41. Composition Filters - A Filter-Based Grey-Box Component Model

Lecturer: Dr. Sebastian Götz

Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de
21. Juni 2018

1. Inheritance Anomaly
2. Design Pattern Decorator
3. Composition Filters
4. Implementations of the Filter Concept in Standard Languages
5. Evaluation

http://trese.cs.utwente.nl

Compose* is the current tool for Composition Filters. It is an extension of Java

http://composestar.sf.net/
Other Literature

- On the TRESE home page, there are many papers available for CF 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
41.1 The Inheritance Anomaly
In a parallel program, where should synchronization code be inserted?

- Stack?
- Queue?
- OrderedCollection?
- Collection?
- Object?

---

**OrderedCollection**

- `add()`

**Stack**

- `pop()`

**LockedStack**

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

**PlainStack**

**Queue**

- `enter()`

**PriorityQueue**
Inheritance Anomaly

- 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?
41.2 The Decorator Design Pattern (Rpt.)
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
Decorator – Structure Diagram

MimicedClass

mimicedOperation()

ConcreteMimicedClass

mimicedOperation()

Decorator

mimicedOperation()

ConcreteDecoratorA

mimicedOperation()

ConcreteDecoratorB

mimicedOperation()

super.mimicedOperation();
additionalStuff();
mimiced.mimicedOperation();
Decorator for Widgets

Diagram:

- Widget
  - draw()

- TextWidget
  - draw()

- WidgetDecorator
  - draw()
  - super.draw();
  - drawFrame();
  - drawScrollbar():

- Frame
  - draw()

- Scrollbar
  - draw()
  - super.draw();
  - drawScrollbar();

mimiced

mimiced.draw()
Decorator for Persistent Objects

**Record**
- `access()`

**TransientRecord**
- `access()`

**PersistentDecorator**
- `access()`
  - `access()`
  - `mimiced.access()`

**PersistentRead**
- `access()`
- `boolean loaded()`
- `load()`

**OnlyRecord**
- `access()`
- `boolean loaded()`
- `load()`

**PersistentRecord**
- `access()`
- `boolean loaded()`
- `load()`
- `boolean modified()`
- `dump()`

```java
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.

![Diagram of Decorator Variants](image)
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
41.3 Composition Filters
Filters are Layers

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

- Filters are decorators that do not suffer from object schizophrenia
- “inner” is the core of the object
- “self” comprises all filters and inner
Filters are Event-Condition-Action rules

**Error.** An error filter tests whether a method exists.
- If not, it stops filtering and execution.
- In statically typed languages, error filters can be replaced by the compiler.

**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, the “inner”
- 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.
Main Advantage of the Filter Concept

- Filters are *built into* an object, they are *grey-box decorators*
  - They avoid object-schizophrenia
- 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
Composing a Locking Stack by Composing Filters

- Filter composition can be specified by selecting filters from superclasses
- Compose* can superimpose filters also dynamically
41.4 Implementations of the Filter Concept in Standard Languages
Implementation with Decorator

- 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 of filter superimposition](image)

- Object Facade
- Input filters
- Object Implementation
- Output filters
- NewDecorator
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
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
Filters In UML

- Realize as inner components

```
move

inputfilters:Filter

* 

Filter

outputfilters:Filter

<<FilteredComponent>>

Robot

Decorator

Robot Implementation
```
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
41.5 Evaluation
CF as Composition System

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

**Composition Technique**
- Static composition of filters by multiple inheritance
- Dynamic composition of filters by filter superimposition
- Dynamic adaptation by filters
- Scaling by exchange of filters

**Composition Language**
- Simple 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