject(this);
        }
    }

    // have to notify observers at the end of the method
    public void tick() {
          :
        notifyObservers();
    }
}

public class Widget {
      :
}

// the observer
// the attaching and detaching are set inside
public class DigitalClock extends Widget implements IObserver {
    private ISubject subject;

    public DigitalClock(ClockTimer s) {
        subject = s;
        subject.attachObserver(this);
    }

    // must handle the detaching manually
    public void finalize() {
        subject.detachObserver(this);
    }

    public void updateBySubject(ISubject s) {
        if(s == subject) {
            draw();
        }
    }

    public void draw() {
          :
    }
}

// client usage
ClockTimer ct = new ClockTimer();
DigitalClock dc = new DigitalClock(ct);
// then dc.draw() will be called after ct.tick() is called
```

Sample Code in DominoJ

```
public class Timer {
      :
}

// the subject
public class ClockTimer extends Timer {
    public ClockTimer {
        tick += notifyObservers;
    }

    public void notifyObservers() {}

    public void tick() {
          :
    }
}

public class Widget {
      :
}

// the observer
public class DigitalClock extends Widget {
    public DigitalClock(ClockTimer s) {
        // bind to single method and let the subject control the notification
        s.notifyObservers += draw;
        // of course you also can bind to tick() directly
        // ex: s.tick += draw;
    }

    public void draw() {
          :
    }
}

// client usage is the same as the one in Java
ClockTimer ct = new ClockTimer();
DigitalClock dc = new DigitalClock(ct);
// then dc.draw() will be called after ct.tick() is called
```

Consequences

Since multiple inheritance is prohibited in Java, we have to use interfaces instead and put pattern code (registration and notification) into ClockTimer and DigitalClock. We may move pattern code to their parent classes, Timer and Widget, but the classes which do not inherit from Timer or Widget still cannot take advantage of this pattern. It is a good idea to put pattern code into ancestor classes such as Object, but the best solution is putting them into language itself—what DominoJ does.

Locality	The code that defines the relation can be gathered up, and no pattern code.
Reusability	This pattern can be applied to any classes easily since it is built-in functionality.
Unpluggability	The subject-observer binding is object-based, and easy to be added or removed.
Composition	The code tangling can be avoided when a class is involved in several subject-object relations. For example, an object is observed by several observer groups for different purposes, and also observes another object at the same time.
Non-inheritance	No need to create classes or interfaces for the subject and the observer.

2.2 Chain of Responsibility Pattern

Intent

"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."— From the GoF book. The chain logic can be described by a series of events, so we can implement it in DominoJ by binding handlers in order.

Sample Code in Java

```java
// there are two choices due to single inheritance limitation
// let every class implements chain interface (HelpHandler)
// or put chain logic into the parent class (Widget)
// here we choose the latter
public class Widget {
    private Widget successor = null;

    public Widget(Widget parent) {
        setHandler(parent);
          :
    }

    public void setHandler(Widget handler) {
        successor = handler;
    }

    public boolean handleHelp() {
        if(successor == null)    return false;
        return successor.handleHelp();
    }
}

public class Button extends Widget {

    public boolean handleHelp() {
        // return true if it can offer help
        // otherwise return super.handleHelp()
        ...
            return true;
        ...
            return super.handleHelp();

    }
      :
}

public class Dialog extends Widget {
    // overrides handleHelp like what Button does
      :
}

// client usage
Dialog dialog = new Dialog();
Button btn1 = new Button(dialog);
Button btn2 = new Button(dialog);
// client may change the successor
btn1.setHandler(btn2);
```

Sample Code in DominoJ

```java
// no need to define HelpHandler interface or put chain logic into Widget
// because the handler chain is built-in
// declare handleHelp method in parent class directly
public class Widget {
    public Widget(Widget parent) {
        handleHelp += parent.handleHelp;
          :
    }

    public boolean handleHelp() {}
}

public class Button extends Widget {
    public boolean handleHelp() {
        // just return if it is handled
        if($ret)    return true;
        // return true if it is handled here, otherwise return false
        ...
```

```java
            return true;
        ...
            return false;
    }
      :
}

public class Dialog extends Widget {
    // overrides handleHelp like what Button does
      :
}

// client usage
Dialog dialog = new Dialog();
Button btn1 = new Button(dialog);
Button btn2 = new Button(dialog);
// clients may also append/prepend/replace the handler
btn1.handleHelp = btn2.handleHelp;
```

Consequences

DominoJ makes the chain relation more clear, especially when there are many chain relations among objects.

Locality	The chain relation among objects can be gathered up.
Reusability	The pattern can be applied anywhere since it can be described by built-in operators.
Unpluggability	The pattern is simply unplugged by removing the binding.
Composition	The code becomes complicated when there are multiple chain relations. It needs more fields and setter to keep successors for different chains. With DominoJ the code tangling can be avoided.
Non-inheritance	In Java the chain relation is usually declared as interface rather than base class because participant classes have to inherit from others. The chain logic has to be implemented by every participant class since the interface cannot contain implementation. Another solution is merging the chain logic into base class as shown in the sample code, but this solution makes it difficult to separate pattern code from main logic.

2.3 Composite Pattern

Intent

"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."—From the GoF book. Usually the purpose of composing objects into tree structures is doing a specified operation. For example, UI libraries maintain a container tree for drawing the widgets in order. We can implement the idea by binding the operation methods.

Sample Code in Java

```java
// in this sample we want to get the sum of the values returned by operation method
// define a leaf class
public class Component {
      :
    // this method return a value
    public int operation() {
        int value = 0;
        // do something and change the value
          :
        return value;
    }
}

// define a composite class
public class Composite extends Component {
    ArrayList<Component> list;

    public Composite() {
        list = new ArrayList<Component>();
          :
    }

    public void add(Component c) {
        list.add(c);
    }

    public int operation() {
        int value = 0;
        // do something and change the value
          :
        // get the values returned by children
        for(int i=0; i<list.size(); i++) {
            value += list.get(i).operation();
        }
        return value;
    }
}

// client usage
Composite comp1 = new Composite();
Component leaf1 = new Component();
Composite comp2 = new Composite();
Component leaf2 = new Component();
Component leaf3 = new Component();
// construct the tree
comp1.add(leaf1);
comp1.add(comp2);
comp2.add(leaf2);
comp2.add(leaf3);
// get the sum
int sum = comp1.operation();
```

Sample Code in DominoJ

```
// Composite class is unnecessary
// bind the operation method of leaves to a leaf directly
public class Component {
      :
    public int operation() {
        int value = 0;
        // do something and change the value
          :
        return $ret + value;
    }
}

