Modeling with Filter Objects in Distributed Systems

Rushikesh K. Joshi

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Powai, Mumbai - 400 076, India.
Email: rkj@cse.iitb.ernet.in

Abstract. Filtering is emerging as an important programming abstraction in distributed object systems. We discuss the modeling capabilities of a first class filter object model in the context of distributed systems. Filter objects are transparent objects that are dynamically pluggable and provide selective filtering of messages. Filters can be injected into a system to dynamically evolve the system. The method is demonstrated with the help of an example application, a Transparent Distributed Decorator. A notation for representing static (class and object) and dynamic (interobject interactions) models in presence of filtering abilities is also discussed.

1 Introduction

Filter Objects [5] are first class objects which can transparently and selectively filter messages to objects to which they are dynamically plugged. Filtering ability is a consequence of a *filter relationship* between two classes. An IDL-centric design and implementation of filter objects for CORBA based distributed systems is presented in [9]. In this model, filter objects, which themselves are full fledged distributed objects, can be specified and plugged to *filter aware* distributed server objects.

Figure 1 depicts a filter relationship between two classes (between two interfaces in the case of component systems). Class *Dictionary* exports two member functions which are accessed by class *Client*. Class *Cache* is in filter relationship with class *Dictionary*. Filter relationship is shown with a special filter symbol at the filter-client end. In this case, class *Dictionary* is the filter-client and class *Cache* is its corresponding Filter class. It is possible to specify multiple filter classes in filter relationship with a filter-client class.

The upward filter function *Cache::searchCache* in Figure 1 filters message invocation of *Dictionary::searchWord* in upward direction. This filter function has the capability to bounce a result on behalf of the destination intended by the source of the message. The upward filter function in this case bounces a result to the client in the case of a cache miss. The bounce is thus a special form of return. The downward filter function *Cache::updateCache* filters the return result from an invocation of *Dictionary::searchCache* and updates the cache.

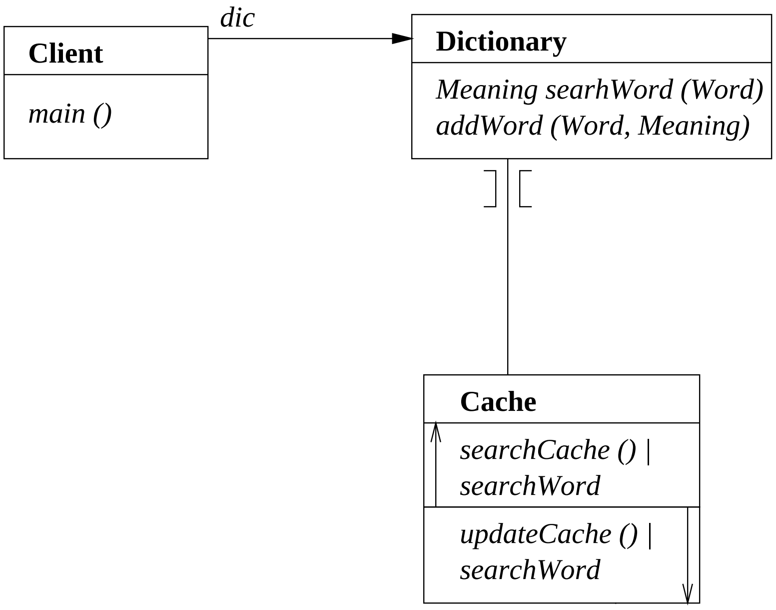

Fig. 1. Interclass Filter Relationship

Filter objects can selectively filter messages sent to their filter-clients. A filter function can be disabled and enabled during runtime. Filter objects are dynamically pluggable. For example, an instance of class *Cache* may be plugged to an instance of class *Dictionary* during runtime and replaced subsequently with another instance of a refinement of class *Cache* providing a better cache policy. Figure 2 models a filter relationship between instances. The filter notation also indicates the transparency of the filter object from the view of the client.

In this paper, we describe a transparent pattern based on filter objects for engineering distributed systems. The pattern is based on the filter configurations for distributed systems described in [4]. Implementations of Design Patterns based on conventional object oriented programming language features have been discussed extensively in literature. Filtering can be viewed as a meta-pattern introduced by Pree [8]. Apart from the hooks, templates and interclass relationships such as part-of, association and dependency, transparent filtering represents a distinct paradigm to engineer object systems.

