Aspect Composition Using Composition Filters

Jianjun Zhao
School of Software,
Shanghai Jiao Tong University

Some slides adapted from Prof. Mehmet Aksit ’s talk
Agenda

• The Composition Filters Model
  – The structure of Composition filters
  – Pseudo variables in composition filters specifications
  – Filter types
• Composition filters in Java – ComposeJ
• Design patterns problems and ComposeJ solutions for them
• AspectJ Vs. HyperJ Vs. ComposeJ
• Opens

Mehmet Aksit (University of Twente)
The Composition Filters Model

• Filters extend the classic object interface, by interfering the object messages sending mechanism
• The extension is not at the level of the object itself (implementation), it is at the level of its interface
• Message are inspected and manipulated at runtime
• Filters are added to objects in a modular way
Normal and Composed Object

incoming message
Object
outgoing message
Composed Object
The structure of Composition filters

• CFs are specified on top of another language in a so-called *interface definition*

• The structure of this interface definition is largely language independent

• An implementation – block, that realizes the defined interface, is part of the filter definition and is language dependent

• Both interface and the implementation specifications are defined separately; thus the language-independent part is specified separately from the language-dependent part
The interface definition

• **Internals**
  – Fully encapsulated objects that are used to compose the behavior of the composition filters objects

• **Externals**
  – Objects that exists outside the composition filter object, such as global objects

```class
class Example interface
  internals
  externals
  conditions
  methods
  inputfilters
  outputfilters
end;
```
The interface definition (cont.)

• **Methods**
  – Names of methods that are available to other objects.

• **Conditions**
  – Typically represent an object’s state, used in a filter to test whether a message should be accepted or rejected.

• **Input filters**
  – Filters are defined in this block for inspecting and manipulating incoming message.

• **Output filters**
  – Filters are defined in this block for inspecting and manipulating outgoing message.
Implementation of USViewMail using CF’s

Filter: multipleViews

Filter: invoke

USViewMail

userView; systemview;

EMail

putOriginator(anOriginator); putReceiver(aReceiver);
putContent(aContent);
getContent;
send;
reply;
approve;
putRoute(aRoute);
deliver;
getOriginator;
getReceiver;
getRoute;
isDelivered;
isApproved;

multipleViews: Error =
{
    userView => {putOriginator, putReceiver,
                 putContent, getContent, send,
                 reply},
    systemView => {approve, putRoute, deliver},
    True => {getOriginator, getReceiver,
             isApproved, getRoute, isDelivered}
}

invoke: Dispatch =
{
    True=> inner.*, mail.*
};
class USViewMail interface

internals
  mail: Email;

/* no externals */

conditions
  userView;
  systemView;

methods

... ...

inputfilters

multipleViews: Error = { userView => {putOriginator, putReceiver, putContent, getContent, send, reply},

          systemView => {approve, putRoute, deliver},
          True => {getOriginator, getReceiver, isApproved, getRoute, isDelivered}

    };

invoke: Dispatch = { True=> inner.*, mail.* }; /* no outputfilters */

end;
CF object: an implementation - block

• Method implementations
  – Corresponds to the implementation of all those methods that are not implemented by the current object signature.

• Condition implementations
  – Idem as methods, but intended for conditions.

• Instance variables
  – Corresponds to variables that can be used by the defined methods and conditions, but cannot used by filters.
Incorporation of Composition – Filters over an Object
From Normal object to composed object
The Filter declaration

- Filter name
- Filter handler
- Filter Initializer
- Filter elements
The filter element

In the USViewMail example:

Conditions: userView
Conclusion Operator: =>
Matching Pattern: {putOriginator, putReceiver, putContent, getContent, send, replay}
Parameters: 

Conclusion Operator O

Operator Matching Pattern
The filter element (cont.)

• **Conditions**
  – A single condition takes no parameters and has no side effects
  – Several conditions can be composed using standard boolean operators and brackets
  – If the condition part evaluates to false, the filter element will always reject the message and continue according to its rejection behavior
  – If the condition part evaluates to true, conclusion operator and matching pattern are considered
The filter element (cont.)

- **Conclusion Operators**
  - The inclusion operator: $\Rightarrow$
    - Causes the acceptance of every message that matches the matching pattern.
  