// client usage
Component comp1 = new Component();
// Component can be used as a leaf or a composite
// you could construct the tree by binding leaves directly
Component leaf1 = new Component();
Component leaf2 = new Component();
Component leaf3 = new Component();
leaf1.operation += leaf2.operation;
leaf1.operation += leaf3.operation;
// get the sum
int sum = leaf1.operation();
// you also could let the tree look like the one in Java
// Component comp1 = new Component();
// Component leaf1 = new Component();
// Component comp2 = new Component();
// Component leaf2 = new Component();
// Component leaf3 = new Component();
// comp1.operation += leaf1.operation;
// comp1.operation += comp2.operation;
// comp2.operation += leaf2.operation;
// comp2.operation += leaf3.operation;
// int sum = comp1.operation();
```

Consequences

Because a class can be a leaf or a composite, it is easy to adjust the structure. We do not have to prepare a composite class for a component class.

Reusability	No need to define and create the composite class, the relation can be applied to any class.
Unpluggability	The unplugging can be done by removing the binding between objects.
Composition	When an object is put into several tree structures for different methods, we do not have to maintain similar code in several places.
Non-inheritance	A class can be used as a composite directly, so we can avoid declaring additional subclasses for composite.

2.4 Decorator Pattern

Intent

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."—From the GoF book. The idea is similar to the idea in DominoJ, which provides a flexible alternative to extend functionality. It is easy to bypass the method calling in DominoJ.

Sample Code in Java

```
// define a class which is wrapped
public class VisualComponent {
      :
    public void draw() {
          :
    }
    public void resize() {
          :
    }
}

// the decorator
public class Decorator extends VisualComponent {
    private VisualComponent component;

    public Decorator(VisualComponent c) {
        component = c;
    }

    public void draw() {
        component.draw();
    }

    public void resize() {
        component.resize();
    }

    // provide other functionality
      :
}
```

Sample Code in DominoJ

```
// the wrapped class is the same as the one in Java
public class VisualComponent {
      :
    public void draw() {
          :
    }
    public void resize() {
          :
    }
}

// the decorator
public class Decorator extends VisualComponent {
    private VisualComponent component;

    public Decorator(VisualComponent c) {
        component = c;
        // bypass some methods
        draw = component.draw;
        resize = component.resize;
    }

    // provide other functionality
      :
}
```

Consequences

There are two reasons that we still let Decorator inherit from VisualComponent. First, this pattern assumes the decorator is used as the wrapped component. Second, we do not have to define empty methods in the decorator for binding. Instead of bypassing, we can also use ^= or += to do some operations before or after the original method.

Locality	All bindings are put in the constructor together.
Unpluggability	Adding or removing the bypassing is easy, even in runtime.
Composition	When the decorator wraps many components at the same time, the relations are clear and can be switched dynamically.

3 Switch by method calls

3.1 Proxy Pattern

Intent

"Provide a surrogate or placeholder for another object to control access to it."—From the GoF book. With DominoJ the behavior of an object can be changed by binding the methods in another object to it. In other words, clients can use the same object whether it has proxy logic or not.

Sample Code in Java

```java
// the original class
public class Image extends Graphic {
    public Image(String f) {
          :
    }

    public void draw() {
          :
    }

    public Point getExtent() {
          :
    }
}

// create a proxy for the original class
public class ImageProxy extends Graphic {
    private Image image;
    private String filename;
    private Point extent;

    public ImageProxy(String f) {
        image = null;
        filename = f;
        extent = Point.ZERO;
    }

    public Image getImage() {
        if(image == null) {
            image = new Image(filename);
        }
        return image;
    }

    public void draw() {
        getImage().draw();
    }

    public Point getExtent() {
        if(extent == Point.ZERO) {
            extent = getImage().getExtent();
        }
        return extent;
    }
}

// client usage
ImageProxy proxy = new ImageProxy("anImageFileName");
TextDocument text = new TextDocument();
// here we assume that insert method accept all Graph classes
text.insert(proxy);
```

Sample Code in DominoJ

```java
// if there is an Image class which cannot be modified
public class Image extends Graphic {
    // in this sample we need two-phase construction
    public void load(String f) {
          :
    }

    public boolean isLoaded() {
          :
    }

    public void draw() {
          :
    }

    public Point getExtent() {
          :
    }
}

// create a proxy class to delay loading and cache some values
// without inheriting from Image or Graphic
public class ImageProxy {
    private Image image;
    private String filename;
    private Point extent = Point.ZERO;

    public ImageProxy(Image i) {
        image = i;
        // it only works for two-phase construction because we cannot replace constructor
        // keep filename and load it later
        image.load = fake;
        // use generic handlers to load file before some operations
        image.draw ^= voidLoad;
        image.getExtent ^= objLoad;
        // get extent value when it is used at the first time
        image.getExtent += getExtent;
    }

    public void fake(String f) {
        filename = f;
    }

    // for methods which have return type
    public Object objLoad(Object[] args) {
        voidLoad(args);
        return null;
    }

    // restore original load function and call it
    public void voidLoad(Object[] args) {
        if(!image.isLoaded()) {
            image.load = image.load;
            image.load(filename);
        }
    }

    // cache the value
    public Point getExtent() {
        if(extent == Point.ZERO && $ret != null)    extent = $ret;
        return extent;
    }
}

// client usage
Image image = new Image();
// create a proxy and still use the image object
ImageProxy proxy = new ImageProxy(image);
image.load("anImageFileName");
// the image will be loaded when it is really used
image.draw();
image.getExtent();
// use cache value in the future
image.getExtent = proxy.getExtent;
TextDocument text = new TextDocument();
// transparent to others, so can be given to which only accepts Image
text.insert(image);
```

Consequences

DominoJ provides a way to do proxy by binding methods instead of using a wrapper. Although it needs some tricks to do something like delay loading, the original object can be used directly. Furthermore, the proxy policy can be applied dynamically. In the sample we toggle the getExtent method manually. A solution to avoid toggling binding manually is inheriting from the original object and put the proxy logic in it. However, the solution falls back to use subclassing, and clients have to decide using proxy or not.

Unpluggability	The proxy can be unplugged by removing the binding.
Transparency	Clients are unaware of proxy. What clients create is the same whether applying proxy or not.
Non-inheritance	When the same proxy logic is used in different classes, we can create a proxy for all. We do not have to create a lot of proxies, and worry about which class should be inherited from.

3.2 State Pattern

Intent

"Allow an object to alter its behavior when its internal state changes. The object will appear to change its class."—From the GoF book. With DominoJ all the state changes can be gathered together. This makes state transition clear.

Sample Code in Java

