

# Synchronization Patterns for Process-Driven and Service-Oriented Architectures

**Till Köllmann**

IBM Business Consulting Services  
c/o IBM Deutschland GmbH  
Pascalstr.100  
70569 Stuttgart, Germany  
e-Mail: [till.koellmann@de.ibm.com](mailto:till.koellmann@de.ibm.com)

**Carsten Hentrich**

IBM Business Consulting Services, SerCon GmbH  
c/o IBM Deutschland GmbH  
Hechtsheimer Str. 2  
55131 Mainz, Germany  
e-Mail: [chentrich@de.ibm.com](mailto:chentrich@de.ibm.com)

*This paper introduces a collection of patterns for solving different sorts of synchronization problems in the area of process-driven architectures that are based on process technology such as commonly known workflow systems according to the Workflow Management Coalition (WfMC) standards [5], for instance.*

## Introduction

Process-driven architectures employ a process engine to coordinate and execute process activities according to a defined process model. These activities are either automatically executed system-activities or involve human interaction. Yet, both types require any sort of program or system that implements the logic to be executed by the activity, as the process engine only orchestrates the activities, but does not implement the functionality represented by them. Rather these functions are executed by systems integrated in the process flow.

Service-oriented architectures (SOA) [4] understand these activity implementations as services that offer a defined functionality which is clearly expressed by its commonly accessible interface description (see also [13]) and that may be invoked from any high-level orchestration independently from the involved technology. A SOA is typically represented as a layered architecture that separates the different responsibilities of the involved components or tasks in order to reach this goal. Figure 1 shows such a layered representation [1]. The Service Composition Layer denotes the high-level orchestration of services that may be accomplished by a process engine. The Client Application and Service Provider Layer include both service consumption on client side and service offering on server side. The lower layers deal with remoting of service calls to and from the service providers and with low-level communication.

This paper deals with the Service Composition and the Service Provider Layer as all considerations are motivated by problems that mainly occur at their boundaries.

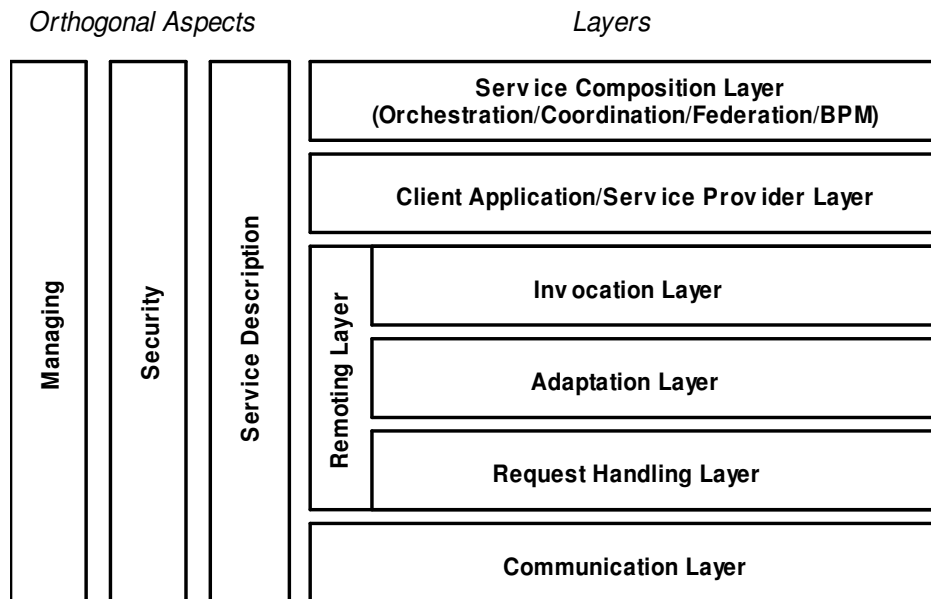


Figure 1: SOA Layers [1]

Even though a clean separation of layers might suggest that almost none or at least very little coupling or interaction exists between process flow and service execution, there are still links and dependencies that are wanted and needed in order to assemble a working solution.

Besides identifying and invoking a service via a name when a process activity is scheduled, parameters will have to be supplied or references to business objects [11] need to be passed, as addressed by the BUSINESS OBJECT REFERENCE PATTERN [3]. The process flow might have to wait at some point for an event to occur asynchronously after a service has been invoked by a previous activity or even by another process. Moreover, if processes are executed in parallel and invoke services that will operate on the same business object, how will the access be synchronized, so that both processes operate with valid data and do not block their execution?

Wherever a service is invoked by a process instance an interaction takes place that may require different degrees of synchronization and that may bear different consequences. The patterns described in this paper tackle some of the problems that may arise from interactions which require synchronization of the process flow or the service execution.

In other words: synchronization of the process flow within the process engine itself as well as synchronization of the process execution and interacting system components is essential for a seamless integration and a stable process-driven architecture.

## Intended Audience

In this paper a collection of patterns is introduced that offer solutions to different sorts of synchronization problems in process-driven and service-oriented architectures.

Some of these patterns mainly deal with business process modeling and are therefore of interest for business analysts or process modelers. Yet, the scope of this paper is not only to present patterns for organizing process flows. Quite some work has been done in this area (see

e.g. [3, 7-10]) and also some patterns for process flow synchronization within single processes and among several process instances exist [8].

This paper does not only take process design into account, i.e. the design time view on problem situations, but rather tries to see the problems from an architectural point of view. That means that not only process models are considered, but also the environment and its underlying software architecture in which the process engine is integrated to execute the business processes. This integration aspect will mainly be of interest for software architects who have to integrate a process engine into an existing or newly designed application or system architecture. Thus, the problem domains for system-wide event synchronization and synchronization of data access, which are presented later on, are addressed rather to software architects than business analysts.

### Pattern Collection Overview

The patterns presented in this paper deal with process and service execution synchronization issues. In detail the covered problems are about controlling the execution timing of single processes, synchronizing parallel process instances, handling of events that occur outside of the process engine and controlling the way process-invoked services access business objects.

Some of the patterns are related to each other as shown in Figure 2. The figure also denotes different categories that further group related patterns.

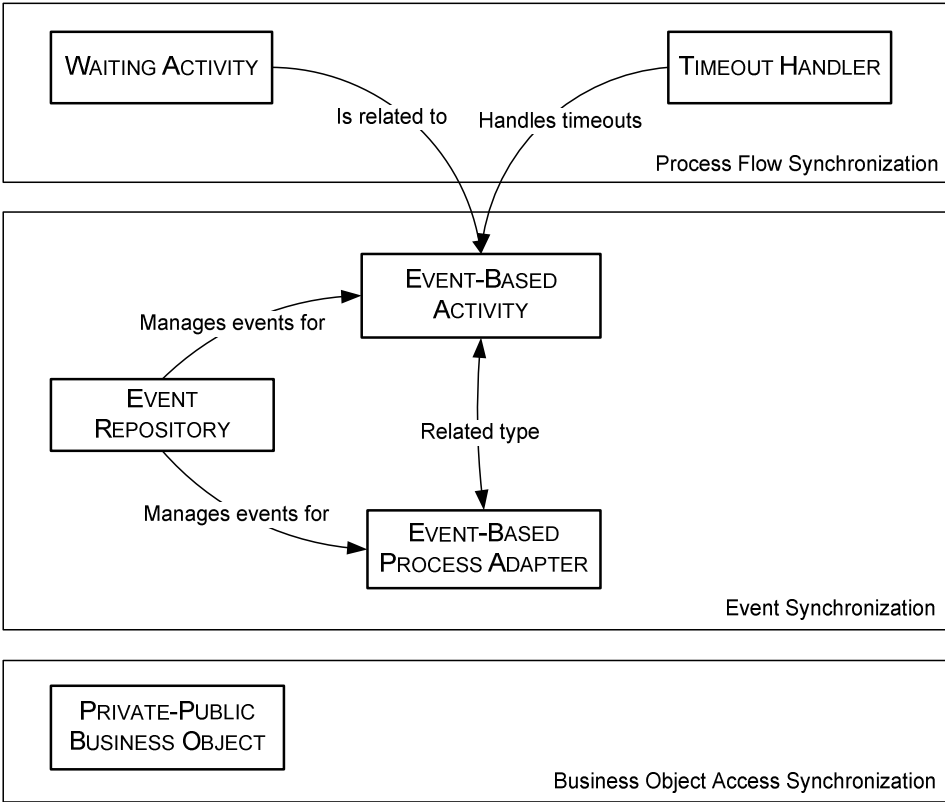


Figure 2: Pattern relationships overview

The table below lists all covered problems and states short problem and solution descriptions. The patterns are divided in three different categories according to the covered problem domain as shown in Figure 2:

- *Process Control Flow Synchronization:* Patterns of this category address synchronization issues in the control flow of one or among several processes.
- *Event Synchronization:* Patterns of this category address synchronization problems in the area of business processes and external events.
- *Business Object Access Synchronization:* Patterns of this category address synchronization problems of business object access form parallel processes.

