5

Event management and composition system

This chapter presents the implementation of an event management and composition system. The chapter is organized as follows: section 5.1 presents the Fractal component model that we have chosen as a base for our event management and composition framework. Section 5.2 describes the basis for the design process of software systems in terms of fractal components. This section presents the fractal architecture of an event-based system constructed by the event management and composition framework. Section 5.3 presents the facility management system adopted like validation system and the implementation aspects. Seccion 5.4 presents the conclusions of this chapter.

5.1 Using Fractal

Fractal [Brun04] is a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and to graphical user interfaces.

It supports the definition of primitive and composite components, bindings between the interfaces provided or required by these components, and hierarchic composition (including sharing). Fractal is a run-time library which enables the specification and manipulation of components and architectures. Fractal is highly dynamic and reflective because of its nature as a run-time library.

In practice, Fractal is presented as an API which can be used to create and manipulate complex architectures using plain Java classes as building blocks. Its programmatic approach makes it an ideal base to build tools on top of it.

Fractal distinguishes two kinds of components: *primitives* which contain the actual code, and *composites* which are only used as a mechanism to deal with a group of components as a whole, while potentially hiding some of the features of the subcomponents. Primitives are simple, standard Java classes conforming to some coding conventions. Fractal does not impose any limit on the levels of composition. Each Fractal component is made of two parts: a *controller* which exposes the component interfaces, and a *content* which can be either a user class in the case of a primitive or other components in the case of a composite. All interactions between components pass through their controller.

The model thus provides two mechanisms to define the architecture of an application: *bindings* between interfaces of components, and *encapsulation* of a group of components into a composite. Because Fractal is fully dynamic and reflective (in the sense that components and interfaces are first-class entities), applications built using it inherently support structural reconfiguration. Fractal also supports component parameterization: if a component can be configured using parameters, it should expose this feature as a Fractal interface identified by a reserved name so that it is available in a standard way. More information about Fractal, including the complete specification of the component model, and several tutorials, can be found at http://fractal.objectweb.org.

5.2 Design

Before programming a component based software system with Fractal, one must first *design* it with components and, in particular, identify the components to be implemented. The *component oriented design* task is quite independent from the subsequent *component oriented programming* task: at programming time, it is possible to merge several or even all the design time components into a single, monolithic piece of code; but then the advantages of component oriented programming, such as modularity and adaptability, are lost.

Some components in a component based application are *dynamic*, i.e. they can be created and destroyed dynamically, while other components are *static*, i.e. their life time is equal to the life time of the application itself. The dynamic components generally correspond to *datas*, while the static ones generally correspond to *services*.

Therefore, in order to identify components in an application, it is easier to begin by identifying the static components (the services that are used in the application). After the services have been specified, the main data structures can be looked for, in order to identify the dynamic components. After the components have been identified, the dependencies between them must be founded, and they must be organized into composite components.

The services for event management and composition that the framework implements are: event observation, filtering and notification. Then, the fractal components for managing the event processing are: the *Observer*, the *Filter* and the *Notifier* components.

Figure 5.1 presents the fractal component architecture of an event-based system. Remember that the framework is instantiated and configured by the system programmer with respect to the event types, the composition operators and the composition algorithm.

The architecture of the event-based system is defined by a composite component. This component is conformed by the components for event management and composition that the framework implements, and also by *Producer* and *Consumer* components.

The event-based system component provides interfaces for: the definition of events by producers; the definition of subscriptions by consumers; the creation of the listener objects (the event receptor and the subscription receptor/notifier); and starting producer, consumer and observer components.