2 Configurational Properties of Filter Objects

Filter objects transparently filter message sent to their corresponding filter clients. This makes client programs unaware of the filtering actions. Since filter objects are dynamically pluggable, new properties can be specified modularly in the path of messages between client and server objects. This ability is referred to as

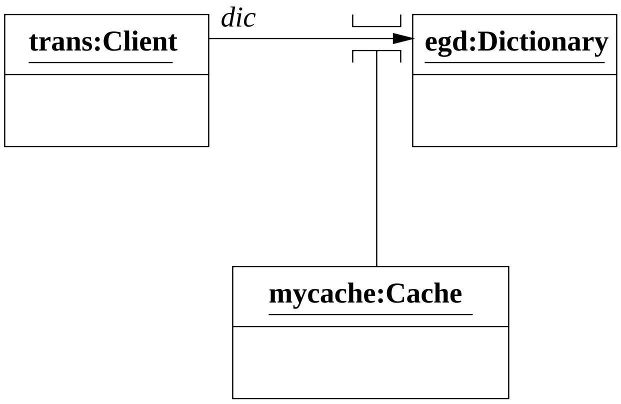

Fig. 2. Filter Relationship between Instances

separation of message control from message processing in [5]. This makes it possible for an architecture to incrementally evolve over time, in some cases without changing the existing source code or even the code in execution. A network of filter objects or individual filter objects may be injected into an executing software at runtime for evolution.

Filter Objects can implement configurations such as logger, replacer, router, value transformer, message transformer and repeater [4]. These configurations may also be implemented with any specific programming paradigm for filtering which provides the required transparency capabilities. The Distributed Transparent Decorator (DTD) that we describe in the next session is based on the *logger* filter configuration which attaches a new responsibility on an existing one without the knowledge of the client.

3 The Distributed Transparent Decorator

The decorator dynamically attaches additional responsibilities to an object [2]. The decorator implementation in [2] requires that the client is given an explicit pointer to the decorator object. Thus, it is not possible to inject a decorator into an existing object oriented system in which clients keep pointed to their intended servers. Filter objects provide an interesting solution to this problem. A filter object can be injected into the path of messages and can attach additional responsibilities in both upward and downward direction. Figure 3 shows the design of the Distributed Transparent Decorator based on interclass filter relationship.

The Decorator class is modeled as a filter class that forms filter relationship with the component class. The client *knows* only of the component while a decorator instance may be dynamically plugged to an instance of the component class. The filter member function *Decorator::addedOperation()* in class *Decorator* filters *Component::operation()* in upward direction. The filter member

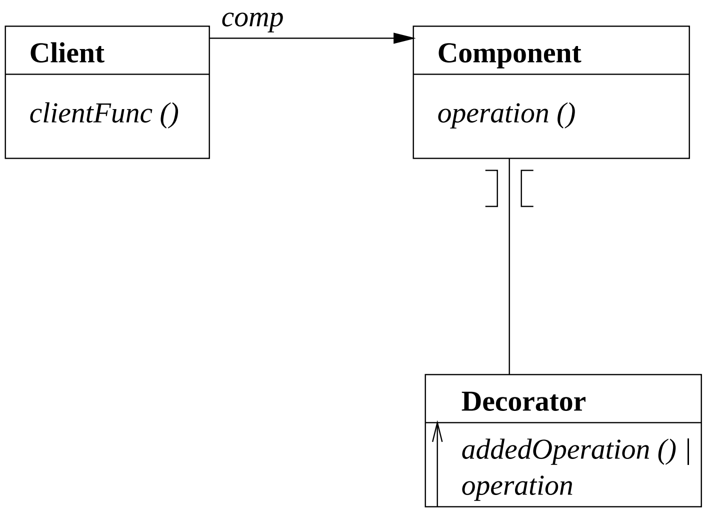

Fig. 3. DTD: Class Diagram

function can perform additional code and *pass on* the invocation to the intended destination.

An example decorator instance diagram is shown in Figure 4. The figure shows a logger as a decorator object that logs incoming requests to a local search engine from a query processor client. It can be noted that since the logger, which is a filter object, is itself an instance of a class, may be related with other classes in the system such as the class *LogFile* in the figure.

The solution to filter object based distributed decorator demonstrates an evolutionary distributed system that attaches new responsibilities in a message path dynamically and transparently, i.e. without the need to change the client or the server code. The dynamic behavior of the decorator implementation is modeled in Figure 5. The figure depicts an intermediate transparent filter object. The client object invokes a method *lse.getPage()*. The invocation in the figure is shown as an apparent invocation indicating the transparency of the filter object. The filter object captures this invocation *on-the-fly* and invokes a direct method on the log file object and subsequently passes the method on to the desired destination. The return result is not filtered in this case. It is also possible to *decorate* the return result with the help of the downward filtering action.