<i>Pattern</i>	<i>Problem</i>	<i>Solution</i>	<i>Category</i>
EVENT-BASED ACTIVITY	How can events that occur outside the space of a process instance be handled in the process flow?	Model an event-based activity that waits for events to occur and that terminates if they do so.	Event Synchronization
EVENT-BASED PROCESS ADAPTER	How can process instances be created on a process engine on the basis of occurring events?	Use an event-based process adapter that instantiates processes if corresponding events occur.	Event Synchronization
EVENT REPOSITORY	How can events that occur outside the scope of a process instance be captured and managed?	Use a central event repository that manages occurring events and which offers a central access point for event-based process components.	Event Synchronization
TIMEOUT HANDLER	How can timeouts of process activities be managed in a process?	Model a timeout handler that defines behavior in the process model in case a timeout has occurred.	Process Control Flow Synchronization
WAITING ACTIVITY	How is it possible to model a waiting position in a process that waits for a defined time to elapse?	Model a waiting activity that terminates after the desired time has elapsed.	Process Control Flow Synchronization
PRIVATE-PUBLIC BUSINESS OBJECT	How can business object modifications be hidden from other users as long as the process activity during which the changes are made is not finished?	Introduce private-public business objects, which expose two separate images, a private and a public image of the contained data.	Business Object Access Synchronization

*Table 1: Problem/solution overview of the patterns*

## Overview of Referenced Related Patterns

There are several important related patterns referenced in this paper, which are described in other papers, as indicated by the corresponding references in the text. Table 2 gives an overview of thumbnails of these patterns in order to provide a brief introduction to them for the reader. For detailed descriptions of these patterns please refer to the referenced article “Six patterns for process-driven architectures” [3].

<i>Pattern</i>	<i>Problem</i>	<i>Solution</i>	<i>Category</i>
BUSINESS OBJECT REFERENCE	How can management of business objects be achieved in a business process, as far as concurrent access and changes to these business objects is concerned?	Only store references to business objects in the process control data structure and keep the actual business objects in an external container.	Technical Architecture
ACTIVITY INTERRUPT	How can an activity in process be interrupted without losing any data?	Model an exit condition that restarts the activity in case the exit condition is false and use the output data of the activity as input data to the restarted activity.	Process Architecture
PROCESS BASED ERROR MANAGEMENT	How can errors that are reported by integrated applications in activities in a process flow be handled and managed	Define special fields for error handling in the process control data structure and embed an activity in an error handling control flow.	Process Architecture
GENERIC PROCESS CONTROL STRUCTURE	How can data inconsistencies be avoided in long running process instances in the context of dynamic sub-process instantiation?	Use a generic process control data structure that is only subject to semantic change but not structural change. Interface	Technical Architecture

*Table 2: Thumbnails of referenced patterns*

---

# Event-Based Activity

---

## Context

As business process design becomes more complex and involves interaction with other business applications and external data sources, handling of external events may have to be considered in process design.

## Problem

**How can events that occur outside the space of a process instance be handled in the process flow?**

During the execution of a process instance the evaluation of defined conditions may have influence on the path the process flow takes. Such conditions may include variables known in the process instance or the result of a previously executed activity, for example. Process design in a complex business scenario may also have to consider handling of events triggered by external systems, waiting for business object status changes or events produced by a process instance executed in parallel.

Reacting upon external events within a process instance bears the problem of the separation of the process space, the process execution environment, and the event space, the environment in which the event occurs or in which the event can be perceived. Furthermore, a simple condition can evaluate a process instance variable, for example, but is not sufficient for retrieving a notification of an event.

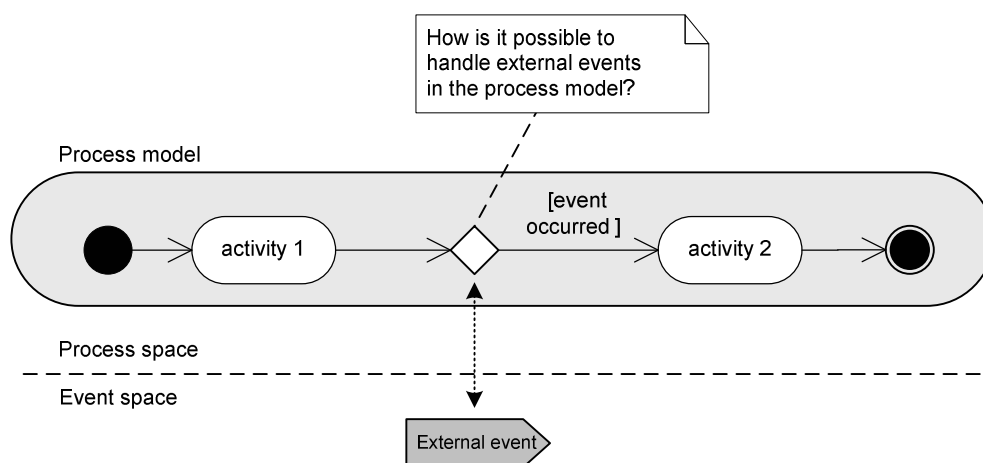


Figure 3: Problem illustration for handling external events

Consider an asynchronous print job that is sent to an output system when a user finishes an activity. In case a subsequent activity of the process instance depends on the newly created document it shall not be scheduled as long as the document is not available. Since the state of the document creation is not known in the scope of the process engine, it can not be evaluated in an internal condition. Furthermore, such a condition could not be simply evaluated once, but would have to be iteratively checked until satisfied. The problem in this scenario is that the creation of the document is an event that occurs outside of the process space. Therefore the scheduling of the activity can not be easily influenced by this external event.

The scenario above can be summarized to the basic problem that a process instance shall wait at a defined process-step for an external event to occur before continuing the process execution. This problem also brings up two closely related aspects to consider: How are errors handled while waiting for an event and what happens if the event never occurs? A strategy for error and timeout handling has to be considered in this context as well.

**Solution**

**Model an event-based activity that waits for external events to occur and that terminates if they do so.**

An event-based activity acts as an event listener and registers to event sources that it would like to retrieve notifications about occurring events from. In case it is notified of the event(s) that the activity is waiting for, the activity terminates and the process execution continues.

The pattern is closely related to the OBSERVER Pattern [2] known in object oriented software design. One component observes the state of another component by registering as observer. The observed component does not need to know about its observers' concrete types or the amount of registered observers. It simply notifies all registered observers in case an event (e.g. a change of its state) occurs.

This pattern can not be directly applied to activities in a process engine, if the subjects to observe, the event sources, are not known in the context of the process instance. Thus, the problem has to be solved outside of the space of the process engine. Therefore we introduce an activity implementation that will be executed in an environment where the external event sources are known (see Figure 4). An event-based activity defined within a process model will forward to its implementation in the event space when it is instantiated by the process engine. The implementation acts as event listener by registering to the event sources of interest and receiving notifications of the occurring events. If it is notified of the event it has been waiting for, the activity implementation terminates the activity in the process space.

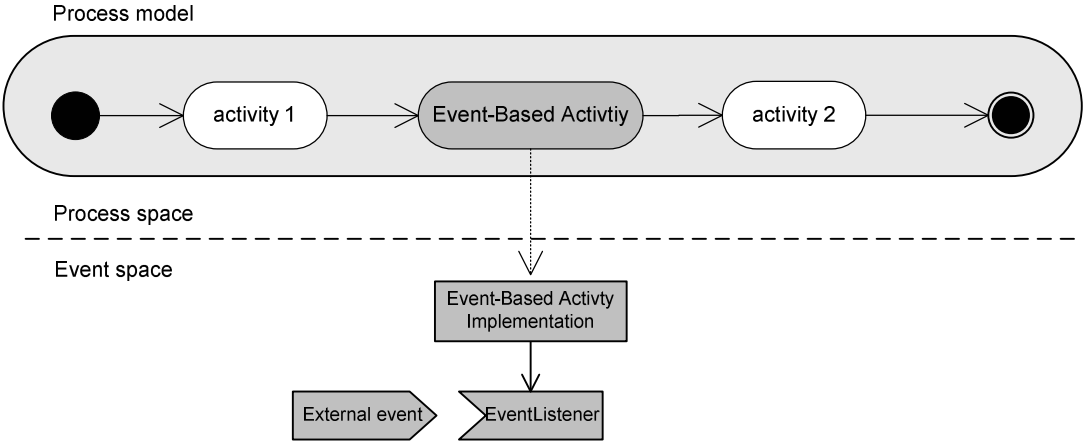


Figure 4: Introduction of an event-based activity to handle external events

As mentioned in the problem description, the pattern requires paying attention to timeout and error handling, since a controllable process execution highly depends on a deterministic behavior of such event-based activities.

The process designer is responsible for handling these situations within the process model. For this purpose the related TIMEOUT HANDLER pattern can be applied. In order to add this functionality, introduce a timeout value for event-based activities that causes the activity to

terminate in case the timeout is reached. In addition a result value shall indicate whether or not the activity terminated successfully (an event notification was received in the given timeframe), a timeout occurred or an error terminated the activity unexpectedly.

