Patterns for Process-Oriented Integration in Service-Oriented Architectures

Carsten Hentrich

CSC Deutschland Solutions GmbH
Abraham-Lincoln-Park 1
65189 Wiesbaden, Germany
e-Mail: chentrich@csc.com

Uwe Zdun

Distributed Systems Group
Information Systems Institute
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria
e-Mail: zdun@acm.org

Service-oriented architectures are increasingly used in the context of business processes, but the proven practices for the integration of services and processes are not well explained so far. For the integration of services and processes many different concerns, ranging from technical to architectural to business-related concerns, must be considered, and combinations of these concerns that are well working in a given architecture are not obvious. In this paper we tackle this problem by presenting patterns for process-oriented integration of services in a service-oriented architecture. The patterns follow a general conceptual distinction into a macroflow and a microflow level, distinguishing long-running, business processes from short-running, technical processes. The architectural pattern PROCESS-BASED INTEGRATION ARCHITECTURE *guides the design of a generic architecture based on this distinction, and it is refined by a number of design patterns which depict individual design decisions within the scope of this architectural pattern.*

Introduction

Service-oriented architectures (SOA) are an architectural concept in which all functions, or services, are defined using a description language and have invokable, platform-independent interfaces that are called to perform business processes [Channabasavaiah 2003 et al., Barry 2003]. Each service is the endpoint of a connection, which can be used to access the service, and each interaction is independent of each and every other interaction. Communication among services can involve simple invocations and data passing, or complex activities of two or more services.

Though built on similar principles, SOA is not the same as Web services, which is a collection of technologies, such as SOAP and XML. SOA is more than a set of technologies and runs independent of any specific technologies.

A SOA is typically organized as a layered architecture (see Figure 1), both on client and server side [Zdun et al. 2006]. At the lowest layer, low-level communication issues are handled. On top of this layer, a Remoting layer is responsible for all aspects of sending and receiving of remote

service invocations, including request creation, request transport, marshalling, request adaptation, request invocation, etc. Above this layer comes a layer of service clients on the client side and a layer of service providers on server side. The top-level layer is the Service Composition Layer at which the service clients and providers from the layer beneath are used to implement higher-level tasks, such as service orchestration, coordination, federation, and business processes based on services.

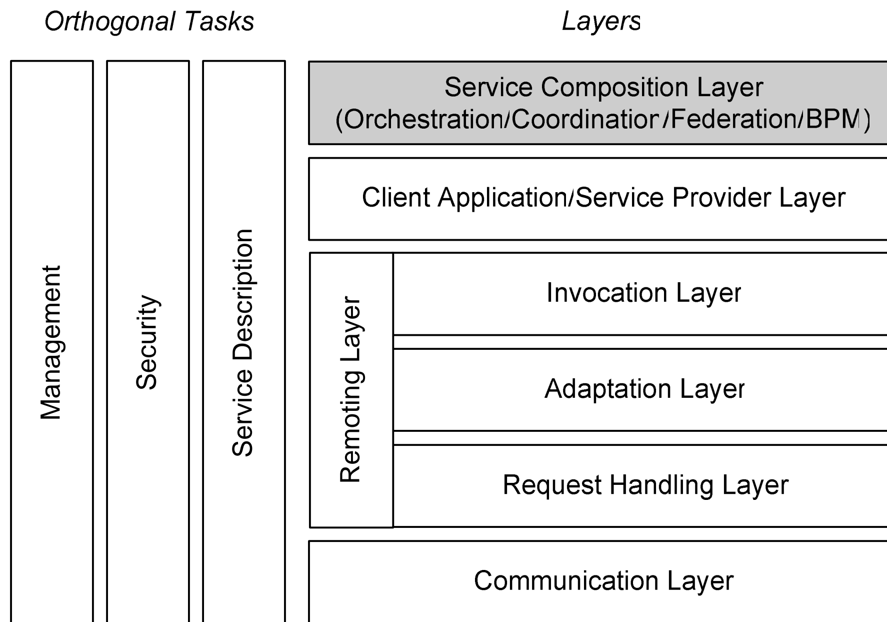


Figure 1: SOA Layers

In this paper we view the SOA concept from the perspective of a Service Composition Layer that is process-driven. That is, the Service Composition Layer introduces a process engine (or workflow engine) which invokes the SOA services to realize individual activities in the process (aka process steps, tasks in the process). The goal of decoupling processes and individual process activities, realized as services, is to introduce a higher level of flexibility into the SOA: Pre-defined services can flexibly be assembled in a process design tool. The technical processes should reflect and perhaps optimize the business processes of the organization. Thus the flexible assembly of services in processes enables developers to cope with required changes to the organizational processes, while still maintaining a stable overall architecture.

In a process-driven SOA the services describe the operations that can be performed in the system. The process flow orchestrates the services via different activities. The operations executed by activities in a process flow thus correspond to service invocations. The process flow is executed by the process engine. In SOAs different communication protocols and paradigms, such as synchronous RPC, asynchronous RPC, messaging, publish/subscribe, etc. can be used and are supported by SOA technologies, like Web Service frameworks or Enterprise Service Bus implementations. For a process-driven SOA, it can generally be assumed, however, that mainly asynchronous communication protocols and paradigms are used. This is because it cannot generally be assumed that a business process blocks until a service invocation returns. In most cases, in the meantime other sensible activities can be performed by the process. In addition, there are many places in a process-driven SOA where invocations must be queued (e.g. legacy systems that run in batch mode). It is typically not tolerable that central architectural components of the process-driven SOA, such as a central dispatcher, block until an invocation returns. Hence, synchronous service invocations are only used in exceptional cases, where they make sense.

In a process-aware system we distinguish two types of data: Process control data and the business objects that are transformed via the process flow. An example of such a business object is a customer order that is being processed via a process flow. The actual processing of that order is controlled by process control data that depicts the routing rules, for instance. Each process activity can be interpreted as a certain state of the business object.

With respect to the process flow, we can generally distinguish two kinds of processes: *long-running, higher-level business processes* and *short running, more technical processes*. The distinction between those two kinds of processes is an important conceptual decision that helps us to design process activities at the right level of granularity. In addition, the technical properties of the two kinds of processes are different. For instance, for long-running processes it is typically not appropriate to use ACID transactions because it is infeasible to lock resources for the duration of the whole process, while this might be perfectly feasible for more short running processes of only a few service invocations. In the remainder of this paper we refer to the long-running, higher-level business process using the term *macroflow*. We use the term *microflow* to refer to the short running, more technical processes.

In order to create the link between an activity of a process and a service, *process integration logic* is required. The process integration logic is typically realized using microflows.

Pattern Language Overview

The pattern language presented in this paper basically addresses the conceptual issues in the Service Composition Layer, when following a process-driven approach to services composition. Following the business process paradigm the Service Composition Layer is structured into four sub-layers as presented in Figure 2. This refined structure will be reflected in the PROCESS-BASED INTEGRATION ARCHITECTURE architectural pattern, and moreover, the more detailed patterns that represent building blocks of the architecture will be described as design patterns in the context of this architectural pattern.

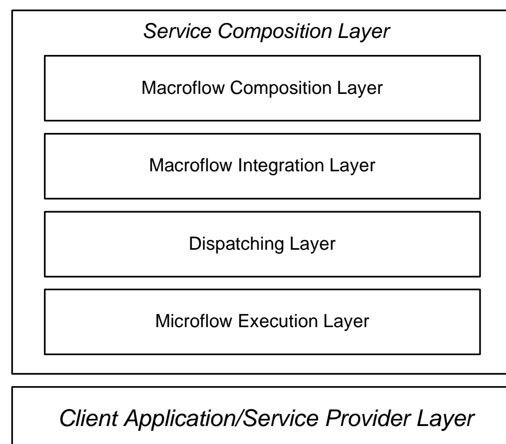


Figure 2: Process-driven refinement of the Service Composition Layer

The patterns and pattern relationships for designing a Service Composition Layer are shown in Figure 3. The pattern MACRO-MICROFLOW sets the scene and lays out the conceptual basis to the overall architecture. The PROCESS-BASED INTEGRATION ARCHITECTURE pattern describes how to design an architecture based on four sub-layers for the Service Composition Layer, which is following the MACRO-MICROFLOW conceptual pattern.

The remaining patterns in the pattern language provide detailed guidelines for the design of a PROCESS-BASED INTEGRATION ARCHITECTURE. In Figure 3 these patterns are thus displayed within the boundaries of the PROCESS-BASED INTEGRATION ARCHITECTURE pattern. Three of the patterns do not (only) refer to the Service Composition Layer: The CONFIGURABLE ADAPTER and BUSINESS-DRIVEN SERVICE patterns explain how to design the interfaces to the Client Application/Service Provider Layer. For this reason, these two patterns refer to both the Service Composition Layer and the Client Application/Service Provider Layer. As a consequence, the CONFIGURABLE ADAPTER REPOSITORY which manages CONFIGURABLE ADAPTERS must also be considered at both layers.

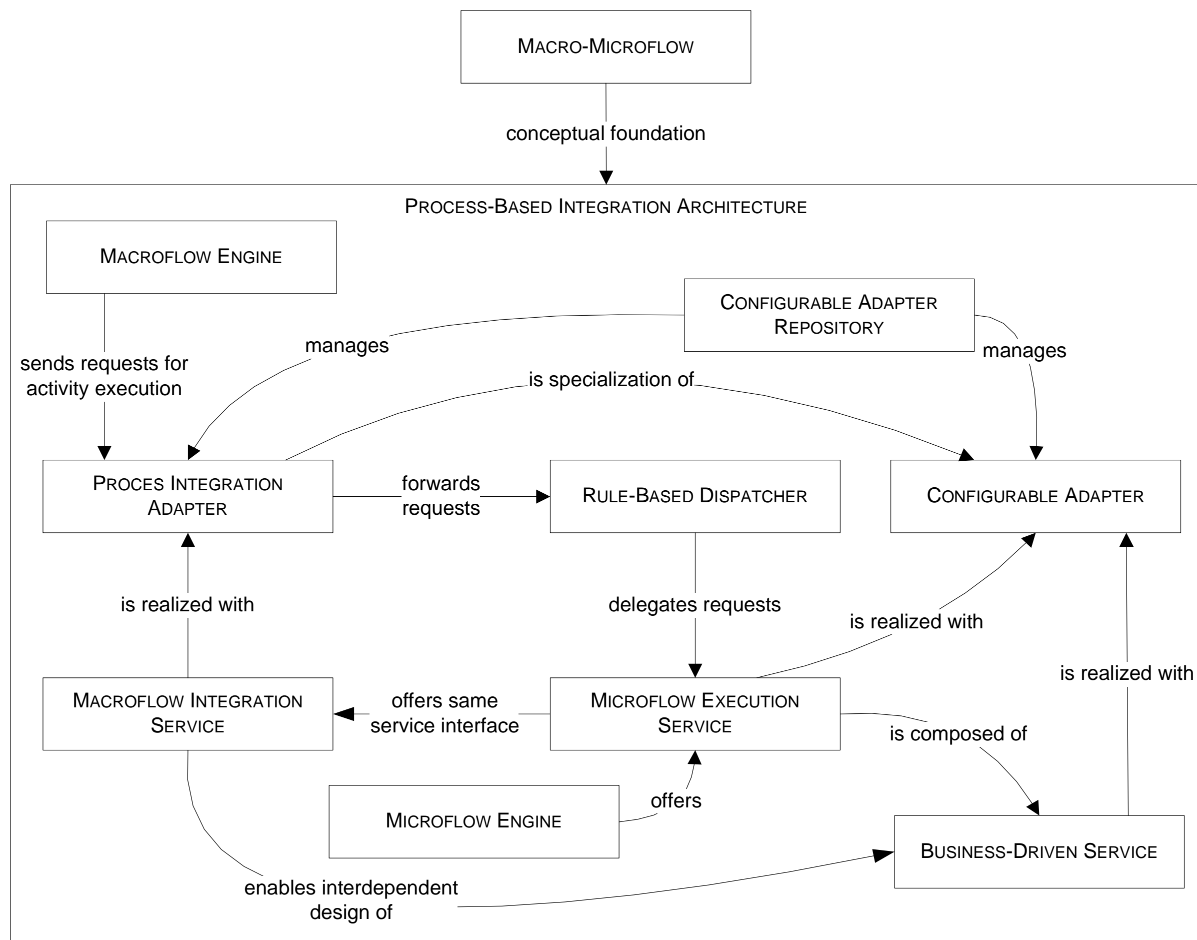


Figure 3: Pattern relationships overview

Table 1 gives an overview of the problem and solution statements of the 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. An overview table for these related patterns is provided in the Appendix.

Pattern	*Problem*	*Solution*
MACRO-MICROFLOW	How is it possible to conceptually structure process models in a way that makes clear which parts will be depicted on a process engine as long running business process flows and which parts of the process will be depicted inside of higher-level business activities as rather short	Structure a process model into macroflow and microflow.

Pattern	*Problem*	*Solution*
	running technical flows?	
PROCESS-BASED INTEGRATION ARCHITECTURE	Which architecture design concepts for process-driven backend systems integration are necessary, in order for the architecture to be scalable, flexible, and maintainable?	Provide a multi-layered PROCESS-BASED INTEGRATION ARCHITECTURE to connect macroflow business processes and the backend systems that need to be used in those macroflows.
MACROFLOW INTEGRATION SERVICE	How can the functionality and implementation of process activities at the macroflow level be decoupled from the process logic that orchestrates them, in order to achieve flexibility, as far as the design and implementation of these automatic functions are concerned?	The automatic functions required by macroflow activities from external systems are designed and exposed as dedicated MACROFLOW INTEGRATION SERVICE with well-defined service interfaces.
PROCESS INTEGRATION ADAPTER	How can interface and technology specifics of a process engine be connected to a different interface and technology of another system, such that the two systems can communicate, as far as requests for activity execution and the corresponding responses are concerned? How to design this connection in a loosely coupled fashion?	Use a PROCESS INTEGRATION ADAPTER that connects the specific interface and technology of the process engine to an integrated system.
RULE-BASED DISPATCHER	How can it be dynamically decided what component has to execute a (macroflow) activity, in order to allow scalability and functional structuring of the architecture, i.e., more generally speaking, how can location-transparency, access-transparency, and scalability-transparency be considered?	Use a RULE-BASED DISPATCHER that picks up the (macroflow) activity execution requests and dynamically decides based on (business) rules, where and when a (macroflow) activity has to be executed.
CONFIGURABLE ADAPTER	How can a system be connected to other systems in a way that allows to easily maintaining the connections, considering that interfaces may change over time?	Implement a CONFIGURABLE ADAPTER to another system that should be connected.
CONFIGURABLE ADAPTER REPOSITORY	How is it possible to manage adapters, such as CONFIGURABLE ADAPTERS or PROCESS INTEGRATION ADAPTERS, in a larger architectural context such that changes to these adapters can be easily implemented at runtime and maintenance effects to connected systems are kept minimal?	Use a central repository to manage the adapters as components and design the adapters as CONFIGURABLE ADAPTERS such that they can be modified at runtime without affecting the components or systems sending requests to the adapters.
MICROFLOW EXECUTION SERVICE	How to expose a microflow as a coherent function with defined input and output parameters without having to consider the technology specifics of the MICROFLOW ENGINE being used, in order to decouple the engine's technology specifics from the actual functionality that is has to offer to execute concrete microflows?	Expose a microflow as a MICROFLOW EXECUTION SERVICE that abstracts the technology specific API of the MICROFLOW ENGINE to a standardised well-defined service interface and encapsulates the functionality of the microflow.
MACROFLOW ENGINE	How is it possible to flexibly configure macroflows in a dynamic environment where business process changes are regular practice, in order to reduce implementation time and effort of these business process changes, as far as the related IT issues are concerned that are involved in these changes?	Delegate the macroflow aspects of the business process definition and execution to a dedicated MACROFLOW ENGINE that executes the business processes described in a business process modelling language. The engine allows developers to configure business processes by flexibly orchestrating execution of macroflow

Pattern	*Problem*	*Solution*
		activities and the related business functions.
MICROFLOW ENGINE	How is it possible to flexibly configure IT systems integration processes in a dynamic environment, where IT process changes are regular practice, in order to reduce implementation time and effort?	Delegate the microflow aspects of the business process definition and execution to a dedicated MICROFLOW ENGINE that allows developers to configure microflows by flexibly orchestrating execution of microflow activities and the related BUSINESS-DRIVEN SERVICES.
BUSINESS-DRIVEN SERVICE	How can the requirements be engineered to decide what services need to be defined, what functionality is actually required, and thus what services must be designed and implemented?	Design BUSINESS-DRIVEN SERVICES that are defined according to a convergent top-down and bottom-up engineering approach, where high-level business goals are mapped to to-be macroflow business process models that fulfil these high-level business goals and where more fine grained business goals are mapped to activities within these processes.

