Six Patterns for Process-Driven Architectures

Carsten Hentrich
SerCon – IBM Business Consulting Services
Germany
e-Mail: *carsten.hentrich@sercon.de*

This paper introduces a set of six patterns for Process-Driven Architectures that are based on workflow middleware technology according to the Workflow Management Coalition (WfMC) standards. The results, as illustrated in this paper, are founded on and extracted from practical project experiences and customer requirements that have been gathered in the past by various experts within IBM.

Introduction

In the last ten years workflow technology [FLDR00] has evolved as an important technological artefact, as far as Business Process Management [Prior03] is concerned. Today, a *process engine* takes the role of a process controller within a Model View Controller (MVC) architecture [FB+96], in order to coordinate and distribute tasks in a heterogeneous system environment. The resulting applications represent socio-technical systems, where business steps are carried out fully automatically, or by human interaction. People and IT systems themselves become an integral part of the process. Thus, in this respect a *process engine* represents an important artefact within an enterprise architecture. IT strategies have furthermore changed accordingly to achieve that process-driven system architecture, in order to flexibly change and adapt business processes in a distributed environment. For this reason, the general aim of *Process-Driven Architectures* is to decouple process logic from business logic.

Some research has been done concentrating on analysing, designing and modelling business processes and workflow, but little work has yet been done on the overall and integral architectural perspective of such systems. Most research efforts have rather been concentrating on single architectural components, but not on interactions and processes between flexible and exchangeable architectural components. As a result, it is actually not enough to concentrate on the *process engine* itself, but to view it as an integral part within an architecture. Only a few design patterns are available in this context. Obviously, a *process engine* represents a special type of middleware component within an architecture as a business process oriented middleware approach is followed that implies different concepts and strategies, compared to common architectures. Some key driving factors are flexible business process control, integration of legacy systems functionality within business processes, automatic invocation of services, open connectivity to various interfaces, scalability, automated measurement and monitoring of business processes, processes as reusable components, B2B, and business process automation.

In order to achieve these strategic goals, architectures must be designed accordingly. Actually, those architectures might as well incorporate advantages of other technologies like object orientation, for instance. As a result, design patterns for those architectures would be very useful if they consider best practices and experiences, as well as aspects of reusability, maintainability, and flexibility.

One bottom-up approach to the issue of identifying patterns is analysis of existing workflow projects and products in order to derive generic elements and solutions from that analysis in terms of design patterns. The aim of this research will be the development of a set of reusable design patterns that incorporate best practices and modern Software Engineering techniques. The results of this research can then be used on designing architectures and applications of this kind, and moreover to avoid mistakes that have been made in the past. This paper illustrates a set of six reusable design patterns for *Process-Driven Architectures* based on workflow technology according to the WfMC standards [WfMC95], [WfMC96], [WfMC99], [WfMC00]. These six patterns address both detailed process modelling and design issues in a technical and architectural context. Thus, this paper is addressed to software architects, workflow designers, and developers.

Process-Driven Architectures

IBM has spent quite some effort on identifying general patterns for e-business. The results of this work have already been documented and published [AKVC03]. In these publications IBM has gathered the practical experiences of the last decade for increasing the efficiency of e-business processes. All those patterns are applicable and the results are very valuable and influential in the context of *Process-Driven Architectures*. Most of these patterns are already reflected by many technologies.

Process-Driven Architectures go a little further as they imply the vision of convergence of business, organisational, and software models and thus aim at providing a framework that allows to design and implement business, technological, and organisational architecture in interdependence. The notion of (*business) process* is the driving and linking concept of all involved aspects. Essentially, this framework establishes the paradigm of a flexibly growing and changing socio-technological organisation.

The very key principle of *Process-Driven Architectures* is separation of business logic from process logic. Practically, this principle means that the process logic will be *taken out* of applications and will be flexibly controlled by a process-driven middleware. Some examples of available products that support such a process-driven approach are *Staffware Process Suite* [StaffPS], *Fujitsu i-Flow* [FuFlow], *Oracle Workflow* [OrFlow], *FileNET Business Process Manager* [FileBPM], or *WebSphere MQ Workflow* [MQWF].

The flexibility is achieved by introducing an abstraction layer for process control with such process-driven middleware that allows to change the process flow definition more easily compared to static process implementation within application program code. For instance, consider a service oriented application infrastructure where services can be flexibly composed in a business process flow with the help of such middleware. The picture in *figure 1* illustrates this principle by giving an example of a customer entering an order, and further how this order will be processed within the organisation in a process-

driven approach that separates process logic from business logic and which extracts the process logic from the application logic.

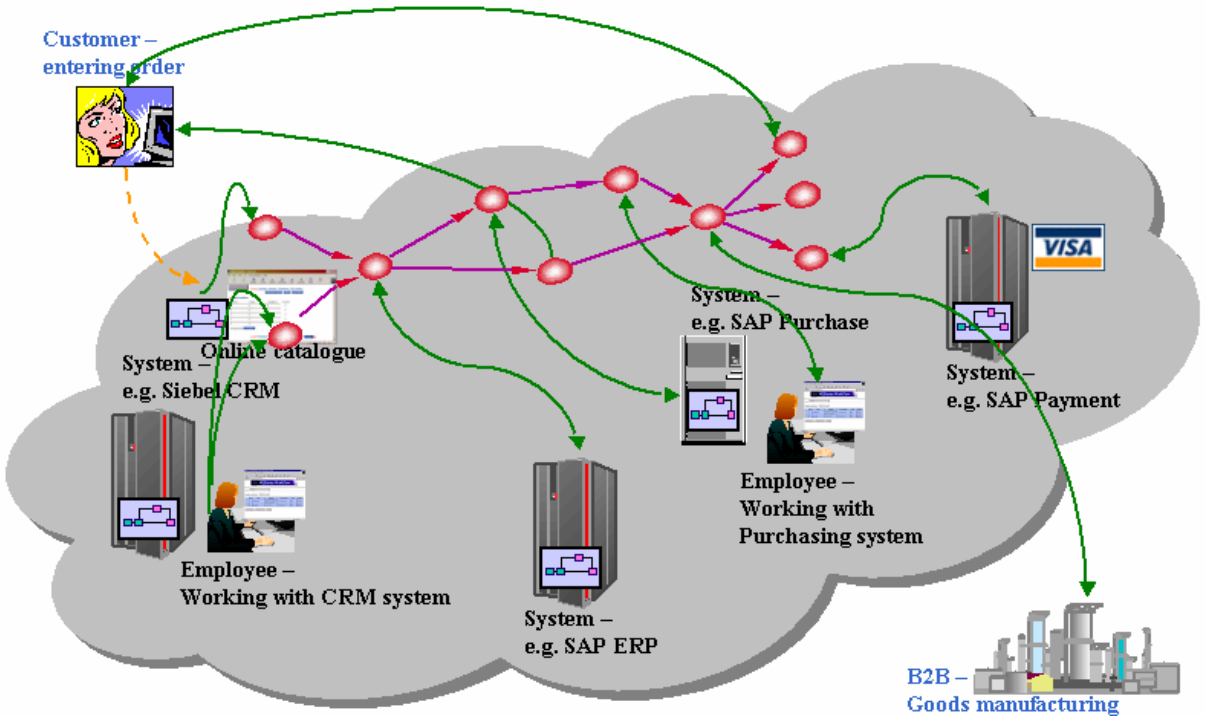

figure 1: separation of process logic from business logic

From an IT perspective, this very principle of decoupling actually represents the most important transformational step that must be taken. Basically, this paper will deal with the architectural implications of this transformational step and will seek to find pattern solutions for arising issues that result from conflicting forces in this context. Thus, on an architectural level, business processes are understood as flexible components that can be easily changed, replaced, and reused on an enterprise wide basis and even across enterprise boundaries in order to connect and integrate the processes of different organisations. However, the following patterns address more detailed issues on a technical level within this paradigm.

Patterns Overview

The diagram in *figure 2* illustrates the relationships between the patterns and *table 1* provides an overview of the problems and solutions that each pattern addresses. Basically, the diagram in *figure 2* expresses that application of certain patterns have influences on other patterns and that there is particularly one central pattern GENERIC PROCESS CONTROL STRUCTURE that is related to all the five other patterns. The arrows associated to the GENERIC PROCESS CONTROL STRUCTURE pattern indicate that this pattern provides the basis for generic solutions of issues addressed in more detail by the five other patterns. The types of influences are indicated by the descriptions on the arrows in the diagram. It will be illustrated what these relationships are about in the pattern descriptions. Moreover, the patterns as listed in *table 1* relate to different categories. The following categories have been defined:

- *Interface:* patterns that address issues concerning interfaces between different components (in this case interfaces between processes).
- *Process*: patterns that primarily address process modelling issues.
- *Architecture:* patterns that address broader architectural issues that view a process engine in context with other architectural components.

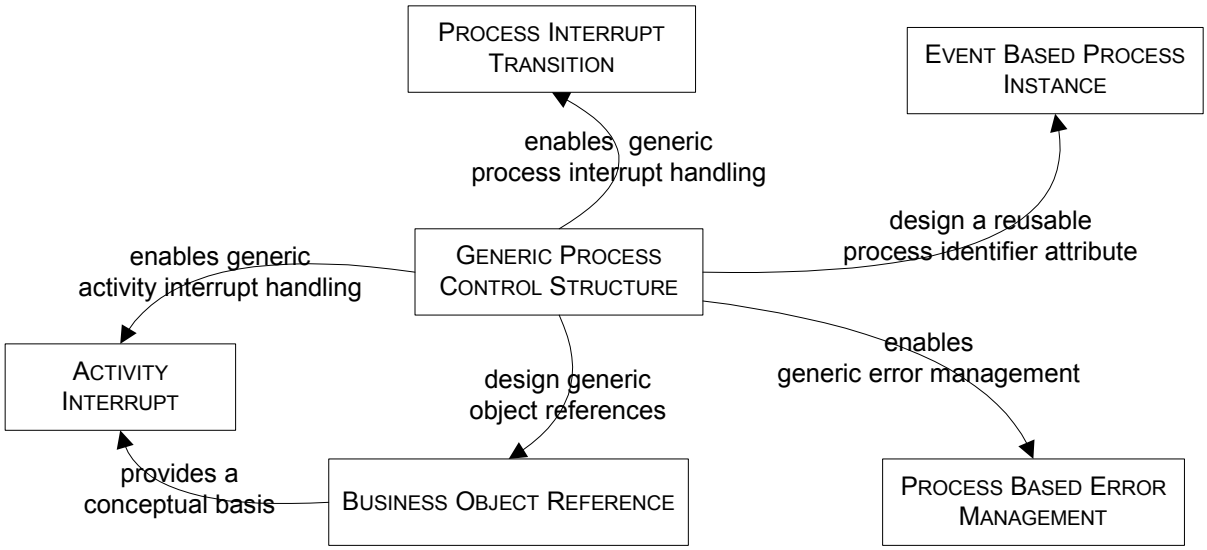

figure 2: patterns overview and relationships

Pattern	*Problem*	*Solution*	*Category*
PROCESS INTERRUPT TRANSITION	How can an instantiated process be interrupted and allocated resources be freed in a controlled way, in case external circumstances force the process instance to stop?	Model the interrupt as a conditional transition involving a cleanup activity in case the interrupt condition is true.	Process
ACTIVITY INTERRUPT	How can an activity in process be interrupted without loosing 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
PROCESS BASED ERROR MANAGEMENT	How can errors that are reported by integrated applications	Define special fields for error handling in the process control	Process

Pattern	*Problem*	*Solution*	*Category*
	in activities in a process flow be handled and managed?	data structure and embed an activity in an error handling control flow.	
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.	Architecture
EVENT BASED PROCESS INSTANCE	How can a process instance be automatically created in case an event based activity occurs in a business process that implies automatic process instantiation, e.g. a customer placing an order?	Externalise event based activities to an external event handler component and split the process model in several parts.	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

table 1: problem/solution overview of the patterns

Process Interrupt Transition

Context

A process instance shall react to an external event by aborting the normal process flow before the normal process-end is reached and allocated resources shall be freed.

Problem

If a process is instantiated it will run according to its flow definition until the defined end is reached (normal process termination). How can an instantiated process be interrupted and allocated resources be freed in a controlled way, in case external circumstances (outside the process engine) force the process instance to stop before the normal process-end is reached?

A high-level business process model only captures the flow logic from a business point of view. Furthermore, once a technical process instance is created on the *process engine*, it will run according to the defined flow logic until the defined end. In this context the problem occurs that it might be necessary to stop a running process instance, because external circumstances may force the process to stop, e.g. a customer cancelled the order that is currently being processed by a process instance. A high-level business process model does usually not capture all these possible external circumstances and the necessary reactions to them. However, technically it is absolutely necessary to stop a superfluous process instance, because it does actually not make sense to process a cancelled order, for instance.