  - The exclusion operator: $\sim\Rightarrow$
    - Causes the acceptance of every message, except for the message accepted by the conclusion operator.
The filter element (cont.)

• Message pattern matching
  – Before message is accepted by specific filter element, it must match the filter element’s matching patterns.
  – Message have the following pattern: Target.Selector
  – Both the target and the selector can be replace by an asterisk wildcard.
  – There are three types of matching
    • Signature matching – check whether the selector is within the signature of the target message
    • Name matching – perform only name matching.
    • Selector matching
Message pattern matching (cont.)

- $\sigma(x)$ denotes the signature of the object $x$
- $msg$ denotes the message being evaluated

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Signature matching</th>
<th>Name matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>aTarget.aSelector</td>
<td>$(\text{selector } msg = \text{aSelector} &amp;&amp; \text{aSelector } \in \sigma(\text{aTarget}))$</td>
<td>$(\text{selector } msg = \text{aSelector} &amp;&amp; \text{target } msg = \text{aTarget})$</td>
</tr>
<tr>
<td>aTarget . *</td>
<td>$(\text{selector } msg \in \sigma(\text{aTarget}))$</td>
<td>$(\text{target } msg = \text{aTarget})$</td>
</tr>
<tr>
<td>* . aSelector</td>
<td>$(\text{selector } msg = \text{aSelector} &amp;&amp; \text{aSelector } \in \sigma(\text{target msg}))$</td>
<td>$(\text{selector } msg = \text{aSelector})$</td>
</tr>
<tr>
<td>* . *</td>
<td>$(\text{selector } msg \in \sigma(\text{target msg}))$</td>
<td>$(\text{true})$</td>
</tr>
</tbody>
</table>
Pseudo variables in composition filters specifications

- OO languages usually provide a certain pseudo variables, such as: super, this etc..
- CF extension introduces a number of pseudo variables.
  - Sender
  - Server
  - Self/Outer/This
  - Inner
  - Message
Pseudo variables (cont.)

- **Thread execution of a message:**
  - Message is sent to aSecretary object.
  - aSecretary dispatches the message to its internal anEmployee.
  - anEmployee triggers the execution of its inner-object’s methods.
  - New message is sent to the object aClerk.
  - aClerk dispatches the message to its internal anEmployee.
  - anEmployee executes one of its inner-object’s methods.
Filter types

- The Composition Filters model currently support the following filter types:
  - Dispatch
  - Error
  - Substitution
  - Send
  - Meta
  - Wait
Filter Types (cont.)

• Dispatch filter
  – Used for initiation of a message execution. The dispatch filter introduces ‘inheritance’ by delegation. The delegation introduced is multiple and dynamic
  – Used in Input filter set
  – The accepted message is delegated to the target object or executed if target object is inner
  – The rejected message continues to the next filter
Filter Types (cont.)

• Error filter

  – Used for restriction of class’ interface
  – Used in Input/Output filter set
  – The accepted message proceeds to the next filter
  – The rejected message is regarded as illegal and the program raises an exception stating where the exception occurred
Filter Types (cont.)

• **Substitution filter**
  
  – Used for substitution of the target, selector or target and selector of the message.
  – Used in Input/Output filter set.
  – The accepted message is substituted by the target and selector specified in the parameter part.
  – The rejected message continues to the next filter.
  – The parameters are the new target and/or selector.
Filter Types (cont.)

- **Send filter**
  - Used for initiation of a message delegation.
  - Used in Output filter set.
  - The accepted message is passed to another object outside the sending object.
  - The rejected message continues to the next filter.
Filter Types (cont.)

• Meta filter

  – Used for reflection on object interactions.
  – Used in Input/Output filter set.
  – The accepted message is reified and delegated to a meta object.
  – The rejected message continues to the next filter.
  – The parameters are the target meta object and the selector of the function to be called.
Filter Types (cont.)

