21) Composition Filters - A Filter-Based Grey-Box Component Model

1. Inheritance Anomaly
2. Design Pattern Decorator
3. Composition Filters
4. Implementations of the Filter Concept in Standard Languages
5. Composition Filters and Role-Object Pattern
6. Evaluation
Literature (To Be Read)

  - http://trese.cs.utwente.nl
Other Literature

- On the TRESE home page, there are many papers available for CF [http://trese.cs.utwente.nl/](http://trese.cs.utwente.nl/)
**Goal**

- Composition Filters (CF) are a solution to many composition problems
- The first approach to grey-box components
- Understand the similarity to decorator/adapter-based component models, and why grey-box provides an advantage
21.1) Inheritance Anomaly
Inheritance Anomaly - Why Software Composition Is Necessary

In a parallel program, where should synchronization code be inserted?
- Stack?
- Queue?
- OrderedCollection?
- Collection?
- Object?

```
LockedStack
s: semaphor;
pop();
wait(s);
super.pop();
free(s);
```

```
OrderedCollection
add()
```

```
Stack
pop()
```

```
PlainStack
```

```
Queue
enter()
```

```
PriorityQueue
```
At the beginning of the 90s, parallel object-oriented languages failed, due to the inheritance anomaly problem.

_Inheritance anomaly_: In inheritance hierarchies, synchronization code is _tangled (interwoven)_ with the algorithm,

- and cannot be easily exchanged
- when the inheritance hierarchy should be extended
- Ideally, one would like to specify algorithm and function independently.
Algorithm and Synchronization are Almost Facets

- But they depend on each other
- How to mix them appropriately?
21.2 The Decorator Design Pattern (Rpt.)
**Decorator Pattern**

- A Decorator is a *skin* of another object.
- It is a 1-ObjectRecursion (i.e., a restricted Composite):
  - A subclass of a class that contains an object of the class as child.
  - However, only one composite (i.e., a delegatee).
- Combines inheritance with aggregation:
  - Inheritance from an abstract Handler class.
  - That defines a contract for the mimiced class and the mimicing class.

---

![Diagram](image-url)
Decorator for Widgets

- **Widget**
  - `draw()`

- **TextWidget**
  - `draw()`

- **WidgetDecorator**
  - `draw()`
  - `mimiced.draw()`

- **Frame**
  - `super.draw(); drawFrame();`
  - `draw()`

- **Scrollbar**
  - `super.draw(); drawScrollbar();`
  - `draw()`
Decorator for Persistent Objects

Record
access()

TransientRecord
access()

PersistentDecorator
access()

PersistentRead
access()

OnlyRecord
access()
boolean loaded()
load()

PersistentRecord
access()
boolean loaded()
boolean modified()
load()
dump()

if (!loaded()) load();
super.access();
if (modified()) dump();
Purpose Decorator

- For extensible objects (i.e., decorating objects)
  - Extension of new features at runtime
  - Removal possible
- Instead of putting the extension into the inheritance hierarchy
  - If that would become too complex
  - If that is not possible since it is hidden in a library
Variants of Decorators

► If only one extension is planned, the abstract superclass Decorator can be saved; a concrete decorator is sufficient.
► Decorator family: If several decorators decorate a hierarchy, they can follow a common style and can be exchanged together.
Decorator Relations

- Decorators can be chained to each other
- Dynamically, arbitrarily many new features can be added
- A decorator is a special ChainOfResponsibility with
  - The decorator(s) come first
  - Last, the mimiced object
21.3 Composition Filters
Filters are Layers

- Composition Filters (CF) wraps objects with *filters*
- Messages flow through the filters
  - are accepted or rejected
  - are modified by them
- A filter is an *interceptor* that is part of an object
**Filters are Special Decorators**

- Filters are decorators that do not suffer from object schizophrenia

Diagram:
- Input filters
- Object Facade
- Decorator
- Output filters
- Object Implementation
- Self
- Inner
Filter Types

- **Error.** An error filter tests whether a message is available.
  - If not, it stops filtering and execution.

- **Wait.** A wait filter accepts methods only if a condition is true, otherwise it waits until the condition becomes true.
  - The condition may refer to a semaphore that is shared by all objects of the class
  - In case the semaphore is not free, the wait filter blocks execution

- **Dispatch.** A dispatch filter dispatches the message
  - to the internal implementation,
  - to other external objects, to a superclass,
  - or to sequences of objects.

- **Meta.** A meta filter converts the message to an instance of class Message and passes it on to the continuation method. Then, the method can evaluate the new message.