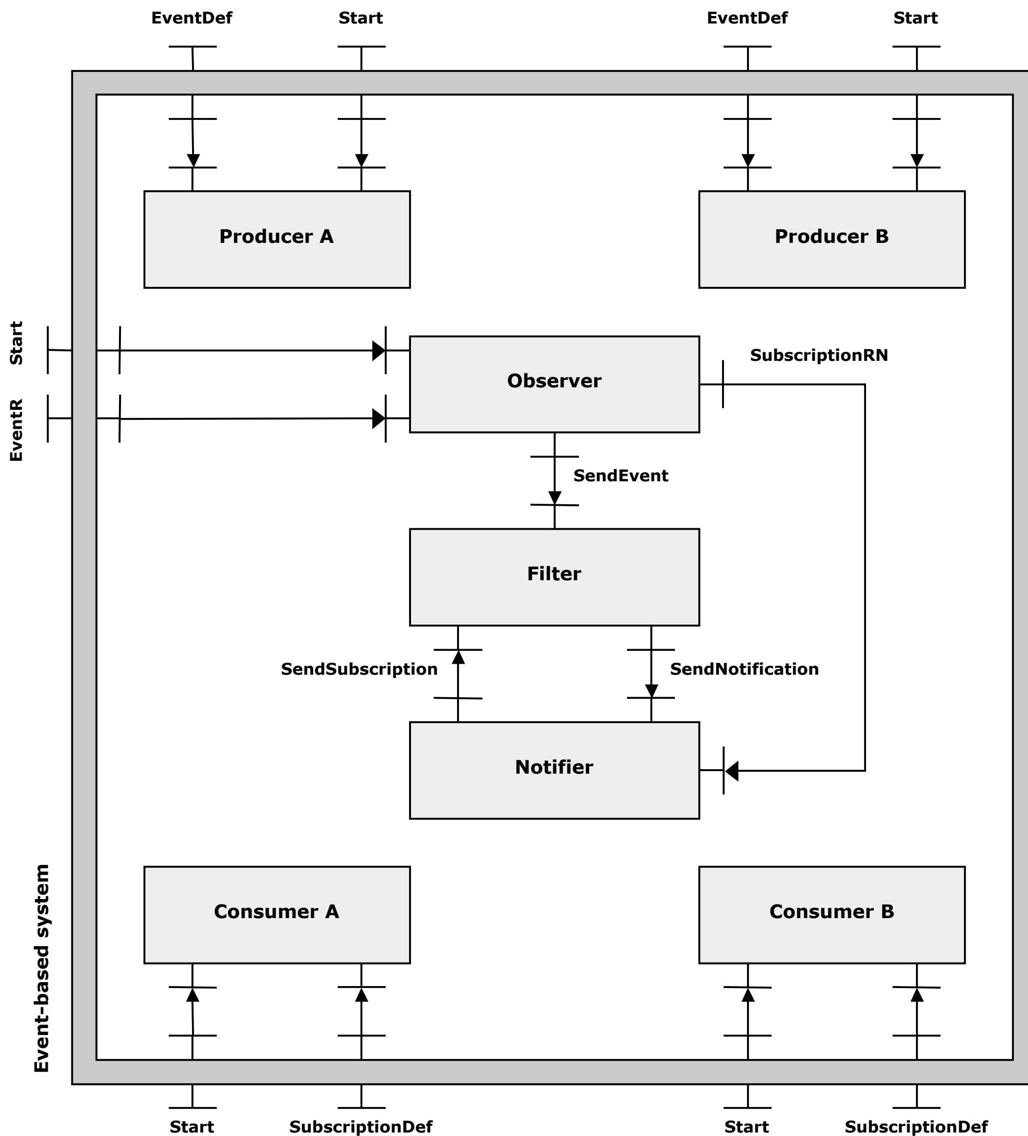

Figure 5.1 Fractal architecture of an event-based system

All the components inside the event-based system component are composite components too. Therefore, we describe each component in the following paragraphs.

The *Producer* component is composed by two components (figure 5.2(a)). The *EventSender* component sends the event definition file to the event receptor. The *Appl* component (figure 5.2(b)) implements the functionality of the producer and then, it is conformed by *Invoker* and *ObjectInformation* components, hence, events occur and are sent to the event receptor.