Table 1: Problem/solution overview of the patterns

MACRO-MICROFLOW

Business processes shall be implemented using process (workflow) technology.

Models of business processes must be developed considering the relationships and interdependencies to technical concerns. If technical concerns are tangled in the business process models, however, business analysts are forced to understand the technical details, and the technical experts must cope with the business issues when they are realizing technical solutions. This should be avoided. On the other hand, to create executable process models, somehow the two independent views need to be integrated into a coherent system.

When developing process models for process-aware information systems, it is necessary to model the process flows from the business perspective, but also to consider IT-related concerns of the process flow. For instance, from the business perspective, process flows are often long-running flows, whereas from the technical perspective rather short-running (transactional) flows need to be considered.

Often both types are mixed in practice and the different concerns are tried to be modelled in one process flow. This practice often causes confusion as business analysts do not understand the level of technical detail, and technical modellers do not have the expertise to understand the business issues fully. Thus, these models tend to fail their primary purpose which is to communicate the overall process/system.

Business processes, which mix the technical level and the business level, often include too many technical details and are not decoupled from technology issues. Hence, the business processes become inflexible and too complex to be managed from the business point of view. Nevertheless, business concerns and IT-related concerns of business processes must be formally linked, because the final result of business process modelling should be process models that can be executed using process technology. This imposes modelling boundaries on the business process models.

Structure a process model into two kinds of processes, macroflow and microflow. Strictly separate the macroflow from the microflow, and use the microflow only for refinements of the macroflow activities. The macroflow represents the long-running, interruptible process flow which depicts the business-oriented process perspective. The microflow represents the short-running transactional flow which depicts the IT-oriented process perspective.

The MACRO-MICROFLOW pattern solves the conceptual problem how to relate business-oriented processes (macroflow) and IT-oriented processes (microflow) by interpreting a microflow as a refinement of a macroflow activity. A microflow represents a sub-process that runs within a macroflow activity. A microflow model can be linked to one or many macroflow activities. The consequence is that the types of relationships between macroflow and microflow are defined. The resulting process models form a basis for implementation with process technology, as they consider technical and business issues separately.

The microflow can be directly invoked as a sub-process that runs automatically, or it can represent an activity flow that includes human interaction. As a result, two types of links between a macroflow activity and a microflow do exist:

- *Link to a microflow for an automatic activity (transaction):* A short-running (transactional) IT process defines a detailed process model of an automatic activity in a higher-level business process. It represents an executed business function or transaction at the business process level.
- *Link to a microflow for human interaction:* In case an activity of a business process is associated to a user interface, the IT process is a definition of the coherent process flow that depicts the human interaction. This process flow is initiated if a human user executes the business process activity.

The microflow level and the macroflow level distinguish conceptual process levels, and the possible links between them must be explicitly considered. Ideally, modelling languages, techniques, and tools should support this conceptual separation and allow the definition of links between macroflow activities and microflows using the two types of links defined above. Figure 4 illustrates this conceptual separation of microflow and macroflow process levels using an example. The figure shows a number of macroflow activities, and in the Macroflow Activities 2 and 3 more fine grained IT process models are linked. Activity 2 has a link of type "transaction" and Activity 3 of type "human interaction".

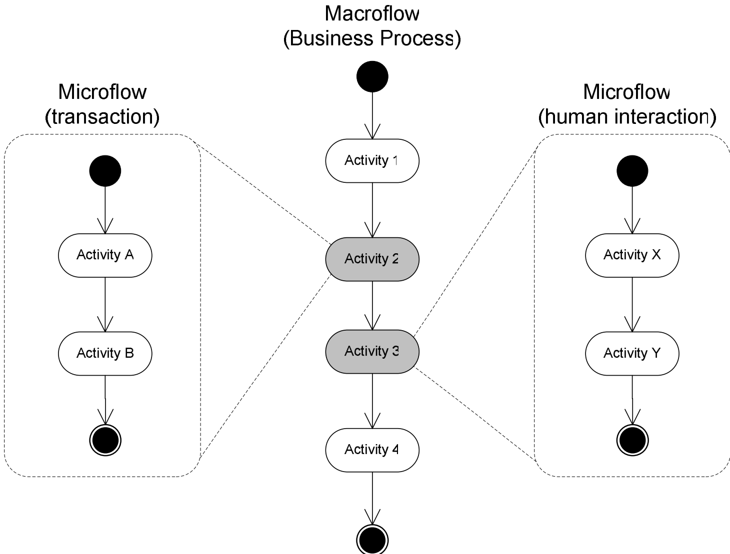


Figure 4: The process levels of microflow and macroflow

It is generally possible to have sub-processes at both the microflow and macroflow level. Within a microflow for human interaction it is also possible to invoke further microflows for an automatic activity (transaction) – a special type of sub-microflow within a microflow.

Figure 5 shows the conceptual model of microflow and macroflow more formally as an exemplary meta-model for realizing MACRO-MICROFLOW. The model shows the different kinds of macroflow and microflow as special process classifiers and the relationships between these classifiers. The MACRO-MICROFLOW pattern generally provides a conceptual basis for the development of meta-models as a foundation for model-driven software design – for instance following a meta-model like the one shown in Figure 5.

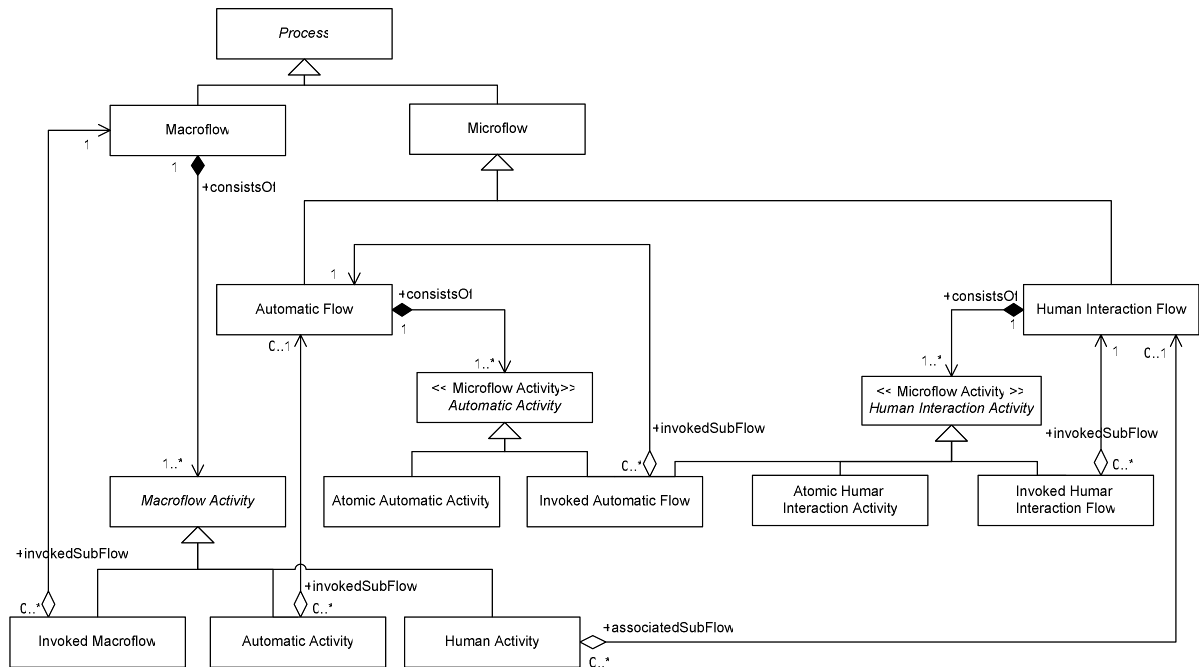


Figure 5: Structural meta-model of macroflow and microflow

When microflow or macroflow work with business objects, the BUSINESS OBJECT REFERENCE pattern [Hentrich 2004] will be used in the flows. The BUSINESS OBJECT REFERENCE pattern provides a solution to referencing business objects that are stored in some container.

The MACRO-MICROFLOW pattern has the following benefits: Modelling can be performed in several steps of refinement. First the higher level macroflow business process can be designed, considering already that business process activities will further be refined by microflows. Vice versa, if certain microflows already exist, the business process can be modelled accordingly, so that these IT processes fit in as business process activities at the macroflow level. MACRO-MICROFLOW thus implies interdependence between the two process levels.

Microflows and macroflows both have a defined input and output, i.e., a well-defined functional interface. By linking these process levels using defined types of relations as described, it is possible to navigate through the overall process, addressing the concerns of both domains – business and IT – separately, while still keeping them in conjunction.

The MACRO-MICROFLOW pattern also has the following drawbacks: The conceptual separation of the MACRO-MICROFLOW pattern must be understood and followed by modellers, which requires additional discipline. Also, the functional interfaces between IT processes and business processes must be understood and considered in the models. The pattern's concepts further require adjusting IT processes and business processes according to the concerns of both domains – business and IT – in order to bring them together. The modelling effort is higher than in usual business modelling, as more aspects are taken into consideration. Designing activities at the right level of granularity in the business processes (macroflows) takes more time, as those activities already describe the later process-aware information system. Modelling is set limitations as IT processes and business processes must strictly fit together like pieces of a puzzle.

Some known uses of the pattern are:

- In IBM's WebSphere technology the general concept introduced by the MACRO-MICROFLOW pattern is reflected by different technologies and methodologies being used to design and implement process-aware information systems. Actually, there are

different kinds of technologies and techniques for both types of flows. On the macroflow level, workflow technologies are used that allow integration of people and automated functions on the business process level. An example is IBM's WebSphere Process Choreographer, which is a workflow modelling component. The microflow level is rather represented by transactional message flow technologies that are often used in service-oriented approaches, for instance. Examples are the WebSphere Business Integration Message Broker and also the WebSphere InterChange Server. At the business process (macroflow) level, a service is invoked that is designed and implemented in detail by a microflow that performs data transformation and routing to a backend application. Moreover, aggregated services are often implemented at the microflow level using these kinds of message flow technologies.

- GFT's BPM Suite GFT Inspire [GFT 2007] provides a modeller component that uses UML activity diagrams as a notation for modelling the macroflow. Microflows can be modelled in various ways. First, there are so-called step activities, which allow the technical modeller to model a number of sequential step actions that refine the business activity. In the step actions, the details of the connection to other systems can be specified in a special purpose dialog. This concept is especially used to invoke other GFT products, like the document archive system or a form-based input. Alternatively, the microflow can be implemented in Java snippets, which can be deployed to the server (together with the business process). Finally, services can be invoked that can integrate external microflow technologies, such as message brokers.
- JBoss' jBPM [JBoss 2007] follows a slightly different model, as the core component is a Java library and hence can be used in any Java environment. The jBPM library can also be packaged and exposed as a stateless session EJB. JBoss offers a graph-based designer for the macroflow process languages, and works with its own proprietary language, jPDL. A BPEL extension is also offered. The microflow is implemented through actions that are associated with events of the nodes and transitions in the process graph. The actions are hidden from the graphical representation, so that macroflow designers do not have to deal with them. The actions invoke Java code, which implements the microflow. The microflows need not be defined directly in Java, but can also be executed on external microflow technology, such as a message broker.

PROCESS-BASED INTEGRATION ARCHITECTURE

Process technology is used, and the basic process design concept follows the MACRO-MICROFLOW pattern.

Process technology is used at the macroflow level, and backend systems need to be integrated in the process flow. The connection between the macroflow level and the backend systems needs to be flexible so that different process technologies can (re-)use the connection to the backend systems. The architecture must be able to cope with increased workload conditions, i.e., it must be scalable. Finally, the architecture must be changeable and maintainable to be able to cope with both changes in the processes and changes in the backends. All those challenges cannot be mastered without a clear concept for the whole SOA. That is, if backend integration in a process-driven SOA is performed per backend, it is highly unlikely that the overall solution is indeed flexible, scalable, changeable, and maintainable.

To properly consider the qualities attributes flexibility, scalability, changeability, and maintainability a number of issues must be addressed. First, there are technology specifics of the process technology being used at the macroflow level. In principle, implementations of macroflow activities represent reusable functions that are not restricted to one specific process technology but which can rather be used with different types and implementations of process engines. If the process technology is tightly coupled to implementations of activities, changes in the process technology may potentially have larger impact on the corresponding activity implementations which means a loss of flexibility.

Activities at the macroflow level are usually refined as microflows following the MACRO-MICROFLOW pattern. Thus, one has to consider where and how these microflows are executed. Aspects of scalability must be considered to cope with increasing workload. As requests for activity execution are permanently initiated and business will usually go on day and night, we additionally have to deal with the question: What further mechanisms are necessary to maintain the whole architecture at runtime?