- **RealTime.** Specify a real-time constraint.
Filters in SINA

Grammar:

InputFilters = 'inputfilters "< Filter*' '>'.
OutputFilters = 'outputfilters' '< Filter* '>'.
Filter ::= Name ':' Type '=' '{' FilterElement // '',' '}.
FilterElement ::= Guard '=>' Match -- All matching messages are accepted
| Guard '~=>' Match -- All matching messages are rejected
| Replacement -- All matching messages are resent
Replacement ::= Guard '=>' '[' Match '] MATCH .
Guard ::= BooleanFunctionCall.
Match ::= TargetObject '.' MethodName | MethodName .
TargetObject ::= 'self' | 'inner' | '*'.
MethodName ::= Name | '*'.

-- All matching messages are accepted
-- All matching messages are rejected
-- All matching messages are resent
Filters in SINA

- Sync Filter example:
  ```
  sync:Wait = { NonEmpty => pop,
               True => * }
  ```

- Meaning:
  ```
  if (sync.Semaphore free)
      if (NonEmpty())
          if (function.name == "pop") inner.pop
      else if (True)
          if (function.name == X) inner.X
  ```
Wrapping Methods with Calls

- Meta-filter example:
  - Full => [put] bufferDistribute.Distribute;
  - Empty => [get] bufferDistribute.Distribute;

- Wrapping Methods with Calls with the Meta filter:
  
  ```
  counterWrapper: Meta {
    isCounting => [put] Counter.increaseCount();
    True => [*] inner.*;
  }
  ```
class PressOrAnimatedPress interface
    internals:
        visualize;
        doIt;
    externals:
        animatedDevice:AnimatedDevice;
    conditions:
        isAnimating;
        isInTracingMode;
        noOneElseIsAnimating;
    methods:
        inputTraceMethod;
        outputTraceMethod;
    inputfilters:
        tracing: Meta = {
            isInTracingMode => [*] inputTraceMethod }
        lockingDisplay: Wait = {
            noOneElseIsAnimating => *; }
    dispatch: Dispatch = {
        isAnimating => [.*] animatedDevice.*;
        True => [*] inner.*; }
    outputfilters:
        tracing: Meta = {
            isInTracingMode => [*] outputTraceMethod }
end

• A press is modeled, either with or without animation.
• There are two Meta filters that call tracing methods when
  the press is in animation mode (precondition isAnimating).
  • The filters match all messages (pattern [*]) and call
    tracing methods.
  • Then, they pass on control to the next filter.
• As an input filter, a Wait filter is executed.
  • It collaborates with other animated devices and
    guarantees with a semaphore that only one device
    at a time uses the display.
  • If another device is animating, the wait filter blocks
    execution until the display is free again.
• The Dispatch filter selects a method for the real
  implementation work.
  • It contains two filter elements.
  • If the press is in animation mode, it forwards every
    message from an arbitrary object (pattern [.*]) to the
    animated device delegatee, otherwise calls its inner
    object.
Main Advantage of the Filter Concept

► Filters are *built into* an object, they are *grey-box decorators*
► Filters are specified in the interface, not in the implementation
  ■ Implementations are free of synchronization code
  ■ Separation of concerns (SOC): synchronization and algorithm are separated
  ■ Filters and implementations can be varied independently
► Filters are specified statically, but can be activated or deactivated dynamically
► Filters are statically composed with multiple inheritance
  ■ One dimension from algorithm,
  ■ one from synchronization strategy
  ■ Filters can be overwritten during inheritance
Filters Can be Multiply Inherited

Filters are composed by boolean AND
Composing a Locking Stack

- Additionally, filter composition has to be specified:

```java
class LockingState interface internals
    superStack: Stack
    locker: BinarySemaphore
inputfilters
    <locker.locking;
    superStack.sync;
    disp: Dispatch={superStack.*, locker.*};
>
```

Sequential AND composition

**Diagram:**
- **OrderedCollection**
  - `add()`
- **LockProtocol**
- **Stack**
  - `pop()`
- **LockingStack**
- **Semaphor**
  - `wait()`
  - `free()`
- **Binary Semaphore**
21.4 Implementations of the Filter Concept in Standard Languages
The superclass of the Decorator pattern implements the object interface:
- The decorating classes are the filters
- Problem: Decorators do not provide access to the “inner” object or the “self” object

Filters also can be regarded as ChainOfResponsibility:
- However, there is a final element of the Chain, the object implementation
Filters Can be Composed From Outside

- Filter superimposition

Diagram:
- Input filters
- Object Facade
- Object Implementation
- Output filters
- NewDecorator
Filters Can be Composed From Outside