```java
// the class which uses states inside
public class TCPConnection {
    private TCPState state;

    public TCPConnection() {
        state = TCPClosed::getInstance();
    }

    public void changeState(TCPState s) {
        state = s;
    }
}

// define state
public class TCPState {
    public void transmit(TCPConnection t, TCPOctetStream s) {
          :
    }

    public void activeOpen(TCPConnection t) {
          :
    }

    public void passiveOpen(TCPConnection t) {
          :
    }

    public void close(TCPConnection t) {
          :
    }

    public void synchronize(TCPConnection t) {
          :
    }

    public void acknowledge(TCPConnection t) {
          :
    }

    public void send(TCPConnection t) {
          :
    }

    public void changeState(TCPConnection t) {
        t.changeState(this);
    }
}

// here only show the detail in TCPClosed class
public class TCPEstablished extends TCPState {
      :
}

public class TCPListen extends TCPState {
      :
}

public class TCPClosed extends TCPState {
    public void activeOpen(TCPConnection t) {
          :
        // change the state at the end
        TCPEstablished.getInstance().changeState(t);
    }

    public void passiveOpen(TCPConnection t) {
          :
        // change the state at the end
        TCPListen.getInstance().changeState(t);
    }
      :
}
```

Sample Code in DominoJ

```
// TCPConnection and TCPState are the same as the one in Java
public class TCPConnection {
    private TCPState state;

    public TCPConnection() {
        state = TCPClosed::getInstance();
    }

    public void changeState(TCPState s) {
        state = s;
    }
}

public class TCPState {
    public void transmit(TCPConnection t, TCPOctetStream s) {
          :
    }

    public void activeOpen(TCPConnection t) {
          :
    }

    public void passiveOpen(TCPConnection t) {
          :
    }

    public void close(TCPConnection t) {
          :
    }

    public void synchronize(TCPConnection t) {
          :
    }

    public void acknowledge(TCPConnection t) {
          :
    }

    public void send(TCPConnection t) {
          :
    }

    public void changeState(TCPConnection t) {
        t.changeState(this);
    }
}

public class TCPEstablished extends TCPState {
      :
}

public class TCPListen extends TCPState {
      :
}

// the only difference is the state transition in TCPClosed
public class TCPClosed extends TCPState {
    public TCPClosed() {
      // state transition code
      activeOpen += TCPEstablished.getInstance().changeState;
      passiveOpen += TCPListen.getInstance().changeState;
    }

    public void activeOpen(TCPConnection t) {
          :
        // no need to change the state at the end
    }

    public void passiveOpen(TCPConnection t) {
          :
        // no need to change the state at the end
    }
```

```
        :
}
```

Consequences

The code of state transition is always specified at the end of methods, so it is a typical usage of appending handler. In addition, the state transition can be adjusted dynamically.

Locality	All state transition statements are defined in the same place, so the transition is easy to understand.

3.3 Strategy Pattern

Intent

"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."—From the GoF book. The GoF book mentions two issues when applying this pattern. The first one is that clients must know how to choose between different strategies. DominoJ can hide the strategies from clients by providing another interface. Clients just give some conditions and do not have to decide which strategy object should be created. The second one is the parameter passing. Every strategy may need different information for its algorithm. The parameter passing is annoying because different strategies may need different information. Using DominoJ the strategy switch is set inside the class.

Sample Code in Java

```
// the class which accepts different strategies
public class Composition {
    private Compositor compositor;

    public Composition(Compositor c) {
        compositor = c;
          :
    }

    public void repair() {
          :
        compositor.compose( ... );
          :
    }
      :
}

// the strategy
public class Compositor {
    public abstract int compose( ... );
}

public class SimpleCompositor extends Compositor {
    public int compose ( ... ) {
          :
    }
      :
}

public class TeXCompositor extends Compositor {
    public int compose ( ... ) {
          :
    }
      :
}

// client usage
// clients must know which strategy is suitable for the situation
Composition quick = new Composition(new SimpleCompositor());
Composition slick = new Composition(new TeXCompositor());
```

Sample Code in DominoJ

```
// the class which has several strategies
public class Composition {
    // the method which will be called to do a strategy
    private void compose() {}

    // different strategies
    private void compose_simple() {
          :
    }

    private void compose_tex() {
          :
    }

    // select a strategy automatically
    // clients do not have to understand which one should be initialized
    public void configure( ... ) {
        ...
        // bind to different strategies according to some conditions
        ...
            compose = compose_simple;
        ...
            compose = compose_tex;
    }

    public void repair() {
          :
        // no need to do annoying parameter passing
        compose();
          :
    }
      :
}

// client usage
Composition composition = new Composition();
// clients give some conditions instead of creating compositor manually
composition.configure( ... );
```

Consequences

If the strategies are also used by other classes, we can separate strategies from Composition by moving them to another class. The configure method is optional. If we hope no switch statement in Composition, we can provide a select method for switching the binding directly or let clients bind the strategy by themselves.

Unpluggability	The pattern can be unplugged by removing the binding. We can leave the default algorithm in compose method, so it can fall back to it when unplugging.
Composition	When many steps in a class are exposed, clients do not have to know how to create those strategy objects. We can also put a family of strategies into another class, but parameter passing becomes necessary.
Transparency	Usually a strategy is set for an algorithm rather than according to the situation, so clients have to know the detail. DominoJ hides the detail from clients.
Non-inheritance	The strategy is not dispatched by inheritance, so it can avoid defining a lot of class families.

3.4 Visitor Pattern

Intent

"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."—From the GoF book. The purpose of this pattern is a little similar to Composite pattern. Both of them provide a way to do an operation on a bunch of components. The main difference is that Visitor pattern lets you put the operation logic outside of component classes. GoF book uses Composite pattern (Chassis class) in this sample, so we also show how to use with composite classes in DominoJ. In fact, with DominoJ composite classes such as Chassis can be eliminated and all components can be connected directly (please refer to Composite pattern).

Sample Code in Java

```
// we want to define a visitor for this class
public class Equipment {
    public int netPrice() {
          :
    }

    public int discountPrice() {
          :
    }

    public abstract void accept(EquipmentVisitor visitor);
      :
}

public class Disk extends Equipment {
    public void accept(EquipmentVisitor visitor) {
        visitor.visitDisk(this);
    }
}

public class Chassis extends Equipment {
    ArrayList<Equipment> parts;

    public Chassis() {
        parts = new ArrayList<Equipment>();
    }

    public void accept(EquipmentVisitor visitor) {
        Iterator<Equipment> iterator = parts.iterator();
        while(iterator.hasNext()) {
            iterator.next().accept(visitor);
        }
        visitor.visitChassis(this);
    }
      :
}

public class EquipmentVisitor {
    public abstract void visitDisk(Disk d);
    public abstract void visitChassis(Chassis c);
}

public class PricingVisitor extends EquipmentVisitor {
    private int total;

    public int getTotalPrice() {
        return total;
    }

    public void visitDisk(Disk d) {
        total += d.netPrice();
    }

    public void visitChassis(Chassis c) {
        total += c.discountPrice();
    }
}