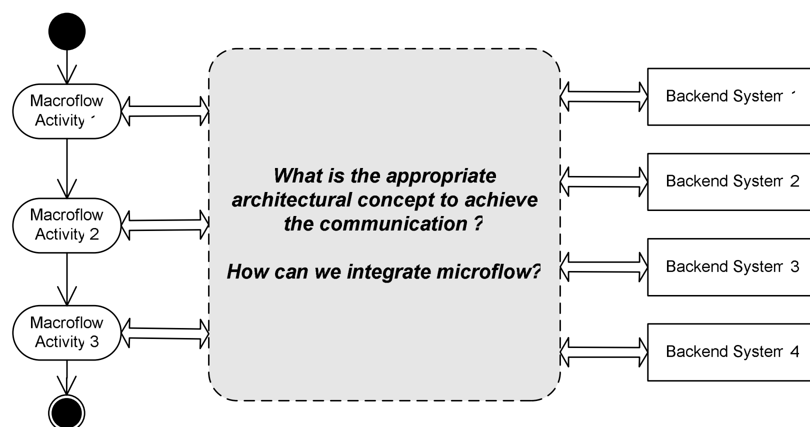


Figure 6: The problem of backend system integration

Changes to the microflow and macroflow should be easy and of low effort. Actual backend system functionality will be invoked at the microflow level, and it is obviously an issue how this can be achieved, as those backend systems are in principle independent and are subject to

individual changes themselves. The impact of these changes must be kept within acceptable limits, in a way that those changes can be managed. Figure 6 illustrates the problem.

Provide a multi-layered PROCESS-BASED INTEGRATION ARCHITECTURE to connect macroflows and the backend systems that need to be used in those macroflows. The macroflows run in dedicated MACROFLOW ENGINES that can invoke MACROFLOW INTEGRATION SERVICES realized by special PROCESS INTEGRATION ADAPTERS. Microflows also run in dedicated MICROFLOW ENGINES. These engines offer MICROFLOW EXECUTION SERVICES that can be invoked to start a specific microflow. The PROCESS INTEGRATION ADAPTERS do not invoke the MICROFLOW EXECUTION SERVICES directly, but use an intermediate RULE-BASED DISPATCHER to achieve scalability. Finally, the microflows invoke BUSINESS-DRIVEN SERVICES that represent the backend systems via CONFIGURABLE ADAPTERS.

The PROCESS-BASED INTEGRATION ARCHITECTURE pattern is an architectural pattern that defines a specific configuration using a number of other patterns that explain more detailed design solutions within its scope. In that sense, this architectural pattern explains the architectural foundation of how the other patterns work together. Thus, this pattern mainly explains the relationships between the sub-patterns in detail. For more detailed information on the sub-patterns themselves, please refer to the corresponding pattern descriptions.

MACROFLOW INTEGRATION SERVICES are the connection between a macroflow activity of a business process and BUSINESS-DRIVEN SERVICES in the backend. A MACROFLOW INTEGRATION SERVICE can be invoked by an activity in a macroflow that runs on a MACROFLOW ENGINE. The MACROFLOW INTEGRATION SERVICE represents the function that satisfies the functional needs of the macroflow activity. Moreover, the MACROFLOW INTEGRATION SERVICE encapsulates the whole and often complex logic for invoking BUSINESS-DRIVEN SERVICES in the backend.

As those backend services can be developed and enhanced independently over time, they stand for themselves and are primarily not dependent on a process model. A BUSINESS-DRIVEN SERVICE thus hides the details and possible changes of a backend system and represents a functional interface to a backend system. This interface of a BUSINESS-DRIVEN SERVICE is independent of technology and interface requirements of macroflow processes.

Integration logic is required to establish the communication between the backend service and the macroflow activity. This integration logic is realized using a microflow that runs on a MICROFLOW ENGINE. Principally, the microflow is based on message routing and message transformation patterns, such as those described in [Hohpe et al. 2003]. The business objects relevant to microflows and macroflows essentially form the CANONICAL DATA MODEL [Hohpe et al. 2003] for storing process relevant business data. The BUSINESS OBJECT REFERENCE [Hentrich 2004] pattern is used to keep the references to the business objects in the process flow.

In larger architectures there might be several MICROFLOW ENGINES and MACROFLOW ENGINES involved that need to be connected. It is desirable to have a flexible concept for process integration services that can be adapted according to changing workload. This is the task of the RULE-BASED DISPATCHER: It exposes an enterprise-wide standard interface for capturing requests of macroflow activities. For each MACROFLOW ENGINE, a PROCESS INTEGRATION ADAPTER is required, to transform the interface and technology specifics of the MACROFLOW ENGINE into the interface of the RULE-BASED DISPATCHER. The RULE-BASED DISPATCHER is responsible for distributing the process integration service requests to various MICROFLOW ENGINES. These engines execute integration logic that is functionally encapsulated by the

MICROFLOW EXECUTION SERVICES. The MICROFLOW ENGINE coordinates the integration activities and invokes the BUSINESS-DRIVEN SERVICES in the backend.

The RULE-BASED DISPATCHER has the main purpose to ensure scalability and handle distribution, according to current workload and business rules. Hence, the RULE-BASED DISPATCHER is an optional component that might be superfluous and unnecessary overhead, if only one or two engines with fixed relations are used and it can be assumed as a fact that no more engines will be added.

The BUSINESS-DRIVEN SERVICES are exposed via CONFIGURABLE ADAPTERS that build the technical connection to the business applications in the backend. Analogously, MACROFLOW INTEGRATION SERVICES are provided by PROCESS INTEGRATION ADAPTERS. Both kinds of adapters are typically managed in CONFIGURABLE ADAPTER REPOSITORIES – to achieve a central management of the adapters for different process engines. Of course, the use of CONFIGURABLE ADAPTER REPOSITORIES is optional and only needed if multiple process engines are used.

Figure 7 shows a simple example configuration for the PROCESS-BASED INTEGRATION ARCHITECTURE pattern. It only shows a minimal configuration, with one MACROFLOW ENGINE (e.g. a BPEL engine), one PROCESS INTEGRATION ADAPTER to map the business concerns into the technical architecture, one simple MICROFLOW ENGINE (e.g. a message broker or even not a "real" engine, but only hard wired service flows implemented in Java), and a number of business applications that are integrated into the SOA using business application adapters.

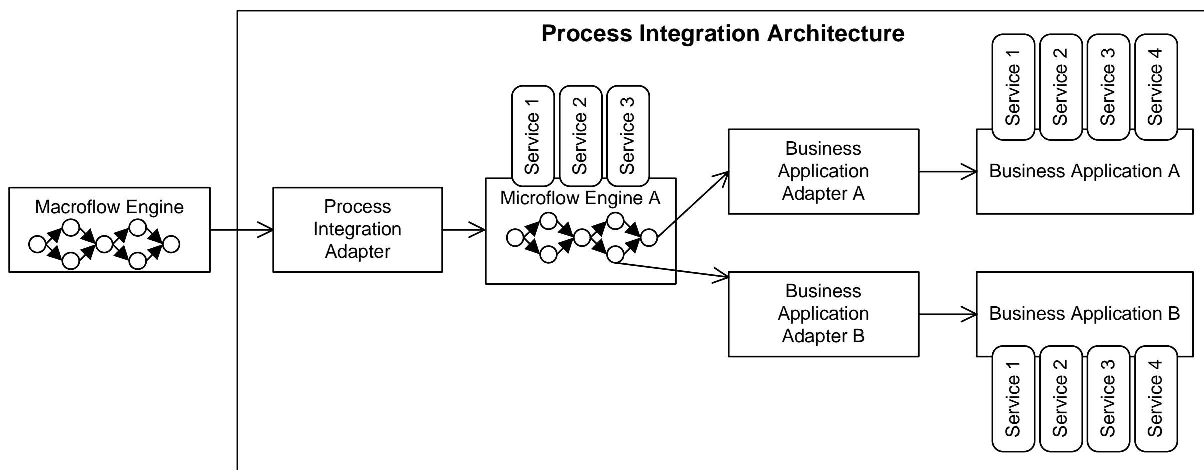


Figure 7: Simple, small-scale configuration of a Process-Based Integration Architecture

Figure 8 shows a larger-scale example configuration for the PROCESS-BASED INTEGRATION ARCHITECTURE pattern in terms of a layered model and its boundaries. As a SOA evolves and grows larger, a simple configuration, such as the one in Figure 7, can grow into such a larger configuration. Here, also the optional patterns explained above are used, and multiple instances of both MACROFLOW ENGINES and MICROFLOW ENGINES are present to support scalability.

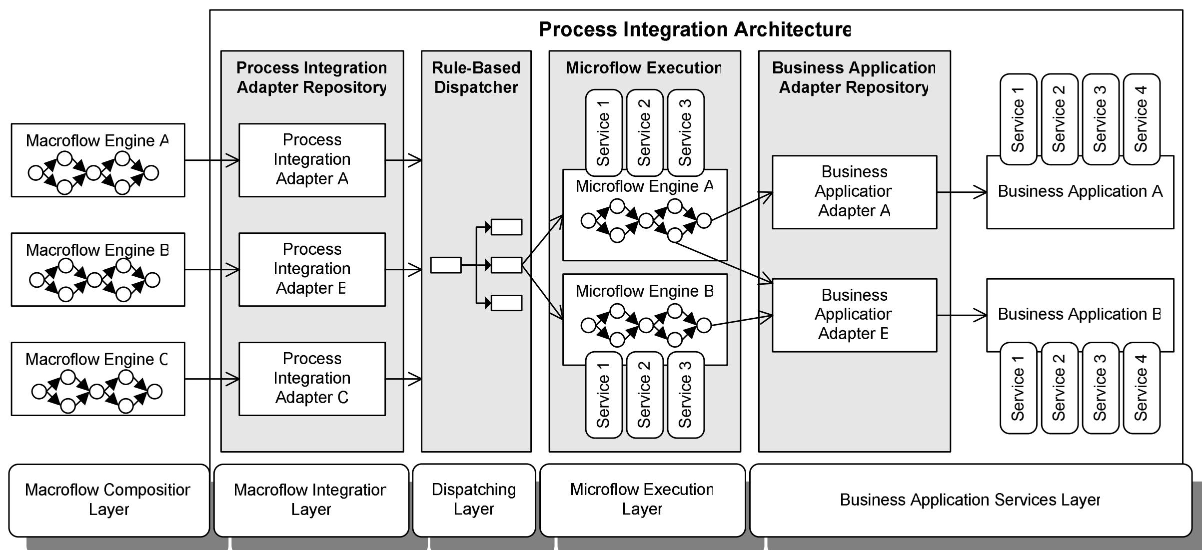


Figure 8: Layers and boundaries of a Process-Based Integration Architecture

The collaborative process between the different layers of a PROCESS-BASED INTEGRATION ARCHITECTURE is managed via exchanging service requests and responses. The CORRELATION IDENTIFIER pattern [Hohpe et al. 2003] allows for relating requests and responses between the different components involved: Each request is assigned a unique ID which is passed back in the response. Thus, a MACROFLOW ENGINE may correlate a response to the original request.

The UML class diagram in Figure 9 illustrates this collaborative process in terms of a static view of the relationships between the involved patterns depicting a CORRELATION IDENTIFIER as an association type. Please refer to the sub-pattern descriptions for more detailed information on the collaborative aspects between the individual patterns.

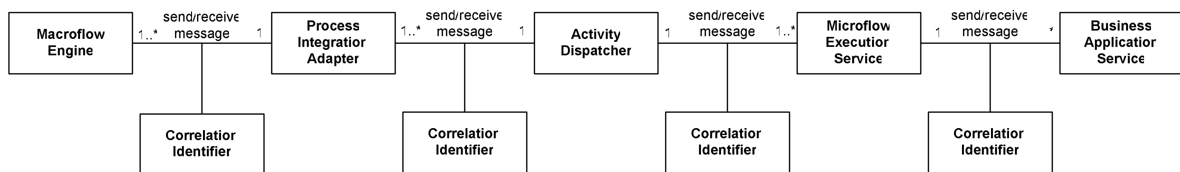


Figure 9: Collaboration using the Correlation Identifier pattern

A PROCESS-BASED INTEGRATION ARCHITECTURE is often implemented in the context of the ENTERPRISE SERVICE BUS (ESB) pattern [Zdun et al. 2006].

Additionally, there are rather indirect relationships to the BUSINESS OBJECT REFERENCE [Hentrich 2004] pattern, as business objects may be subject of microflows and macroflows and may form a CANONICAL DATA MODEL [Hohpe et al. 2003] for storing process relevant business data.

The PROCESS-BASED INTEGRATION ARCHITECTURE pattern has the following benefits: It enables a service-oriented and process-driven approach to architecture design. It offers a flexible and scalable concept for integrating backend systems in a process flow, and, at the same time, it explains how to integrate different kinds of MICROFLOW ENGINES and MACROFLOW ENGINES. Hence, MICROFLOW ENGINES and MACROFLOW ENGINES can be treated as exchangeable components. Thus, also the business applications, macroflows, and microflows can be maintained as independent components as long as the service interfaces do not change. Load balancing and prioritized or rule-based processing of requests via the RULE-BASED DISPATCHER is supported. Many existing off-the-shelf engines can be used as components of a PROCESS-

BASED INTEGRATION ARCHITECTURE, which might reduce the necessary in-house development effort.

The PROCESS-BASED INTEGRATION ARCHITECTURE pattern also has the following drawbacks: Greater design effort might be necessary compared to simpler alternatives, because of the layered model with corresponding loosely coupled interfaces. If only one MICROFLOW ENGINE and MACROFLOW ENGINE is used which shall not be exchangeable, the RULE-BASED DISPATCHER and the CONFIGURABLE ADAPTER REPOSITORY might be superfluous – it is thus possible to customize the architectural concepts according to concrete situations and the architecture can be extended, if necessary.

To buy (and customize) different off-the-shelf engines for MICROFLOW ENGINES, MACROFLOW ENGINES, RULE-BASED DISPATCHER, etc. can be costly, just like inhouse-development of these components. Hence, for small, simple process-driven SOAs, it should be considered to start-off with a single process engine and follow the MACRO-MICROFLOW pattern only conceptually. A PROCESS-BASED INTEGRATION ARCHITECTURE pattern can then still be introduced later in time, when requirements for higher flexibility, scalability, changeability, and maintainability arise.

Some known uses of the pattern are:

- In a supply chain management solution for a big automotive customer in Germany this architectural pattern has been applied. WebSphere MQ Workflow has been used as the MACROFLOW ENGINE. The integration adapters, the dispatching layer, and the microflow execution level have been implemented in Java. The application services are implemented using MQ messaging technology. In this realization of the pattern, a Java architecture has been implemented to represent the RULE-BASED DISPATCHER, a MICROFLOW ENGINE, and the application adapters. No off-the-shelf middleware has been used.
- For a telecommunications customer in Germany, the pattern has been used in a larger scale variant. The MICROFLW ENGINE has been implemented by an enterprise service bus based on WebSphere Business Integration Message Broker. WebSphere MQ Workflow has been used as the process engine at the macroflow layer. The off-the-shelf MQ Workflow adapters provided by the message broker served as the PROCESS INTEGRATION ADAPTER. The architecture has been laid out initially as to support different instances of MQ Workflow engines to cope with growing workload using a dispatcher represented as a routing flow that routes the messages received by the adapter to another message broker instance. New message broker instances have been created according to the growing workload.
- A simple variant of the pattern is implemented in IBM's WebSphere Integration Developer, which includes WebSphere Process Server, a process engine that represents both the micro- and macroflow levels. It further offers an architectural concept called Service Component Architecture (SCA) to wire up services, including the corresponding adapters.