NewDecorator

Object Implementation

superimposition

Object Implementation

Object Implementation

Object Implementation

Object Implementation
Superimposing a Decorator in Hand-Written Code

► Walk through the list of decorators
► Insert a new decorator where appropriate

► Example: superimposing synchronization:
  ■ Do for all objects involved:
    • Get the first decorator
    • Append a locking decorator, accessing a common semaphore

► Removing synchronization
  ■ Do for all objects involved:
    • Get the synchronizing decorator
    • Dequeue it
Superimposing Several Filters Produces Filter-Connector Pattern

- All Decorator-Connectors can be realized with filters
Filters in MOP-Based Languages

- In languages with a MOP, a filter can be implemented as a specific object that is called during the functions
  - enterObject
  - accessAttribute
  - callMethod
class Filter {
    // Test whether the filter can be applied to a method.
    public boolean matches(Method method) { .. }
    // Filter executes accept. Also, it substitutes a continuation.
    public Object acceptAction(Method method) {
        return substitute(method);
    }
    // Filter executes reject. Also, it substitutes a continuation.
    public Object rejectAction(Method method) {
        return substitute(method);
    }
    public Object substitute(Method method) {
        if (<<filtering should be stopped>>)
            return null;
        ..
        return <<continuationMethod>>;
    }
}

class FilteredClass extends Class {
    Filter[] inputFilters;
    public FilteredClass() { .. }
    public void enterMethod() {
        // First assign the called inner method to be the continuation
        Method continuation = thisMethod;
        // Run the input filters and calculate the real continuation
        for (int i = 0; i < inputFilters.size(); i++) {
            if (filter.matches(continuation))
                continuation = filter.acceptAction(continuation);
            else
                continuation = filter.rejectAction(continuation);
        // If the filter returned null, stop here
        if (continuation == null)
            return;
        // Continue at next filter
        if (continuation == inputFilters.getNext())
            continue;
        // Otherwise, continue at continuation
        else
            continuation.execute();
    }
    // Similar for output filters...
    Filter[] outputFilters;
    ..
A Specialized Filter

class TracingFilter extends Filter {
    public void matches(Method method) { return true; }
    public Object acceptAction(Method method) {
        trace();
        return substitute(method);
    }
    public Object substitute(Method method) {
        return method;
    }
    public void trace() {
        System.out.println("Here is the class "+getClass().getName());
    }
}

Class WorkPiece = new FilteredClass("WorkPiece",
    new Filter[] {TracingFilter},
    new Filter[] {});
21.5 Filters and The Role Object Pattern
Filter Layers

- Instead of role objects, filter objects can be used.
- Then, filters belong to layers:
  - Layers are like slices through the application.
  - We get a *layered object model*.
- The filters are separate objects (role objects):
  - Which can be exchanged separately.
  - Which can be superimposed appropriately.
Aksit's Filter Pattern in Framework Layers

Core Layer
- Customer
  - CustomerCore
  - CustomerRole
- Account
  - AccountCore
  - AccountRole

Security Layer
- TrustedCustomer
- Safe Account

Personalization Layer
- Personalized Customer
- Personalized Account
Using Filters

- Filters can implement a supercall (upcall) in the inheritance hierarchy
  - Delegating to an object of the superclass
  - In languages without inheritance

- Filters can implement multiple and mixin inheritance in languages with single inheritance

- Filters are applicable to all types of components
  - Filters are appropriate to implement the DCOM/COM+ facade-based component model
    - The dispatch filter delegates to aggregated objects
  - or to UML components
Filters In UML

- Realize as inner components
Insight: Greybox Composition Relies on Extensibility

- Composition Filters is a greybox composition technology
  - Because it inlines Decorators into objects
- Superimposition of filters can be used for greybox composition
  - Adding filters changes objects extensively, but the “self” identity does not change
  - Connectors can be made grey-box with the Filter-Connector pattern
21.6 Evaluation as Composition System
**CF - Component Model**

- Development environments
- Business services
- Infrastructure
- Versioning
- Parameterization
- Binding points
- Contracts
- Distribution
- Types
- Secrets
- Filters
CF – Composition Technique and Language

- Adaptation
- Product quality
- Software process
- Metacomposition
- Extensibility
- Connection
- Filters
- Separation of Concerns
- Aspect Separation
- Fully scalable distribution
- Scalability
CF as Composition System

Component Model
- Content: Filtered objects
- Binding points: ports

Composition Technique
- Dynamic adaptation by filters
- Scaling by exchange of filters

Simple composition language

Composition Language
What Have We Learned?

- CF extends the standard object model to a new component model *FilteredComponent*
  - The objects have filters and can be adapted easily
- Any component model that provides interceptors or decorators can be used as filtered component
- Filtered components support
  - Adaptation
  - Greybox composition
The End