Moreover, a process instance may allocate certain resources, like storing temporary process data in a database or making changes to the state of an external system, e.g. entering an order in a central ordering system. Thus, just physically deleting the process instance will not be a sufficient solution, because temporary allocated resources will not be freed and state changes to external systems will not be rolled back. It might also be required to document the process interrupt for revision purposes, i.e. information must be kept why the process was terminated abnormally. For this reason, a concept is required to interrupt a process instance in a controlled way and to systematically free allocated resources.

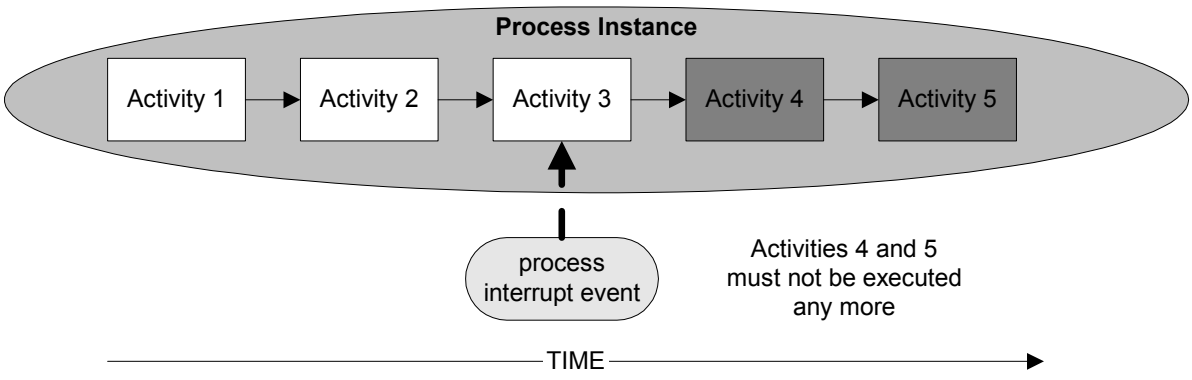

figure 3: illustration of a process interrupt caused by an external event

Solution

Consider the possible abnormal process termination already in the process flow definition and model the process interrupt as a conditional transition involving a cleanup activity in case the interrupt condition is true.

The problem definition indicates that we are dealing with an issue that is similar to handling exceptions in a program. The atomic unit in a process is an activity. For this reason, a controlled interrupt of a process can only be achieved between activities, i.e. in a transition from one activity to the next. As a result, a controlled process interrupt must be modelled in the process by a corresponding transition condition that indicates that *something* has happened in the outside world that forces the process to terminate.

The actual reason why the process must be interrupted may vary widely. The process model is not interested in the very reason, but only in the fact that *something* has happened that forces the process to stop. For this reason, it is only necessary to concentrate on the basic procedure how to handle such an event. Thus, the whole issue can be reduced to handling a whole class of possible failure events, because for the process model it does not matter what specific failure it was.

Moreover, the question is where and how is determined that such an event has happened? The answer is: in implementations of process-activities. The implementation of an activity is the application that executes the task of an activity (one has to remember that the process model only triggers execution of activities). Thus, activities are controlled by the outside world from the viewpoint of the *process engine*. That means the application that implements an activity is the place where business logic is executed, and it is actually business logic to determine whether an event has happened that forces a process to stop or not. Thus, the solution is to define a standardised way of handling these events in *activity implementations*. This pattern can then be implemented in a technical programming framework, for instance.

Summing up, from an architectural point of view, it is possible to extract an interesting aspect, i.e. that the complete solution consists of two parts. One part of the solution references process modelling, and the other part of the solution references implementation of activities in a way that handles events that finally result in process interrupts. Both aspects are strongly interdependent, as they rather represent two aspects of one single pattern. The conceptual process pattern for modelling a process interrupt is depicted in the next *figure 4*.

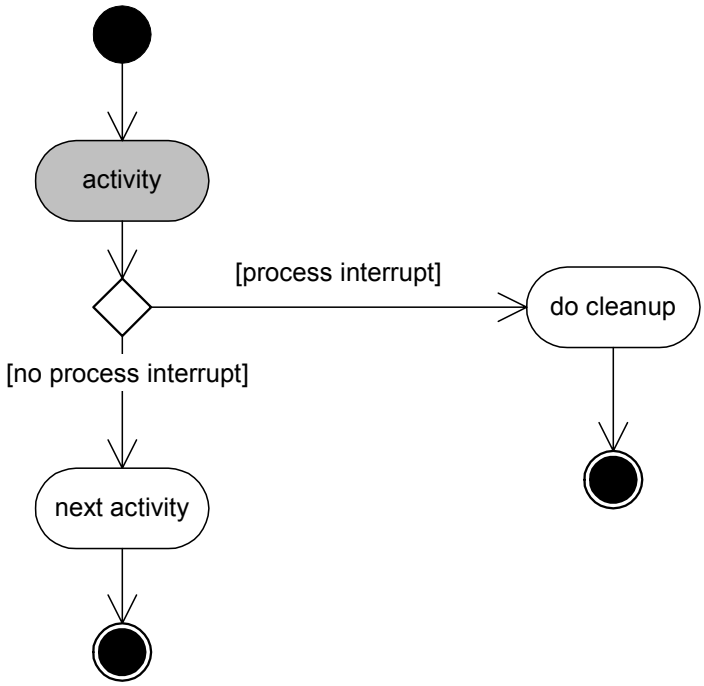

figure 4: process interrupt modelling pattern

The second part of the pattern will provide a solution for handling events in *activity implementations* that result in process interrupts, which are then handled by the actual business process model as it has just been defined in the previous diagram. The *activity implementation* has to determine whether something has happened that causes a process interrupt and, if that is the case, the *activity implementation* has to inform the process instance by setting the corresponding attribute of the process control data structure that informs the process instance about the interrupt and allows the process instance to react accordingly. The next diagram in *figure 5* defines the pattern how an *activity implementation* principally has to respond to such events.

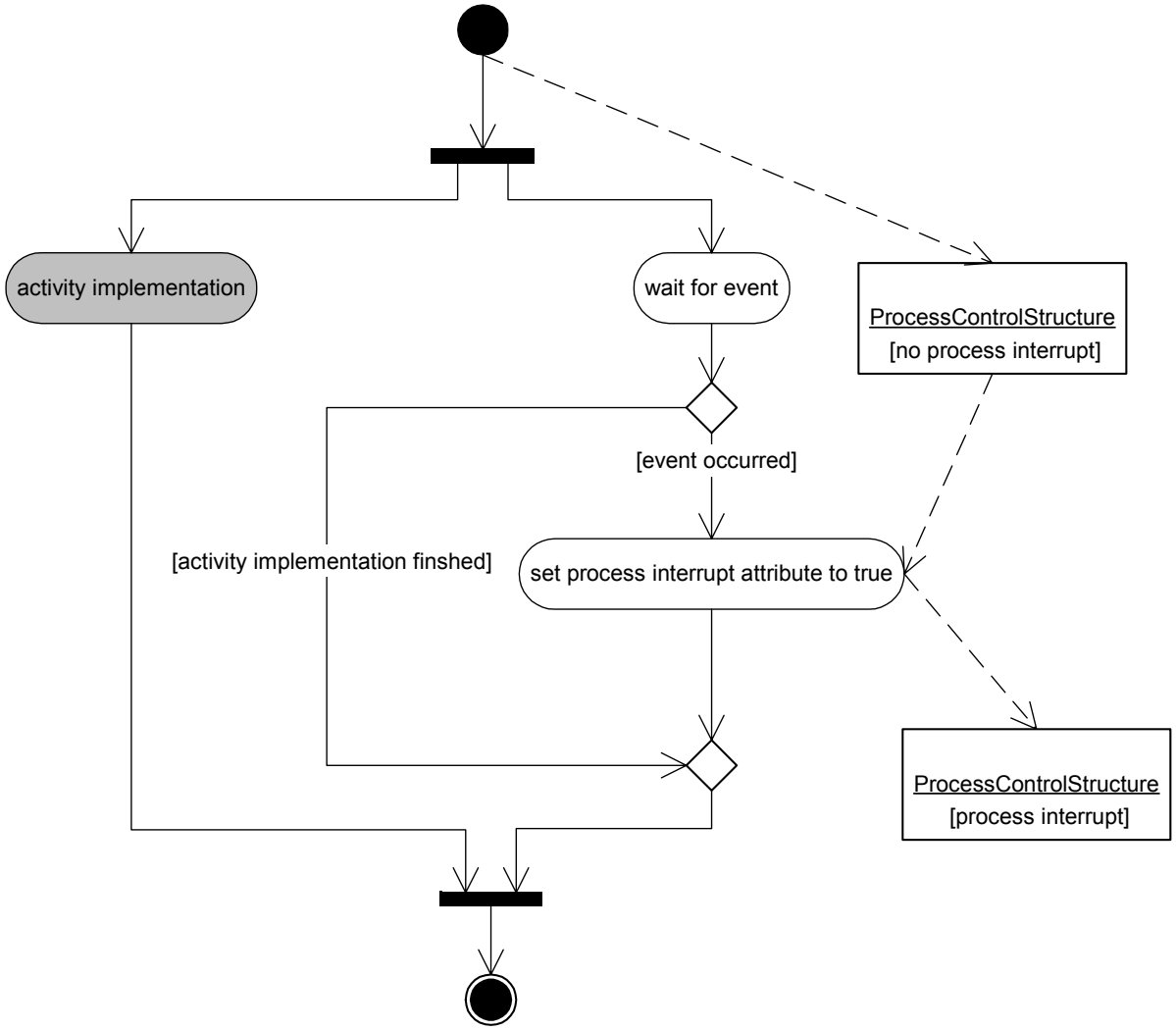

figure 5: handling of events that result in a process interrupt

The recommended design decision is to define a reusable generic attribute in the process control data structure to handle the process interrupt. It can thus be easily tracked when a process has been interrupted according to the logged value of this attribute.

Consequences

- The process control data structure must contain an attribute for reporting the process interrupt to the process instance.
- Implementation of a generic mechanism for handling process interrupts.
- The pattern provides a basis for an implementation in a programming framework, because it defines a general behaviour for any activity that could be simplified for an application developer by implementing it in a programming framework.
- There is additional development effort necessary in order in implement the event handling in *activity implementations* and in to implement the cleanup activity.

- Actually, the cleanup activity is optional. It might be possible that it is not necessary to free any resources, because none have been allocated. In that case a cleanup might not be required.
- The process models are becoming a little bit more complex, because the *process-interrupt* and *no-process-interrupt* conditions must be modelled in addition to the normal transition conditions. Thus, the logic of the transition conditions is getting more complex. In case a transition condition already exists from one activity to the next, the logical AND operator must be applied, in order to define a complete transition condition that considers a process interrupt. Such a complete transition condition would generally look as follows: *no-process-interrupt AND normal transition condition,* as the process has to only make the transition to the next activity in case there is no process interrupt and the normal transition condition is true.

Related Patterns

Application of this pattern principally has an impact on the design of a process control data structure as an attribute is required to report the process interrupt. Consequently, it will influence the design of a generally applicable process control structure. For this reason, GENERIC PROCESS CONTROL STRUCTURE is related to this pattern. Thus, if the GENERIC PROCESS CONTROL STRUCTURE pattern is applied, then a generic attribute to handle process interrupt transitions must be defined in the generic process control data structure.

Example

The diagram in *figure 6* illustrates a sample process model that has been implemented with *WebSphere MQ Workflow*, which applies the PROCESS INTERRUPT TRANSITION pattern. The model is defined with *MQ Workflow Buildtime* [WFBT03] and shows the transition from one activity to a next activity only in case the interrupt condition is false. In case the condition is true, a cleanup activity is executed and the process terminates.