// client usage
PricingVisitor visitor = new PricingVisitor();
// assume equipment can be gotten by the following method
Equipment[] components = getEquipments();
for(int i=0; i<components.length; i++) {
    components[i].accept(visitor);
}
int total = visitor.getTotalPrice();
```

Sample Code in DominoJ

```
// first define visitor interface for accept method
// the accept method in this sample is used for composite class
// actually there is no need to define composite class
public interface IEquipmentVisitor {
    // provide a slot for visitor
    public Equipment visit();
}

// we want to define a visitor for this class
public class Equipment {
    public int netPrice() {
          :
    }

    public int discountPrice() {
          :
    }

    public Equipment visit() {
        return this;
    }

    // give clients a method to set the binding, you also can do it in client
    public void accept(IEquipmentVisitor visitor) {
        visitor.visit += visit;
        visitor.visit += visitor.visit;
    }
      :
}

public class Disk extends Equipment {
      :
}

// with DominoJ you can avoid defining such composite class
// please refer to Composite pattern section
public class Chassis extends Equipment {
    ArrayList<Equipment> parts;

    public Chassis() {
        parts = new ArrayList<Equipment>();
    }

    public void accept(IEquipmentVisitor visitor) {
        Iterator<Equipment> iterator = parts.iterator();
        while(iterator.hasNext()) {
            visitor.visit += iterator.next().visit;
            visitor.visit += visitor.visit;
        }
        visitor.visit += visit;
        visitor.visit += visitor.visit;
    }
}

// the purpose of the visitor is calculating total price
public class PricingVisitor implements IEquipmentVisitor {
    private int total;

    public int getTotalPrice() {
        return total;
    }

    public Equipment visit() {
        if($ret != null) {
            if($ret instanceof Disk)    total += $ret.netPrice();
            else if($ret instanceof Chassis)    total += $ret.discountPrice();
        }
        return null;
    }
}

// client usage
PricingVisitor visitor = new PricingVisitor();
// assume equipment can be gotten by the following method
Equipment[] components = getEquipments();
// set the binding
for(int i=0; i<components.length; i++) {
    // if we do not use composite class, we can set the binding here, for example:
    // visitor.visit += components[i].visit;
    // visitor.visit += visitor.visit;
    components[i].accept(visitor);
}
// start visiting
visitor.visit();
// get result
int total = visitor.getTotalPrice();
```

Consequences

Using DominoJ can avoid the similar method calling in every subclass such as visitDisk method and visitChassis method. The visit method can be defined in parent class and simply return itself. All handling for different subclasses are moved to visitor class. When adding a new subclass, only the visitor should be modified. Furthermore, if we remove Chassis in this sample, accept method and IEquipmentVisitor are also unnecessary. Clients can set the binding by themselves.

Locality	Unlike accept method, visit method is defined in parent class. We do not have to define accept method in subclasses. On the other hand, putting the handling for subclasses in one method of visitor class makes it possible to share the same logic for different subclasses. For example, maybe the handling for Disk and Chassis are the same, and we do not have to maintain the same code in visitDisk and visitChassis. Besides, it can provide default behavior for the subclass that may be declared in the future.
Unpluggability	The visitor pattern can be disabled simply by removing the visit method in the parent class.
Composition	When there are different class families and the behaviors of their visitors are the same, the visitors can be shared by setting the return type of visit method to Object.

4 An alternative to inheritance

4.1 Abstract Factory Pattern

Intent

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes."—From the GoF book. In some cases the objects in different product families have the same feature, for example, MotifScrollBar and MotifWindow in the following sample. We cannot extract the feature to their parent class because they have inherited from different products and multiple inheritance is prohibited in Java. The following sample shows how to use DominoJ to resolve the problem.

Sample Code in Java

```java
// define the factory for creating products
public abstract class WidgetFactory {
    public abstract ScrollBar createScrollBar();
    public abstract Window createWindow();
      :
}

// a concrete factory
public class MotifWidgetFactory extends WidgetFactory {
    public ScrollBar createScrollBar() {
        return new MotifScrollBar();
    }

    public Window createWindow() {
        return new MotifWindow();
    }
}

// define the first product family
public abstract class ScrollBar {
      :
}

public class MotifScrollBar extends ScrollBar {
    // the common feature which can be used in MotifScrollBar and MotifWindow
    public void transparent() {
          :
    }
      :
}

// define the second product family
public abstract class Window {
      :
}

public class MotifWindow extends Window {
    // the common feature which can be used in MotifScrollBar and MotifWindow
    public void transparent() {
          :
    }
      :
}

// client usage
// create a concrete factory
MotifWidgetFactory factory = new MotifWidgetFactory();
// create products without specifying their concrete classes
```

```
Window w = factory.createWindow();
ScrollBar s = factory.createScrollBar();
```

Sample Code in DominoJ

```
// MotifScrollBar and MotifWindow also need to inherit from Motif for some features
// but Motif, ScrollBar, and Window cannot be interfaces
// because they all have implementation
// the factory is the same as the one in Java
public abstract class WidgetFactory {
    public abstract ScrollBar createScrollBar();
    public abstract Window createWindow();
      :
}

// a concrete factory
public class MotifWidgetFactory extends WidgetFactory {
    public ScrollBar createScrollBar() {
        return new MotifScrollBar();
    }
    public Window createWindow() {
        return new MotifWindow();
    }
}

// you can define an interface for those methods and bind implementation to them
public interface IMotif {
    public void transparent();
}

// then put the implementation in Motif class
public class Motif implements IMotif {
    // let all Motif widgets share the same object
    private static motif = null;

    public static Motif getInstance() {
        if(motif == null)    motif = new Motif();
        return motif;
    }

    // the common feature which can be used in MotifScrollBar and MotifWindow
    public void transparent() {
          :
    }
      :
}

// replace the implementation with the one in Motif
public class MotifScrollBar extends ScrollBar implements IMotif {
    public MotifScrollBar() {
        transparent = Motif.getInstance().transparent;
          :
    }

    // empty method for binding
    public void transparent() {}
      :
}

// maybe there are additional operations for MotifWindow
// we use += or ^= instead of =
public class MotifWindow extends Window implements IMotif {
    public MotifWindow() {
        // call transparent method in Motif after transparent method is called
        transparent += Motif.getInstance().transparent;
          :
    }

    // put additional operations in this method
    public void transparent() {
          :
    }
      :
}