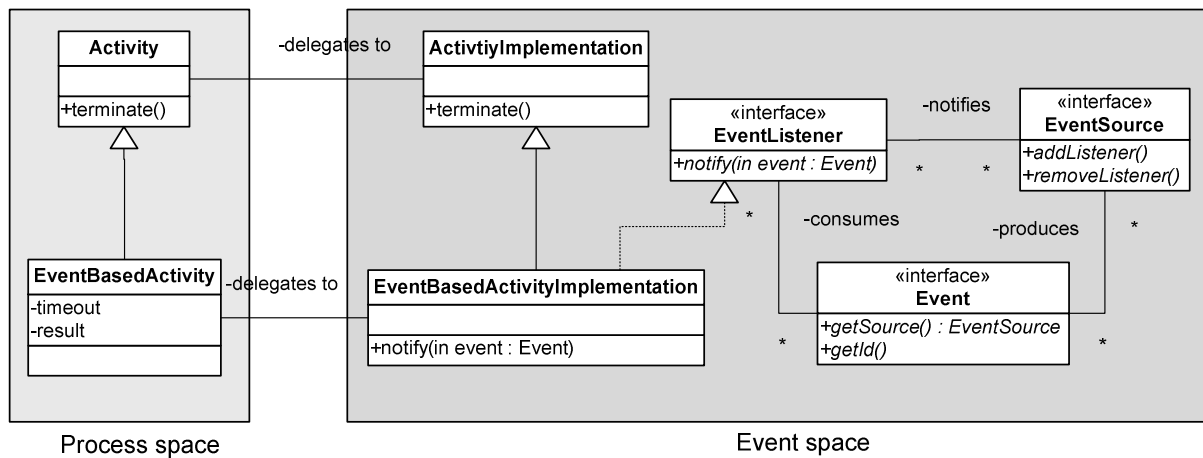


Figure 5: The event-based activity pattern

The pattern shown in Figure 5 decouples the process from the event space. A coupling is only present in terms of a reference from the activity to its implementation. This has the advantage that the process engine and also the process designer are independent of concrete events and event sources. On the other hand all logic concerning which events to wait for until the process flow continues is contained in the event-based activity implementation outside of the process space. If a common knowledge about events and event source is desirable, corresponding identifiers can be supplied in the activity's data container during design time and evaluated in the implementation during runtime.

## Consequences

- The process flow is interrupted and its further execution depends on the occurrence of one or more defined external events.
- Business process design can take system-wide resources like external business applications or data sources into account, as the process flow may be synchronized with external systems.
- Additional effort has to be paid to timeout and error handling in order not to block the process execution for an undefined amount of time.

## Related Patterns

The EVENT REPOSITORY pattern introduces a possibility to decouple event listeners from event sources and thereby added a higher degree of abstraction from concrete implementations. Apply the pattern in this context if the system has to deal with many event sources and frequent changes in this area.

The TIMEOUT HANDLER pattern describes a solution for handling timeouts that are reported by components integrated in the process flow. Event-based activities are typically considered as such components as listening for an event should include a defined timeout value.

The WAITING ACTIVITY pattern describes a solution for controlling process execution based on a dynamic or static timer. In the context of event-based activities the timer can be perceived an event source and the waiting activity as a special type of event-based activity. This is only true though in case the timer is not available within the space of the process instance. As many process engine implementations offer integrated support for this feature, the timer can not generally be considered an external event source.

## Example

The pattern can be implemented differently depending on the process engine used, the environment in which the event occurs and depending on how generic or specific the solution design is required to be. The following example assumes a process engine according to the WMFC standards [5, 6] denoted as the process space and any object oriented application execution environment (e.g. offered by a J2EE application server [11]) for the activity implementations in the event-space.

If the process flow often has to wait for a specific status change of business objects (BO) managed outside of the process engine, for example, a generic event-based activity implementation can be considered for this purpose.

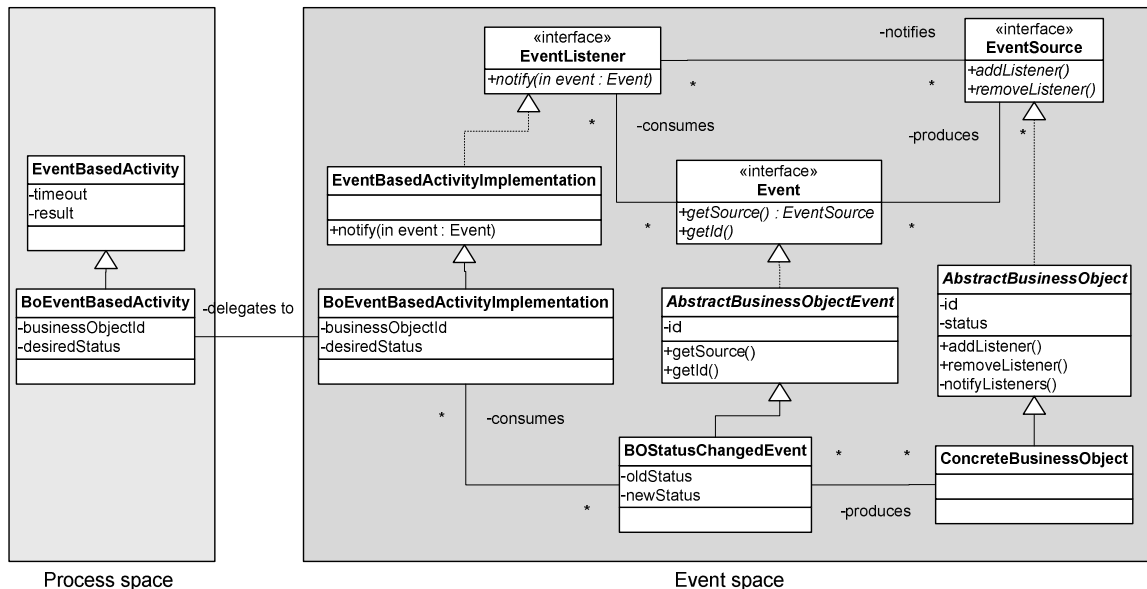


Figure 6: Example for a generic implementation for BO status change events

This specific implementation registers as listener at a business object that is referenced by the activity in terms of its system-wide unique ID (compare the business object reference pattern in [2]). Once notified of a “BoStatusChangedEvent” the delegate compares the new status to the one specified in the activity’s input container. If the new status is equal to the desired status, the activity implementation terminates the activity and the process execution continues.

The pseudo code below gives a more detailed insight into the implementation of such event-based activity implementation:

```

public abstract class EventBasedActivityDelegate implements EventListener {
    // field initialized from activity's input container
    private int timeout;

    // Initialization
    public EventBasedActivityDelegate() {

```

```

        // Create a timeout event source
        EventSource source = new TimeOutEventSource(timeout);
        // Register as listener
        source.addListener(this);
    }

    public synchronized void notify(Event event) {
        if (event.getId() == TIMEOUT_EVENT) {
            // Terminate the activity
            terminate(TIMEOUT);
        }
    }
}
...
}

public class BoEventBasedActivityDelegate extends EventBasedActivityDelegate{
    // fields initialized from activity's input container
    private String businessObjectId;
    private int desiredStatus;

    // Initialization
    public BoEventBasedActivityDelegate() {
        // Initialize timeout counter
        super()
        // Get the business object to register to
        EventSource source = BoManager.getBusinessObject(businessObjectId());
        // Register as listener
        source.addListener(this);
    }

    public synchronized void notify(Event event) {
        if (event.getId() == BO_STATUS_CHANGED_EVENT) {
            BoStatusChangedEvent statusEvent = (BoStatusChangedEvent) event;
            // Compare new status to desired status
            if (statusEvent.getNewStatus() == desiredStatus) {
                // Terminate the activity
                terminate(SUCCESS);
            }
        } else {
            // handle ths event in the super class
            super.notify(event);
        }
    }
}
...
}

```

---

# Event-Based Process Adapter

---

## Context

One or more process instances shall be instantiated and started upon occurrence of defined events that occur outside of the scope of the process engine.

## Problem

**How can process instances be created by a process engine on the basis of occurring events?**

In a typical process-driven environment process instances are instantiated and started by the process engine due to user interaction, e.g. a user triggers the creation of a process instance from within an application. Process instances are also created during execution of the process model if it involves sub-processes, which will be instantiated by the process engine once the execution has reached the corresponding process-step.

These types of process creation either involve human interaction or require the processes to be statically modeled as sub-processes that will be called from within another process model.

In some situations it will not be possible to statically define the processes that are to be started from within another process model, though. This may be the case if the amount of instances or the type of the process to instantiate depends on a parameter that is only available during runtime, for example. Furthermore, it may not always be desirable to start processes from other process instances, as this requires the “super processes” to run during that time.

If the creation of process instances shall neither be triggered directly by a user interaction nor due to a static model, it shall be possible to automatically instantiate and start a process if an event takes place within the system environment.

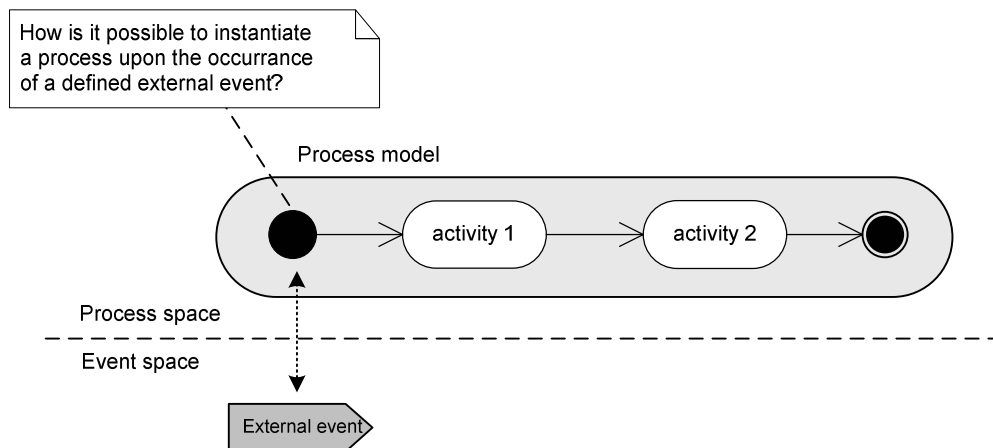


Figure 7: Problem illustration for the instantiation of process upon events

The basic problem to achieve this goal is that the event to occur will not always be visible in the scope of the process engine and the engine may not support dynamic process instantiation. How is it possible to create instances of defined process models when an event occurs outside of the scope of the process engine? And how can the instantiated process be supplied with correct input data?