MACROFLOW INTEGRATION SERVICE

Macroflows that represent long-running business processes are executed on a dedicated MACROFLOW ENGINE.

Automatic activities in macroflows must be executed by integrated systems. If the integrated systems are directly called from the macroflow activities, the process logic is tightly coupled with the implementation of these automatic functions. This means, that technical details pollute the macroflow, and thus it is impossible to keep a clean business function view in the macroflow. Also, it is not be possible to change the process logic with low effort.

Usually, there are many activities that represent automatic functions in a business process at the macroflow level. That is, these activities imply that something must be done by some external systems. From the business process perspective, such activities are coherent and consistent business functions and, for this reason, they are designed as automatic activities. In most cases these functions should be reusable, which means that there may be many processes that require one and the same function to be executed. These functions might be quite complex in their implementation because there might be many systems involved to realize them in terms of a whole microflow (see also MACRO-MICROFLOW pattern).

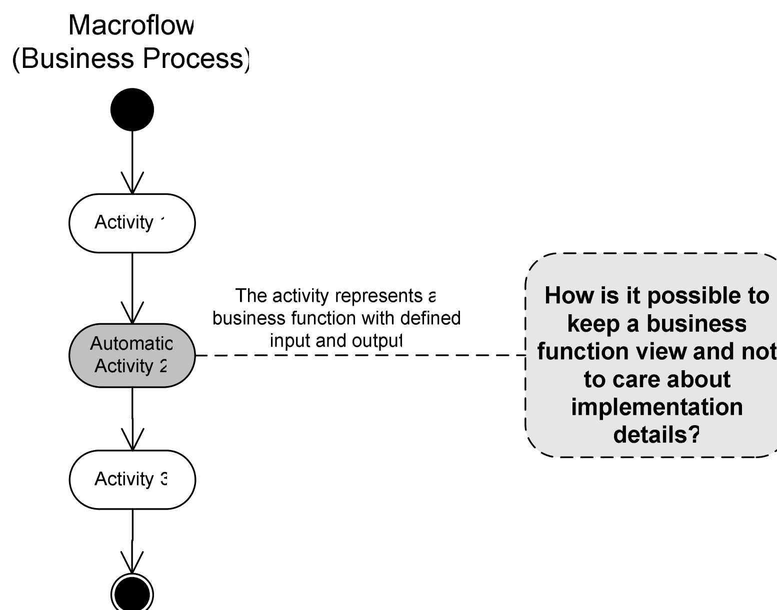


Figure 10: The problem of keeping the business function view in the macroflow

A MACROFLOW ENGINE only orchestrates the process activities and thus decouples the process logic. It should not be depending on the physical location where a function is executed, the systems that actually realize the function, or the function's implementation details. Changes to these details should be possible with low effort. This problem is illustrated in Figure 10.

The automatic functions required by macroflow activities from external systems are designed and exposed as dedicated MACROFLOW INTEGRATION SERVICES with well-defined service interfaces. These services integrate external systems in a way that suits the functional requirements of the macroflow activity. That is, they are designed to expose a specific business process' view – needed by a macroflow – onto an external system. The macroflow activity can thus be designed to contain only the functional business view and invoke MACROFLOW INTEGRATION SERVICES for interaction with external systems.

The functional requirements of the macroflow activity must be turned into a functional specification with a well-defined interface in terms of a complete definition of input and output parameters. The input parameters must match the available data in the corresponding macroflow process or macroflow activity respectively. The definition of output parameters must also be mapped to elements of the data structure in the process. In this context, as far as the process control data structure design is concerned, the GENERIC PROCESS CONTROL STRUCTURE and the BUSINESS OBJECT REFERENCE patterns should be considered [Hentrich 2004].

As a result, there is a data interdependency between the macroflow process and the service interface definition, as it is necessary to map the input and output parameters to elements in the process control data structure. However, the implementation of the service may vary as the macroflow activity only considers the functional interface and does not have to care how the service is implemented. Thus, the macroflow considers only aspects of business functionality. This enables the use of a consistent approach from design of business functionality to implementation. Moreover, requirements for services can be deduced from process specifications. It is actually a process design issue to specify activities at a level of granularity that is appropriate to shape reusable blocks of business functions in terms of services.

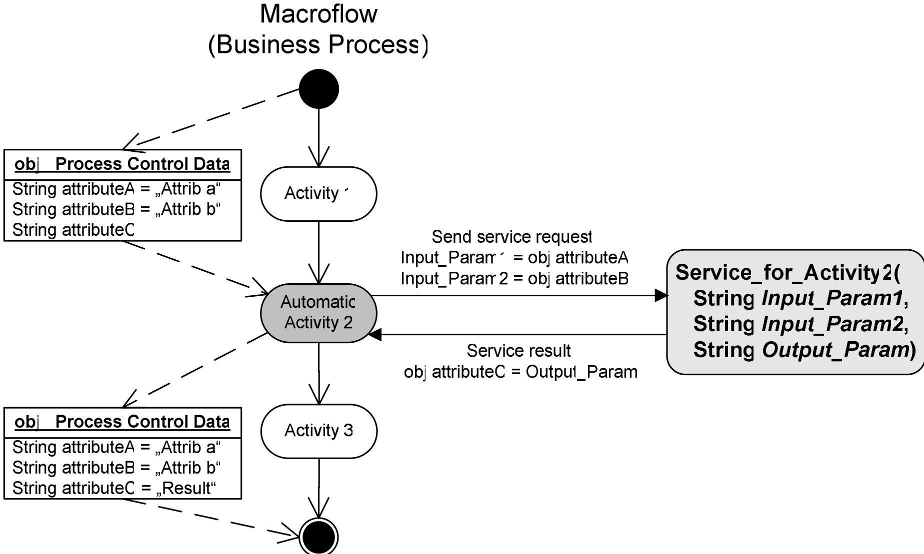


Figure 11: Macroflow activity invoking a service that represents the required business function

Figure 11 illustrates how a service can be invoked in a macroflow. The service corresponds to the functionality of an automatic macroflow activity. It is also shown that input and output parameters of the process control data must match the parameters in the service interface. Thus, the service interface is designed according to information that can be provided by the process control data structure. The parameters of the service match the available attributes in the process control data structure.

The MACROFLOW INTEGRATION SERVICE itself is reusable across different macroflows and may be invoked by many processes. Hence, no redundant functionality will be built. It is also possible to scale the performance of the service implementation according to increased service request workload, without affecting the macroflow business processes, because the macroflow and the MACROFLOW INTEGRATION SERVICES are loosely coupled.

Technically, the functionality specified by the service is realized by a PROCESS INTEGRATION ADAPTER that establishes the connection between a macroflow process engine and a service provider. The service provides executes the desired service and communicates the result back to the process engine.

The MACROFLOW INTEGRATION SERVICE pattern is related to the MACROFLOW ENGINE pattern, as it deals explicitly with macroflow business processes. As the pattern also addresses issues of designing process control data structures, the BUSINESS OBJECT REFERENCE and GENERIC PROCESS CONTROL STRUCTURE patterns must be considered [Hentrich 2004].

The MACROFLOW INTEGRATION SERVICE pattern has the following benefits: Automatic macroflow activities and the services are loosely coupled. The business process at the macroflow level defines the functional requirements for services that fulfil the tasks of automatic activities. The design and modelling of macroflow processes, as well as the implementation, remains at the level of business functions. The MACROFLOW INTEGRATION SERVICE pattern thus enables a consistent approach from process design to service identification and service specification.

Implementation of MACROFLOW INTEGRATION SERVICES may vary according to performance requirements, for instance, while the interfaces to the macroflow stay stable. Defined and implemented MACROFLOW INTEGRATION SERVICES are, in most cases, reusable building blocks.

The MACROFLOW INTEGRATION SERVICE pattern also has the following drawbacks: For each new automatic macroflow activity a service must be defined that depicts the functional requirements of the macroflow activity. This might not be a big drawback: because MACROFLOW INTEGRATION SERVICES are reusable building blocks, often a suitable service implementation already exists that fulfils these requirements, and thus this service can be reused.

The large amount of generated services from various automatic activities in business processes needs to be managed in some way.

Some known uses of the pattern are:

- IBM's WebSphere Integration Developer directly supports the pattern in its Service Component Architecture (SCA) concept to wire up services with activities in a BPEL process.
- In a workflow-based implementation of a customer enrolment business process for a telecommunications company in Spain, the pattern has been applied to integrate services in a workflow implemented with WebSphere MQ Workflow. The actual services were realized via WebSphere MQ messages.
- In a business requirements analysis project for a telecommunications customer in Germany the pattern has been applied to identify services from business processes. The business processes have been modelled with BOC's Adonis and the services have been linked to process activities and were thus identified and modelled in a systematic way. The pattern has enabled a systematic approach to design services from the requirements given by business processes.

PROCESS INTEGRATION ADAPTER

A process engine is used together with other integrated systems.

Process engines and external systems, to be integrated in the processes, often have different interfaces and technology specifics. However, the systems in a process-driven SOA must be able to communicate with each other, and the connection should be designed in a loosely coupled fashion. If loose coupling is not achieved, changes to integrated external systems will have effects on the processes that run in process engines. This generates unmanageable complexity and efforts for change implementation.

A process engine only acts as a coordinator of activities. Some of these activities are tasks that need to be executed by some other system. For this reason, requests for activity execution and the corresponding responses must be aligned with the target system that captures the requests, takes over the task of activity execution, and generates a response. How can this communication be achieved if the process engine and the target system have different interfaces and technologies?

Furthermore, how can a response be related to an original request and how can a request be transformed such that it is understood by a target interface and technology? Vice versa, how can the response be generated from that target interface and technology be transformed backwards, such that the response can be understood by the process engine? Another question is how can we detect what kind of function is associated with a request, i.e., what function is requested to be executed by the process activity? Figure 12 illustrates the problem.

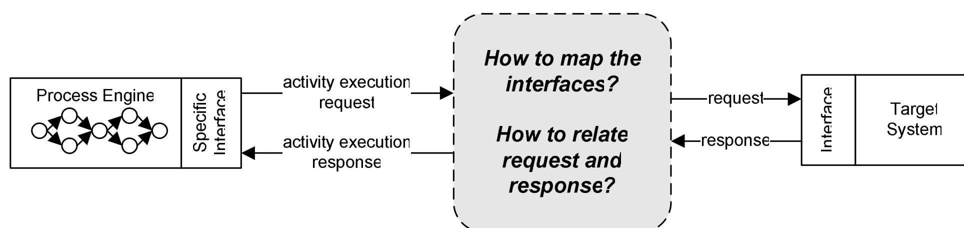


Figure 12: Delegating activity execution requests to an integrated target system

Use a PROCESS INTEGRATION ADAPTER that connects the specific interface and technology of the process engine to an integrated system. It transforms activity execution requests into requests that can be understood by the target system's interface and technology, and transforms responses from the target system backwards to the interface and technology of the process engine. CORRELATION IDENTIFIERS are used to relate requests and responses.

The core principle of the solution of the PROCESS INTEGRATION ADAPTER pattern is the same as in the classical ADAPTER pattern [Gamma et al. 1994]: An adapter connects to the interface and technology of the process engine and the target system, and translates between the two interfaces. The process engine acts as a sender in terms of sending out requests for activity execution, which are received by the adapter.

An activity execution request must contain a CORRELATION IDENTIFIER [Hohpe et al. 2003]. The CORRELATION IDENTIFIER identifies the activity instance in the process engine that is associated to the request, i.e., the activity instance that has sent the request. Additionally, the request must contain an identifier for the desired function to be executed and input parameters, such as BUSINESS OBJECT REFERENCES [Hentrich 2004], for instance. All these input data must be defined and carried in the process control data, so that it is accessible to an activity instance. As a result, this input data can be sent with the request.

The PROCESS INTEGRATION ADAPTER transforms the request into a format that can be understood by the target system's interface and technology. To achieve this transformation the adapter must implement transformation rules for mapping an activity execution request including all its data to the interface and request format of the target system. The request will be forwarded to the target system after the transformation is performed.

If supported by the target system, the CORRELATION IDENTIFIER will also be used on the target system's side to relate the response of the target system back to the original request. Consequently, the target system will have to send the CORRELATION IDENTIFIER back in its own response so that the adapter can re-capture it. The response will also contain the result of the execution of the desired function, which has been executed by the target system. If CORRELATION IDENTIFIERS cannot be used with the target system, for instance, because it is a legacy system that we cannot change, the PROCESS INTEGRATION ADAPTER must implement its own mechanism to align requests and results.

After the adapter has received a response of the target system, the response is transformed back into the format and technology used by the interface of the process engine. Again, transformation rules apply to achieve the transformation. Finally, the generated response to the original activity execution request, including the CORRELATION IDENTIFIER, is sent back to the interface of the process engine. The process engine is thus able to relate the response to the sending activity instance, and the process instance may proceed according to the result contained in the response.