// client usage is the same as the one in Java
// create a concrete factory
MotifWidgetFactory factory = new MotifWidgetFactory();
Window w = factory.createWindow();
ScrollBar s = factory.createScrollBar();
```

Consequences

DominoJ helps to extract the common implementation among objects in different product families such as transparent method in this sample.

Locality	In this example, common features in MotifScrollBar and MotifWindow can be extracted into another class, Motif.
Non-inheritance	Unlike interface, using this approach can gather similar implementation together, and allow the objects to inherit from other classes.

4.2 Bridge Pattern

Intent

"Decouple an abstraction from its implementation so that the two can vary independently."—From the GoF book. The following sample shows how to use the implementor in the abstraction and its subclasses. The detail of implementor (WindowImpl) is defined in concrete implementors (XWindowImpl and PMWindowImpl), and use inheritance to delegate. If we also want to let XWindowImpl inherit from XImpl for some reasons, we have to choose between WindowImpl and Ximpl. With DominoJ XWindowImpl and PMWindowImpl can be delegated without using inheritance.

Sample Code in Java

```java
// the abstraction class, which only needs to know WindowImpl
public class Window {
    private WindowImpl impl = null;

    public WindowImpl getWindowImpl() {
        if(impl == null) {
            // return a XWindowImpl or PMWindowImpl object
            impl = WindowSystemFactory.getInstance().makeWindowImpl();
        }
    }
      :
}

// the class delegate its method to subclasses
public abstract class WindowImpl {
      :
    public abstract void deviceBitmap(String filename, int l, int t, int r, int b);
}

public class XWindowImpl extends WindowImpl {
      :
    public void deviceBitmap(String filename, int l, int t, int r, int b) {
          :
    }
}

public class PMWindowImpl extends WindowImpl {
      :
    public void deviceBitmap(String filename, int l, int t, int r, int b) {
          :
    }
}

public class WindowSystemFactory {
    private WindowSystemFactory factory = null;

    public static getInstance() {
        if(factory == null) {
            factory = new WindowSystemFactory();
        }
        return factory;
    }

    public WindowImpl makeWindowImpl() {
        WindowImpl impl;
        // create an object of subclass of WindowImpl according to some conditions
          :
            impl = new XWindowImpl();
          :
            impl = new PMWindowImpl();
        return impl;
    }
}

// usage
```

```
public class IconWindow extends Window {
    private String filename;
      :
    public void drawContents() {
        WindowImpl impl = getWindowImpl();
        if(impl != null)    impl.deviceBitmap(filename, 0, 0, 0, 0);
    }
}
```

Sample Code in DominoJ

```
// the abstraction class is the same as the one in Java
public class Window {
    private WindowImpl impl = null;

    public WindowImpl getWindowImpl() {
        if(impl == null) {
            // return a XWindowImpl or PMWindowImpl object
            impl = WindowSystemFactory.getInstance().makeWindowImpl();
        }
    }
      :
}

// maybe XWindowImpl has inherited from XImpl
// and PMWindowImpl has inherited from PMImpl
// so we cannot create WindowImpl class as their parent
// define IWindowImpl for consistency
public interface IWindowImpl {
    public void deviceBitmap(String filename, int l, int t, int r, int b);
}

// WindowImpl is used directly, so it cannot be abstract
public class WindowImpl implements IWindowImpl {
      :
    // empty method for binding
    public void deviceBitmap(String filename, int l, int t, int r, int b) {}
}

public class XWindowImpl extends XImpl implements IWindowImpl {
      :
    public void deviceBitmap(String filename, int l, int t, int r, int b) {
          :
    }
}

public class PMWindowImpl extends PMImpl implements IWindowImpl {
      :
    public void deviceBitmap(String filename, int l, int t, int r, int b) {
          :
    }
}

// WindowSystemFactory always creates a WindowImpl object
// but replace its implementation
public class WindowSystemFactory {
    private WindowSystemFactory factory = null;
    private IWindowImpl real;

    public static getInstance() {
        if(factory == null) {
            factory = new WindowSystemFactory();
        }
        return factory;
    }

    public WindowImpl makeWindowImpl() {
        // the object will be used
        WindowImpl impl = new WindowImpl();
        // create an object of subclass of WindowImpl according to some conditions
        ...
            real = new XWindowImpl();
        ...
            real = new PMWindowImpl();
        // and bind its implementation to the WindowImpl object
        impl.deviceBitmap = real.deviceBitmap;
        return impl;
    }
}

// the usage is the same as the one in Java
public class IconWindow extends Window {
    private String filename;
      :
```

```java
    public void drawContents() {
        WindowImpl impl = getWindowImpl();
        if(impl != null)    impl.deviceBitmap(filename, 0, 0, 0, 0);
    }
}
```

Consequences

This sample shows a typical usage of DominoJ. The methods are delegated by binding instead of inheritance.

Locality	XWindowImpl and PMWindowImpl can inherit from XImpl and PMImpl respectively to avoid implementing the same logic again.
Non-inheritance	The approach allows the implementor is delegated by non-inheritance way, and lets concrete implementors inherit from other classes for getting some common implementations.

4.3 Builder Pattern

Intent

"Separate the construction of a complex object from its representation so that the same construction process can create different representations."—From the GoF book. Sometimes the concrete builders need to inherit from other classes for its implementation, but we still want to put some common code in the builder class. Using DominoJ the delegation of builder class can be done by binding.

Sample Code in Java

```java
// the output
public class Maze {
      :
}

// the director class
public class MazeGame {
      :
    // this method calls builder to construct
    public Maze createMaze(MazeBuilder builder) {
        builder.buildMaze();
        builder.buildRoom(1);
        builder.buildRoom(2);
        builder.buildDoor(1, 2);
        return builder.getMaze();
    }

    // construct another Maze
    public Maze createComplexMaze(MazeBuilder builder) {
        builder.builderRoom(1);
          :
        builder.buildRoom(1001);
        return builder.getMaze();
    }
}

// builder class
public abstract class MazeBuilder {
    public void buildMaze() {}
    public void buildRoom(int n) {}
    public void buildDoor(int from, int to) {}
    public Maze getMaze() {
        return 0;
    }
}

// define a concrete builder
public class StandardMazeBuilder extends MazeBuilder {
    private Maze maze = null;

    public void buildMaze() {
        maze = new Maze();
    }

    public void buildRoom(int n) {
          :
    }

    public void buildDoor(int from, int to) {
            :
    }

    public Maze getMaze() {
        return maze;
    }
}

// client usage
MazeGame game = new MazeGame();
StandardMazeBuilder builder = new StandardMazeBuilder();
game.createMaze(builder);
Maze maze = builder.getMaze();
```

Sample Code in DominoJ