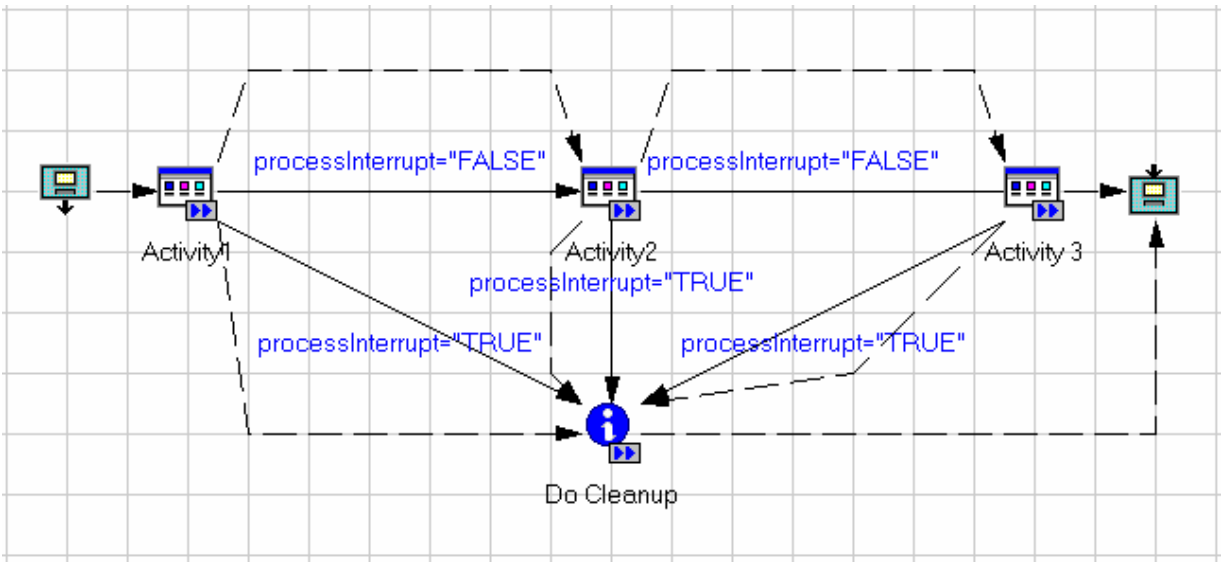

figure 6: process interrupt implementation example

The activity processing application, i.e. the *activity implementation* sets the *processInterrupt* attribute in case an event has occurred that causes the process instance to terminate abnormally. Some projects have implemented this behaviour in a programming framework in terms of an event listener that automatically sets the attribute in case a process interrupt event is fired.

Activity Interrupt

Context

The process control data should be updated but the process instance must not move on to the next process step but rather stay at the current position.

Problem

Process control data can only be manipulated via a check-out and check-in of an activity, i.e. normally processing an activity. However, if an activity is checked-in, the process engine will usually assume that the activity is finished and will move on to the next process-step. How can an activity "in process" be interrupted without loosing any data, which means that the process control data must be updated but the process instance remains in the current process-step?

Especially as far as activities with human interaction are concerned, another issue has been observed that requires attention. The issue occurs due to the fact that humans often start working on a task but will not finish the whole task in one session. Rather, they want to store what they have done, thus keep the changes they have made, leave their work for a while, e.g. until the next day or after the lunch break, and finish their work later.

For this reason, a concept is required that allows in a way to "interrupt" an activity, i.e. to store the changes for later use and to finish the activity later. A business process model does usually not consider these details but a technical representation of the business process apparently has to consider this. As a result, the final process model must consider that activities can be interrupted. Thus, there must be a control mechanism to inform the process instance not to move on to the next step but to remain at the same process-step and to store the changes that are output of the current process-step.

The problem in this context is that a process instance usually moves on to the next process-step in case the current process-step has been finished. Activities (process-steps) are atomic units in a technical representation of a business process when they are executed on a *process engine*. Usually, if an activity is finished, the output data of the activity will be transferred to the process instance and will be passed further as input data to the next activity in the transition, i.e. the process instance makes a transition into the next state represented by the following activity. However, in case of an activity interrupt, an activity will actually not be completely finished yet but will only be interrupted in order to finish work later. For this reason, the process instance must not move on to the next process-step yet, but stay at the same activity and the process control data must be updated. As a result, an activity interrupt occurs every time the process instance must not move on to the following activity but the process control data must be updated.

In order to process an activity, a *process engine* provides check-out and check-in mechanisms. A check-in of a checked-out activity means that the activity has been finished and the process control data can be updated during check-in. Usually, this implies that the process instance will move on according to the process model. That is actually part of the concept of a *process engine*, because a check-in of an activity usually implies a change in the process-state of the process instance. In case of an activity interrupt it is not

desired that the process instance moves on, but rather the check-in means that the current activity should be restarted with updated process control data. Thus, it is necessary to provide a mechanism that distinguishes these two types of check-in. The illustration in *figure 7* shows the transformation of a business process into a technical representation that considers restarting activities with updated control data (activity interrupt). As illustrated in *figure 7*, in this context the very problem is how that activity restart can be achieved.

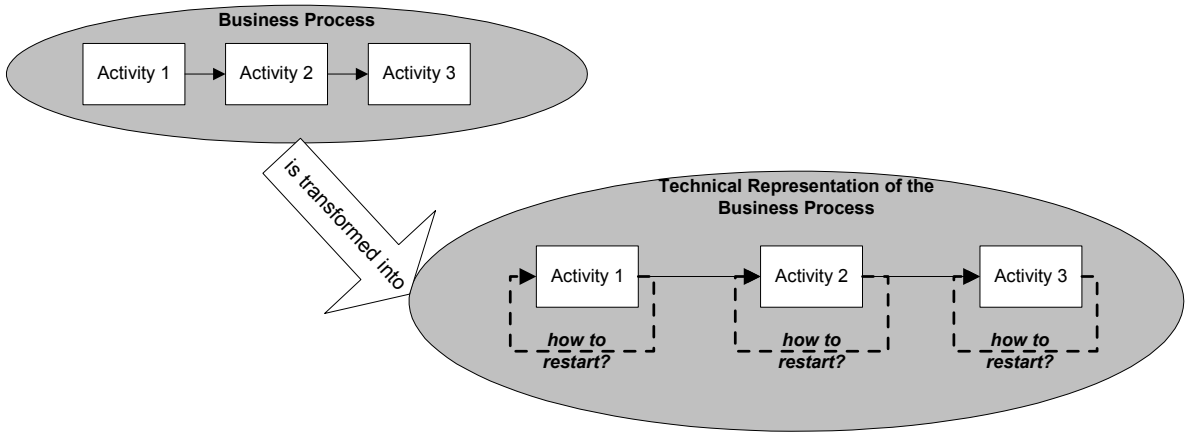

figure 7: problem context for an activity interrupt

Solution

Model an exit condition that restarts an activity in case the exit condition is false and use the output data of the activity as input data to the restarted activity. This concept allows to check-in an activity, updating the process control data, but the exit condition indicates the process engine not to move on to the next process-step, but rather to restart the current activity with updated process control data.

Generally, the solution is to update the process control data by finishing the activity, passing the updated data to the process instance, and by simultaneously telling the *process engine* not to move on to the next process step but to restart the current activity by applying the updated control data as input data to the restarted activity. The activity restart is achieved by modelling an exit condition for an activity in the process model, which initiates the restart of an activity in case the condition is false.

The concept of separation of business logic from process logic implies that actual changes to business data will be made in a business object which is managed outside the *process engine* (compare the related BUSINESS OBJECT REFERENCE pattern). Conclusively, the process model will not be concerned with changes to business data but with changes to process control data. It might happen that a new business object is created somewhere in the outside world around the *process engine* and the process instance must store the reference to that business object.

For this reason, theoretically, the only event that might happen that implies manipulation of process control data in this context is creation of new business objects, which need to be referenced in the further process flow. If a changed business object is already known by the process instance, it will only be necessary to stay in the same process-step and not to move on to the next step. No further changes to process control

data are necessary. If new business objects are created, it will additionally be necessary to store the references to them. Ultimately, storing these new references will be necessary in any case, whether the activity is interrupted or not.

Unfortunately, this is the theory but not the reality. The process control data structure may also include attributes that have helper and support functions. Those attributes might be affected by those changes to business data. The application will then store the modified control information, which are in fact related to some business data changes, in the process control data. For instance, in order to reflect an application specific state of the referenced business objects in the process instance. For this reason, the practical answer to the question is: new business object creations and modification of process related helper and support attributes concerning business object changes will imply changes to process control data in the context of an activity interrupt. The next model in *figure 8* illustrates the design pattern that can be used for modelling activity interrupts in a process.

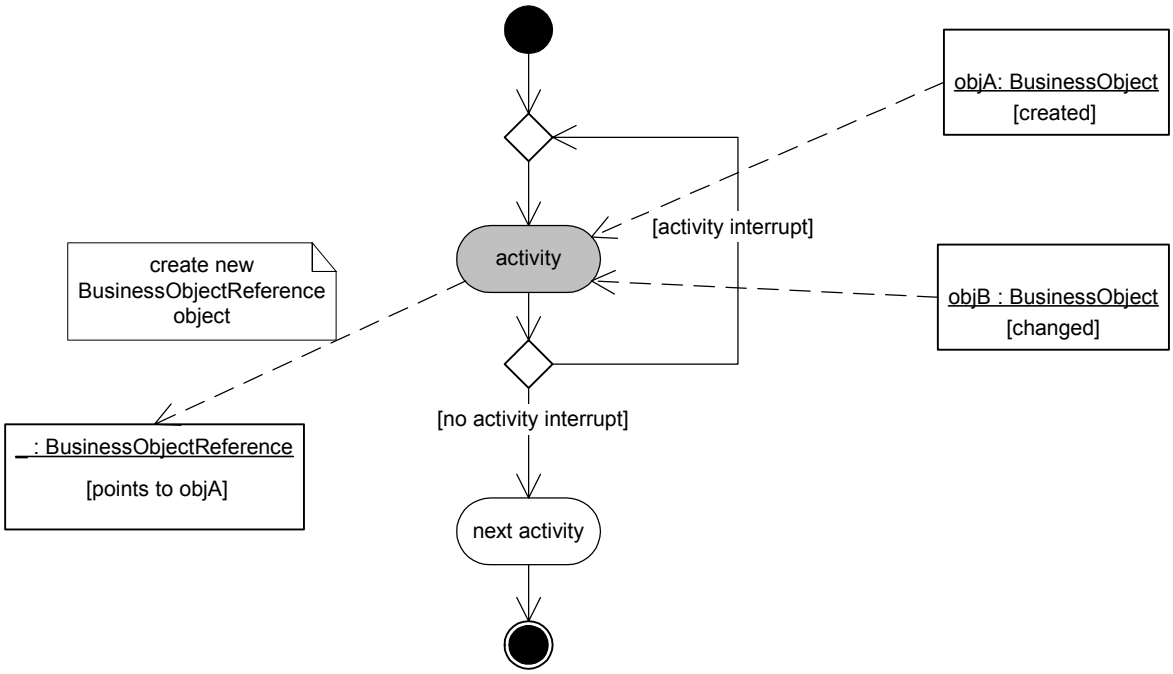

figure 8: activity interrupt process modelling pattern

The diagram in *figure 8* illustrates that the reference to a newly created business object (*objA* in *figure 8*) will be stored in the process control structure, but a change to an already referenced business object (*objB* in *figure 8*) has no effect on the process control structure, because the object reference is already there. Changing a business object is simply a change in the objects internal state and has for this reason no impact on the process control data, because of the concept of separation of concerns. On the one hand, one can argue that in case the business object is deleted, the reference in the process control structure should be deleted as well. On the other hand, the behaviour in this case depends on application specific issues, there might also be reasons to keep the reference. However, the recommended design decision in this context is to define a reusable generic attribute in the process control data structure that handles the activity interrupt.

Consequences

- Activities can be interrupted without losing any relevant data.
- The state of the outside world (business objects states) is kept consistent with the state in the process instance.
- Higher security, e.g. in case the *process engine* crashes the system can set-up on a consistent state after restart.
- The process control data structure must contain an attribute to report the activity interrupt to the process instance.
- People can stop working on a task related to an activity and can go on working on the task later, starting from the last change made (imagine the coffee or lunch break).
- The *process engine* must support implementation of this concept, i.e. looping activities with backward data mapping. For instance, IBM *WebSphere MQ Workflow* [WFBT03] allows implementation of this concept.
- The process models are becoming a little more complex, because an additional exit condition must be modelled, together with the backward data mapping.

Related Patterns

From the BUSINESS OBJECT REFERENCE pattern results the idea to only keep references to business objects in the process control data structure. For this reason, this pattern is a conceptual basis of ACTIVITY INTERRUPT. Moreover, application of ACTIVITY INTERRUPT principally influences the design of a process control data structure as an attribute is required to handle the restart of an activity. Consequently, it will influence the design of a generally applicable process control structure. For this reason, GENERIC PROCESS CONTROL STRUCTURE is related to this pattern.

Example

This example illustrates an application with *WebSphere MQ Workflow*. The ACTIVITY INTERRUPT pattern has been implemented using *Buildtime* [WFBT03] process modelling constructs. In order to implement an activity restart, an exit condition has been assigned to an activity. FDL (the process definition language of *WebSphere MQ Workflow*) directly supports the definition of exit conditions for activities. In order to implement the concept of reusing the output data of an activity as input data for the restarted activity, data mapping from the output of an activity to its input container has been defined (data loop connector). The defined generic control structure contains a special reusable attribute called *activityInterrupt* to implement the exit condition.