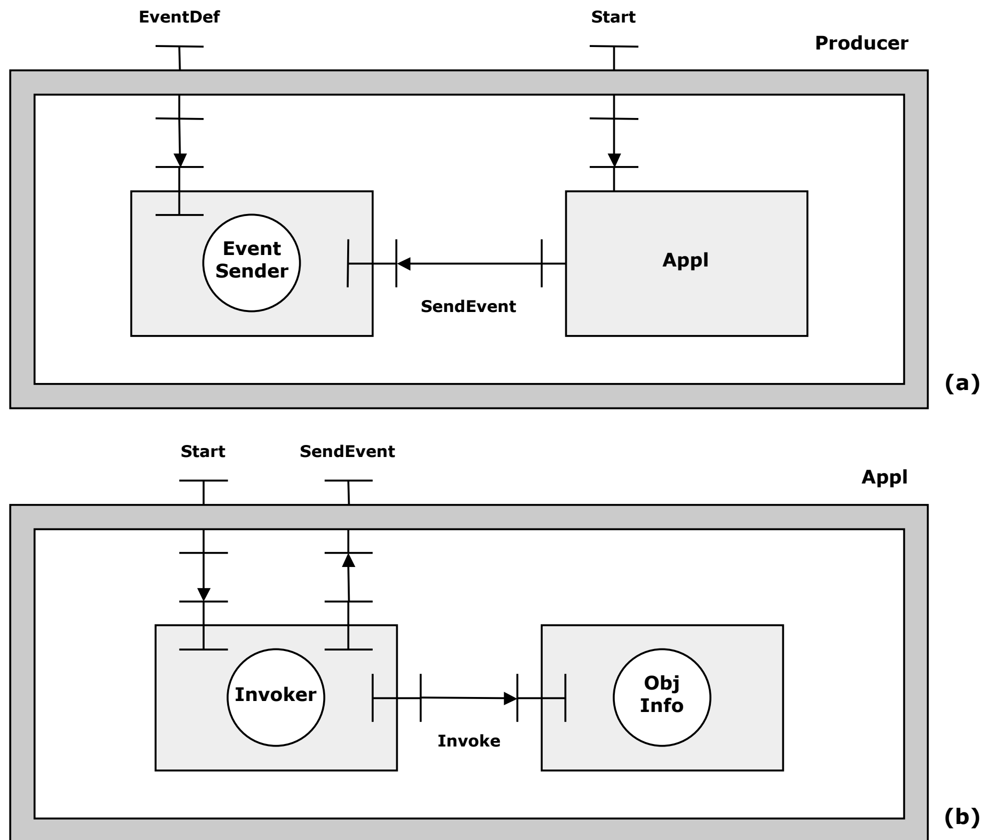

Figure 5.2 Producer components

The *Observer* component (figure 5.3) is composed by the *EventR* component and the *Appl* component. The *EventR* component creates the event message receptor. The *Appl* component receives the occurred events, verifies them with respect to the event definition file and sends the allowed events to the filter.

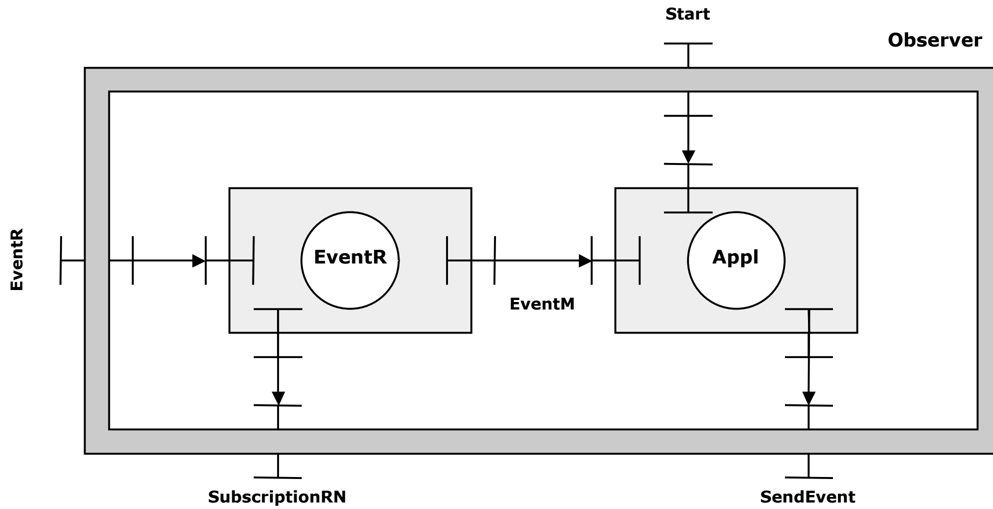

Figure 5.3 Observer components

The *Filter* component is composed by the *Subscription*, *Operator*, *Algorithm*, and *StoredEvents* components. The *Subscription* component receives and stores the consumer subscriptions to the system. Subscriptions are verified with respect to the operator semantics defined at the *Operator* component. The *Algorithm* component receives the occurred events and sends the events that match with the defined subscriptions to the notifier. The occurred events are stored by the *EventStore* component.