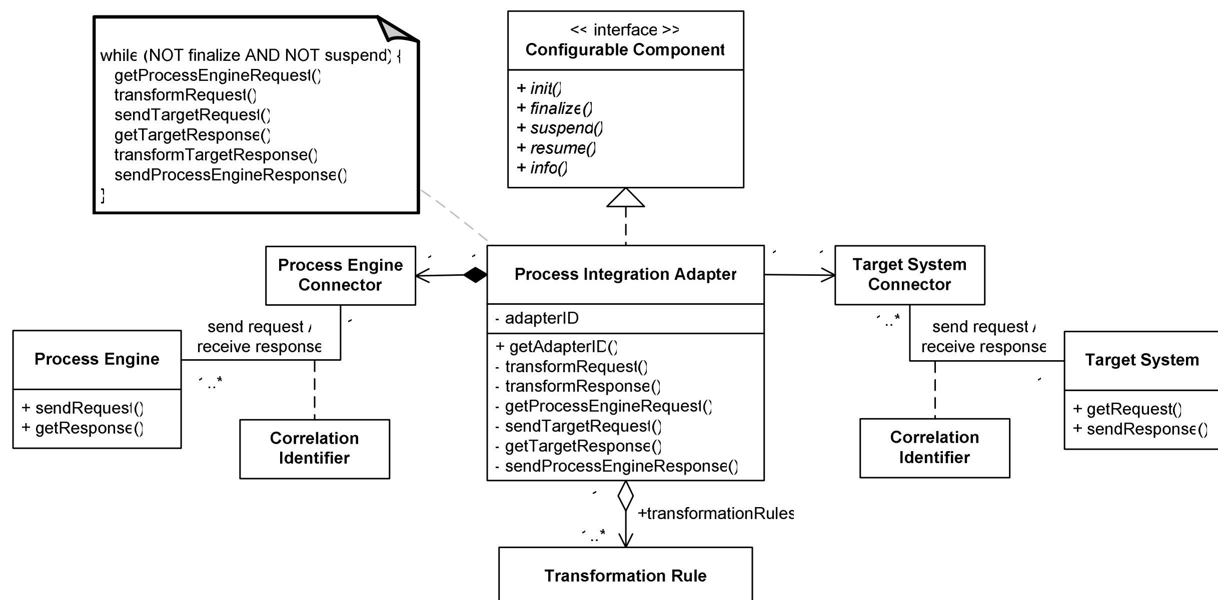


Figure 13: Conceptual structure of a process integration adapter

To make the PROCESS INTEGRATION ADAPTER maintainable at runtime, the COMPONENT CONFIGURATOR pattern [Schmidt et al. 2000] should be applied. That means, it is possible to initialise the adapter when it is put in use the first time and to suspend it at runtime, e.g., for updating the transformation rules, while activity execution requests are constantly arriving. Later on, after maintenance activities are finished, the adapter can resume its work and process all requests that have arrived in the meantime. The PROCESS INTEGRATION ADAPTER also offers a finalising function such that it finishes all ongoing activities properly and then terminates itself, e.g., in case it shall be replaced by a different adapter implementation. The PROCESS INTEGRATION ADAPTER is actually a specialization of the CONFIGURABLE ADAPTER pattern, which focuses on this aspect of configurability. Figure 13 shows an example for the conceptual structure.

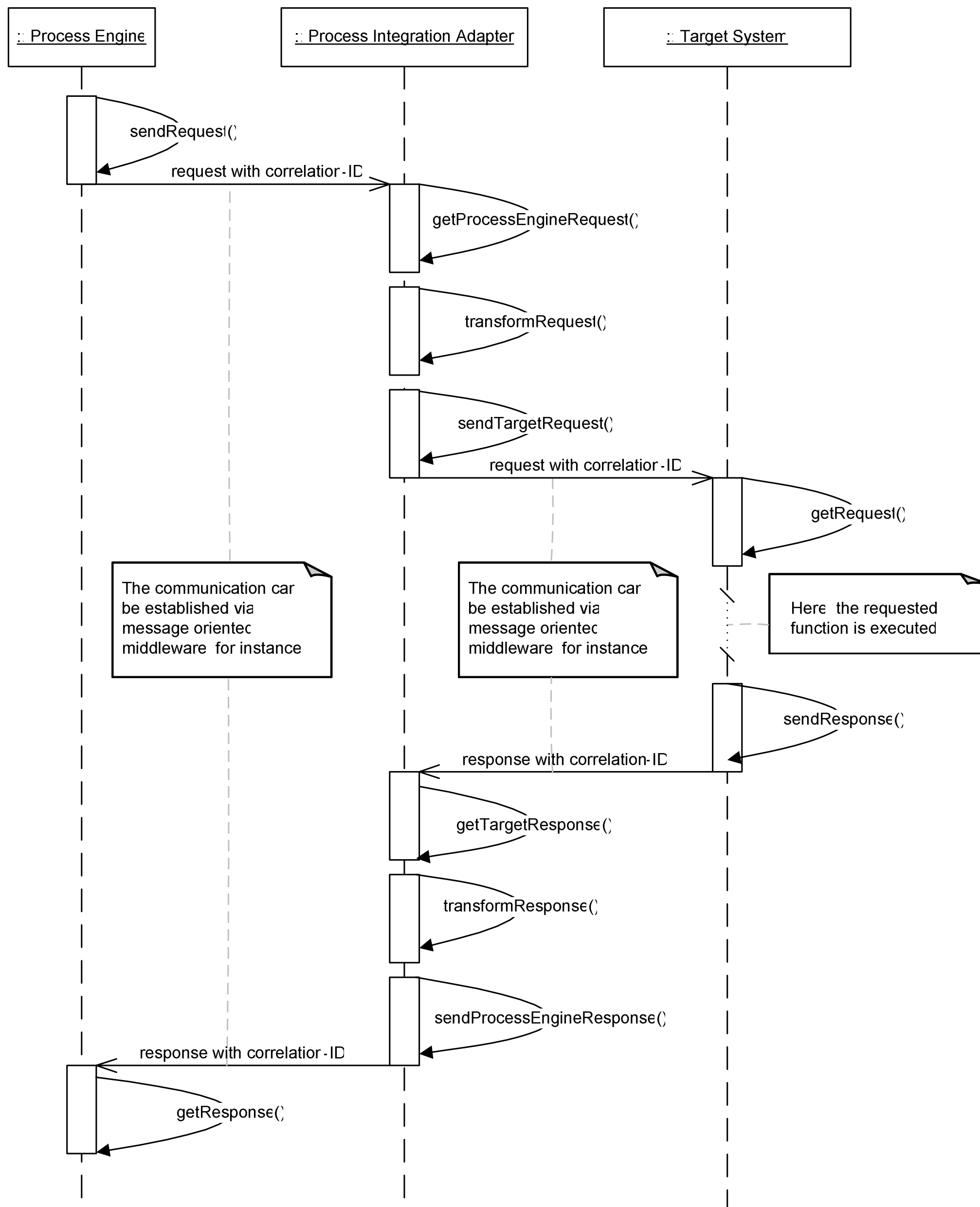


Figure 14: Principle interaction sequence

In principle, receiving a request or response can work via push or pull mechanisms. Thus, there are two general options of implementation, as far as this aspect is concerned. How the

requests and responses are sent and what kind of vehicle is used may vary as well. Often message-oriented middleware is used as a communication platform. Figure 14 shows the typical interaction.

As illustrated in Figure 13, the PROCESS INTEGRATION ADAPTER also has an attribute holding the ID of the adapter, which is usually a unique name. As PROCESS INTEGRATION ADAPTERS can be managed in a CONFIGURABLE ADAPTER REPOSITORY, this ID can be used to identify a specific PROCESS INTEGRATION ADAPTER in a CONFIGURABLE ADAPTER REPOSITORY, for instance.

In conjunction with a PROCESS-BASED INTEGRATION ARCHITECTURE the PROCESS INTEGRATION ADAPTER pattern is used to connect a MACROFLOW ENGINE to an RULE-BASED DISPATCHER. That means the target system is an RULE-BASED DISPATCHER in this case. Moreover, a MACROFLOW INTEGRATION SERVICE is technically realized with the help of the PROCESS INTEGRATION ADAPTER pattern, and a CONFIGURABLE ADAPTER REPOSITORY manages a whole set of those integration adapters.

The PROCESS INTEGRATION ADAPTER pattern has the benefit that it offers a clear model for the communication between a process engine and a target system.

The PROCESS INTEGRATION ADAPTER pattern also has the following drawbacks: In order to use PROCESS INTEGRATION ADAPTERS, the process engine must have an interface for communication with other systems. Transformation rules must be defined to transform requests forward to the target interface and responses backwards to the process engine.

Some known uses of the pattern are:

- WebSphere MQ Workflow offers a technical concept called a User-Defined Program Execution Server (UPES), which implements this pattern. The UPES concept is a mechanism for invoking services via XML-based message adapters. Basis of the UPES concept is the MQ Workflow XML messaging interface. The UPES concept is all about communicating with external services via asynchronous XML messages. Consequently, the UPES mechanism invokes a service that a process activity requires, receives the result after the service execution has been completed, and further relates the asynchronously incoming result back to the process activity instance that originally requested execution of the service (as there may be hundreds or thousands of instances of the same process activity).
- CSC offers within their e4 reference meta-architecture the concept of PROCESS INTEGRATION ADAPTERS. For an insurance customer in the UK the e4 adapter concept has been used to integrate FileNet P8 Business Process Manager with an enterprise service bus based on WebSphere Business Integration Message broker.
- Within the Service Component Architecture (SCA) concept of IBM's WebSphere Integration Developer various PROCESS INTEGRATION ADAPTERS are offered off-the-shelf, e.g., for WebSphere MQ, Web services, or JMS.

RULE-BASED DISPATCHER

Functional requests of automatic (macroflow) activities must be executed by backend systems.

As system architectures usually change over time, it is necessary to add, replace, or change systems in the backend for executing process activities. In many process-driven systems, this must be possible at runtime. That is, it must be dynamically decided at runtime which component has to execute a (macroflow) activity. If the architecture does not consider these dynamics, then modifications to the backend structures will be difficult to implement at runtime. On the other hand, scalability and the functional structuring might be negatively influenced by introducing more dynamics.

Let us consider the "larger" context and that architecture is subject to evolution and change. A typical evolution scenario is that the architecture must cope with an increased workload for automatic activity execution. Concerning evolution it is important to consider that the very component that actually executes a macroflow activity might be subject to change. This component might even get exchanged while the activity is executed. For instance, the component might be moved to another physical location.

Concerning increased workload, a high scalability is required, and typically supported by adding new instances of components that execute the additional requests. It must be dynamically decided at runtime which component has to execute a macroflow execution request. In addition, other concerns like priorities for execution of requests must be considered as well.

All these aspects actually point to well known issues in distributed architectures and can be conceptually classified as dimensions of transparency [Emmerich 2000]: access transparency, location transparency, migration transparency, replication transparency, concurrency transparency, scalability transparency, performance transparency, and failure transparency. The core problem is thus how to consider those dimensions of transparency appropriately in order to keep the architecture flexible and scalable.

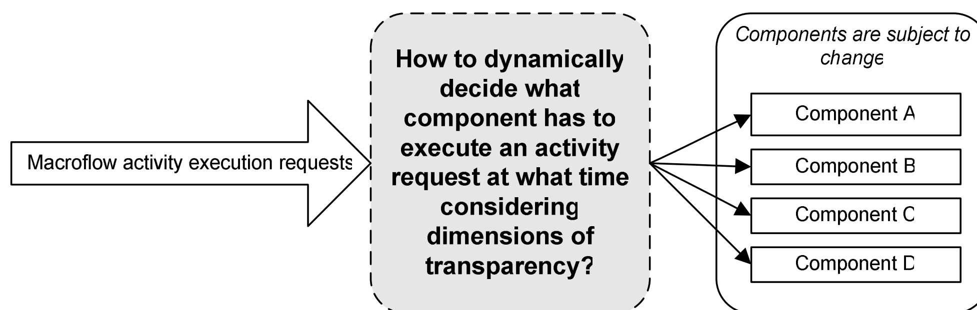


Figure 15: Transparency issues with macroflow activity execution

Use a RULE-BASED DISPATCHER that picks up the (macroflow) activity execution requests and dynamically decides on basis of (business) rules, where and when a (macroflow) activity has to be executed. After making the decision, the RULE-BASED DISPATCHER forwards the requests to the corresponding functional unit (component).

In a PROCESS-BASED INTEGRATION ARCHITECTURE, the requests for macroflow activity execution are sent by a PROCESS INTEGRATION ADAPTER. The RULE-BASED DISPATCHER is the connected target system of the adapter (see PROCESS INTEGRATION ADAPTER pattern). The

dispatcher applies dispatching rules to the requests to determine when, e.g., according to priorities, and where to forward the request. These dispatching rules are applied dynamically at runtime and the rules can be changed.

To achieve flexible configuration of the dispatching rules, the dispatcher applies the COMPONENT CONFIGURATOR pattern [Schmidt et al. 2000]. The requests, which must not be forwarded immediately according to the rules, are put on a hold queue and are forwarded later on. The dispatching rules are based on data in the requests, e.g., the request type, the function associated to the request in terms of a MACROFLOW INTEGRATION SERVICE, or the priority of the request. The rules are defined according to business requirements, and the request format designer must consider these requirements to provide the right data in the requests to evaluate the rules.

The dispatcher also has the task to pick up the request result from the component and send it back to the adapter. Again it is possible to apply special rules in this context. As the communication is usually achieved via messaging, a CORRELATION IDENTIFIER [Hohpe et al. 2003] is used to correlate the requests and responses.

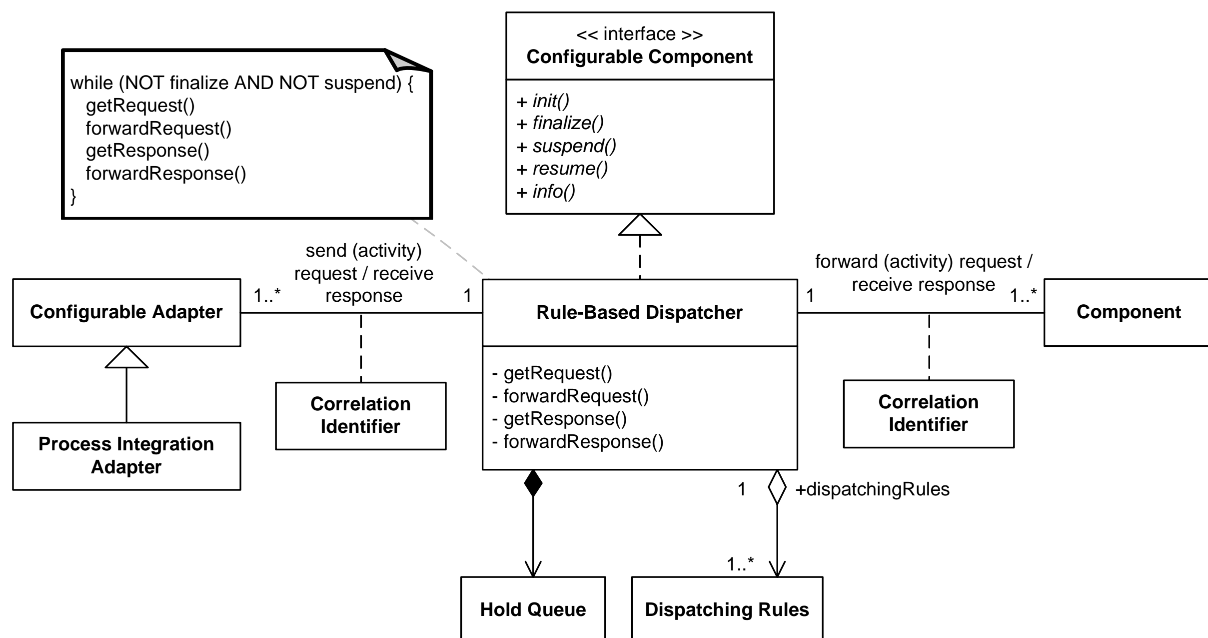


Figure 16: Rule-based dispatcher exemplary design

In case the component to execute the macroflow activity request is a MICROFLOW ENGINE, forwarding the request means invoking a MICROFLOW EXECUTION SERVICE. It is also possible that the dispatcher does some additional message transformation to suit the (service) interface of a component. However, this transformation logic should be kept as simple as possible, for the purpose of separation of concerns, as transformation is rather the task of the PROCESS INTEGRATION ADAPTER. Instead the functional interfaces of the components, which are used to execute the requests, should be designed accordingly, to avoid additional transformations.