The next FDL fragment illustrates the implementation using the exit condition and the data loop connector. The FDL contains a simple process with one activity that implements the ACTIVITY INTERRUPT pattern. The important lines are highlighted in bold font.

```
PROCESS 'ActivityInterruptSample' (
   'GenericProcessControlStructure',
   'GenericProcessControlStructure' )
  DO NOT PROMPT_AT_PROCESS_START
  PROGRAM_ACTIVITY 'Activity' (
   'GenericProcessControlStructure',
   'GenericProcessControlStructure' )
    START MANUAL WHEN AT_LEAST_ONE CONNECTOR TRUE
    EXIT AUTOMATIC WHEN "activityInterrupt=""FALSE"""
    PRIORITY DEFINED_IN INPUT_CONTAINER
    PROGRAM 'Foo'
    SYNCHRONIZATION NESTED
  END 'Activity'
  DATA
    FROM SOURCE 1 TO 'Activity'
    MAP '_STRUCT' TO '_STRUCT'
  DATA
    FROM 'Activity' TO SINK 1
    MAP '_STRUCT' TO '_STRUCT'
  DATA
    LOOP 'Activity'
    MAP '_STRUCT' TO '_STRUCT'
END 'ActivityInterruptSample'
```

Process Based Error Management

Context

Errors must be managed that are reported by various components that are integrated in a process flow.

Problem

An activity in a process can be executed by an external component, i.e. an application or a service that is integrated in the process flow. How can errors that are reported by such integrated components in activities in a process flow be handled and managed?

During execution of activities it is possible that errors occur that must be reported to the *process engine*, in order to manage the error. One must keep in mind that we are dealing with a highly distributed environment and the business process is the central control mechanism. Thus, all different kinds of systems and applications can be integrated in the business process. As a result, there can be many experts and teams involved in resolving system errors—each of these people/teams could be responsible for a certain application.

Usually, error management cannot be delegated to a single person or even a single team, due to the potentially large amount of integrated applications and the high distribution of all systems. If one keeps in mind that a business process might move across national and organisational boundaries, it becomes clear that managing occurring errors is a significant issue.

The *Process-Driven Architecture* approach is actually an attempt to provide some standardised layer where anything can connect to that is relevant to doing business. Consequently, a standardised way of managing errors is required. It must be clear that we are not dealing with errors that occur inside the *process engine*, as the *process engine* is *one* architectural component which has its own error handling mechanisms (log-files, exceptions, error queues, etc.), but we are dealing with errors from *various* integrated components. Management of errors that occur inside the *process engine* can be delegated to one team that is responsible for the *process engine* as a component. However, as the business process is the central point where everything is connected, one can conclude that also a process oriented approach will be necessary for managing errors of integrated components.

Solution

Define special attributes for error handling in the process control data structure and embed any activity that integrates external components in an error handling control flow. The integrated component can thus report errors to the process instance and the process instance can react to those errors. This concept allows a process oriented approach to handle errors as managing errors is captured by the process model itself.

If an error occurs, this will be an event that must be handled by the process model. The process must then move into an exception handling mechanism in order to provide

the right team/person with the necessary information to manage the error. Conclusively, there must be an error handling process-activity defined in the process model. Output of this activity will be information, whether the activity where the error occurred must be *repeated*, or the error can be *ignored*, or to *abort* the whole process instance. The process model is again only interested in the result of the error management activity. Consequently, the process model just has to delegate the issue to someone who handles the error and who provides information on how to proceed in the process.

The *process engine* is responsible for assigning the error handling task/activity to the right person or team as it is responsible for distributing *any* activity. Information who to assign the error handling activity must be provided by the component that reports the error, because we can assume that each component has the knowledge about who is responsible for error messages that occur in that component. For this reason, if a component reports an error to the *process engine,* it also has to provide information who is responsible for managing the error. Summing up, one can say that a component that reports an error to the *process engine* must provide the following information, which must be kept in the process control data structure:

- A standardised or normalised error code that indicates the general type or class of error. This error code can be used by the process model to react on an error in a standard way, as this error code represents agreed and unified error code semantics.
- The component specific error code.
- The component specific error message or error description.
- The name/identifier of the system or component that reports the error.
- The actor (person, team, organisational unit) that is responsible for managing the error, i.e. the *administrator* of the error.

It has already been mentioned that output of an error handling activity can generally be three different decisions: *retry*, *ignore*, and *abort*. As a result, attributes are necessary in the process control data structure to report that decision to the process instance. The next picture in *figure 9* shows the overall process design pattern that can be used for managing errors in the process flow. Basically, the pattern is about embedding an activity that integrates external components in an error handling control flow. In practice, the error handling activity is often embedded in a reusable sub-process. This sub-process can then be modelled in the process flow whenever required.

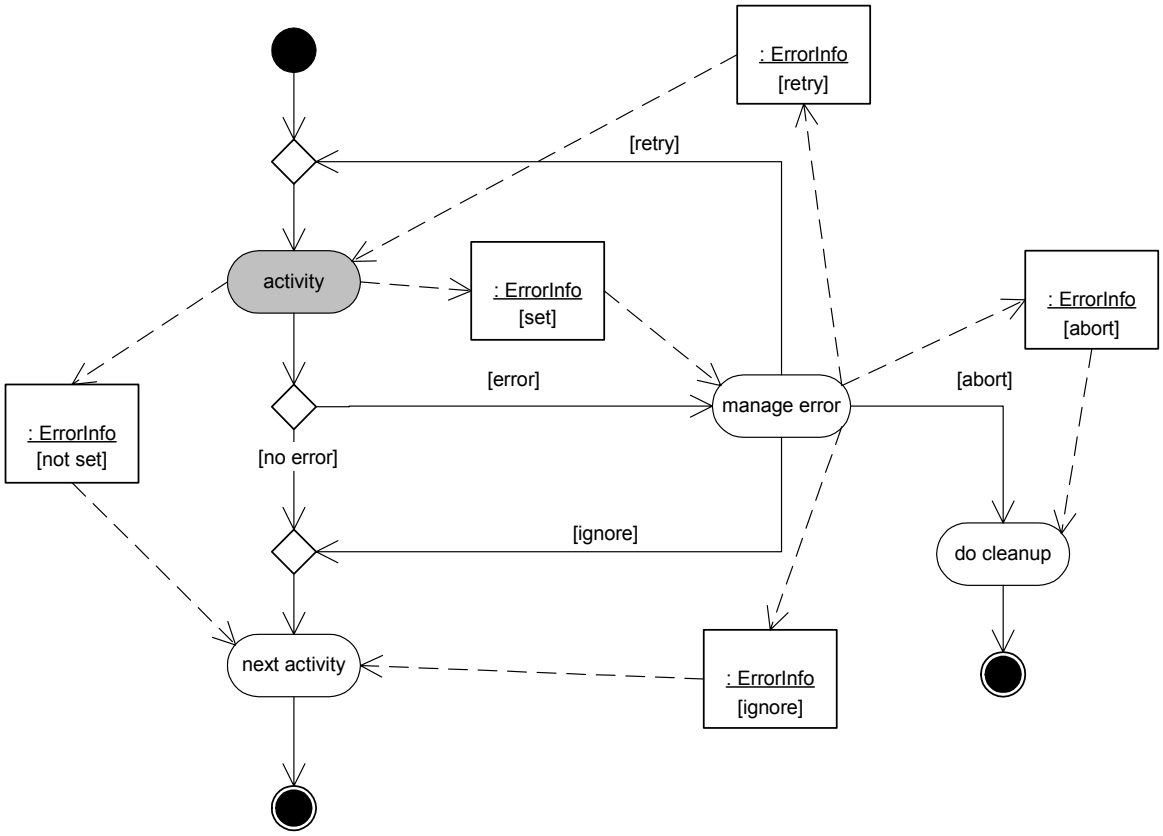

figure 9: process based error management pattern

Consequences

- The pattern provides a standardised way of handling and managing errors of various integrated components.
- The *process engine* must provide mechanisms of *dynamic staff resolution*, because the administrator of the "manage error" activity will be dynamically defined by the error reporting component. For instance, IBM *WebSphere MQ Workflow* allows *dynamic staff resolution*.
- The process control data structure must contain attributes to report the necessary error information.
- For each occurring error, a "manage error" activity instance will be generated. This may result in a large amount of such activity instances, because errors are reported to every process instance. Thus, one error source might report the same error to many process instances repeatedly.
- Additional development effort is necessary to implement the "manage error" activity. It might be necessary to provide sort of bulk operations on the "manage error" activity instances. For example, in order to set the retry, ignore, or abort option for a whole set of "manage error" activity instances. This is very useful in case the same errors are reported repeatedly, as indicated before.

Related Patterns

Application of this pattern principally influences the design of a process control data structure as special attributes are required to report and manage errors. Consequently, it will influence the design of a generally applicable process control structure. For this reason, GENERIC PROCESS CONTROL STRUCTURE is related to this pattern.

Example

WebSphere MQ Workflow offers a mechanism for integrating applications via XML based message adapters. The whole mechanism is encapsulated in a concept called *User Defined Program Execution Server* (UPES). Basis of the UPES concept is the *MQ Workflow* XML messaging interface. *MQ Workflow* does not communicate with applications directly but uses *WebSphere MQ* (MQ Series) [WFPG03].

The UPES concept is all about communicating with external applications via asynchronous XML messages, in order to execute external business logic of an activity in a process automatically. Consequently, the UPES concept is literally about "informing" an application that a process activity requires execution of business logic (which is actually implemented by the application), "receiving" the result after the execution has been completed by the application, and further relate the asynchronously incoming result back to the activity instance that originally requested execution of the business logic (as there may be hundreds or thousands of instances of the same process activity). Thus, a UPES is an XML adapter that represents an interface to one or many applications, related to integrated business logic in process-activities.

If an activity instance sends out an XML request to a UPES implementation, that message will have a defined format. The request is automatically generated by *MQ Workflow*. It contains the data structures and their contents associated to the activity. Furthermore, it is important to mention that the request contains an *activity implementation correlation ID*. Via this correlation ID, the request is associated to an activity instance in the *process engine*. The reply message must contain this correlation ID in order to relate the response back to the corresponding activity instance. Fundamentally, the UPES concept is an application of the ASYNCHRONOUS COMPLETION TOKEN pattern [DS+00].

As far as an implementation of the PROCESS BASED ERROR MANAGEMENT pattern is concerned, the concrete UPES implementation assures that the result message delivered to *MQ Workflow* does contain the necessary error information, such that the process model can react to the error information as defined by the pattern. The actual error information will be accessible in the process control data, as *MQ Workflow* automatically picks up result messages, associates them with an activity instance, and transfers the message data to the process control data of the process instance (the data structures of the result message and the process model must match). For this reason, the basic process fragment modelled in *Buildtime* that handles a reported error looks as depicted in *figure 10*.

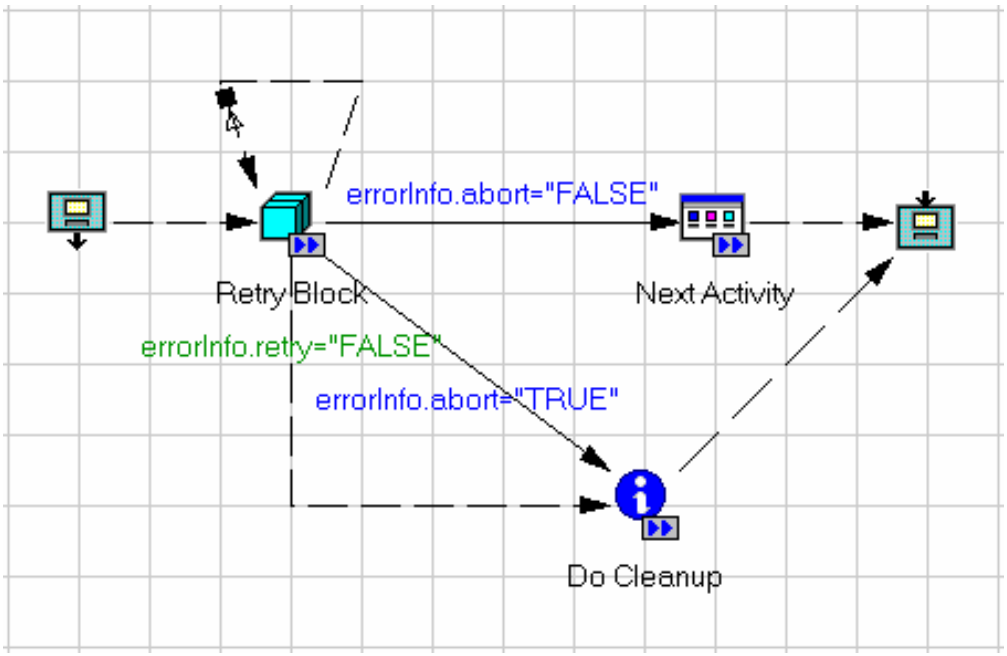

figure 10: implementation of error management

The picture in *figure 10* shows a block activity which contains the actual error reporting activity. The block activity is necessary in order to implement the loop in case the *retry* flag has been set. For this reason, the block activity implements an exit condition that the *retry* flag must not be set (*retry="FALSE"*). The detailed diagram of this block activity is pictured in *figure 11*. However, *figure 10* shows the transition to the subsequent activity in case the process should not be aborted and the transition to the cleanup activity in case the process instance must be terminated.