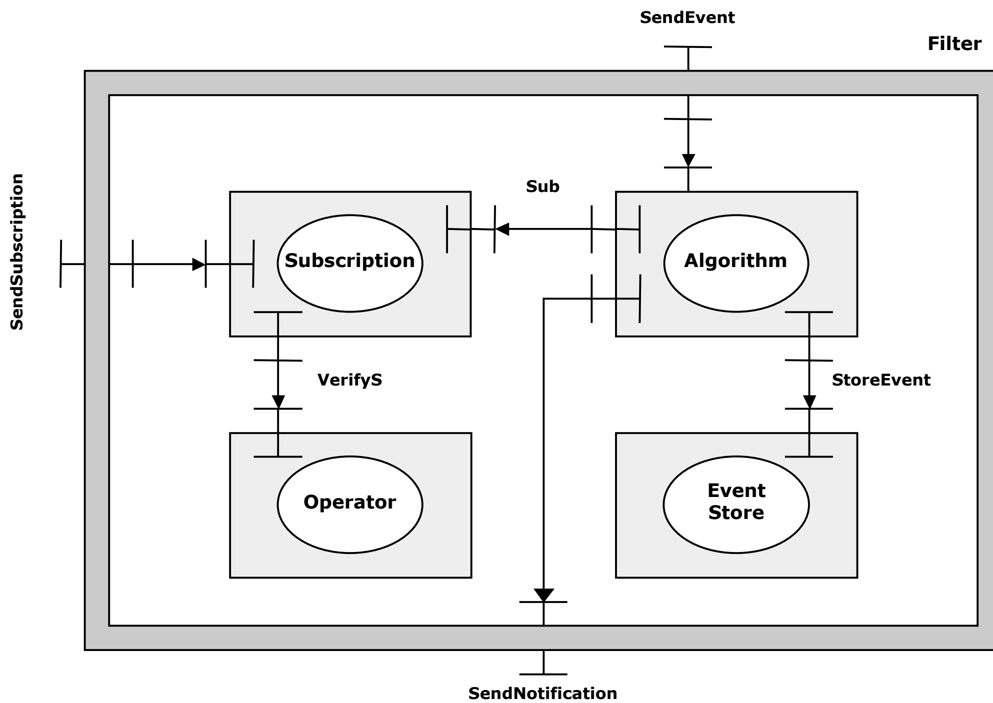

Figure 5.4 Filter components

The *Notifier* component is composed by the *SubRNotS* component which creates the subscription receptor (or notification sender); and the *Appl* component which receives event messages from the filter and send notification to consumers by the notification sender.

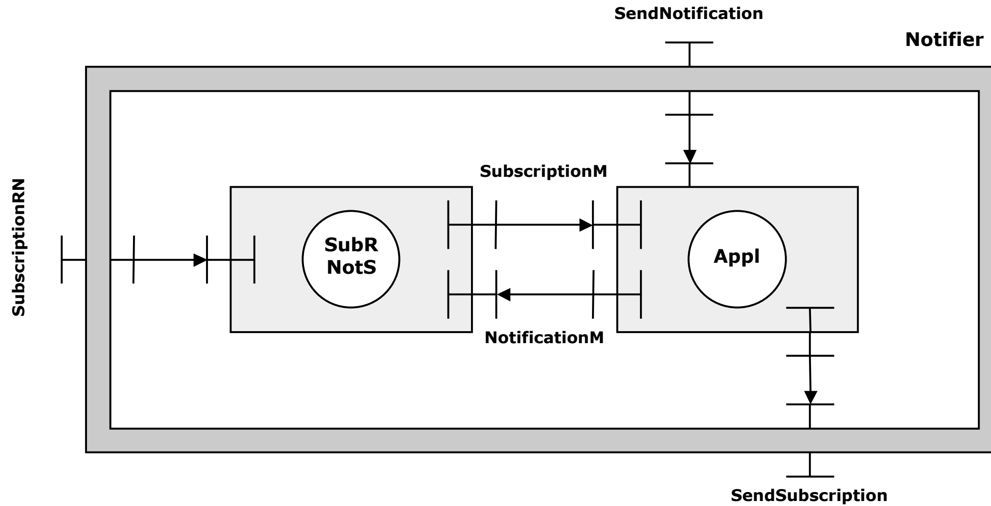

Figure 5.5 Notifier components

The *Consumer* component (figure 5.6) is composed by the *SubSNotR* which sends the subscription definition file to the system. Event notifications are received at this component. The *Appl* component implements the functionality of the consumer and receives the event notifications received by the notification receiver.

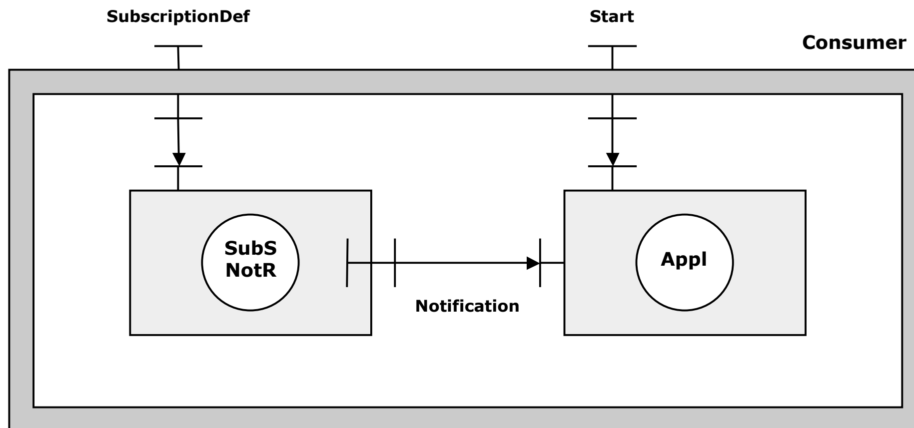

Figure 5.6 Consumer components

5.3 Implementation

For the system implementation we adopt a *facility management system* for big buildings that requires efficient integration of event information form different sources under various or changing application requirements.