```
// Maze class is the same as the one in Java
public class Maze {
      :
}

// MazeGame class is the same as the one in Java
public class MazeGame {
      :
    // this method calls builder to construct
    public Maze createMaze(MazeBuilder builder) {
        builder.buildMaze();
        builder.buildRoom(1);
        builder.buildRoom(2);
        builder.buildDoor(1, 2);
        return builder.getMaze();
    }

    // construct another Maze
    public Maze createComplexMaze(MazeBuilder builder) {
        builder.builderRoom(1);
          :
        builder.buildRoom(1001);
        return builder.getMaze();
    }
}

// MazeBuilder will be used directly, so it cannot be abstract
public class MazeBuilder {
    public void buildMaze() {}
    public void buildRoom(int n) {}
    public void buildDoor(int from, int to) {}
    public Maze getMaze() {
        return 0;
    }
}

// assume that StandardMazeBuilder has to inherit from another class Standard
public class StandardMazeBuilder extends Standard {
    private Maze maze = null;
    private MazeBuilder builder;

    public StandardMazeBuilder() {
        builder = new MazeBuilder();
        // redirect the calling
        builder.buildMaze = buildMaze;
        builder.buildRoom = buildRoom;
        builder.buildDoor = buildDoor;
        builder.getMaze = getMaze;
    }

    public MazeBuilder getBuilder() {
        return builder;
    }

    public void buildMaze() {
        maze = new Maze();
    }

    public void buildRoom(int n) {
          :
    }

    public void buildDoor(int from, int to) {
          :
    }

    public Maze getMaze() {
        return maze;
    }
}

// client usage
MazeGame game = new MazeGame();
StandardMazeBuilder standard = new StandardMazeBuilder();
MazeBuilder builder = standard.getBuilder();
game.createMaze(builder);
Maze maze = builder.getMaze();
```

Consequences

Although in most cases the builder looks like an interface, sometimes we need to keep some fields in it. However, the concrete builders also want to take advantage of other classes for their construction process. Using DominoJ the issue can be resolved.

Locality	StandardMazeBuilder can get the feature from Standard by inheritance rather than copying the code.
Non-inheritance	This sample shows how to use the binding in DominoJ instead of inheritance. Concrete builders can inherit from other classes for reusing the code in their construction process. It avoids maintaining similar code in different places.

4.4 Factory Method Pattern

Intent

"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."—From the GoF book. In this pattern DominoJ can replace product instantiation methods easily, so we do not need to subclass the creator for replacing those methods.

Sample Code in Java

```
// the creator class, which has factory methods for instantiate products
public class MazeGame {
    public Maze makeMaze() {
        return new Maze();
    }

    public Room makeRoom(int n) {
        return new Room(n);
    }

    public Wall makeWall() {
        return new Wall();
    }

    public Door makeDoor(Room r1, Room r2) {
        return new Door(r1, r2);
    }

    // call factory methods to instantiate products
    public Maze createMaze() {
        Maze m = makeMaze();
        Room r1 = new MakeRoom(1);
        Room r2 = new MakeRoom(2);
        Door d = makeDoor(r1, r2);
        m.addRoom(r1);
        m.addRoom(r2);
          :
        return m;
    }
}

// a concrete creator class
public class BombedMazeGame extends MazeGame {
    // some product instantiation method are replaced
    public Wall makeWall() {
        return new BombedWall();
    }

    public Room makeRoom(int n) {
        return new RoomWithABomb(n);
    }

    // common "Bombed" feature
    public void bomb() {
          :
    }
      :
}

// another concrete creator class
public class EnchantedMazeGame extends MazeGame {
    // some product instantiation method are replaced
    public Room makeRoom (int n) {
        return new EnchantedRoom(n, castSpell());
    }

    public Door makeDoor(Room r1, Room r2) {
        return new DoorNeedingSpell(r1, r2);
    }

    public Spell castSpell() {
          :
    }

    // common "Enchanted" feature
    public void enchant() {
          :
    }
      :
}

// client usage
// create different subclasses for different products
BombedMazeGame bmg = new BombedMazeGame();
Maze m1 = bmg.createMaze();
EnchantedMazeGame emg = new EnchantedMazeGame();
Maze m2 = emg.createMaze();
```

Sample Code in DominoJ

```
// MazeGame class is the same as the one in Java
public class MazeGame {
    public Maze makeMaze() {
        return new Maze();
    }

    public Room makeRoom(int n) {
        return new Room(n);
    }

    public Wall makeWall() {
        return new Wall();
    }

    public Door makeDoor(Room r1, Room r2) {
        return new Door(r1, r2);
    }

    // call several methods to instantiate products
    public Maze createMaze() {
        Maze m = makeMaze();
        Room r1 = new MakeRoom(1);
        Room r2 = new MakeRoom(2);
        Door d = makeDoor(r1, r2);
        m.addRoom(r1);
        m.addRoom(r2);
          :
        return m;
    }
}

public class Bombed {
    // common "Bombed" feature
    public void bomb() {
          :
    }
}

// if BombedMazeGame has to inherit from another class Bombed
public class BombedMazeGame extends Bombed {
    MazeGame mg;

    public BombedMazeGame() {
        mg = new MazeGame();
        mg.makeWall = makeWall;
        mg.makeRoom = makeRoom;
        createMaze = mg.createMaze;
    }

    public Maze createMaze() {
        return null;
    }

    public Wall makeWall() {
        return new BombedWall();
    }

    public Room makeRoom(int n) {
        return new RoomWithABomb(n);
    }
      :
}

public class Enchanted {
    // common "Enchanted" feature
    public void enchant() {
          :
    }
}

// if EnchantedMazeGame has to inherit from another class Enchanted
public class EnchantedMazeGame extends Enchanted {
    MazeGame mg;

    public EnchantedMazeGame() {
        mg = new MazeGame();
        mg.makeRoom = makeRoom;
        mg.makeDoor = makeDoor;
        createMaze = mg.createMaze;
    }

    public Maze createMaze() {
        return null;
    }

    public Room makeRoom (int n) {
        return new EnchantedRoom(n, castSpell());
    }

    public Door makeDoor(Room r1, Room r2) {
        return new DoorNeedingSpell(r1, r2);
    }

    public Spell castSpell() {
          :
    }
      :
}

// client usage is the same as the one in Java
BombedMazeGame bmg = new BombedMazeGame();
Maze m1 = bmg.createMaze();
EnchantedMazeGame emg = new EnchantedMazeGame();
Maze m2 = emg.createMaze();
```

Consequences

In this sample we replace inheritance with the binding for delegating factory methods.

Locality	In order to avoid implementing the same logic twice, BombedMazeGame and EnchantedMazeGame can inherit from Bombed and Enchanted, respectively.
Non-inheritance	The factory methods in creator are delegated by binding, so concrete creators can inherit from other classes.

4.5 Template Method Pattern

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."—From the GoF book. This pattern divides an operation in one class into some steps and let subclasses have the ability to redefine the implementation in the steps. On the other hand, we cannot let the subclasses take advantage of other classes by multiple inheritance. With DominoJ the "subclasses" role can inherit from another class, and the original "parent class" role still can defer the steps by binding.

Sample Code in Java