## Solution

Introduce an **Event-Based Process Adapter**, which acts as event listener and instantiates a process or a set of processes once the defined events occur.

An event-based process adapter is responsible for instantiating one or more processes and starting them with a defined initial state upon the occurrence of an external event. It acts as event listener, i.e. it waits to be notified about occurring events from defined event sources that it registers to.

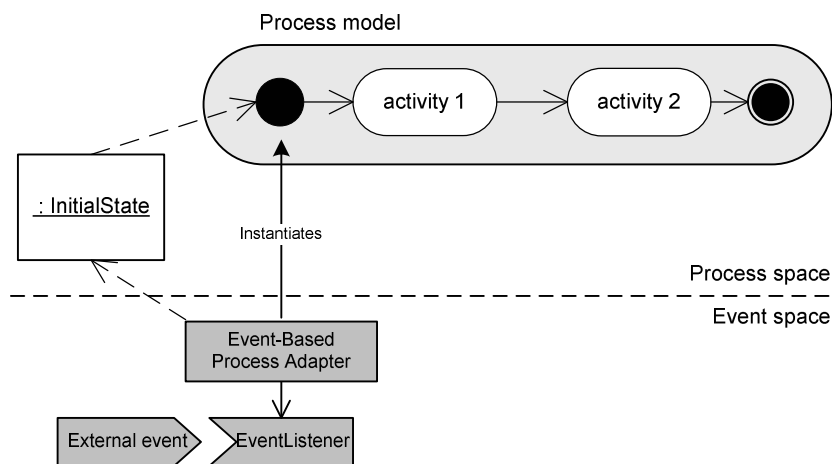


Figure 8: Solution employing an event-based process adapter

Since external events will occur outside of the scope of the process engine, the process adapter resides within the event space, the space in which the events occur or where the events can be perceived.

If the process to start requires a predefined set of information, the event-based process adapter is responsible for supplying this initial state to the newly created process instance. This is either possible by the means of an external configuration or any other data storage.

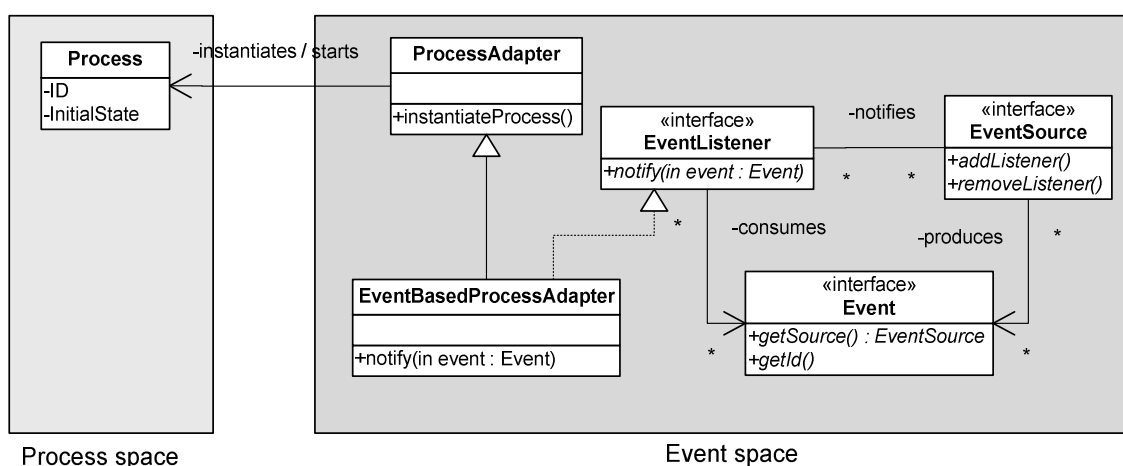


Figure 9: The Event-Based Process Adapter Pattern

Figure 9 shows the components involved in this pattern. The event-based process adapter is a specialized type of a process adapter that is capable of instantiating and starting processes in the

process engine. Like the related EVENT-BASED ACTIVITY pattern this pattern distinguishes between event listeners and event sources. An event source is a component that fires events and notifies all listeners that have registered to this source about the events. When notified the event listener can check if the event is of interest using its unique ID or checking the event source it was emitted from.

## Consequences

- Process instances can be instantiated and started depending on an external event.
- If the process to start requires an initial state to be supplied, the event-based process adapter is responsible for it and therefore the corresponding information has to be stored outside of the process engine. This may result in additional effort for assuring data consistency.
- If the event the event-based process adapter is listening never occurs it may result in undesired situations. A process based error or timeout handling is not possible, because the adapter resides in the event space and not the process space.

## Related Patterns

The EVENT REPOSITORY pattern introduces a possibility to decouple event listeners from event sources and thereby added a higher degree of abstraction from concrete implementations. Apply the pattern in this context if the system has to deal with many event sources and frequent changes in this are.

The EVENT-BASED ACTIVITY pattern employs the same event listener / event source mechanism and describes, like an EVENT-BASED PROCESS ADAPTER pattern, an event-based component.

The EVENT-BASED PROCESS INSTANCE pattern [3] is a specialized type of this pattern. It offers a solution for the problem of processes that are suspended for a long time, because of an event-based activity waiting for an external event. The process is split into two parts and an event-based process adapter is used to instantiate the second process when the event occurs. This special usage requires the state of the first process to be stored and used as initial state for the instantiated process.

---

# Event Repository

---

## Context

Events that occur outside of the scope of the process engine shall be centrally managed in order to decouple event listeners and event sources and have one defined access point for event-based process components.

## Problem

**How can events that occur outside the scope of a process instance be centrally captured and managed?**

The introduction of event-based process components such as the EVENT-BASED ACTIVITY or the EVENT-BASED PROCESS ADAPTER leads to the question how the events that occur are best captured and managed.

If each event-based process components has to register at the event sources directly to obtain notifications from events of this source, it implies that the component needs knowledge not only about the event it is waiting for, but also about the source that fires this event. Problems arise where there might be several sources for the same event. The activity implementation has to know which sources these are and register to all of them.

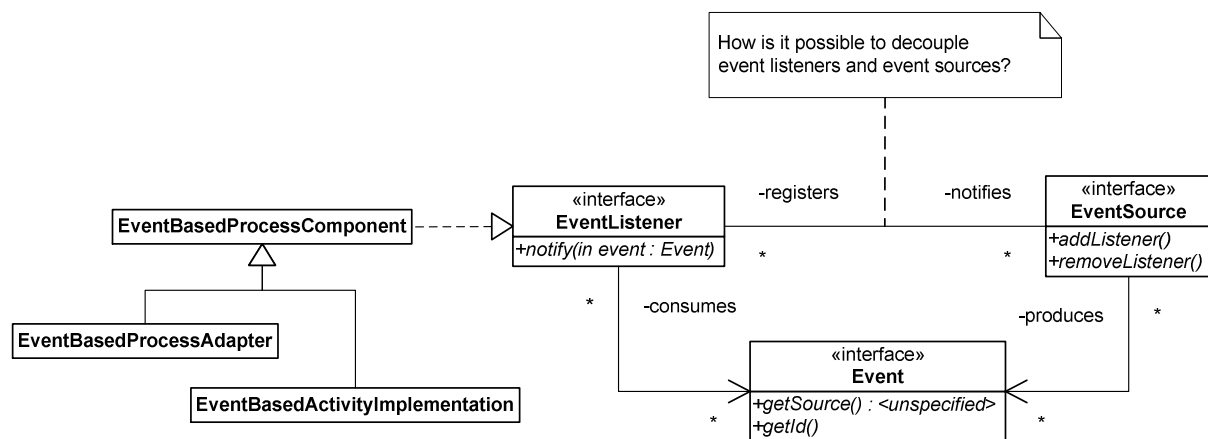


Figure 10: Problem illustration for decentralized event handling

Another problem arises if one takes the runtime aspect into account. If an event-based process component is instantiated it shall register immediately to the event source in order not to miss an occurring event during its lifetime. This may be a problem if the concrete event source is not always present. If the event source is an external system that offers a constantly available interface to access, there will be no problem. In case an event source is implemented by a component that has limited lifetimes, i.e. an instance of the component will be created and destructed several time during the applications runtime, an event listener might not always have the chance to register.

Consider a scenario in which a process shall wait for a business object state to change. If the process designer employs an event-based activity to realize this requirement, this component will have to register with an event source that fires such a business object event. In case the business object is defined as the event source itself, listeners would have to register directly with it. This will be a problem if the business objects are only instantiated on demand.

The scenario above can be summarized to the basic problem that event listeners not only need knowledge of the concrete event source to register to, but also depend on runtime aspects that may not always suite in the area of process-driven architectures.