The RULE-BASED DISPATCHER pattern mediates service requests of the macroflow with corresponding microflow services. It is an optional design element in a PROCESS-BASED INTEGRATION ARCHITECTURE, where it is typically placed between a PROCESS INTEGRATION ADAPTER and a MICROFLOW ENGINE that offers MICROFLOW EXECUTION SERVICES. The RULE-BASED DISPATCHER applies the COMPONENT CONFIGURATOR [Schmidt et al. 2000]

pattern to be configurable at runtime, and the CORRELATION IDENTIFIER [Hohpe et al. 2003] pattern is used to correlate service requests and responses.

The RULE-BASED DISPATCHER pattern has the following benefits: It supports the flexible dispatch of requests based on configurable rules. These dispatching rules can be changed at runtime by suspending the dispatcher and updating the rules. The workload can be managed by scaling the architecture in terms of adding instances of components to execute the requests, and all components can be changed independently.

The RULE-BASED DISPATCHER pattern also has the following drawbacks: A RULE-BASED DISPATCHER might be superfluous in case one can be sure that no more than one component will be used to execute the activity requests. To realize the pattern, additional specification and design effort might be necessary for defining the language and format of the dispatching rules. A central component like a RULE-BASED DISPATCHER is a single-point-of-failure. It might be a bottleneck and hence have a negative influence on the performance of the whole system.

Some known uses of the pattern are:

- Using IBM's WebSphere Business Integration Message Broker a RULE-BASED DISPATCHER can be implemented with a message flow definition that represents the dispatching logic. The dispatching rules are stored in a database and are accessed via a database access node in the flow.
- The Service Container of the Mule Enterprise Service Bus [Mule 2007] offers support for content-based and rule-based routing. Inbound and outbound message events, as well as responses, can be routed according to declarative rules that can be dynamically specified. A number of predefined routers are available (based on the patterns in [Hohpe et al. 2003]). Pre-defined (or user-defined) filters, like a payload type filter or an XPath filter, can be used to express the rules that control how routers behave.
- Apache ServiceMix [ServiceMix 2007] is an open source Enterprise Service Bus (ESB) and SOA toolkit. It uses the rule-language Drools to provide rule-based routing inside the ESB. The architecture is rather simple: A Drools component is exposed at some service, interface, or operation endpoint in ServiceMix and it will be fired, when the endpoint is invoked. The rule base is then in complete control over message dispatching.

CONFIGURABLE ADAPTER

Systems, such as business applications, must communicate with other systems.

When different systems are interconnected and the individual systems evolve over time, the system interfaces change. Often many systems are affected by an interface change, and thus each change causes high maintenance efforts and costs. Many changes cannot even be avoided because they are caused in external vendor systems.

Often the API of a business system changes with each new release of the system. If the business system has to communicate with other systems, the functional interfaces of those systems, as well as the supported requested formats and technological interfaces must be connected to the business system. In many cases such integrated systems are independent components that may change over time, and thus their interfaces may change as well. The effect is multiplied if several systems are dependent on the interface of another system.

In this context of ongoing change and maintenance, the costs and efforts of changes should be kept at a minimum level. The impact of changes and the related testing efforts must also be kept within acceptable limits.

Sometimes it is possible to circumvent these problems by avoiding changes that influence other systems. Such changes cannot be avoided, however, if the business applications under considerations are systems of external vendors, like SAP, for instance, and changes that occur with a new version must be taken into account. Migration to a new release is often forced as old releases are not supported anymore, or the new functionality is simply required within the business context.

Apart from migration to a new version, the problem also occurs if a business application shall be replaced by a completely different system. In such cases, the technology and functional interfaces of the new system are often highly different, causing a significant change impact.

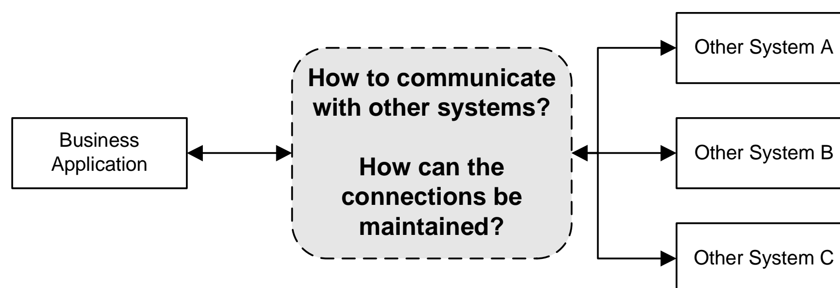


Figure 17: How to communicate with other systems

Implement a CONFIGURABLE ADAPTER to another system that should get connected. The adapter abstracts the specific interface (API) of that system. Make the adapter configurable, by using asynchronous communication protocols and following the COMPONENT CONFIGURATOR pattern, so that the adapter can be modified at runtime without impacting the systems sending requests to the adapter.

To make the adapter configurable it must be loosely coupled to other systems, which is in first place achieved by asynchronous communication protocols. As requests must be accepted at any

time, no matter whether an adapter is at work or temporally suspended, an asynchronous connector should be used to receive the requests and to send the responses. That is, the connector must be decoupled from the adapter to still accept requests in case an adapter is not active.

Basically, asynchronous communication is only really required on requestor side, i.e., for systems that access the adapter. The system, the adapter applies to, does not necessarily need to be connected asynchronously. For instance, a connected system might only offers a synchronous API, or the system is a database which is connected via synchronous SQL. That also means, the connector may accept requests and queue them until they are processed by the adapter. Requests and responses are related by applying the CORRELATION IDENTIFIER pattern [Hohpe et al. 2003]. That is, the adapter is responsible for putting the same correlation ID that is sent with the request into its response, so that the connected system can relate the response to its respective request.

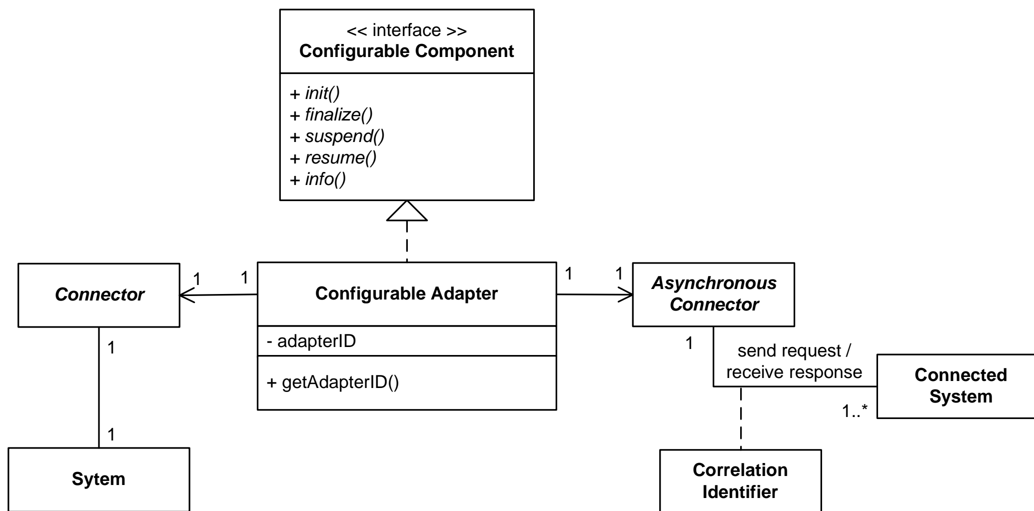


Figure 18: Exemplary structure of the configurable adapter

The adapter design should follow the COMPONENT CONFIGURATOR [Schmidt et al. 2000] pattern to configure the actual adapter. Examples of re-configuration are to replace an old adapter with a different adapter, update an adapter with a new version or configuration, or temporarily suspend the adapter from processing in case the connected system is currently under maintenance. Figure 18 illustrates an example structure of a CONFIGURABLE ADAPTER.

A CONFIGURABLE ADAPTER is very useful for flexible integration of business applications from external vendors. It also gets more popular to provide interconnectivity by supporting generic adapters for common standards, such as XML and Web Services. That is the reason why many vendors deliver such adapters off-the-shelf and provide open access to their APIs. As standard adapters can be provided for most common standards or products, solutions following the CONFIGURABLE ADAPTER pattern are usually reusable.

The BUSINESS-DRIVEN SERVICE pattern is typically realized using the CONFIGURABLE ADAPTER pattern: In this pattern a business service is offered by a business application, which is implemented via a CONFIGURABLE ADAPTER.

The PROCESS INTEGRATION ADAPTER is a specialisation of the CONFIGURABLE ADAPTER pattern. It describes how to connect a process engine with some target system and process requests from process activities being executed on that engine.

The CONFIGURABLE ADAPTER pattern has the following benefits: It enables the flexible connection of one system to another. Because the adapters can be configured at runtime, new

versions of the adapter can be deployed in a controlled way without affecting the connected systems.

The CONFIGURABLE ADAPTER pattern also has the following drawbacks: Potentially many CONFIGURABLE ADAPTERS needs to be managed, if many systems exist where adapters for different purposes, systems, or technologies are required. Hence, maintenance and deployment of adapters might become problematic and must be done in a controlled way. The CONFIGURABLE ADAPTER REPOSITORY offers a way to manage those adapters in a centralised and controlled way.

Some known uses of the pattern are:

- WebSphere InterChange Server offers a very large set of CONFIGURABLE ADAPTERS for most common technologies and applications. Users can extend the set of adapters with self-defined adapters.
- The transport providers of the Mule ESB [Mule 2007] provide CONFIGURABLE ADAPTERS for transport protocols, repositories, messaging, services, and other technologies in form of their connectors. A connector provides the implementation for connecting to an external system. The connector sends requests to an external receiver and manages listeners to receive responses from the external system. There are pre-defined connectors for HTTP, POP3/SMTP, IMAP, Apache Axis Web Services, JDBC, JMS, RMI, and many other technologies. Components can implement a common component lifecycle with the following lifecycle interfaces: Initialisable, Startable, Callable, Stoppable, and Disposable. The pre-defined connectors implement only the Disposable and Initialisable interfaces.
- iWay's Universal Adapter Suite [iWay 2007a] provides so-called intelligent, plug-and-play adapters for over 250 information sources and broad connectivity to multiple computing platforms and transport protocols. It provides a repository of adapters, an special-purpose MICROFLOW ENGINE for assembling adapters called the Adapter Manager, a graphical modelling tool for adapter assembly, and integration with the MACROFLOW ENGINES and EAI frameworks of most big vendors.

CONFIGURABLE ADAPTER REPOSITORY

Various systems shall be connected via adapters in a larger architectural context.

Many systems require a larger number of adapters, such as CONFIGURABLE ADAPTERS or PROCESS INTEGRATION ADAPTERS. To manage these adapters is an issue, often simply because of the sheer mass of adapters that need to be maintained. In addition, when the adapters evolve, new adapter versions need to be supported as well, meaning that actually multiple versions of each adapter need to be maintained. This also introduces a deployment issue: Usually connected systems should not be stopped for deploying a new adapter or adapter version, instead it should get "seamlessly" deployed at runtime.

Adapters are important to connect systems that have incompatible interfaces. However, if there are several systems being connected with each other via adapters, it is actually an issue to deploy an adapter update or implement a new adapter, for instance, as those systems may keep on sending requests to the adapters that must be processed. It is often not acceptable to stop all the connected systems just to deploy an adapter modification.

Often it is also not acceptable that implementations of a new or modified adapters result in a recompilation and redeployment of some larger parts of the components. The recompilation and redeployment should be limited to the adapter that needs to be modified. Maintaining adapters and controlling adapter functionality should be easy and of low effort. For instance, if a connected system is temporarily disabled this should have minimum effect on the systems that need to communicate with the disabled system.

Figure 19 illustrates that this problem especially occurs in larger architectural contexts, where different systems have to communicate and larger sets of adapters exist. The problem does not have such a great impact within the boundaries of one closed component or application, as the whole component or application needs to be redeployed if changes are made.

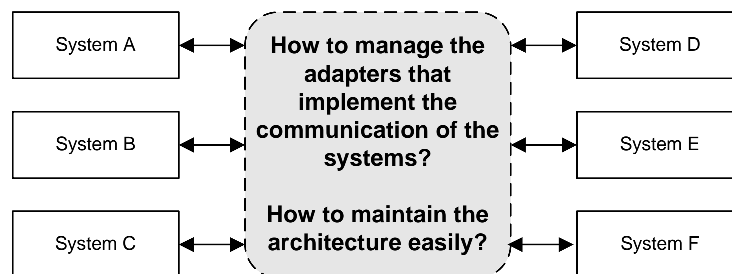


Figure 19: Problems in managing adapter changes

Use a central repository to manage the adapters as components and design the adapters as CONFIGURABLE ADAPTERS such that they can be modified at runtime without affecting the components or systems sending requests to the adapters. The CONFIGURABLE ADAPTER REPOSITORY manages the access to its adapters based on the configuration state of the adapter. That is, for instance, requests sent to an adapter that is suspended for maintenance are queued until the adapter is available again.

Adapters can be stored in a central repository that offers operations to add, retrieve, and remove adapters, or even to search for adapters by given attributes. The adapters in the

repository are CONFIGURABLE ADAPTERS (or a specialization of this pattern, such as PROCESS INTEGRATION ADAPTERS). That is, maintenance or deployment tasks are supported because each single adapter can be stopped and restarted, new adapters can be deployed, and old adapters can be removed via a centralised administration interface.

It is important that requests sent to adapters are processed asynchronously (see CONFIGURABLE ADAPTER pattern) to bridge maintenance times when the adapters are modified. The requests are queued while the adapter is suspended. The pending requests can be processed when the adapter restarts work after maintenance, or after an adapter is replaced by a new adapter. Figure 20 illustrates an example structure of a CONFIGURABLE ADAPTER REPOSITORY.

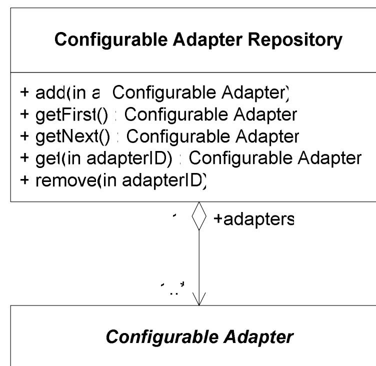


Figure 20: Exemplary structure of a configurable adapter repository

The CONFIGURABLE ADAPTER REPOSITORY pattern has the following benefits: The pattern addresses the flexible management of adapters (at runtime). Following the pattern, changes to adapters can be implemented rather quickly and easily.

The CONFIGURABLE ADAPTER REPOSITORY pattern also has the following drawbacks: The pattern requires changing the adapters because an administration interface is necessary for maintaining the adapters. If an adapter is suspended for a long time or if the amount of requests sent to a suspended adapter is very high, then the request queue may contain large amounts of requests that take a long time to be processed or the requests may even have timed out. The workload of requests and the amount of requests that an adapter can process must be in balance. Middleware is required to queue the requests.