```java
// clients always call display method of View
public class View {
    public void display() {
        // this operation are divided into 3 steps
        // only doDisplay method can be replaced by subclasses
        setFocus();
        doDisplay();
        resetFocus();
    }

    private void setFocus() {
          :
    }

    // doDisplay do nothing in this class. subclasses can override it
    protected void doDisplay() {}

    private void resetFocus() {
          :
    }
      :
}

// subclasses override doDisplay method to add specific drawing behavior
public class FancyView extends View {
    protected void doDisplay() {
        // add specific drawing behavior here
          :
    }

    // common "Fancy" feature
    public void getInfo() {
          :
    }
}

// client usage
FancyView view = new FancyView();
view.display();
```

Sample Code in DominoJ

```java
// clients always call display method of View
public class View {
    public void display() {
        // this operation are divided into 3 steps
        // only doDisplay method can be replaced by binding
        setFocus();
        doDisplay();
        resetFocus();
    }

    private void setFocus() {
          :
    }

    // doDisplay must be public
    public void doDisplay() {}

    private void resetFocus() {
          :
    }
      :
}

public class Fancy {
    // common "Fancy" feature
    public void getInfo() {
          :
    }
}

// if FancyView has to inherit from another class Fancy
public class FancyView extends Fancy {
    View view;

    public FancyView() {
        view = new View();
        view.doDisplay = doDisplay;
    }

    public View getView() {
        return view;
    }

    public void doDisplay() {
        // add specific drawing behavior here
          :
    }
}

// client usage
FancyView my = new FancyView();
View view = my.getView();
view.display();
```

Consequences

In the sample FancyView can use the logic in Fancy by inheritance rather than copy it to FancyView. And the redefinition of doDisplay method still can be used by View.

Locality	FancyView can inherit from Fancy to avoid duplicating code.
Non-inheritance	DominoJ provides an alternative to inheritance for this pattern. This pattern can be applied even if FancyView does not inherit from View. We can decide whether using inheritance or binding depending on the circumstances.

5 Parameters passing

5.1 Adapter Pattern

Intent

"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces."—From the GoF book. The binding can pass parameters between methods, but the signatures of methods must be the same except their names. The simplest case is that the methods in adapter and adaptee are almost the same except their names, but it is a rare case. In the following example taken from [1], the binding can be used in the adapter for bypassing to its component and setting execution order.

Sample Code in Java

```
// if you would like to rewrite an awt program using JFC
// JFC JList is adaptee, and awt List is adapter
// define an interface according to awt List
public interface awtList {
    public void add(String s);
    public void remove(String s);
    public String[] getSelectedItems();
}

// object adapter
// wrap JFC JList and let it looks like awt List
// client can use it without changing original methods in the program
public class JawtList extends JScrollPane implements awtList {
    private JList window;
    // use JListData as model. its definition is shown below
    private JListData contents;

    public JawtList(int rows) {
        contents = new JListData();
        window = new JList(contents);
          :
    }

    public void add(String s) {
        contents.addElement(s);
    }

    public void remove(String s) {
        contents.removeElement(s);
    }

    public String[] getSelectedItems() {
        Object[] obj = window.getSelectedValues();
        String[] s = new String[obj.length];
        for(int i=0; i<obj.length; i++) {
            s[i] = obj[i].toString();
        }
        return s;
    }
}

// define JListData, the model used in JList
public class JListData extends AbstractListModel {
    private Vector data;

    public JListData() {
        data = new Vector();
    }

    public void addElement(String s) {
        data.addElement(s);
        fireIntervalAdded(this, data.size()-1, data.size());
    }

    public void removeElement(String s) {
        data.removeElement(s);
        fireIntervalRemoved(this, 0, data.size());
    }
}
```

Sample Code in DominoJ

```
// the interface is the same as the one in Java
public interface awtList {
    public void add(String s);
    public void remove(String s);
    public String[] getSelectedItems();
}

// can be used to bypass implementation or pass parameters
// when the signature of methods are the same
public class JawtList extends JScrollPane implements awtList {
    private JList window;
    private JListData contents;

    public JawtList(int rows) {
        contents = new JListData();
        window = new JList(contents);
        // bypass to contents
        // maybe some methods can be bypassed directly
        add = contents.addElement;
        remove = contents.removeElement;
          :
    }

    // empty methods for binding
    public void add(String s) {}
    public void remove(String s) {}

    public String[] getSelectedItems() {
        Object[] obj = window.getSelectedValues();
        String[] s = new String[obj.length];
        for(int i=0; i<obj.length; i++) {
            s[i] = obj[i].toString();
        }
        return s;
    }
}

// define JListData
public class JListData extends AbstractListModel {
    private Vector data;

    public JListData() {
        data = new Vector();
        // you may put data related code here
        // and set the execution order
        addElement ^= data.addElement;
        removeElement ^= data.removeElement;
    }

    public void addElement(String s) {
        fireIntervalAdded(this, data.size()-1, data.size());
    }

    public void removeElement(String s) {
        fireIntervalRemoved(this, 0, data.size());
    }
}
```

Consequences

The method bypassing is quite intuitive and makes code clear. The execution order setting is useful when there are more complicated execution dependencies. By setting the execution order we can avoid calling them one by one in methods explicitly.

Locality	The code of method bypassing and execution order setting can be gathered together. This makes the code easy to understand.

5.2 Facade Pattern

Intent

"Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use."—From the GoF book. For this pattern, the binding only can be applied when the interface of subsystems are designed for it. Because the signature of methods in a binding must be the same, it needs some trick to define the interface of subsystems. If the interfaces can be bound by DominoJ, the interaction among subsystems is more clear and easy to change.

Sample Code in Java

```
// there are 3 subsystems: authentication, authorization, and database
public class Authentication {
    public Token authenticate(String id, String pw) {
          :
    }
      :
}

public class Authorization {
    public Ticket authorize(Token t) {
          :
    }
      :
}

public class Database {
    public FileList getFileList(Ticket t) {
          :
    }
      :
}