## Solution

**Use a central event repository that manages occurring events and which offers a central access point for event-based process components.**

An event repository is a central component that is responsible for gathering, managing and delegating events from event sources to event listeners. It offers a common interface for event listeners and sources to register to and to remove from the event repository.

In order to be able to act as central component all event sources of the considered application or system must register at the event repository, thereby offering their service of event notification in the system. When an event source registers, the event repository itself registers as listener at that source. Consequently the event repository does not initially have to know about all concrete event sources, yet is informed about each event that will occur in the system. If the event source no longer fires events or if it will be destructed, it removes itself from the repository.

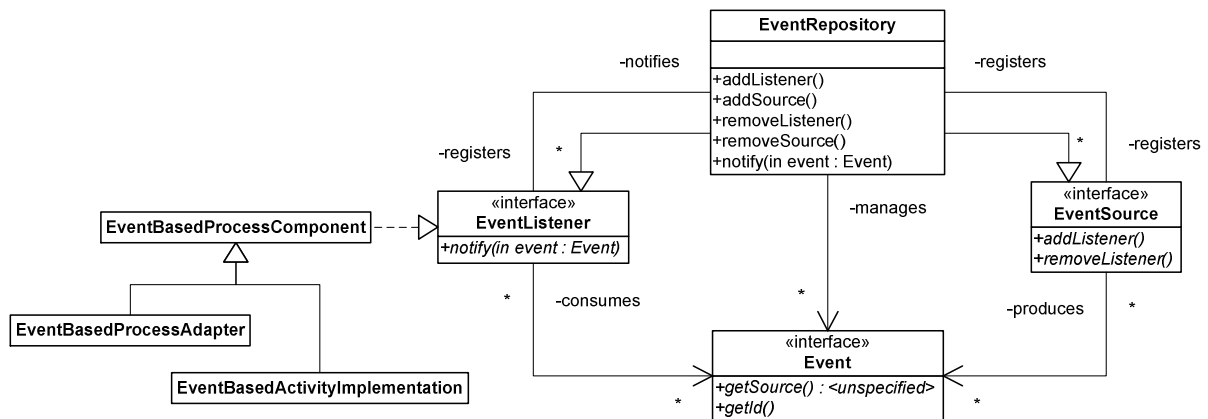


Figure 11: The Event Repository pattern

All event listeners, e.g. event-based process components that are waiting for a defined event to occur, simply have to register at the event repository. The repository is responsible for delegating the correct events to the correct event listeners. For this to work the listeners must provide a system-wide unique identifier of the events that they are interested in. In case the event source is known, the listener may also state to be notified about all events of that specific source.

Thereby the event repository acts as broker delegating events from one or more sources to zero or more listeners. The information which events to route to which listener must be provided by the listeners or may be centrally configured.

Furthermore, the introduction of an event repository allows a centralized common handling for all events. This includes aspects such as centrally configured logging, verification or filtering.

## Consequences

- By introducing an event repository event listeners and event sources of the application or system can be decoupled. This also decouples process components from other application resource.



The example shown in Figure 12 introduces an event repository in this context. The event-based activity registers at the event repository and supplies an identifier for the event source it is interested in. The example uses a business object ID which serves as event source id in this case.

Once other application components access business objects, the business objects will be instantiated and register at the event repository during initialization. If the application component commits a modification that results in a change of the business object's state, the event source, the business object, fires an according event and notifies its listeners (the repository). The repository checks the event source and compares all registered listeners. It forwards the event to the event-based activity, which now checks for the desired state it is waiting for.

---

# Timeout Handler

---

## Context

If the process model contains activities that may finish with a timeout, this situation has to be taken into account and structures have to be defined that handle the timeout correctly according to the business requirements.

## Problem

### How can timeouts of process activities be handled generically in the process model?

During the execution of a process instance activities are scheduled according to the definition of the process model. Once an activity is instantiated and started by the process engine the execution of this process path suspends until the activity is finished. If an activity implementation contains a rather time consuming logic or if it waits for an event to occur (see the EVENT-BASED ACTIVITY Pattern) it may be desired to introduce a timeout counter, that finishes the activity before its regular termination if the timeout value has been reached.

The process model that contains such an activity has to handle this situation in a defined way in order to continue the process execution in the correct manner. Normally the process flow shall not continue as if the activity was finished regularly. On the other hand this might even be desired in some cases.

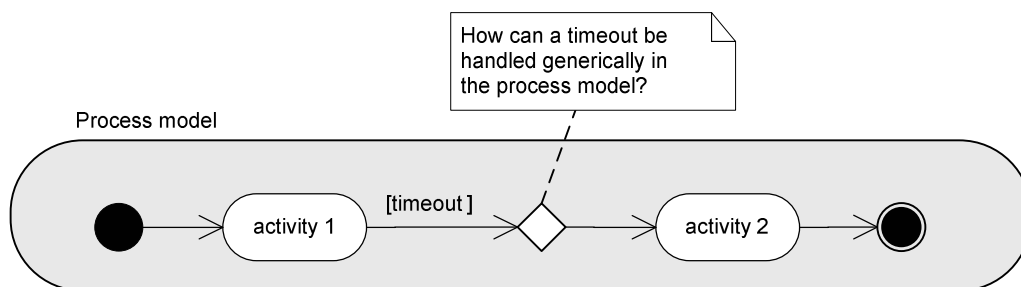


Figure 13: Problem illustration for handling timeouts in the process model

When employing such activities each time a specific solution would have to be modelled depending on the business requirements. Yet, it might also not always be possible to statically model these decisions during process design time. The process might have to handle the timeout situation differently depending on certain parameters that are only known during runtime.

How is it possible to model the process to handle timeouts in a defined way, and how can such a solution be reused in different scenarios?

## Solution

### Model a reusable timeout handler activity that allows reacting in defined ways in the process model in case a timeout has occurred.

A time handler is an activity that is responsible for handling timeout situations in a process model. Wherever an activity that may raise a timeout is defined in the process, model a decision point that checks whether a timeout has taken place or not. In case of a timeout it branches to the timeout handler (see Figure 14).

When the timeout handler is instantiated it evaluates the given situation, processes this input according to its configuration and terminates with the resulting decision how to handle the situation. This result might either be to retry the execution of the activity that caused the timeout, to ignore the timeout and proceed in the process flow or to abort the process execution and clean up any process resources.

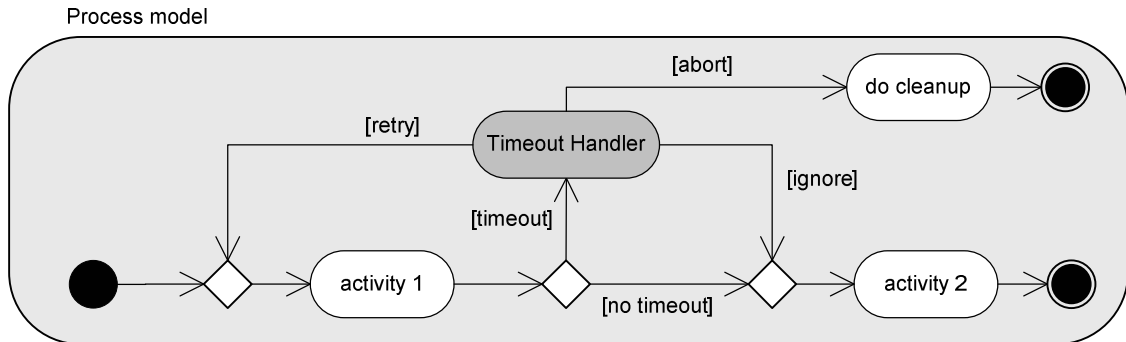


Figure 14: Introduction of an event-based activity including timeout and error handling

In order to determine the correct result the timeout handler depends on information about the timeout situation. Figure 3 depicts that an appropriate process control data structure is to be passed to and from the timeout handler. This structure shall typically consist of the following information:

- The name or unique identifier of the process in which the component exists.
- The name or unique identifier of the activity that caused the timeout.
- A retry count that is initially zero and will be incremented with each retry by the timeout handler.
- The context in which the timeout occurred (e.g. a business object reference or the ID of an event source that the activity was registered to or the ID of an external system, which was not accessible etc.)

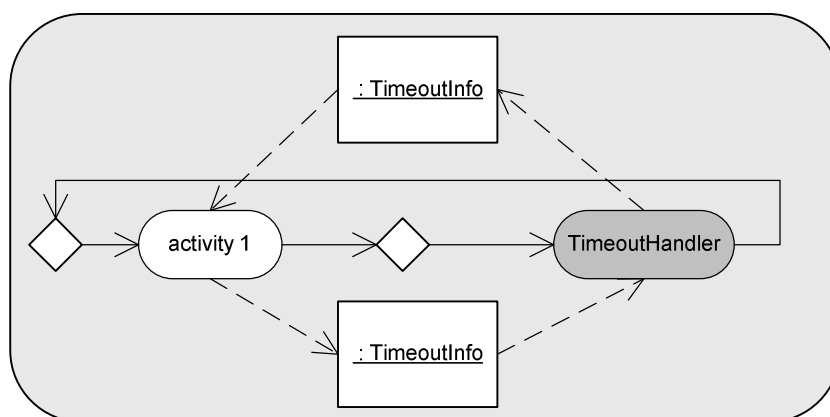


Figure 15: Passing of timeout information to and from the timeout handler

Provided with this information the timeout handler shall be able to decide how to react. To achieve a high degree of reuse it makes sense to externalize some handling logic in configurations of the timeout handler. This may e.g. include a configuration for each activity that states the

maximum retry count before aborting the process or a configuration that ignore timeouts of certain event sources.