Some known uses of the pattern are:

- WebSphere InterChange Server offers a CONFIGURABLE ADAPTER REPOSITORY where the pre-defined large set of CONFIGURABLE ADAPTERS, as well as self-defined adapters, can be added, for instance.
- The connectors of transport providers of the Mule ESB [Mule 2007] are, like all other components in Mule, managed either by the Mule container or an external container like Pico or Spring. The container manages the lifecycle of the connectors using the component lifecycle interfaces, which the components can optionally implement. Thus the container acts as a CONFIGURABLE ADAPTER REPOSITORY for the connectors.
- iWay's Universal Adapter Suite [iWay 2007a] provides a repository of adapters in the Adapter Manager [iWay 2007b]. The graphical modeller of iWay, the Adapter Designer, is used to define document flows for adapters. The Adapter Designer can be used to maintain and publish flows stored in any Adapter Manager repository. The adapters in the repository can be deployed to the Adapter Manager, which is the MICROFLOW ENGINE used for executing the Adapter flows.

MICROFLOW EXECUTION SERVICE

Microflows are exposed to macroflows (and other microflows) following the MACRO-MICROFLOW pattern.

A microflow is a function that actually represents a service. Consequently, the microflow should be exposed as a coherent function with defined input and output parameters. This might be difficult because developers have to consider the technology specifics of the MICROFLOW ENGINE being used. The engine's technology specifics might be hard to decouple from the actual functionality that it offers to execute concrete microflows. In addition, different microflows in a PROCESS INTEGRATION ARCHITECTURE typically require common functionality that is not offered by the MICROFLOW ENGINE. This common functionality should not be reimplemented for each microflow.

A MICROFLOW ENGINE usually offers some kind of API to access the engine and initiate the execution of a microflow model. This API depends on the technology specifics of the engine being used. However, as it is necessary in a PROCESS INTEGRATION ARCHITECTURE to make the functionality represented by microflows accessible to arbitrary systems that require the functionality, the MICROFLOW ENGINE'S external interface should remain independent of these technology specifics.

In the context of a PROCESS INTEGRATION ARCHITECTURE**,** microflows might include some activities which many microflows generally have in common. For instance, the BUSINESS OBJECT REFERENCE pattern suggests that the business objects containing the business data are principally stored in a repository outside the process engine. To orchestrate and invoke backend services in a microflow, business data is used as parameters of these backend services. Thus fetching the business data is a typical standard functionality. The question is what this general functionality in microflows is and whether it can be captured somehow. The goal is to build it only once and re-use it as a common framework.

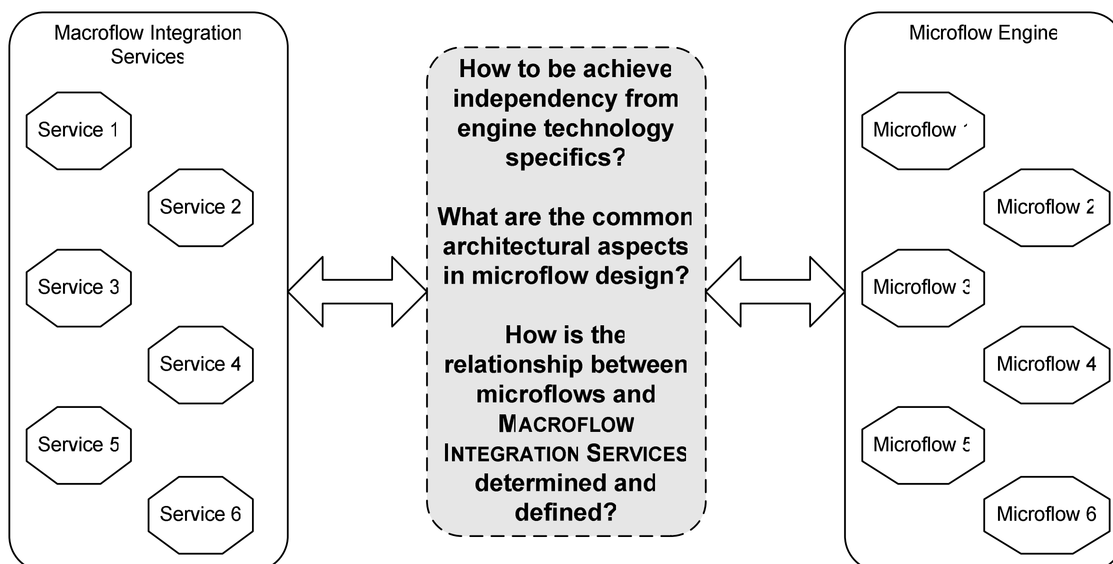


Figure 21: Three principle questions regarding microflow

Additionally, MACROFLOW INTEGRATION SERVICES represent functions requested by activities in a macroflow. As a microflow is a refinement of a macroflow activity, there must be some kind of relationship between a MACROFLOW INTEGRATION SERVICE and a microflow. The relationship must be defined in a way that offers enough flexibility as far as accessibility, scalability, and configuration issues are concerned. The problem is illustrated in Figure 21.

Expose a microflow as a MICROFLOW EXECUTION SERVICE that abstracts the technology specific API of the MICROFLOW ENGINE to a standardised well-defined service interface and encapsulates the functionality of the microflow. Define the interface of this service according to a particular MACROFLOW INTEGRATION SERVICE, as there is a one-to-one relationship between the two services that corresponds to the activity the microflow is related to. Each microflow typically invokes standard functions to access the data of business objects, transform this data according to the interface requirements of invoked BUSINESS-DRIVEN SERVICES, and write possible results back to business objects.

For each microflow model, design a service that encapsulates the functionality of the microflow model and has the required input and output parameters in its service interface. This type of service is called MICROFLOW EXECUTION SERVICE.

A MACROFLOW INTEGRATION SERVICE corresponds to an activity in a macroflow. In case this activity represents an automatic business function that is executed by backed systems, there is a one-to-one relationship to a MICROFLOW EXECUTION SERVICE that encapsulates the corresponding microflow of the macroflow activity. The interfaces of these two related services must be aligned because they actually represent the same function at different levels of integration and in relation to different architectural components. Ideally, the service interfaces match exactly. But transformations can also be performed by a RULE-BASED DISPATCHER (or some other intermediary component) that works as a link between the two services. Any kind of unnecessary transformation, however, will make the dispatcher more complex and should be avoided, if possible.

If a MICROFLOW EXECUTION SERVICE is invoked, the microflow that is encapsulated by the MICROFLOW INTEGRATION SERVICE will be executed on a MICROFLOW ENGINE. To achieve this, the microflow model that corresponds to the service must be identified and can then be executed by the engine. The API of the engine usually identifies the model by a unique name. To abstract the technology specific API of the engine, the MICROFLOW EXECUTION SERVICE connects to a MICROFLOW ENGINE via a CONFIGURABLE ADAPTER. Figure 22 shows an example structure of a MICROFLOW EXECUTION SERVICE.

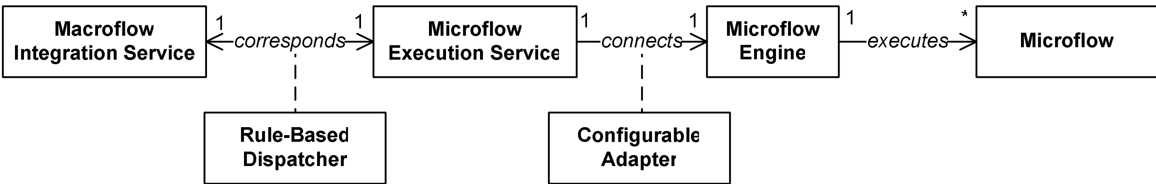


Figure 22: Exemplary structure of a Microflow Execution Service

Microflow models often orchestrate service invocations of BUSINESS-DRIVEN SERVICES. BUSINESS-DRIVEN SERVICES use business objects that are stored in a container outside a process engine, following the BUSINESS OBJECT REFERENCE pattern [Hentrich 2004]. In this context, five typical, reoccurring steps can be identified in microflow models:

1. To invoke a BUSINESS-DRIVEN SERVICE, business data is required, which must be retrieved from business objects that are accessed via their BUSINESS OBJECT REFERENCES.
2. Some kind of transformation must be performed to invoke a service, as the service interface may not match the structure of business objects.
3. The actual service invocation is (perhaps optionally) performed.
4. The result of the service invocation is transformed into a business object.
5. The business object is saved back into a container outside the engine to keep the result.

Not necessarily all these five steps occur, but if they occur, they will occur in this order. Some steps may be left out. For instance, in case the service interface directly understands the format of the business objects, no transformation is needed. Or, if there is read access only to business objects they will not be saved. In any case, the MICROFLOW EXECUTION SERVICE pattern implementation should offer the common functionality of these steps such that microflow models can reuse the implementations of these steps.

The MICROFLOW EXECUTION SERVICE pattern has the following benefits: Microflows can be flexibly invoked as services. The design of the interfaces of MICROFLOW EXECUTION SERVICES corresponds to the design of the interfaces of MACROFLOW INTEGRATION SERVICES; hence understandability of the overall architecture is supported. The services are decoupled from their implementation, and it is possible to change the engine that implements the services. The technology specifics of the MICROFLOW ENGINE are hidden.

The MICROFLOW EXECUTION SERVICE pattern also has the following drawbacks: The technology specific API of a MICROFLOW ENGINE must be abstracted and additional effort might be necessary to implement a service-oriented view on the MICROFLOW ENGINE.

Some known uses of the pattern are:

- iWay's Adapter Manager [iWay 2007b] is an engine that allows clients to access adapter-specific microflows via various protocols, such as TCP, SOAP, TIBCO, JMS, MQ, etc. A listener is provided, which invokes the adapter flow representing the microflow.
- Within WebSphere Business Integration Developer a microflow can be exposed as a service via the Service Component Architecture (SCA) concept. SCA supports Web services, JMS, or MQ bindings of the services.
- In a SOA project for a telecommunications customer, the pattern has been used to expose service orchestration flows implemented with WebSphere Business Integration Message Broker at the service bus.
- In an insurance company in Switzerland the pattern is used in the company's global SOA standard as to expose complex service orchestration flows as services based on BEA Aqualogic.

MACROFLOW ENGINE

The macroflow concept is used to represent long-running business processes.

Business processes need to adapt, for instance, due to changed market conditions or business optimization initiatives. In a dynamic business environment where business process changes are regular practice, the IT systems must cope with the pace of business process changes. This cannot be reached, if the business processes are statically implemented in IT systems, and long and costly development cycles are needed for changing the business process implementations. Thus, to provide organisationally flexible business processes, macroflows must be easily configurable to reduce implementation time and effort of these business process changes.

Changes to business are reflected by changes in the corresponding business processes. Today a lot of IT systems support business processes, and the required changes often involve significant changes in IT systems with high costs and long timelines. In a dynamic business environment, these costs and long timelines are often not acceptable, as conditions might have already changed when old requirements are implemented and new requirements have already popped up. The time to react on change requirements is simply too long and the involved costs are too high.

One of the major reasons for this problem is that business process logic is statically implemented in IT systems, i.e., in the program code of these systems. The required changes thus imply to change program code in various systems. Often a lot of different skills are required to achieve this, as the systems are implemented on varying platforms with different technology and applying different programming paradigms and languages. The heterogeneity of systems and the different concepts of these systems also imply difficulties to the end-users, who have to learn the adaptations of the changed systems. Often the desired business process, as it was originally designed, cannot be realized due to limitations given by existing systems and/or because of the high efforts required to implement the changes.

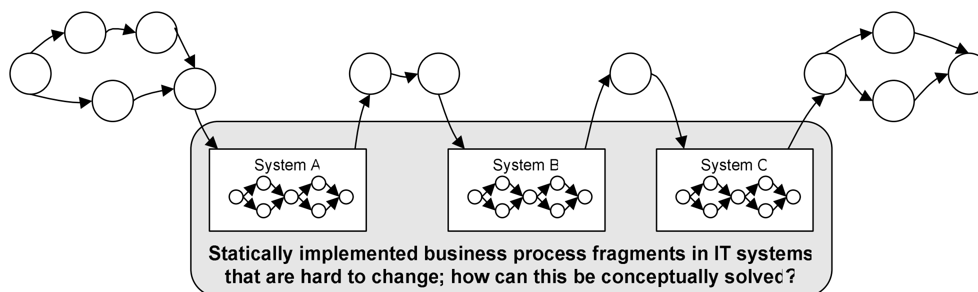


Figure 23: Fragmentation of business processes

The complexity generated by this heterogeneity and the interdependencies between the systems let projects fail even before they have started, as the involved risks and the costs may be higher than the estimated benefit of the business process change. Thus incremental evolution cannot be achieved. As a result, IT has gained the reputation of just being a cost driver but not a business enabler anymore. This is very often the reason why no significant and innovative changes are made, but solving prevalent problems is postponed as long as possible and changes are reduced to the most important maintenance activities. The reason is that regular business process changes have not yet been conceptually considered enough in IT.

Additionally, often a fragmentation of business processes in conjunction with IT systems that support them occurs, if business process logic is statically implemented in IT systems. This happens because the IT systems are not designed according to the higher-level business processes' structure, but instead the IT systems make certain assumptions about the business processes' structure. Business process changes often lead to situations, where the technological assumptions turn out to be invalid, and as quick changes of the IT are not possible, a fragmentation (or structural gap) between the business process models and the IT systems occurs. Figure 23 illustrates the problem.

Extract the statically implemented business process logic from systems, and model the business processes in a business process modelling language. Delegate the macroflow aspects of the business process definition and execution to a dedicated MACROFLOW ENGINE that executes the business processes described in the business process modelling language. The engine allows developers to configure business processes by flexibly orchestrating the execution of macroflow activities and the related business functions.

As organisational inflexibility results from business process fragmentation and static implementation of business process fragments in IT systems that are hard to change, the business process logic is extracted from IT systems. Control of the business processes is delegated to a dedicated component, the MACROFLOW ENGINE. This component allows developers to configure the business process logic by easily changing the business process definitions and executing the defined business processes. This implies that applications will be understood as modules that offer business functions (services) and can be orchestrated by business process logic. The MACROFLOW ENGINE realizes the following distinct features of macroflow definition and execution:

- Supports full-automatic and semi-automatic macroflow activities with human interaction.
- Offers an API to access the functionality of the engine, i.e., processing of automatic and semi-automatic tasks.
- Offers functions for macroflow definition.
- Offers functions for long-running macroflow execution.
- Concentrates on orchestration issues of macroflow activities but not on the implementation of these activities; the actual implementation of macroflow activities is delegated to functionality of other systems that the engine communicates with.