// the facade provides a simple method to get file list by id and password
// without sending request to several subsystems in order
public class Facade {
    public FileList getFilesByIdPw(String id, String pw) {
        // get a token from authentication subsystem
        Authentication authen = Authentication.getInstance();
        Token token = authen.authenticate(id, pw);
        if(token == null)    return null;
        // give authorization subsystem the token to get a ticket
        Authorization author = Authorization.getInstance();
        Ticket ticket = author.authorize(token);
        if(ticket == null)    return null;
        // give database subsystem the ticket for querying file list
        Database database = Database.getInstance();
        return database.getFileList(ticket);
    }
      :
}
```

Sample Code in DominoJ

```
// the three subsystems
public class Authentication {
    public Token authenticate(String id, String pw) {
          :
    }

    // add an adapter method
    public Object authenticate(Object[] obj) {
        if(obj.length < 2 || ! (obj[0] instanceof String) || ! (obj[1] instanceof String))    return null;
        return    authenticate((String)obj[0], (String)obj[1]);
    }
      :
}

public class Authorization {
    public Ticket authorize(Token t) {
          :
    }

    // add an adapter method
    public Object authorize(Object[] obj) {
        if(! ($ret instanceof Token))    return null;
        return    authorize((Token)$ret);
    }
      :
}

public class Database {
    public FileList getFileList(Ticket t) {
          :
    }

    // add an adapter method
    public Object getFileList(Object[] obj) {
        if(! ($ret instanceof Ticket))    return null;
        return    getFileList((Ticket)$ret);
    }
      :
}

// the facade class
public class Facade {
    public Facade() {
        Authentication authen = Authentication.getInstance();
        Authorization author = Authorization.getInstance();
        Database database = Database.getInstance();
        // authenticate always should be done before authorize
        author.authorize ^= authen.authenticate;
        // authorize always should be done before getFileList
        database.getFileList ^= author.authorize;
        // define more execution order
          :
    }

    public FileList getFilesByIdPw(String id, String pw) {
        // just focus on main logic
        Database database = Database.getInstance();
        Object obj = database.getFileList(new Object[]{id, pw})
        if(obj instanceof FileList)    return (FileList)obj;
        else    return null;
    }
}
```

Consequences

With the defined execution order, we can avoid putting similar logic into every method. In this sample most methods in Facade need authentication and authorization. We move the calling from getFilesByIdPw to the constructor, so we do not have to write the calling for other methods again. The return value of getFilesByIdPw also can be sent to other methods by binding.

Locality	Similar method calling in methods of the facade class can be collected. The execution order can be expressed more meaningfully. When the basic behavior is changed, for example the parameters of authorize method is changed or adding one operation between authenticate and authorize, we do not need to check the calling in every method.

5.3 Mediator Pattern

Intent

"Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."—From the GoF book. In this pattern the binding can be applied only when the interface of objects are designed for it. In order to bind between methods, the signature of methods must be the same except their names.

Sample Code in Java

```java
// the base class of colleague classes. it knows its mediator
public class Widget {
    // keep its mediator
    private DialogDirector director;

    public Widget(DialogDirector d) {
        director = d;
    }

    public void changed() {
        director.widgetChanged(this);
    }
      :
}

// colleague class
public class Button extends Widget {
      :
}

// colleague class. a widget to let user picks up a string for the list
public class ListBox extends Widget {
    // keep current selected string
    private String current;

    public String getSelection() {
        return current;
    }

    // this method will be called when user clicks on an item in the list
    public void setSelection(String s) {
        current = s;
        changed();
    }
      :
}

// colleague class. a widget which shows a string
public class EntryField extends Widget {
    // the string shown in the field
    private String text;

    public void setText(String s) {
        text = s;
        changed();
    }
      :
}

// the mediator
public class DialogDirector {
    public abstract void widgetChanged(Widget w);
    protected abstract void createWidgets();
      :
}

// a concrete mediator for coordinating colleagues in FontDialog
public class FontDialogDirector extends DialogDirector {
    private Button ok;
    private Button cancel;
    private ListBox fontList;
    private EntryField fontName;

    public void widgetChanged(Widget w) {
        if(w == fontList) {
            // pass the selected string from ListBox to EntryField
            fontName.setText(fontList.getSelection());
        } else if(w == fontName) {
              :
        } ...
    }

    protected void createWidgets() {
        ok = new Button(this);
        cancel = new Button(this);
        fontList = new ListBox(this);
        fontName = new EntryField(this);
    }
}
```

Sample Code in DominoJ

```
// the base class of colleague class. no need to keep its mediator
public class Widget {
    // no need to notify director manually
      :
}

// colleague class
public class Button extends Widget {
      :
}

// colleague class. a widget to let user picks up a string for the list
public class ListBox extends Widget {
    // keep current selected string
    private String current;

    public String getSelection() {
        return current;
    }

    // this method will be called when user clicks on an item in the list
    public void setSelection(String s) {
        current = s;
        // no need to call changed()
    }
      :
}

// colleague class. a widget which shows a string
public class EntryField extends Widget {
    // the string shown in the field
    private String text;

    public void setText(String s) {
        text = s;
        // no need to call changed()
    }
      :
}

// the mediator
public class DialogDirector {
    protected abstract void createWidgets();
      :
}

// concrete mediator which coordinates colleague objects
public class FontDialogDirector extends DialogDirector {
    private Button ok;
    private Button cancel;
    private ListBox fontList;
    private EntryField fontName;

    protected void createWidgets() {
        ok = new Button(this);
        cancel = new Button(this);
        fontList = new ListBox(this);
        fontName = new EntryField(this);
        // define the behavior here
        // no need to call changed() in every widget,
        // and the director doesn't have to switch the action in widgetChanged()
        fontList.setSelection += fontName.setText;
    }
}
```

Consequences

In this sample we bind setText method in EntryField to setSelection method in ListBox because the parameter passing is simple. When user clicks on a string, the string will be sent to EntryField automatically. The binding avoid calling getSelection method and switching actions in widgetChanged method of the mediator class.

Locality	The behavior is defined in the mediator, not depends on the calling in colleague classes. We can manage the behaviors in one place.
Unpluggability	We can just remove the mediator class to unplug this pattern. Colleague classes are unaware of the mediator because they do not contain the pattern code.

6 No benefit to structure

Method slots provide no benefit to Command pattern, Flyweight pattern, Interpreter pattern, Iterator pattern, Memento pattern, Prototype pattern, and Singleton pattern. The implementation in Java and in DominoJ are the same.

Why they cannot benefit from DominoJ

These patterns focus on their structures and object creation, but DominoJ does not have the ability to refine structure. Although DominoJ can replace method implementation, it cannot replace the constructor to refine object creation.

References

[1] James W. Cooper. *The Design Patterns Java Companion*. Addison-Wesley, 1998.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[3] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: modular event-driven object interactions in Scala. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 227–240, New York, NY, USA, 2011. ACM.

[4] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 161–173, New York, NY, USA, 2002. ACM.