## Consequences

- Timeouts reported by activities during process execution the can be handled generically within the process model according to defined rules.
- The activities that may finish with a timeout are not responsible for handling the timeout themselves.
- The process designer does not need to evaluate each single timeout situation and model the process respectively; rather the timeout handler component will be configured to direct the process flow accordingly.
- Activities that can finish with a timeout might have to supply additional information to the timeout handler so that the correct decisions can be made.

## Related Patterns

The PROCESS BASED ERROR MANAGEMENT [3] pattern describes a solution for handling errors that are reported by components integrated in the process flow. The TIMEOUT HANDLER pattern functions in a similar way, yet exclusively handles timeout situations. Thus, the TIMEOUT HANDLER can be viewed as special subset of the PROCESS BASED ERROR MANAGEMENT.

The EVENT-BASED ACTIVITY pattern introduces activities that listen for events occurring outside of the scope of the process instance. Activities of that type will usually be defined with a timeout value and might require the use of a timeout handler.

---

# Waiting Activity

---

## Context

The execution of a process instance shall wait for a defined period of time at a specific process step.

## Problem

**How is it possible to model a waiting position in a process that suspends the process execution until a defined period of time has elapsed?**

The execution of a process instance may consist of system-side processing and human interaction. Human activity processing can be rather considered time consuming in comparison to system-side execution of activities. Using time constraints to control interactively processed activities such as timeout or expiration mechanisms can serve as means to gain control of the activity's execution timing. But not only is the execution of single activities to be considered in this context. Controlled timing of the whole process flow is an important issue in process management systems.

Consider a process for order entry and order processing. Law Constraints might force the order processing to be started not before the defined period of legal order revocation has passed. The execution of the process instance has to wait a defined period of time before the next activities are scheduled.

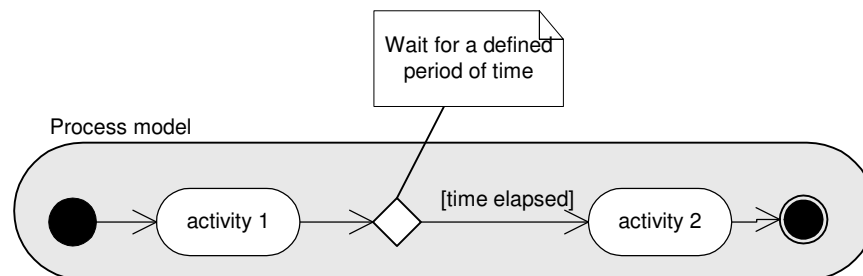


Figure 16: Problem illustration of defining waiting states in the process flow

The scenario mentioned above deals with a fixed or static time frame which does not change depending on other data processed in the system. The period is known during process design time and it can be supplied within the process model or in an external configuration. Yet, the amount of time to wait for may not always be known during process design, e.g. because it depends on a user input during process execution. In that case the process execution shall wait for a time that is to be evaluated during runtime.

## Solution

**Model a waiting activity which uses an internal timer to wait for the defined time frame to pass until it automatically terminates causing the process execution to continue.**

A waiting activity is a special system-side activity that employs a statically or dynamically configured timer. The waiting activity is responsible for starting the timer when the activity is instantiated and automatically terminating when the timer reaches the configured value. Thus, the process flow will be suspended for the desired time at that process step if the process engine schedules this activity.

The value specifying the time frame to wait for is either statically supplied to the activity, for example as metadata within the process model or from an external configuration source, or it is dynamically evaluated during runtime and supplied from within the process instance via the activity's input data.

Correspondingly, we distinguish static and dynamic timers. A static timer is configured during design time, a dynamic timer may be configured during runtime, e.g. by a previously executed activity (see Figure 17). That means the timer value is not statically bound to the process model, but is defined dynamically during process execution.

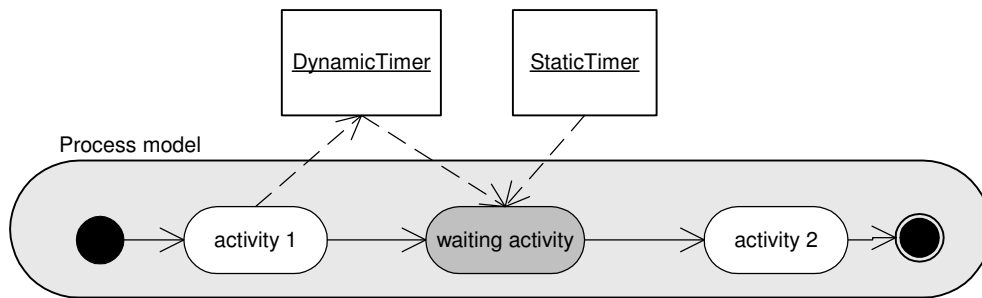


Figure 17: Introduction of a waiting activity

Use a static timer in case the time to elapse is not influenced by any measure within the system environment. Model a dynamic timer in case the timer's value can only be evaluated during runtime or will undergo current change by means of other system components.

## Consequences

- By modeling a waiting activity the process flow can be suspended for a defined period of time at a defined process step.
- When using a dynamic timer, it must be assured that the evaluated time frame is reasonable in the context where it is used. A wrong evaluation could result in an unexpected long suspension of the process execution.

## Related Patterns

The **EVENT-BASED ACTIVITY** pattern describes a general event-aware activity. An activity of that kind terminates upon the occurrence of a defined external event. The **WAITING ACTIVITY** pattern could be perceived as a special **EVENT-BASED ACTIVITY**. The event the activity subscribes to is created by a timer. This is only valid, though, if the timer is not known in the scope of the process instance and has to be seen as external event source. As many process engines offer a timer feature this is not always the case.

## Example

Most process engines offer integrated support for implementing waiting activities with static or dynamic timers. The example illustrated in Figure 18 is an implementation using the expiration mechanism of IBM WebSphere MQ Workflow a WMFC compliant process engine implementation [5]. The model and configuration shown in the next figures are defined with MQ Workflow Buildtime [12].

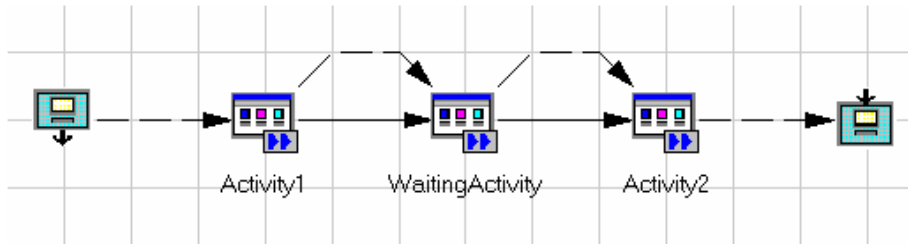


Figure 18: Example for a waiting activity

Figure 19 shows the configuration dialog for the expiration mechanism, which offers support for both static and dynamic implementations. The input value for a dynamic configuration can be obtained from the activity's input container. In order to implement a dynamic waiting activity using the expiration mechanism, model a sequence of two activities, the first is responsible for acquiring the value for the timer and passing it to the second one that uses this value available in its input container for the expiration configuration.

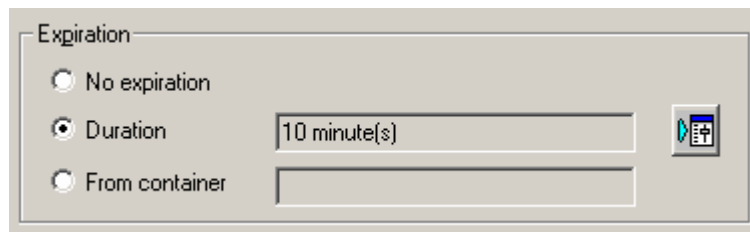


Figure 19 Configuration of WebSphere MQ Workflow's expiration mechanism

---

# Private-Public Business Object

---

## Context

The use of parallel business processes and activities implies that associated business objects may be accessed in parallel as well. To assure data consistency and integrity appropriate business object access and locking strategies have to be introduced.

## Problem

**How can parallel business processes and activities access shared business objects in a controlled way to avoid concurrent updates and to limit the visibility of uncommitted changes. That means how can persistent business object modifications be hidden from other system users and how can a business object be write-locked until explicitly freed by an activity?**

Business objects representing a company's or system's business data are usually accessed from within defined business process activities in process-driven applications. Such objects are typically managed and persistently stored in central data stores or business applications. Activities that process business objects during their execution may access these with a reference uniquely identifying the object. If such activities are executed in parallel conflicts may arise by a parallel write access, but also by a parallel read access to a business object that is currently being processed in another activity.

A parallel read access may become a problem, because business object modifications made in the context of one activity will be visible immediately to other system activities when the corresponding business object transaction is committed. If a business activity involves user interaction, for example, a referenced business object may be modified subsequently by multiple user invoked transactions. The activity can also be called a microflow in this case. The modifications are immediately accessible by parallel activities if the changes are committed after each interaction. Yet, this behavior may not be desired, because the microflow may span a long running transaction (LRT) that is not finished until the microflow is processed completely. A microflow may offer a user to undo modifications, for example. If a parallel activity reads the processed business object it may operate on an old business object state.