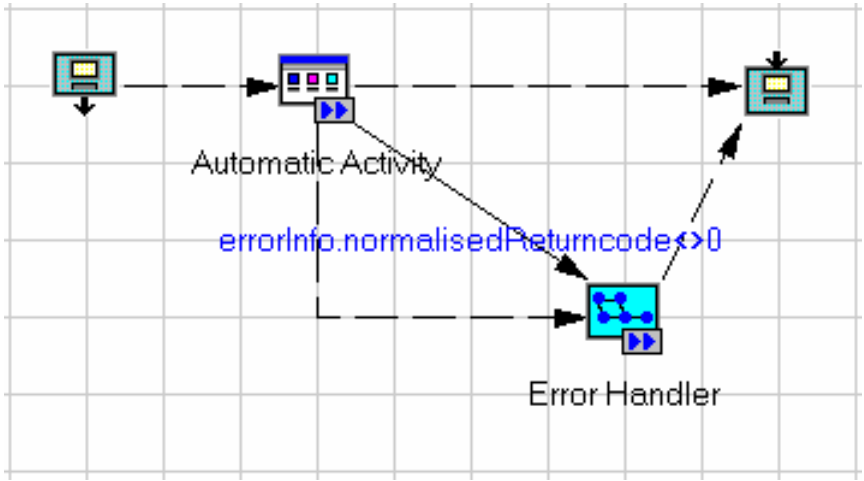

figure 11: detailed view of the block activity

The picture in *figure 11* shows the automatic activity that implements the XML message based communication with an external application via the UPES mechanism. In case the result of this activity indicates an error, a transition to an error handler is made. In case no error is reported, the block activity is terminated normally. One has to consider that the *retry*, *abort*, and *ignore* flags must be reset for every single loop. The *Error Handler* process is

a small process model with only one activity that represents the error management by an administrator.

The error administrator is dynamically associated to a concrete actor, e.g. a person or role, by the value of the attribute *errorInfo.administrator* in the process control data. Within the error managing activity, the error administrator can view all the error information and thus try to fix the problem. Result of the error management activity will be the decision whether to *retry* the activity, *abort* the process, or *ignore* the error. The decision is reported in the corresponding attributes in the process control data structure. The following FDL fragment demonstrates the definition of the *Error Handler* process and highlights the dynamic association of the error administrator according to the value of the corresponding attribute in the process control data structure.

```
PROCESS 'Error Handler' (
   'GenericProcessControlStructure',
   'GenericProcessControlStructure' )
  DO NOT PROMPT_AT_PROCESS_START
  PROGRAM_ACTIVITY 'Manage Error' (
   'GenericProcessControlStructure',
   'GenericProcessControlStructure' )
    START MANUAL WHEN AT_LEAST_ONE CONNECTOR TRUE
    EXIT AUTOMATIC
    PRIORITY DEFINED_IN INPUT_CONTAINER
    DONE_BY PERSON TAKEN_FROM 'errorInfo.administrator'
    PROGRAM 'Foo'
    SYNCHRONIZATION NESTED
  END 'Manage Error'
  DATA
    FROM SOURCE 1 TO 'Manage Error'
  DATA
    FROM 'Manage Error' TO SINK 1
END 'Error Handler'
```

Business Object Reference

Context

The principle of separation of concerns, in terms of separation of business logic from process logic, shall be applied as a concept for process definition.

Problem

If business data is part of the process control data structure the principle of separation of concerns is violated. Moreover, it will not be possible to access the same business data, i.e. business object from different process instances, as the process control data are exclusively accessible by the corresponding process instances the control data belongs to. How can management of business objects be achieved, as far as concurrent access and changes to these business objects is concerned?

The concept of separation of concerns, i.e. separation of business logic from process logic in the worldview of a *Process-Driven Architecture*, implies that business objects are managed outside the *process engine* and will for this reason not be modelled in the process control data structure. Apart from the aspect of separation of concerns, the conflict that can be observed here is that static binding of business objects to process instances is simply not sufficient in this case.

Long running processes require flexibility concerning structural changes to business objects, because business requirements may change over time. As data structures will be bound to a process instance at instantiation time, structural changes to the business objects will not be possible if the business objects are directly modelled in the process control data structure. Moreover, it would imply structural changes to the process control data structure which should actually be avoided, as already illustrated by the GENERIC PROCESS CONTROL STRUCTURE pattern. Apart from that, static modelling of business objects in the process control data would mean that each process instance works with separate objects, which is actually not the case. Rather, there may be several process instances reading and modifying the same business objects concurrently.

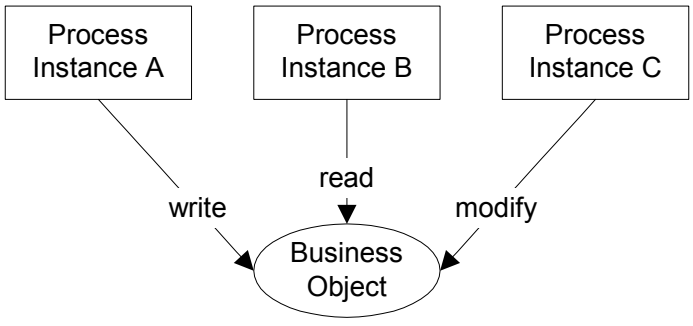

figure 12: different process instances accessing the same business object

Solution

Only store references to business objects in the process control data structure and keep the actual business objects in an external container. Via these references, access to business objects in the container can then be established whenever necessary within activity implementations.

The technical representation of the business process will consider only references to external business objects but not the actual business objects themselves. Those business objects will be stored in a separate container outside the *process engine* and the technical business process will carry references to business objects in the container. As a result, different process instances can modify the same business objects concurrently and they can react to state changes of the business objects, e.g. a process instance wants to access a business object that has been deleted by a concurrent process instance (a cancelled order would be a practical example).

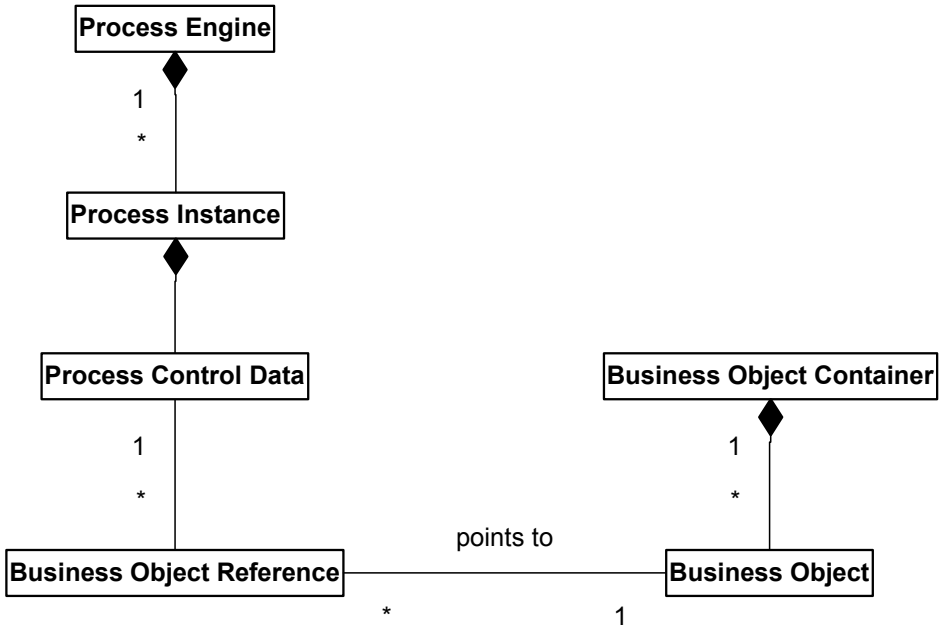

figure 13: business object reference

Consequences

- Flexible management and maintenance of business objects in a *Process-Driven Architecture*.
- Controlled concurrent access to business objects in different process instances.
- Allows changes to business objects without affecting the process control data, as only references are stored in the process control data.
- Business object data will only be loaded from the container when required during process execution in the application.
- Performance increase in the *process engine*, because no large amount of business data is carried for process execution.

- On the other hand, there is also a trade-off concerning performance, because the business object must be retrieved from the container in every activity.
- The process control data structure must keep the references to those external business objects and must contain attributes in order to store these references.
- This solution implies higher development effort compared to modelling the business data in the process control data structure.

Related Patterns

Application of this pattern principally influences the design of a process control data structure as attributes are required to store references to external business objects. Consequently, it will influence the design of a generally applicable process control structure. For this reason, GENERIC PROCESS CONTROL STRUCTURE is related to this pattern. Thus, if the GENERIC PROCESS CONTROL STRUCTURE pattern is applied then generic attributes to store references to business objects must be defined in the process control data structure.

Example

This pattern has been implemented using reusable attributes in a generic process control structure that reference business objects in a database. The chosen DBMS in these projects has been DB/2 or Oracle. All projects have used Java as the programming platform in order to access the business objects from applications. The *metadataID* of each business object reference identifies the type of business object, which basically points to a concrete database where the object is contained in. The *objectID* attribute of a business object reference points to an entity in that database.

Via access classes in a Java framework implementation, a business object can be dynamically determined by a given reference (*metadataID* and *objectID*). The framework implementation searches for the *objectID* in the database identified by the *metadataID* and generates a Java access object that allows to read and write attributes of the business object. Additionally, check-out and check-in operations are provided to enable concurrent access to business objects from various applications.

Event Based Process Instance

Context

A business process shall be implemented where an activity is defined that is event based, and the event implies automatic instantiation of a process on a *process engine*.

Problem

The technical representation of a business process implies that a process instance is created and started on a process engine with a defined initial state after an event occurred. How can a process instance be created automatically, in case an event based activity occurs in a business process that implies automatic process instantiation, e.g. a customer placing an order?

There is a very practical example of this issue. Sometimes there are event based activities modelled in business processes that have a very long duration. For instance, an activity that waits for something to happen, i.e. an event, and then initiates the next process-step. Often it takes weeks and even months until that event occurs. Principally, there is nothing wrong about modelling those event based activities—it is actually the preferred way of modelling.

However, it may cause serious performance problems, because all those process instances exist in the *process engine* but nothing really happens with them, as they are all in a waiting position. This may result in a lot of unnecessary data in the *process engine* (more and more waiting process instances are in the queue) and may thus make the response time of the *process engine* slower and slower. We have to keep in mind that a *process engine* manages all kinds of processes and it is efficient not to generate unnecessary workload in the *process engine* in order to keep the performance of the engine adequate. Moreover, one has to remember that in some projects several thousand process instances are created every day. It would thus be useful if those "temporarily useless" process instances could be removed from the *process engine* and restored later, after the desired event has occurred.

Solution

Externalise event based activities to an external event handler component and split the original business process in several parts at the points where event based activities appear. The event handler stores the relationships between events and processes to be created on a process engine, as well as the initial state of these processes. If an event is fired, the event handler automatically looks up which processes are associated to that event and creates an instance of the processes on a process engine with the defined initial state.

The solution to this issue is externalisation of those problematic event based activities to an external event handler component and to split the process model in several parts. That means, an event based activity is transformed into an activity that externalises the state of the process instance to that external event handler component (actually, this is an application of the MEMENTO design pattern [GoF94]) and the process model is divided at the point where such an event based activity occurs. The event handler component

observes whether the event occurs and activates the next part of the process if the desired event occurs. The state of the previous part of the process is restored when the next part of the process is created. Thus, an original process model including one event based activity will be split into two process models. The first part is instantiated normally and terminates after having registered its state at the event handler. The second part is automatically instantiated by the event handling component right after the event has occurred.

Principally, those two process instances are independent for the *process engine*. The dependency must be managed by that external event handler component. In order to achieve this, it is useful to provide an artificial process-ID in order to link the different instances. That artificial process-ID will be part of the externalised state and will thus be restored in the following part of the process. The process control data structure must contain a *process-ID* attribute for these purposes.