The following pseudo code represents an example decorator implementation. The filter member in the decorator performs its local computation before passing the message on to the intended destination. The filter interface that specifies filter member functions is not exported as a public contract, nor the methods

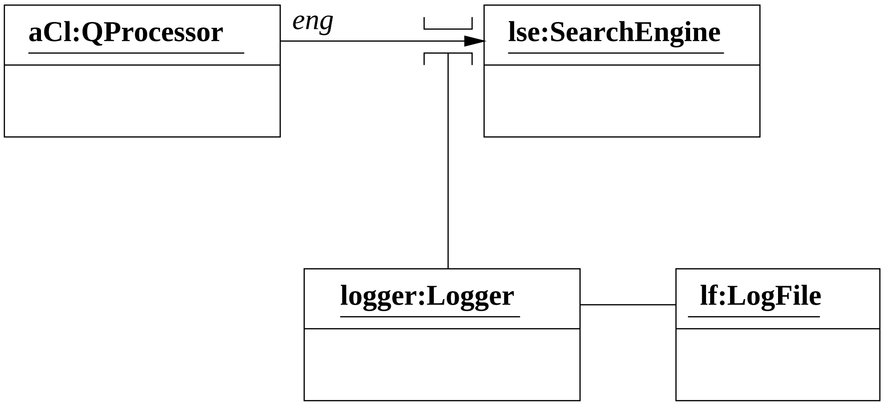

Fig. 4. DTD: Instance Diagram

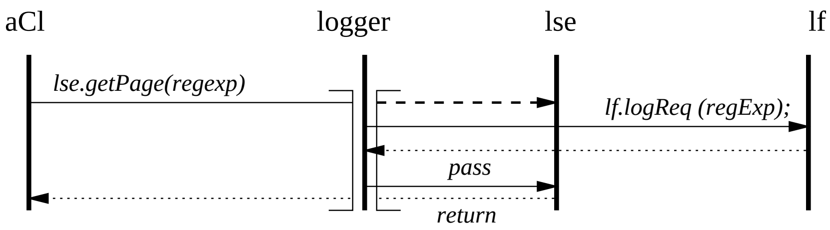

Fig. 5. DTD: Dynamic Model

are available locally for explicit invocation. Methods defined in the filter interface are invoked implicitly as filtering invocations.

```
Logger | SearchEngine {
...
filter interface:
  updateHits () upfilters SearchEngine::getPage (REGEXP rexp) {
     details = extractDetails (rexp);
     lf -> logReq (details);
     pass;
   }
}
```

4 Related Work and Conclusions

Filtering has been in use in software systems in various forms such as shell level filters, packet filters, mail filters etc. However, programming language support for filtering is not in abundance. Aksit et. al. discuss Composition Filters for language Sina [1]. Composition Filters provide for filter specifications of pre-defined filter types embedded into objects. Recent CORBA implementations such as Orbix [3] support per-process and per-object filtering mechanisms. COM specification also discusses a filtering mechanism [6]. Context Relationships of Lieberherr et al. [10] is a related mechanism which allows context dependent code to be plugged to an object. Proxy pattern of Gamma et al. [2] provides a place-holder object for its corresponding server object, however the client *knows* the place-holder and must possesses an explicit handle to it.

Our approach to filtering in object oriented systems is based on an interclass filter relationship leading to transparency of filter objects [5]. Filter specifications are separated from their respective filter-clients. The unit of modularity for filters is class. In a distributed system, a filter object is just another distributed object with additional capabilities of filtering. Filtering member functions cannot be explicitly invoked. A filter object may however export a public contract and be a part of the system through conventional relationships. We discussed applicability of filter objects to distributed object systems engineering and demonstrated it through a transparent distributed decorator pattern with the help of a notation modeling filtering abilities.

References

1. M. Aksit, K. Watika, J. Bosch, L. Bergmans, A. Yonezawa.: Abstracting Object Interactions Using Composition Filters, Proceedings of ECOOP'93 Workshop, Springer Verlag, (1993) 152–184.
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.
3. IONA Technologies Ltd.: Orbix Advanced Programmer's Guide, 1995.

4. R. K. Joshi.: Filter Configurations for Transparent Interactions in Distributed Object Systems, (to appear in Journal of Object Oriented Programming).
5. R. K. Joshi, N. Vivekananda, D. Janaki Ram.: Message Filters for Object-Oriented Systems, Software Practice and Experience, **27** (1997) 677–700.
6. Microsoft Corporation.: The COM Core Technology Specification, 1998.
7. Object Management Group.: CORBA Specifications, http://www.omg.org.
8. Wolfgang Pree.: Design patterns for object-oriented software development, Addison-Wesley, 1995.
9. G. S. Reddy, Rushikesh K. Joshi.: Filter Objects for Distributed Object Systems, (to appear in Journal of Object Oriented Programming).
10. Linda M. Seiter, K.L. Lieberherr.: Evolution of Object Behavior using Context Relations. IEEE Transaction on Software Engineering. **24** (1998) 79-92.