• Wait filter
  – Used for concurrency control and synchronization
  – Used in Input/Output filter set
  – The accepted message message continues to the next filter
  – The rejected message is blocked until one or more conditions are satisfied. The message is then re-evaluated by the wait filter
Composition filters in Java - ComposeJ

- ComposeJ is a Composition-Filters implementation for the Java language
- **Compile time oriented source code modification**
  - CF definition is considered as a meta-Java program, that can be transformed back to Java language by using a preprocessor
  - Compile time oriented approach is used, mainly due to performance issues
  - CF implement by translating them to Java and weaving them through a Java file
- **A gap emerges between the conceptual view and implementation**
Composition filters in Java – ComposeJ (cont.)

• ComposeJ preprocessor extends the Java compiler
• Filter definitions extend Java files
• To define Composition Filters class in ComposeJ there must be two specifications:
  – A class-interface (a “.cf” file).
  – An implementation that realizes the class-interface (a “.jwf” file which is a “.java” file without filters).
• Current implemented Filters:
  – Dispatch Filter
  – Error Filter
class CFOMailMessage 

    public String getContent() throws Exception 
            
            String senderName; 
            if (sender == null) 
                senderName = sender.getClass().getName(); 
            else 
                senderName = "An unknown object"; 
            return (senderName + " is sending this request: " + 
                    server.getClass().getName() + " is serving this request. In Hello dear... ")

    ```
```
Example I: The State pattern

```
DrawingController
  MousePressed ()
  ProcessKeyboard ()
  Initialize ()

currentTool

Tool
  name
    HandleMousePress ()
    HandleMouseRelease ()
    HandleCharacter ()
    GetCursor ()
    Activate ()

CreationTool
SelectionTool
TransformTool
```
**Example I: The State pattern (cont.)**

- **Filter definition written in ComposeJ**

```java
class DrawingController interface
    internals
        importdeclaration
            import Graphics.DrawingTools.*
    objectdeclaration
        CreationTool creator = new CreationTool();
        SelectionTool selector = new SelectionTool();
        TransformTool transformer = new TransformTool();
    conditions
        public boolean creating();
        public boolean selection();
        public boolean transforming();
    methods
        public void MousePressed (MouseEvent e);
            ...
            ...
    Inputfilters
        invoke: Dispatch = { inner.*,
            creating()  => creator.*,
            selecting() => selector.*,
            transforming() => transformer.*
        end;
```
Example II: The Composite pattern

Graphic
- Draw ()
- Add (Graphic)
- Remove (Graphic)
- GetChild (child_id)

Line
- Draw ()

Rectangle
- Draw ()

Text
- Draw ()

Picture
- Draw ()
- Add (Graphic)
- Remove (Graphic)
- GetChild (child_id)
Example II: The Composite pattern (cont.)

- Filter definition written in ComposeJ

```java
class GraphicWithViews interface
externals
    Graphic graphic;

conditions
    public boolean compositeView();
Inputfilters
    multipleviews: Error = {compositeView() => {add, remove, getchild},
                        true ~> {add, remove, getchild}};

end;
```

```java
class LockableGraphicWithViews interface
externals
    GraphicWithViews graphic;
conditions
    public boolean isLocked();
methods
    public void lock(); public void unlock();
Inputfilters
    precondition: Error = {isLocked() => {unlock, lock}, !isLocked() => *};

end;
```
AspectJ vs. HyperJ vs. ComposeJ

How does AspectJ solve crosscutting problems?

– Several aspects can be applied over a system
– Specified events by pointcut definitions
– The specified events can be caught, and advices can be applied to take care of them
– The advices can be executed before, after & around the code that generated the event
– Desired code can be added to existing applications at the level of class definitions
How does HyperJ solve crosscutting problems?

- Multi-Dimensional Separation of Concerns
- A **Hyperspace** contains specifications for dimensions and concerns of importance
- A **Concern-Matrix** specifies relations among concerns
- A **Hyperslice** is a set of concerns that are declaratively complete
- An **Hypermodule** is a set of **Hyperslices**, that indicates the integration mechanism among each **Hyperslice**
- Crosscutting problems are specified into **Hypermodules** and **Hyperslices**
- Concerns are woven with system code and are statically applied at compile-time
AspectJ vs. HyperJ vs. ComposeJ (cont.)

• How does ComposeJ solve Crosscutting problems?

  - This is a question of a well definition of filters and interfaces and how filters defines the correct semantic of the modeled problem
Who is the best?

- Is something missing?
- Can we merge them all?
  - Yes, ConcernJ