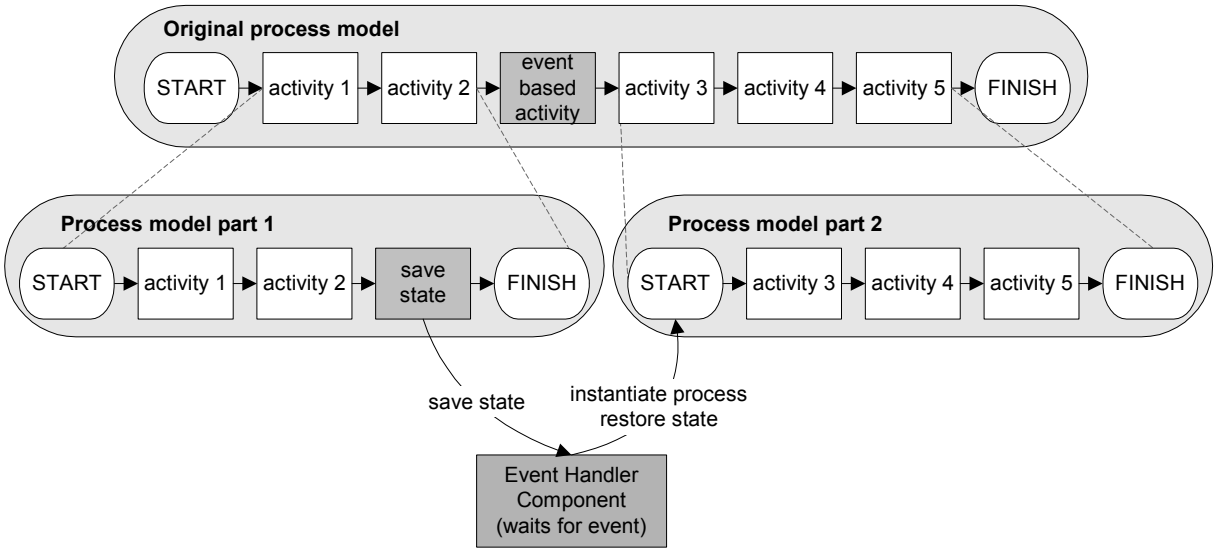

figure 14: process split via external event handler

A client may register at an event handler to listen to certain events in order to instantiate a business process with a defined starting state. The event handler observes events and instantiates the defined business process with the defined starting state if an event occurs that someone registered listens to. As a result, the event handler is connected to several event originators that fire events. Thus, a client could be any application and any system and not only a process-activity implementation.

For this reason, the solution is an event based interface to the *process engine*. The mentioned *starting state* of the business process can be defined as the current contents of the process control data structure, because it will define the internal process data of the business process, i.e. its state. The structure of the design pattern can then be modelled as pictured in the UML diagram below.

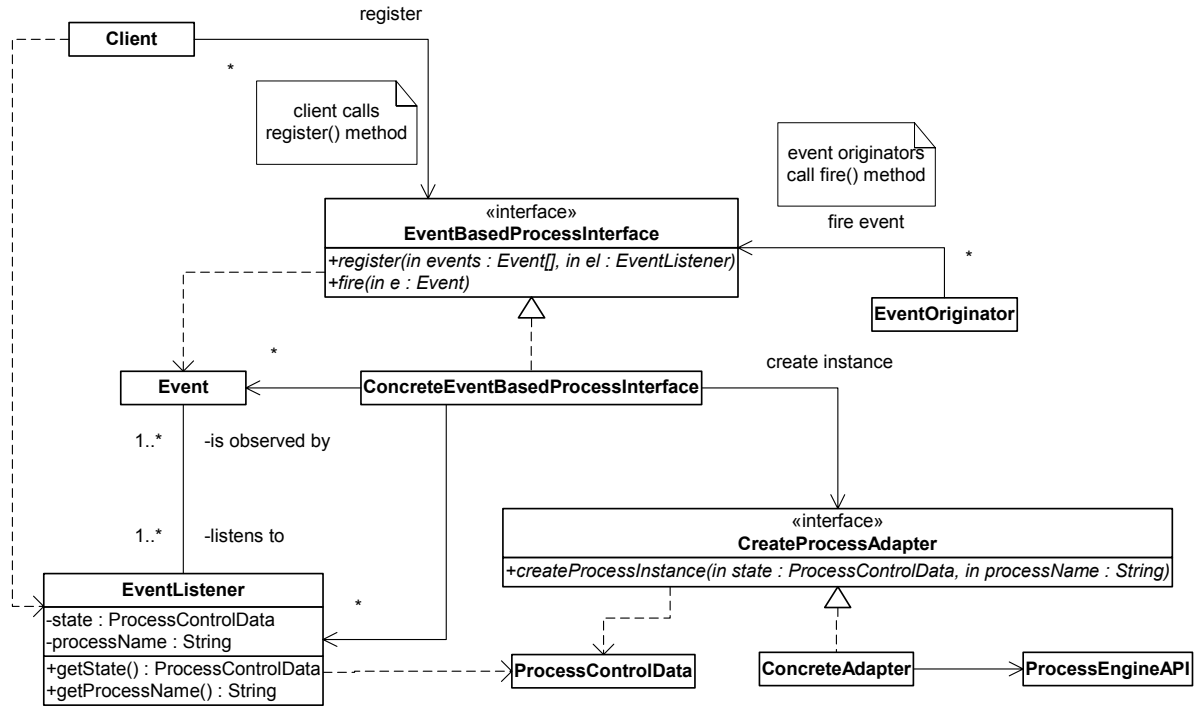

figure 15: event based process instantiation

The diagram in *figure 15* shows that the pattern applies the basic principles of *asynchronous event processing* and *messaging* on the context of process instantiation. Moreover, there is some relationship to the principle of *callback*, as the control is temporarily transferred to an event handler and the specified object to be called is the *CreateProcessAdapter* in case an event occurs.

Consequences

- Automated event based process instantiation can be implemented generically.
- The pattern solves the performance problem related to waiting activities with very long duration. There is a conflict between the demand of a direct implementation of a business process and the technical limitations that might be involved. The design pattern solves this conflict by splitting the original business process into several parts, externalising the event handling, and by loosely coupling the partial processes via an artificial process-ID.
- Flexible event based networking of processes, as a process instance might fire events that cause other processes to be instantiated dynamically on a different *process engine* in a B2B constellation, for instance.
- The process control data structure must contain an attribute to store the artificial process-ID.

Related Patterns

Application of this pattern principally has an impact on the design of a process control data structure as a special attribute is required to store the artificial process-ID.

Consequently, it will influence the design of a generally applicable process control structure. For this reason, GENERIC PROCESS CONTROL STRUCTURE is related to this pattern. Thus, if the GENERIC PROCESS CONTROL STRUCTURE pattern is applied then a reusable process identifier attribute must be defined in the generic process control data structure.

Example

Apart from the application already mentioned in the pattern solution section, which deals with splitting an original process definition in several parts, there are some other types of applications. The very point in this context is that not necessarily a whole cross-boundary business process (end-to-end) is depicted with workflow technology, but only a certain part of the original business process, which might only be related to the internal business of a company or an organisational unit within a company. Thus, it certainly depends what boundaries are defined that shape the relevant aspects to be considered using workflow technology.

For instance, in a Supply Chain Management system this pattern has been applied for automatic process instantiation in a situation where an event based interface to several other legacy applications needed to be implemented. These applications are creating events via MQ messages. The messages are picked up by an event handler, as an incoming message represents an event that needs to be processed. A special input queue has been defined for the event handler.

Moreover, that event handler transforms this application specific input message into an XML message that suits the format of the *WebSphere MQ Workflow* XML interface and sends the XML message to this messaging interface. Depending on the data in the input message, which basically identifies the application firing the event, different processes are automatically instantiated. The event handler has been implemented in Java and runs on *WebSphere* application server. In this application, the event handler manages the relationship to the internal business, whereas those legacy applications are viewed as external components that are only indirectly integrated into the process flow by an event processing mechanism.

Other implementations of this pattern can be found in many document management applications, where a process is automatically instantiated after a document has been scanned. The type of process to be instantiated depends on the type of document that has been scanned. Usually, special scanning software with *Object Character Recognition* (OCR) features is used, such that the software recognises the document type automatically and instantiates the corresponding business process to process the document.

Events that happen in this context are newly scanned documents posted to the event handler. Conclusively, the scanning software plays the role of the event originator. Furthermore, the event handler thus queues the documents that have been scanned and creates the process instances. Often, the process instances then carry references to scanned documents that are stored in a content management system, which is another application of the BUSINESS OBJECT REFERENCE pattern. Applications of this kind are very common in the banking and insurance industry.

Furthermore, there are applications generally related to B2B or B2C constellations, where service interfaces are offered to create certain process instances. Actually, the offered service is a business process. The type of business process to be instantiated depends on the service request. Some implementations already use *WebServices* as the service interface implementation. For instance, one project has been conducted in the German railway industry, where business customers may request certain railway routes from point A to point B, via several stations at a certain date and time. The request is made via a web-based user interface and a corresponding business process is instantiated to construct the route and to report the finished route back to the customer. Thus, in this case placing a service request is an event to automatically instantiate a process.

Generic Process Control Structure

Context

Process models are viewed as reusable components in terms of providing the possibility of invoking any process as a sub-process or directly invoking them from various independent applications.

Problem

When process models are viewed as reusable components in terms of reusable sub-processes, there is a problem in case the data structures of the called processes do change. This problem occurs due to the data dependency between a calling and a called process. That data dependency results from the fact that changes to data structures that actually represent interfaces between calling and called processes will imply data inconsistencies, because the calling process does not know about the change and thus assumes an invalid version of the interface. How can those data inconsistencies be avoided in long running process instances in the context of dynamic sub-process instantiation?

When a process is instantiated, a clone of its process template will be generated and the clone, i.e. the process instance, will be executed on the *process engine*. Analogous to object instantiation in Java, for instance, the process definition will be determined at instantiation time. Thus, the process instance will execute the model that has been valid at instantiation time. Furthermore, a process instance may run for several months, depending on the business process that it implements, as there are business processes that have a duration of several months, e.g. supply chains. This longevity of processes is rather the general rule than the exception. The biggest issue in this context is the changing of data structures in conjunction with sub-processes, because the data structure definitions are statically bound to the process instance at instantiation time, which causes data inconsistencies when the data structures of dynamically instantiated sub-processes are changed.

Consider the following scenario: process A is instantiated with a certain definition of a data structure. This process contains a sub-process B. That means, there will be a certain point in time, when process A dynamically instantiates process B as a sub-process. The process model defines how process A transfers the necessary data to process B by mapping rules, as process B may have different data structures. This mapping of data is defined by the corresponding implementation of the processes. Thus, there is a data dependency of these processes. Hence, we imagine there is a running instance of process A but we assume that process B has not yet been instantiated, i.e. process A has not come to the point where process B is dynamically instantiated. In *figure 16* this situation is indicated by event 1, or rather the timeframe between event 1 and event 3.

Furthermore, we assume that a new requirement has been implemented in a new version of process B. This new version modifies the data structures of process B, for example, by changing the data type of an attribute in the structure, and is deployed after process A has been instantiated. In *figure 16* this situation is indicated by event 2. We have to keep in mind that the data mapping rules for process A instantiating process B have

been statically bound at instantiation time of process A. If our running instance of process A now comes to the point where process B is dynamically instantiated as a sub-process, it will instantiate the new version of B by applying the old data mapping rules. The result is data inconsistency.

In *figure 16* the described situation is indicated by event 3. By this example, it becomes clear, how process control data structures represent interfaces between processes and what impact changes to these interfaces can have.