A *facility management system* is a distributed system for the remote monitoring and control of multiple heterogeneous big buildings across the Internet from a single control center. The component for event monitoring and notification is called *surveillance system*.

A surveillance system for several buildings monitors lighting, heating, air conditioning, sun protection, and visitor movements. Various sensors are located within each of the monitored buildings (figure 5.7).

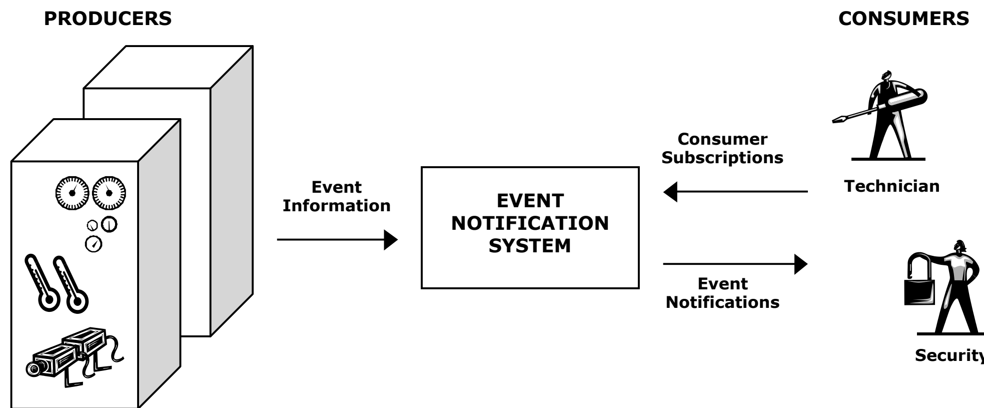

Figure 5.7 Facility management system

Some sensors send status information on a regular basis to the system. Other sensors send only critical events, i.e., if the status values cross a predefined threshold. A third group of sensors passively collects data and is to be observed by the system. Sensors may also have different reaction times and granularities.

For the tracking of visitor movements in a building several techniques may be employed such as personalized badges for client wearing. The badges transmit a signal every few seconds, the signal is received by widely deployed sensors throughout the building or marked transmitter stations. An event source module can retrieve all badge sightings for all clients. The event notification system can be used, for instance, to set alarm whenever someone enters a restricted area of the building.

Several different applications may use the data from the event notification service: access management, security, maintenance, energy management, laboratory safety, and budget management. Each of these applications may use the data in different ways.

In multi-purpose buildings, the applications may change frequently. For example, depending on the actual usage of multi-purpose buildings and rooms, the surveillance system of a building covers certain subscriptions and events: for festive arrangements, the guests security has to be ensured while for cultural exhibitions, strict environmental conditions have to be maintained for the presented pieces of art. The following examples show client subscriptions in a facility management system:

- *Notify a technician if the air conditioning system fails for the third time.*
- *Notify service personnel if a sensor did not send data for more than half an hour.*
- *Notify a technician if in a certain room the temperature rises above 35ºC within a time interval of 1 week length after a failure in the air conditioning system.*

- *Notify security personnel if a window is broken (during the night) and after this a presence detector sends a signal.*

Depending on the sensor type and the application, the technician has to be notified, for example, about every occurrence of that event or just the first one. Similarly, for each of the contributing events, different evaluation methods may have to be applied.

The architecture of the feature management system corresponds to the system architecture presented in figure 5.1. The system implements sensors like event producers: air conditioning, temperature, light. The consumers are represented as technician, security and service personnel.

The feature management system was implemented using Julia, the reference implementation of the Fractal component model in Java. The system uses the Java Message Service (JMS) for asynchrnous communication. JMS is well suited to develop applications that asynchronously send and receive data and events. Indeed, JMS supports both messaging models: point-to-point (queuing) and publish/subscribe.

5.4 Conclusion

This chapter presented the implementation aspects for an event-based system. The system implementation was defined using the Fractal component model. The system prototype was conceived like a facility management system. The system implementation demonstrates that the framework is a well suited infrastructure for building personalized systems which require event management and composition.