The problem of a parallel read access to a business object is especially given in the context of interruptible activities. The ACTIVITY INTERRUPT pattern [3] offers a solution for the problem of updating process control data during the lifecycle of such activities. Considering business object modifications in this scenario, it becomes obvious that it is not only desirable to update process control data when interrupting an activity, but it will also be necessary to save the current work done on business objects in order not to lose information and to be able to re-establish the context when resuming the activity.

When the user interrupts his/her work, modified business objects may not always be in a consistent state. Consider an activity which involves entering a large amount of data, for example a detailed customer feedback form. When the user processing the activity wants to interrupt his work, the data entered so far shall not be lost, but shall not be publicly visible either, because not all data of the feedback form has been entered. Thus, it may be necessary that interruptible activities allow saving inconsistent states of associated business objects. These states shall not be publicly visible, i.e. not become available in the system, until all data is consistent and the activity is finished. The read access to the business object shall be limited to the state it had before the changes were made.

Furthermore, parallel changes to a business object shall be prohibited. If an activity processes a business object and another activity performs parallel changes the object may enter an inconsistent state.

In the context of the ACTIVITY INTERRUPT pattern, if an object has been modified before an activity interrupt, the corresponding user expects to resume at the same point, i.e. the activity's context has to remain consistent. Therefore a business object modified within this interruptible activity shall be locked by the activity to prohibit parallel changes. In other words, write access shall be limited to the processing activity in order to guarantee a deterministic behavior of the business transaction spanned by the activity.

## Solution

**Introduce Private-Public business objects, which expose two separate images of the contained data - a private and a public image. A business object of that kind allows making changes on its “private” image prior to publishing the changes to its “public” state and enforces a write-lock for other activities as long as a private image exists.**

The concept of separation of business logic from process logic, which is also addressed by the related BUSINESS OBJECT REFERENCE [3] pattern, implies that the modification of business data is done on business objects that are managed outside of the process engine. Consequently, the PRIVATE-PUBLIC BUSINESS OBJECT Pattern solves the above mentioned problem within the business object domain.

The fundamental concept of this pattern is related to existing mechanisms in the area of transactional systems, like for example database transaction and locking features that prevent parallel updates by write locks and assure limited data visibility during a running transaction. Yet these technical solutions to data visibility and write control, do not address the problem within the context of process-driven systems.

Even if business objects are stored via a database management system (DBMS) updates to a business object from within a business activity will often be performed in more than one single database transaction. Processing of a business activity can involve several user interactions, for example. Each user interaction typically spans a database transaction in this scenario. The activity itself, though, spans a business transaction that shall not be committed until the activity is finished.

The requirements on business object visibility and write control within a business transaction are addressed by introducing private-public (p-p) business objects. A business object of that kind has a public and a private image (see Figure 20). Both images contain the same business object attributes.

The private image is created during a business transaction and reflects the object's state which is only visible from within this transaction, i.e. the processing activity. The public image of the object reflects its state which is always publicly visible. When the activity commits the first modifications on a private-public business object, a private image is created that exposes the business object's data including the changes. Any read access to the business object from within another business activity returns the object's public image that does not contain these modifications. Any parallel write access will be prohibited if a private image exists.

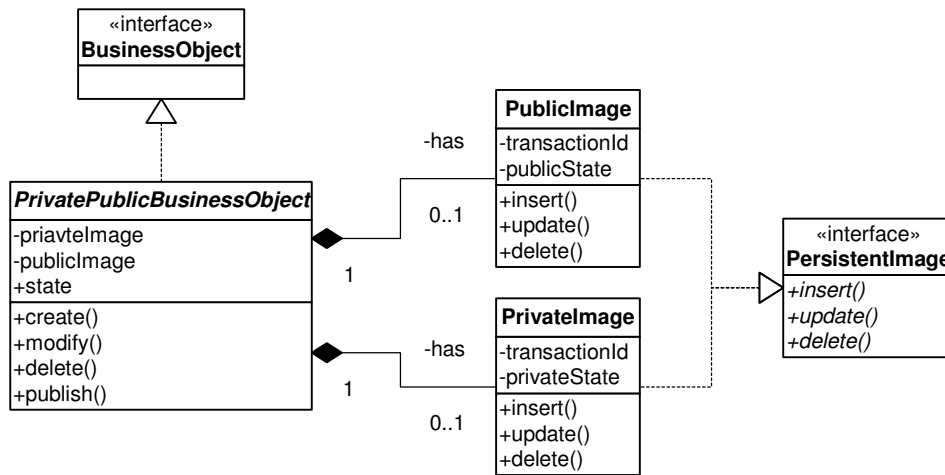


Figure 20: Class diagram for P-P Business Objects

A read access to the business object issued from the activity which initially performed the changes will return the private image and thus reflects all modifications made in that business transaction so far. When the activity is finished it explicitly publishes these modifications. The changes stored in the private image are merged into the public image and will then be accessible by all system activities.

To be able to differentiate between the activities that access the object, a unique activity or business transaction identifier (ID) has to be introduced. The identifier has to be compared with the owning transaction ID during each read and write request. Thus, the business transaction which created the private image is stored in that image (see attribute “transactionId” in Figure 20).

Figure 21 shows the possible transitions between private and public states of p-p business objects. The object is said to be in a private state if a private image of the object exists and in a public state if it has no private image. So the p-p business object is either in one of the private states (“created”, “modified” and “deleted”) or it is in public (“published”) stated. Figure 21 also shows the state “locked” which is the state of the public image once a private image exists.

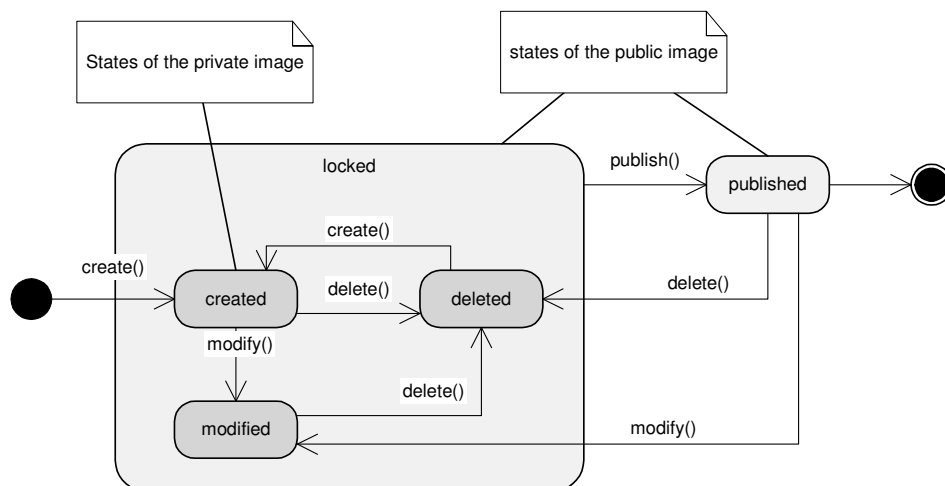


Figure 21 Public and private state transitions of P-P business objects

The differentiation between the private states “created”, “modified” and “deleted” allow applying the correct merge operation when publishing the changes. When the object is initially created no

public image exists. The publish operation will invoke the public image to be created. If the object is being deleted before the first publish operation is called then no public image has to be created. Once a public image exists the business object can either enter the states private modified or private deleted. If a private image exists or the business object is not in the published state, its public image is write-locked.

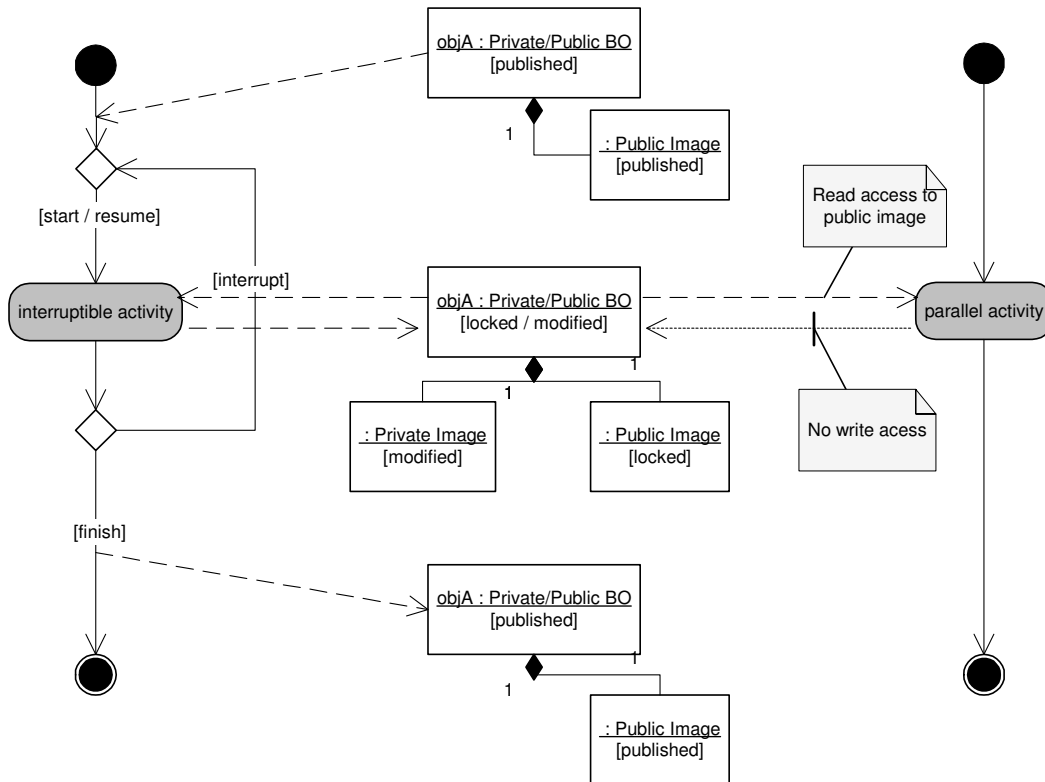


Figure 22: Parallel access to private-public business objects

Figure 22 gives an example of parallel access to a private-public business object which is modified during the execution of an interruptible activity. The first modification of the object within the activity creates a private image reflecting the changes and enforces a write-lock on the object's public image. Parallel activities can still read the objects public state, but are not eligible to perform any changes. The public image remains in state "locked" during the processing and interruption of the activity. The activity may modify the object's private image until the user finishes the activity. When finishing the activity the private image is merged into the public image and deleted afterwards and the write-lock is freed and the object is in state "published" again.