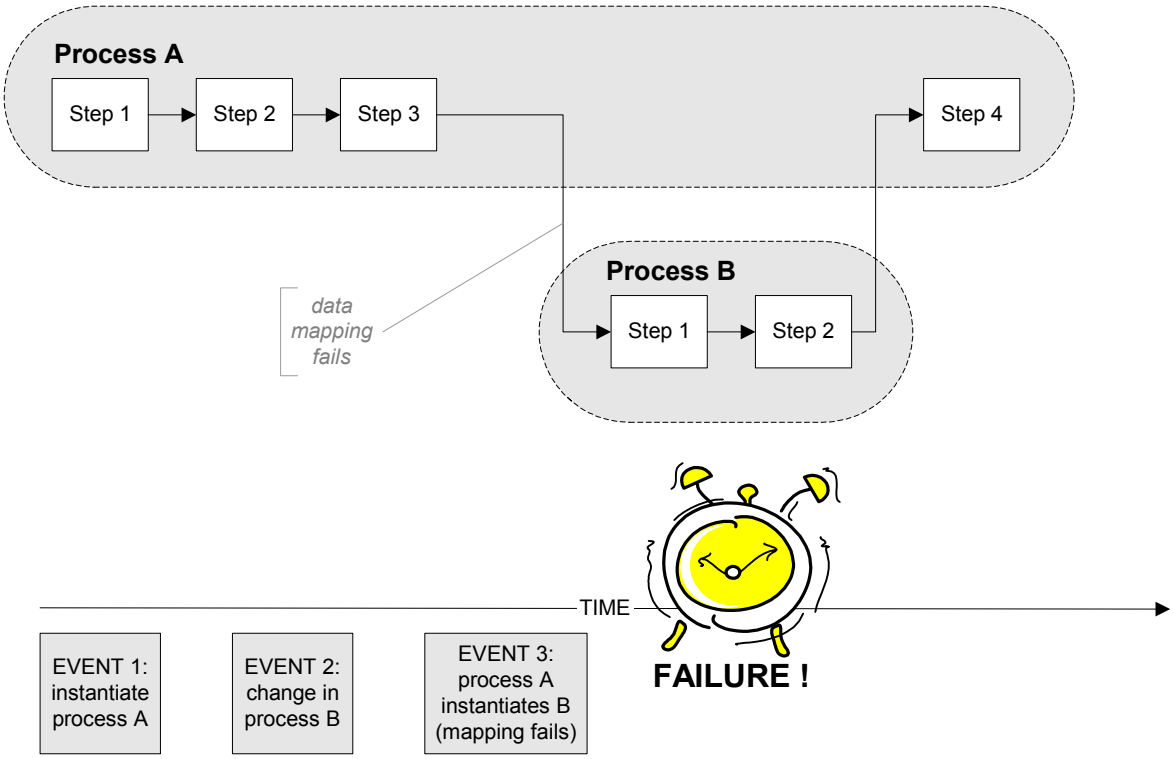

figure 16: data inconsistency during process execution

Unfortunately, even version control may not really provide a suitable approach to a solution, because it is actually not possible to version the single data structure interfaces without versioning the process models that use them, as the data structures are statically bound to the process models. In order to introduce a new version of a data structure it will thus be necessary to create a new version of all processes using this data structure as an interface when calling another process.

For instance, in the example in *figure 16* it would be necessary to create a new version of processes A and B in order to introduce a change in the data structure interface between them. That means, the data dependencies between processes must be managed somehow, because it is necessary to identify the effect that a potential change to a data structure will have and to find out what processes are affected.

Additionally, this change will only take effect with a new instance of process A—all the running instances of processes A and B still have to apply the old version. It may thus take a long time until the change really has an effect, namely at the point in time when all old instances of process A have terminated and only new versions of process A and B are instantiated. The effects that many cascading changes will have over a longer period of

time result in a large amount of different versions of the same processes running in parallel and all these different versions must be handled by the applications using them. Usually, this results in unacceptable maintenance effort and complexity of the system. It might be a solution in case the processes are only running over a very short period of time, e.g. a few seconds, but in businesses processes that run for several weeks and months a nearly unmanageable complexity might be generated.

Solution

Use a generic process control data structure that is only subject to semantic change but not structural change. That means, a generic interface in terms of a generic process control structure is defined between processes that standardises all necessary aspects for controlling a process. Attributes in this generic structure can be used for different purposes and thus with varying, i.e. generic semantics. The concrete semantics of attributes are defined by providing additional attributes that contain meta-information.

If a generic and reusable process data structure is used for all processes it is actually possible to avoid those structural changes on data structures. This generic data structure has to depict a meta-model for process control data structures, i.e. it has to include all concerns for process control. This solution introduces the concept of designing a meta-structure that captures the requirements of many concrete process control data structures by applying the principle of semantic abstraction. The key aspect here is that a structural change to data is crucial but a semantic change is not. This is a practical example that there may be situations where a structural change to data can have unacceptable effects. Consequently, a structural change must be avoided.

The conflict that can be observed here results from the design comfort of flexible definitions of data structures on the one hand and the negative effects that flexible definitions of data structures do actually have on the other hand. This conflict is resolved by the concept of a meta-structure that abstracts the semantics of concrete data structures and which thus allows semantic flexibility but not structural flexibility. The next UML diagram in *figure 17* illustrates the generic relationships. Consequently, it will not be necessary to change the control structure of a process as the generic structure depicts all necessary general aspects for controlling processes.

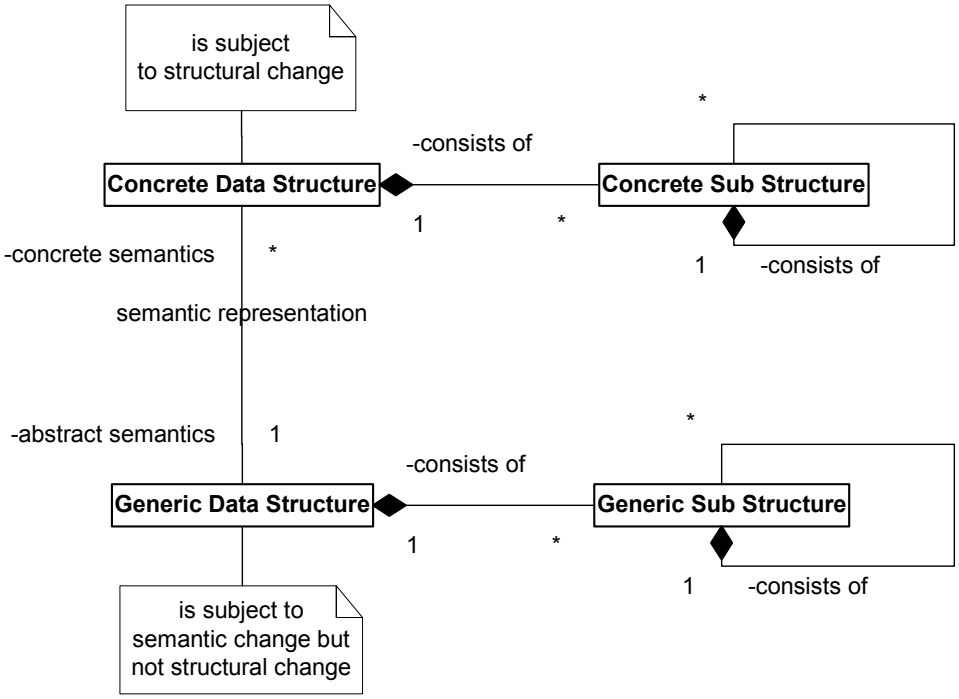

figure 17: the principle of a generic process control structure

In order to define such a generic process control structure, it is necessary to gather the general requirements for controlling a process. If the BUSINESS OBJECT REFERENCE pattern is applied this is even easier, as no business data must be included in the process control data structure. Thus, one can exclusively focus on aspects that are necessary for controlling the process flow.

Consequences

- Data mapping problems are solved, because no structural changes will occur.
- Process modelling is much simpler and less error prone, because no difficult mapping rules must be applied.
- Process models can be easily reused, because interfaces between these processes are structurally standardised.
- Applications have to take care of interpreting the semantically flexibly used data structures.
- There might be quite some design effort necessary to develop a standardised generic data structure that suits all purposes of the enterprise and that captures all necessary meta-information. However, it is actually possible to achieve this, because the general concerns of process control can usually be clearly classified and defined if the principle of separation of concerns is followed, which means that no application specific business data is contained in the process control data. Moreover, the principle of encoding generic semantics in the data structure provides a tool for designing the process control structure according to generic requirements.

- The semantic flexibility usually implies additional development effort, because it must dynamically be decided, based on the meta-information, how to interpret the contents of the structure. Thus, complexity is increased at this level.
- There is also additional effort necessary to handle semantic errors. Semantic errors will usually be detected by inconsistencies between the meta-information and the contents. These error detection and handling algorithms must be implemented by the application or a general framework.
- It is recommended to define a format for the meta-information, e.g. an XML schema. However, the specification of this data format represents an additional development effort and the format might also be subject to change.
- Type safety is becoming more complex, because type safety must be ensured by programming the consistency check, based on a self defined meta-model.

Related Patterns

In case the patterns ACTIVITY INTERRUPT, PROCESS INTERRUPT TRANSITION, BUSINESS OBJECT REFERENCE, PROCESS BASED ERROR MANAGEMENT, and EVENT BASED PROCESS INSTANCE are applied, this will have implications on the design of a generic process control data structure. Thus, applying these patterns will consequently imply the definition of generic attributes that depict the concerns of these patterns in a generic process control structure.

- If the ACTIVITY INTERRUPT pattern is applied, a generic attribute to handle activity interrupts must be defined in the generic process control data structure.
- If the PROCESS INTERRUPT TRANSITION pattern is applied, a generic attribute to handle process interrupt transitions must be defined in the generic process control data structure.
- If the BUSINESS OBJECT REFERENCE pattern is applied, generic attributes to store references to business objects must be defined in the generic process control data structure.
- If the PROCESS BASED ERROR MANAGEMENT pattern is applied, generic attributes to handle error messages must be defined in the generic process control data structure.
- If the EVENT BASED PROCESS INSTANCE pattern is applied, a reusable process identifier attribute must be defined in the generic process control data structure.

Example

The following list of requirements represents an example how a requirements specification for a generic process control data structure could look like. It will further be demonstrated how these requirements can be transformed into a design for a generic process control data structure and how this design can be implemented using *WebSphere MQ Workflow*. Furthermore, the following requirements specification considers all mentioned related patterns:

- Multiple references to external business objects that are subject of the business process are necessary. This requirement results from the BUSINESS OBJECT REFERENCE pattern.
- Staff information must be defined in order to dynamically assign who/what is the actor of which activity (it could be a person, an organisational unit , or a role, etc.).
- It must be possible to set actual process control data to handle conditional transitions between activities.
- A set of flexibly usable filter and sorting attributes is required, in order to define dynamic filter and sorting constraints on user work-lists based on different application data.
- Error management, e.g. in case an integrated business service reports an error. This requirement results from the PROCESS BASED ERROR MANAGEMENT pattern.
- It must be possible to abort a process in case the process must not finish normally due to external circumstances. This requirement results from the PROCESS INTERRUPT TRANSITION pattern.
- Controlled activity interrupt must be possible in case the data of an activity is manipulated but the process must not yet continue to the next step. This requirement results from the ACTIVITY INTERRUPT pattern.
- Text messages shall be used as a tool for interpersonal communication between activities.
- An application specific business process ID is needed in order to implement relationships between principally independent sub-processes. This requirement results from the EVENT BASED PROCESS INSTANCE pattern.
- A set of *multi-purpose string objects* are required that can be flexibly used. Sometimes it is necessary to have some kind of container for application specific process information. For this reason, it is necessary to have such a container in the data structure. As no complex business data should be transported in the process, string objects are absolutely sufficient.
- It is necessary to store meta information about the data structure contents in order to provide some variability on the semantics of the data structure. One could imagine several versions that unambiguously define different semantics of the *multi-purpose objects*, for instance. The meta information thus indicates how to interpret the contents of the flexible parts of the data structure.
- A set of numeric and string helper attributes are required that temporarily store data for later reuse, e.g. for data mapping purposes.
- An attribute is required for setting expirations of activities dynamically. Some *process engines* have a feature that an activity may expire, in case the activity has not been processed in a defined time. This expiration time can be defined statically in the process model or dynamically at runtime. For this reason, an attribute is required to set the expiration time dynamically, if required.

The resulting generic process control structure that depicts these requirements could look as defined in *figure 18*. This structure can then be implemented on the *process engine* and can be used in every process. As a result, the interfaces between processes are structurally standardised.