## Consequences

- The pattern offers a concept for accessing a business object from within different activities. The business object is publicly write-locked by the first activity that creates a persistent private image.
- Business objects may be leaved write-locked for a long time, which may block execution of parallel processes. As a consequence this aspect enforces a clear and integrated design of business processes and business objects.

- Users of a process management system involving interruptible activities may save modifications on business objects made within the activity in inconsistent states prior to finishing the activity.
- The realization of the pattern requires implementing a custom persistence framework for such business objects or extension of existing frameworks, which causes implementation effort.

## Related Patterns

The **ACTIVITY INTERRUPT** pattern describes activities that allow interrupting and resuming work done within the activity. The pattern concentrates on process control data to be saved within the work item container.

The **BUSINESS OBJECT REFERENCE** pattern takes the separation of business data and process engine into account. This separation of concerns is achieved by linking business objects associated to an activity and storing only references to these objects in the process control data.

Considering these concepts the **PRIVATE-PUBLIC BUSINESS OBJECT** pattern introduces a way to handle business objects referenced by and modified in activities, especially interruptible activities.

## Example

The pattern can be implemented in several ways depending on the technology employed to store business objects in the system and the way service access these objects. When using a relational database to store business objects, the concept of a private-public business object can be realized by introducing two database schemas, one for the private images and one for the public images of the business objects.

Ideally, activity implementations (services) do not have to deal with the different images themselves, but simply perform operations on the business object. The special persistence mechanism required for private-public business objects is handled by an application's persistence layer that encapsulates the required write and read logic.

Figure 23 gives an example of the persistence logic using a relational database. In order to hold information about the transaction locking a business object the column "TransactionId" is contained in each table of the public and private schema. The states of the private and public images are stored in the "PrivateState" and "PublicState" column.

In the example a customer object is being updated and another is being deleted during a long running transaction, e.g. an interruptible activity. The changes performed during this activity are logged in a transaction log table. This is needed to keep track of the operations to be executed when the long running transaction is committed, i.e. the business objects are published.

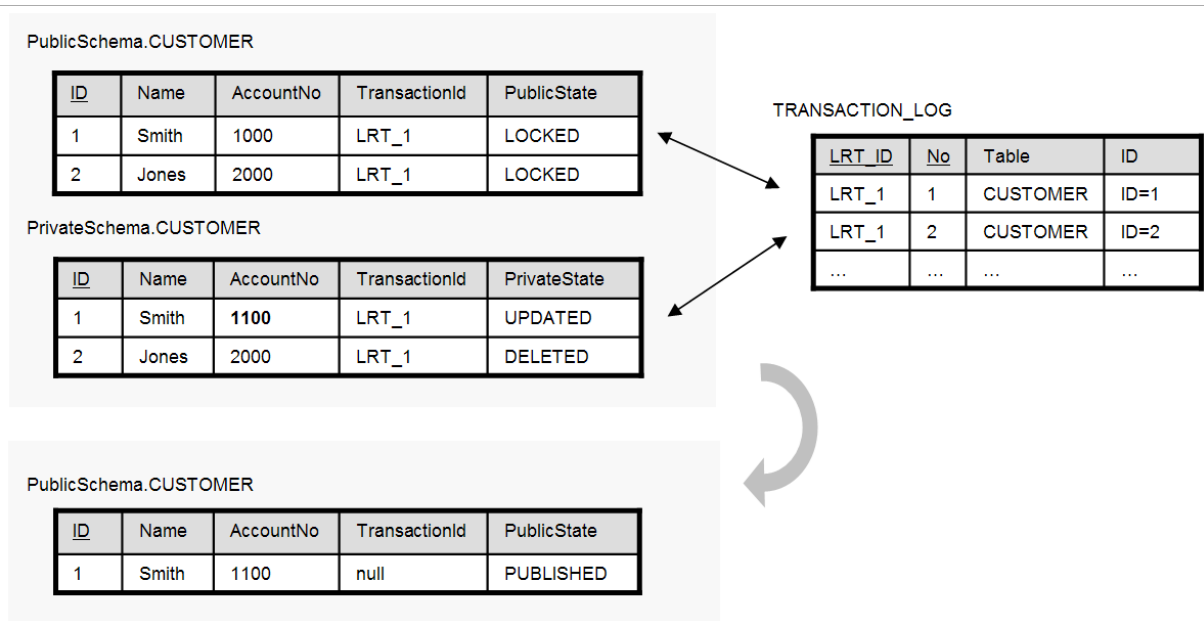


Figure 23: Realization with a relational database

The pseudo code below gives a high level example of an implementation of the private-public business object’s publish method. Depending on the state a database row in the public schema will be inserted, updated or deleted for an existing private image. The database operations (e.g. executed via JDBC) are supposed to be encapsulated within the implementation of the “PublicImage” class.

```

public abstract class PrivatePublicBusinessObject implements BusinessObject {
    private PersistentImage privateImage;
    private PersistentImage publicImage;
    private int state;

    public synchronized void publish() {
        if (state==STATE_CREATED) {
            publicImage = new PublicImage();
            // Copy all attributes of private to public image
            copyAttributes(privateImage, publicImage);
            // Set the state and insert the row
            publicImage.setState(STATE_PUBLISHED);
            publicImage.setTransactionId(null);
            publicImage.insert();
            // Delete private image
            privateImage.delete();
        } else if (state==STATE_MODIFIED) {
            // Merge all changes of private to public image
            mergeAttributes(privateImage, publicImage);
            // Set the state and update the row
            publicImage.setState(STATE_PUBLISHED);
            publicImage.setTransactionId(null);
            publicImage.update();
            // Delete private image
            privateImage.delete();
        } else if (state==STATE_DELETED) {
            // Persistently delete public and private images
            publicImage.delete();
            privateImage.delete();
        }
    }
    ...
}

```

Simple data access becomes more complex in this scenario. If all Customer objects visible in the context of a specific long running transaction are to be retrieved a simple query like:

```
SELECT NAME FROM CUSTOMERS WHERE ((NAME LIKE 'AB%'))
```

has to be changed to retrieve all public images except the ones currently processed in this long running transaction (identified by <current\_LRT\_ID>) and perform a union with the private images that are in this transaction, but are not marked as deleted:

```
SELECT NAME FROM PublicSchema.CUSTOMERS
WHERE (NAME LIKE 'AB%') AND TransactionId <> <current_LRT_ID> OR
TransactionId is null
UNION
SELECT NAME FROM PrivateSchema.CUSTOMERS
WHERE (NAME LIKE 'AB%') AND TransactionId = <current_LRT_ID> AND
PrivateState <> <deleted>
```

This algorithm to retrieve the correct data depending on the current scope (within a long running transaction or not) should be encapsulated in an adequate way, e.g. by offering special database union views.

## References

- [1] Zdun, U. and Hentrich, C. and van der Aalst, W.M.P. (2005). A Survey of Patterns for Service-Oriented Architectures, *International Journal of Internet Protocol Technology*, 2005
- [2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [3] C. Hentrich. Six patterns for process-driven architectures. In *Proceedings Conference on Pattern Languages of Programs (EuroPLoP 2004)*, 2004.
- [4] D. K. Barry. *Web Services and Service-oriented Architectures*, Morgan Kaufmann Publishers, 2003.
- [5] WMFC. *The Workflow Reference Model (WFMC-TC-1003)*, Workflow Management Coalition, 1995.
- [6]. WMFC. *Terminology and Glossary (WFMC-TC-1011)*, Technical report, Workflow Management Coalition, 1996.
- [7] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. *Workflow Patterns*. BETA Working Paper Series, WP 47, 2000.
- [8] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced workflow patterns. In *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, Berlin, 2000.
- [9] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. *Workflow Resource Patterns*. BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, 2004.
- [10] P. Lawrence, editor. *Workflow Handbook 1997*, Workflow Management Coalition. John Wiley and Sons, New York, 1997.
- [11] Deepak Alur, Dan Malks, John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall PTR, 2003.
- [12] IBM corporation. *WebSphere MQ Workflow 3.4 – Getting Started with Buildtime*, IBM corporation, 2003.
- [13] M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns*. Pattern Series. John Wiley and Sons, 2004.