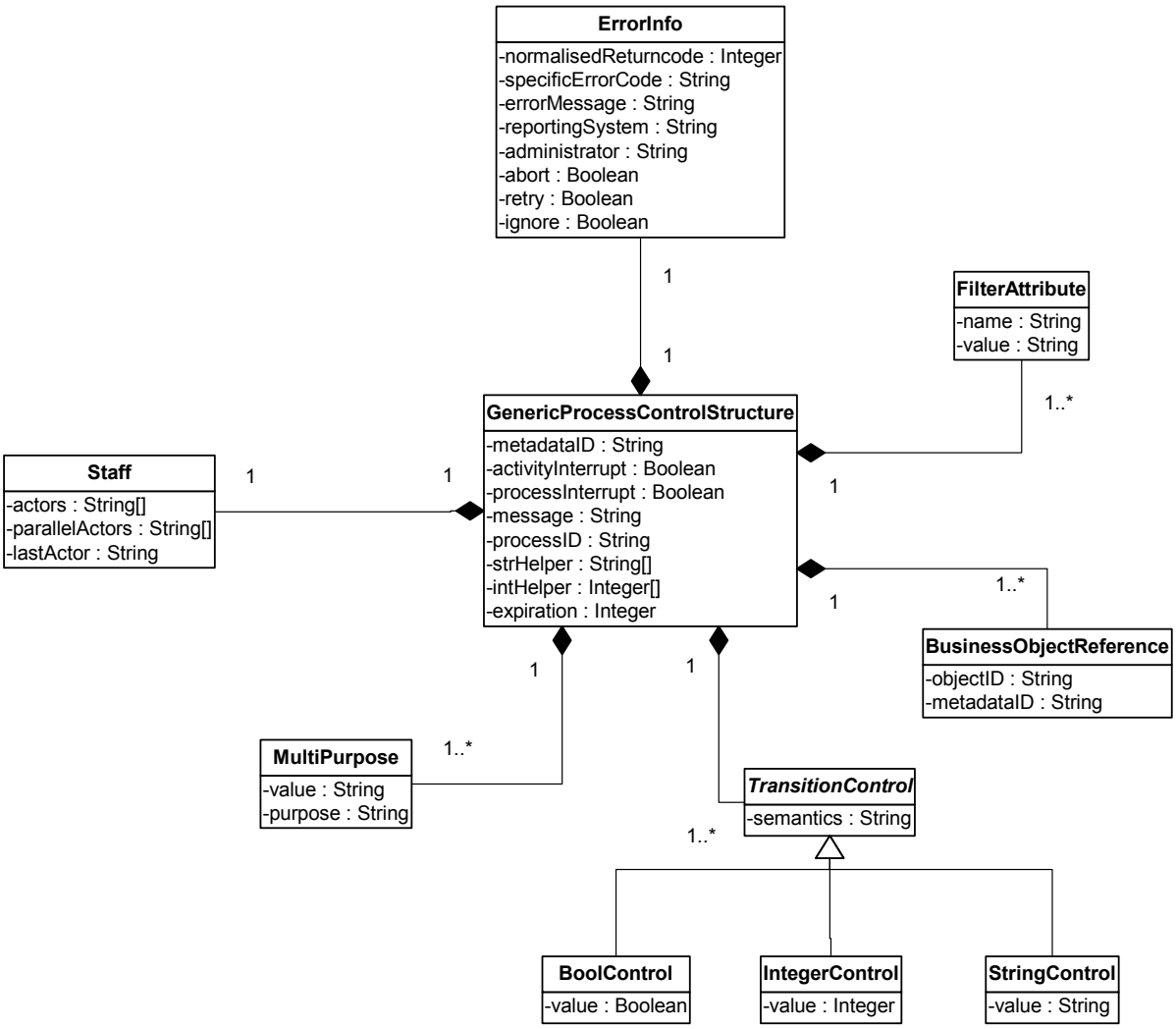

figure 18: example of a generic process control structure

The model in *figure 18* illustrates how the semantics of the control data can be variably defined. For instance, the abstract class *TransitionControl* defines the attribute "semantics" to set the semantics of a control object dynamically. Thus, a *BoolControl* object might contain the semantics "credit limit > 50000 Euro". In another process model the same object might be associated to different semantics. The corresponding Boolean attribute "value" will indicate whether the condition is true or false.

Another example is attribute *metadataID* from class *GenericProcessControlStructure*. Via this attribute meta-information about the concrete semantics of the structure can be obtained. For instance, the actual meta-information could be stored in a container outside the *process engine* as an XML schema, and the *metadataID* just points to that metadata object. In this case, it would be an application of the BUSINESS OBJECT REFERENCE pattern. However, the meta-information provides a unique interpretation of the contents of the structure for the applications using it. The other elements of the presented generic process control data structure also follow this principle of semantic abstraction.

This example demonstrates how a generic process control data structure can encode generic semantics and thus define an interface between processes that does not need to change as far as its structure is concerned, because it is possible to standardise the

necessary general structural aspects for controlling processes by this interface. As a result, interface dependencies between processes are reduced to a minimum level of complexity and changes to the control structure are manageable, as those changes will only reference varying semantic interpretations of certain attributes within a process as a component, and will not populate among different processes (loose coupling).

The following code is a *WebSphere MQ Workflow* implementation of the example of the GENERIC PROCESS CONTROL STRUCTURE pattern in *figure 18*. The code is presented in *WebSphere MQ Workflow's* process definition language called FDL (Flowmark Definition Language). Multiplicities of the originally presented model in *figure 18* have been restricted to fixed values, as FDL only supports fixed sized arrays. Moreover, FDL does not support the *Boolean* data type. For this reason, *Boolean* attributes have also been depicted as strings, defining a business rule that only string values of "TRUE" and "FALSE" are allowed to represent the *Boolean* values. Moreover, the type *Integer* is not supported as well but only type *Long*. As a result, attributes of type *Integer* are depicted as *Long*.

```
STRUCTURE 'ErrorInfo'
  'normalisedReturncode': LONG;
  'errorMessage': STRING
    DESCRIPTION "";
  'reportingSystem': STRING
    DESCRIPTION "";
  'administrator': STRING
    DESCRIPTION "";
  'abort': STRING
    DESCRIPTION "";
  'retry': STRING
    DESCRIPTION "";
  'ignore': STRING
    DESCRIPTION "";
END 'ErrorInfo'

STRUCTURE 'FilterAttribute'
  'name': STRING;
  'val': STRING
    DESCRIPTION "";
END 'FilterAttribute'

STRUCTURE 'BusinessObjectReference'
  'metadataID': STRING;
  'objectID': STRING;
END 'BusinessObjectReference'

STRUCTURE 'Staff'
  'actors': STRING(20);
  'parallelActors': STRING(30);
  'lastActor': STRING;
END 'Staff'

STRUCTURE 'MultiPurpose'
  'val': STRING;
  'purpose': STRING;
END 'MultiPurpose'
```

```
  STRUCTURE 'StringControl'
    'semantics': STRING;
    'val': STRING;
  END 'StringControl'
  STRUCTURE 'IntegerControl'
    'semantics': STRING;
    'val': LONG;
  END 'IntegerControl'

  STRUCTURE 'BoolControl'
    'semantics': STRING;
    'val': STRING;
  END 'BoolControl'

  STRUCTURE 'GenericProcessStructure'
    'metadataID': STRING
      DESCRIPTION "";
    'activityInterrupt': STRING
      DESCRIPTION "";
    'processInterrupt': STRING
      DESCRIPTION "";
    'message': STRING
      DESCRIPTION "";
    'processID': STRING
      DESCRIPTION "";
    'strHelper': STRING(5)
      DESCRIPTION "";
    'intHelper': LONG(5)
      DESCRIPTION "";
    'expiration': LONG
      DESCRIPTION "";
    'errorInfo': 'ErrorInfo'
      DESCRIPTION "";
    'filterAttributes': 'FilterAttribute'(6)
      DESCRIPTION "";
    'objects': 'BusinessObjectReference'(30)
      DESCRIPTION "";
    'staff': 'Staff'
      DESCRIPTION "";
    'multiPurpose': 'MultiPurpose'(30)
      DESCRIPTION "";
    'boolCtrl': 'BoolControl'(30)
      DESCRIPTION "";
    'strCtrl': 'StringControl'(30)
      DESCRIPTION "";
    'intCtrl': 'IntegerControl'(30)
      DESCRIPTION "";
END 'GenericProcessStructure'
```

The data structure has originally been modelled with the *Buildtime* [WFBT03] process modelling tool of *WebSphere MQ Workflow*. Once defined with *Buildtime,* the data structure can be associated to any process model defined in *Buildtime*. Exported process models will then contain the data structure. The contents of the data structure can then be passed from one activity to the next. The next FDL example illustrates how this is done. The sample shows a simple process definition with two activities. The generic process control

structure is associated to the process as input and output data structure as well as to the two activities. Rows that represent this assignment of the data structure are in bold font.

```
PROCESS 'GenericControlStructureSample'(
   'GenericProcessControlStructure',
   'GenericProcessControlStructure' )
  DO NOT PROMPT_AT_PROCESS_START
  PROGRAM_ACTIVITY 'Activity1' (
   'GenericProcessControlStructure',
   'GenericProcessControlStructure' )
    START MANUAL WHEN AT_LEAST_ONE CONNECTOR TRUE
    EXIT AUTOMATIC
    PRIORITY DEFINED_IN INPUT_CONTAINER
    PROGRAM 'Foo'
    SYNCHRONIZATION NESTED
  END 'Activity1'
  PROGRAM_ACTIVITY 'Activity2' (
   'GenericProcessControlStructure',
   'GenericProcessControlStructure' )
    START MANUAL WHEN AT_LEAST_ONE CONNECTOR TRUE
    EXIT AUTOMATIC
    PRIORITY DEFINED_IN INPUT_CONTAINER
    PROGRAM 'Foo'
    SYNCHRONIZATION NESTED
  END 'Activity2'
  CONTROL
    FROM 'Activity1' TO 'Activity2'
  DATA
    FROM 'Activity1' TO 'Activity2'
    MAP '_STRUCT' TO '_STRUCT'
  DATA
    FROM 'Activity2' TO SINK 1
    MAP '_STRUCT' TO '_STRUCT'
  DATA
    FROM SOURCE 1 TO 'Activity1'
    MAP '_STRUCT' TO '_STRUCT'
END 'GenericControlStructureSample'
```

Known Uses

The patterns described in this paper have been used in IBM development projects in various industries such as telecommunications, insurance, railway, and banking all around Europe, based on *WebSphere MQ Workflow* [WFAC03]. For instance, IBM has developed a large scale Supply Chain Management solution for one of the largest automobile companies in Germany where all patterns described in this paper have been applied. The system has gone productive in 2002.

WebSphere MQ Workflow implements a *process engine* that has originally been designed according to the basic standards of the WfMC [WfMC95]. Fundamentally, it consists of a *Buildtime* component [WFBT03] to model the process flows, and a *Runtime* component [WFRT03], where the defined processes are instantiated and executed. Moreover, it offers APIs in several programming languages, e.g. Java and C++ [WFPG03]. Cross-platform application integration is supported via XML messaging interfaces. The process models are exported from the *Buildtime* component in a process definition language format called *Flowmark Definition Language* (FDL) [WFBT03]. This process definition language is derived from the *Workflow Process Definition Language* (WPDL) standard [WfMC99], which has been developed by the WfMC. Exported process definitions can then be imported into the *Runtime* component. FDL language thus functions as the process definition interchange format between the *Buildtime* and *Runtime* components.

References

[AKVC03] "Patterns for e-Business – A Strategy for Reuse" – J. Adams, S. Koushik, G. Vasuveda, G. Calambos, IBM Press, 2003 4th ed.

[DS+00] "Pattern-Oriented Software Architecture. Volume 2: Patterns for Concurrent and Networked Objects" – D. Schmidt et al., John Wiley & Sons 2000

[FB+96] "Pattern-Oriented Software Architecture" – F. Buschmann et al, John Wiley & Sons 1996

[FLDR00] "Production Workflow, Concepts and Techniques" – Frank Leymann, Dieter Roller, Prentice Hall 2000

[GoF94] "Design Patterns – Elements of Reusable Object Oriented Software" – E. Gamma, R. Helm, R. Johnson, J.Vlissides, Addison Wesley 1994

[Prior03] "Workflow and Process Management" – Carol Prior, Maestro BPE Pty Limited, Australia 2003

[WFAC03] "WebSphere MQ Workflow 3.4 – Concepts and Architecture" – IBM corporation 2003

[WFBT03] "WebSphere MQ Workflow 3.4 – Getting Started with Buildtime" – IBM corporation 2003

[WFMC00] "Workflow Management Coalition – Workflow Standard Interoperability Wf-XML Binding" – WfMC May 2000

[WfMC95] "The Workflow Reference Model" – WfMC 1995

[WfMC96] "Workflow Client Application (Interface 2) Application Programming Interface (WAPI) Specification" – WfMC 1996

[WfMC99] "Interface 1: Process Definition Interchange Process Model" – WfMC 1999

[WFPG03] "WebSphere MQ Workflow 3.4 – Programming Guide" – IBM corporation 2003

[WFRT03] "WebSphere MQ Workflow 3.4 – Getting Started with Runtime" – IBM corporation 2003

References to Websites

[FileBPM] http://www.filenet.com/English/Products/Business_Process_Manager/

[FuFlow] http://www.fapl.fujitsu.com/services/software_wkflow_iflow_01.html

[MQWF] http://www-306.ibm.com/software/integration/wmqwf/

[OrFlow] http://otn.oracle.com/products/ias/workflow/release261/workflow_ds.html

[StaffPS] http://www.staffware.com/products/