Various concepts are used to achieve the orchestration of macroflow activities in a MACROFLOW ENGINE. Examples are:

- Strictly structured process flow, e.g., in terms of directed graphs with conditional paths
- Flexibly structured flow of activities, e.g., by defined pre- and post-conditions of macroflow activities

Macroflow definitions can be made changeable by applying a process definition language. Macroflows are in this case modelled in the process definition language and are imported in the engine that executes the models. Changes occur via modified models of macroflows.

A macroflow definition consists of macroflow steps. The steps transform data that is used to control the orchestrations of macroflow activities and invoke a business function of an IT system, where the function can either be:

- Completely automatic
- Semi-automatic representing a human interacting with a system

For this reason, a macroflow step is assigned to a resource, where a resource can be some virtual actor like an IT system acting in a certain role, or a human actor who interacts with an IT system. As far as a human actor is concerned, constraints may be applied to make the macroflow step only accessible to a defined set of users, e.g., by roles and rights that a user must have in order to be able to process a macroflow step. However, these issues rather relate to concepts of a concrete MACROFLOW ENGINE implementation.

The macroflow step thus always invokes a business function, whether the business function is executed with support of a human being or whether it is completely automatic. Figure 24 shows an exemplary, general structure of a MACROFLOW ENGINE.

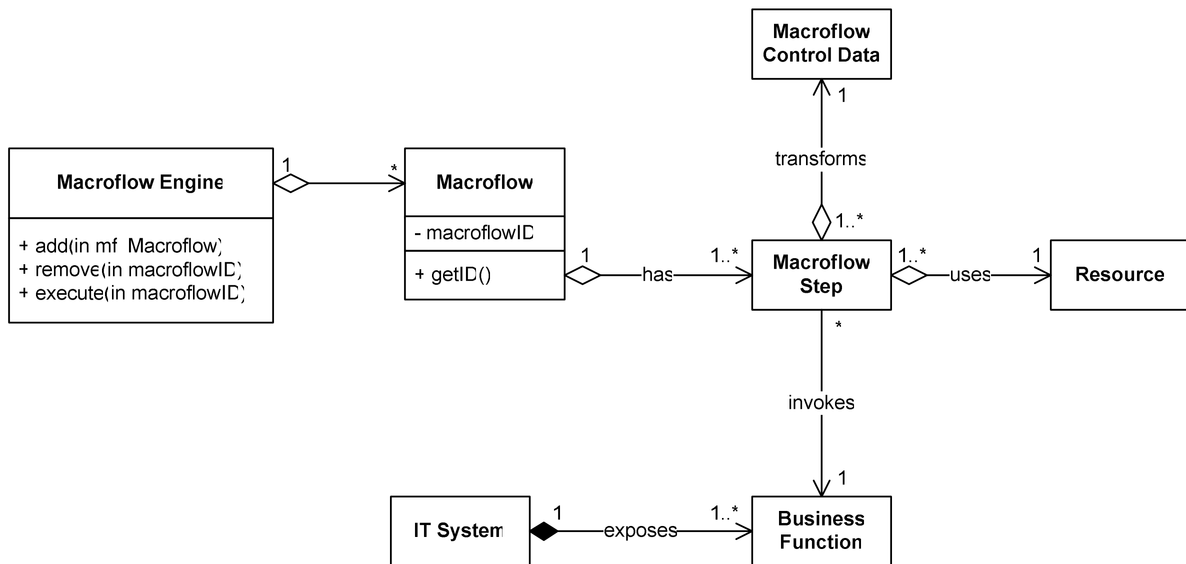


Figure 24: Exemplary structure of a Macroflow Engine

The MACROFLOW INTEGRATION SERVICE pattern represents a service-oriented approach to designing full-automatic business functions and invoking them in macroflows that are executed on a MACROFLOW ENGINE. To invoke these services, the PROCESS INTEGRATION ADAPTER connects the MACROFLOW ENGINE in the context of a PROCESS-BASED INTEGRATION ARCHITECTURE und establishes communication with backend systems via the API of the engine.

The MACROFLOW ENGINE pattern has the following benefits: The business process logic is architecturally decoupled from the IT systems. Business process definitions can be flexibly changed and the corresponding processes in IT systems can be adapted more easily. Organisational flexibility is conceptually supported by applying the business process paradigm to IT architecture. Hence, an IT strategy to organisational flexibility is supported.

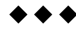
The MACROFLOW ENGINE pattern also has the following drawbacks: Efforts must be invested to extract the business process logic out of existing systems implementations and to modify the architecture. The approach has best effects if applied as long terms approaches to architecture design and application development. Short term goals may not justify the efforts involved.

Some known uses of the pattern are:

- IBM's WebSphere Process Choreographer is the workflow modelling component of WebSphere Studio Application Developer Studio, Integration Edition, which provides a MACROFLOW ENGINE. The workflow model is specified in BPEL.
- In the latest WebSphere product suite edition, the two products WebSphere Process Choreographer and WebSphere InterChange Server have been integrated into one product which is called WebSphere Process Server. Consequently, this new version offers both, a MACROFLOW ENGINE and a MICROFLOW ENGINE.
- GFT's BPM Suite Inspire [GFT 2007] provides a designer for macroflows that is based on UML activity diagrams. The business processes can be deployed to an application server that implements the MACROFLOW ENGINE for running the business processes. The engine also offers an administrator interface for monitoring and management of the processes.
- JBoss' jBPM [JBoss 2007] is an open-source MACROFLOW ENGINE for graph-based business process models that can be expressed either in jPDL or BPEL as modelling languages.
- ActiveBPEL [Active Endpoints 2007] is an open-source BPEL engine that acts as a MACROFLOW ENGINE for business processes modelled in BPEL.

MICROFLOW ENGINE

The microflow concept is used to represent short-running, technical processes.



In a dynamic environment, where IT process changes are regular practice, it takes considerable time and effort to realize and change technical IT integration processes, if the microflow details are statically implemented. But, to provide organisationally flexible IT integration processes, microflows must be configurable, and changes must be quickly implemented. Furthermore, the IT integration processes must be conceptually aligned to the business processes they realize to make the process-driven SOA as a whole understandable and flexible.

Modifications in business processes often result in changes to the IT-related integration processes (represented as microflows). As a consequence, in dynamic environments where business process changes are regular practice, the corresponding changes to IT integration processes must be achieved with minimum effort, in terms of time and budget. For instance, if the communication structures are statically implemented by point-to-point connections between systems, it is very hard to realize a desired process change. This is because the whole architecture is highly coupled and thus a single change requirement often results in modifications in many systems. Hence, changes imply a high effort, and high risks are involved. These issues are related to integration concepts that follow the idea of a messaging backbone that enables point-to-point communication by exchanging messages.

The concept of an integration hub solves some of the technical issues and improves flexibility by offering centralized management of the connections. However, still a variety of different and heterogeneous interfaces, protocols, and technologies needs to be managed, and there is still no direct relationship to the business processes. The issues being solved rather remain on a technical level, and the solutions are not business-driven.

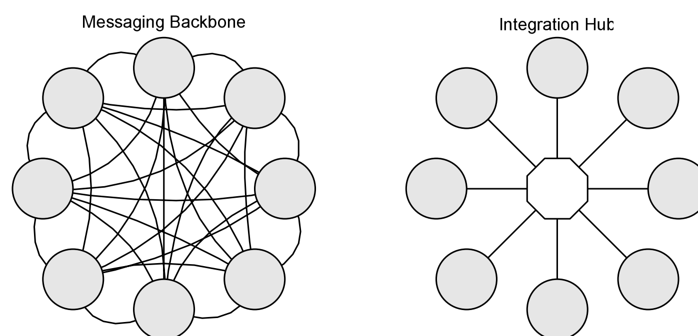


Figure 25: Integration concepts

A more business process driven approach to IT systems integration is thus required, which, on the one hand, focuses on the technical IT system integration issues but which, on the other hand, also bridges to the higher level business process view.

Apply the business process paradigm directly to IT integration process design and implementation by decoupling the integration logic. Delegate the microflow aspects of the business process definition and execution to a dedicated MICROFLOW ENGINE that

supports configuration of microflows by flexibly orchestrating the execution of microflow activities and the related BUSINESS-DRIVEN SERVICES.

According to the MACRO-MICROFLOW pattern the more technical IT integration processes relate to the microflow level. Microflows are thus special types of processes that are conceptually related to business process reflected at the macroflow level. To achieve organisational flexibility at the microflow level, integration logic is delegated to a dedicated component, the MICROFLOW ENGINE. This component allows developers to configure the integration logic by easily changing the integration process definitions and executing the defined integration processes. This implies that connected systems will be understood as modules that offer functions (services) to be orchestrated by integration process logic.

As the component represents the microflow aspects of business process, certain features are supported that are representative for microflow definition and execution:

- Supports full-automatic transaction safe integration processes only.
- Support of technology and/or application adapters, e.g. ODBC, JDBC, XML, Web service, SAP, Siebel.
- Offers an API to access the functionality of the engine.
- Offers functions for microflow definition.
- Offers functions for short-running transactional microflow execution.
- Concentrates on orchestration issues of microflow activities but not on implementation of these activities; the actual implementation of microflow activities is delegated to functionality of integrated systems the engine communicates with.

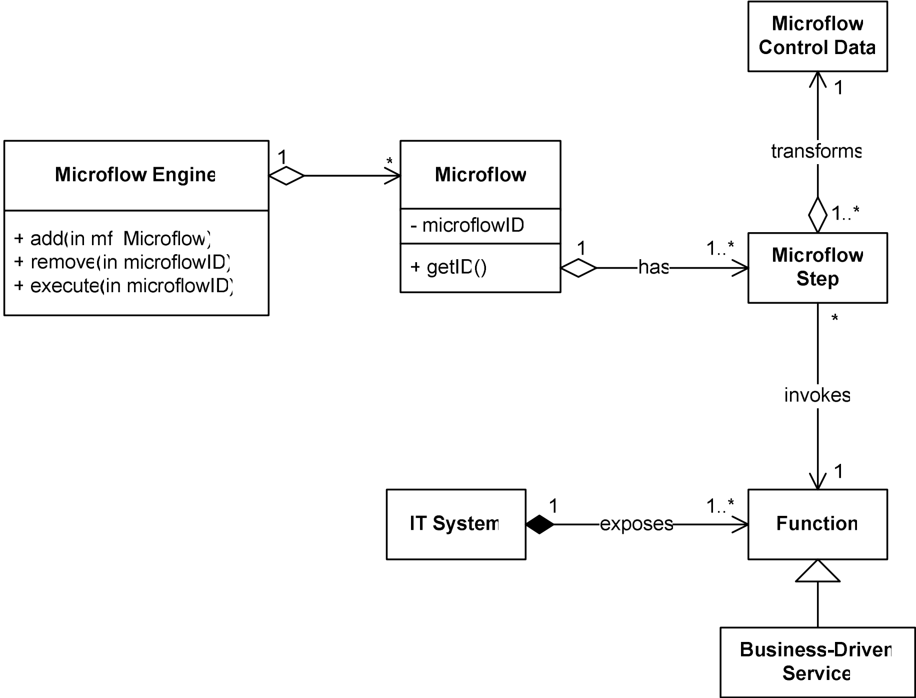


Figure 26: Structure of a Microflow Engine

The basic feature of a MICROFLOW ENGINE is execution of defined microflow integration process logic by orchestrating microflow activities. Analogous to the MACROFLOW ENGINE, the

concepts to achieve that orchestration may vary. The engine fulfils the tasks of executing integration logic and for defining it in a manner that can be flexibly changed.

Just as in the MACROFLOW ENGINE pattern, microflow definitions are changeable by applying the concept of a process definition language. A microflow definition consists of microflow activities. Each step transforms data that is used to control the orchestrations of microflow activities and invokes a function of an IT system. Such invocations are performed automatically and in a transaction-safe way. Within the SOA context this function is actually represented as a BUSINESS-DRIVEN SERVICE. Figure 26 shows the general structure of a MICROFLOW ENGINE.

To invoke these services, CONFIGURABLE ADAPTERS can be used to connect the MICROFLOW ENGINE with other systems in the context of a PROCESS-BASED INTEGRATION ARCHITECTURE. Often there are CONFIGURABLE ADAPTERS delivered off-the-shelf for most common technologies and applications with a MICROFLOW ENGINE middleware.

The MICROFLOW ENGINE pattern has the following benefits: The IT system's integration logic is architecturally decoupled. Integration processes can be flexibly changed. Organisational flexibility is conceptually supported by applying the business process paradigm to IT architecture, and thus an IT strategy to organisational flexibility is supported. Business processes are directly represented in the IT.

The MICROFLOW ENGINE pattern also has the drawback that the conceptual separation of integration logic might involve additional efforts, which might be too much, for small-scale problems.

Some known uses of the pattern are:

- The WebSphere Business Integration Message Broker and also the WebSphere InterChange Server both do represent MICROFLOW ENGINES. Both middleware products can also be used in conjunction.
    - Often the WebSphere Business Integration Message Broker is used for simpler functions, e.g., to implement a PROCESS INTEGRATION ADAPTER, to offer MACROFLOW INTEGRATION SERVICES, and a RULE-BASED DISPATCHER, as this product has strong features concerning off-the-shelf adapters, message routing, and transformation.
    - WebSphere InterChange Server has very strong features if used as a MICROFLOW ENGINE, as it offers transaction safe integration process execution. Process definition is done via a GUI and the product also offers a very large set of CONFIGURABLE ADAPTERS for most common technologies and applications. It also implements a CONFIGURABLE ADAPTER REPOSITORY where also self-defined adapters can be added, for instance.
- webMethods' Integration Server (now integrated in the Fabric BPM suite) [webMethods 2007] provides a MICROFLOW ENGINE that supports various data transfer and Web services standards, including JSP, XML, XSLT, SOAP, and WSDL. Its offers a graphical modeller for microflows that models the microflow in a number of sequential steps (including loop steps and branching), as well as a data mapping modeller.
- iWay's Universal Adapter Suite [iWay 2007a] provides an Adapter Manager [iWay 2007b] for its intelligent, plug-and-play adapters. The Adapter Manager is a component that runs either stand-alone or in an EJB container and executes adapter flows. The basic adapter flow is: It transforms an application-specific request of a

client into iWay's proprietary XML format, invokes an agent that might invoke an adapter or perform other tasks, and transforms the XML-based response into the application specific response format. The Adapter Manager provides a graphical modelling tool for assembling the adapters, the Adapter Designer. It allows developers to specify special-purpose microflows for a number of adapter-specific tasks, such as various transformations, routing through so-called agents, encryption/decryption, decisions, etc. Multiple agents, transformations, and decisions can be combined in one flow. The Adapter Manager hence provides a special-purpose MICROFLOW ENGINE focussing on adapter assembly.

BUSINESS-DRIVEN SERVICE

Services must be designed within the context of a service-oriented architecture approach.

In a SOA, the business goals must be properly mapped to services. This is difficult because on the one hand business goals must be broken down into requirements for services. On the other hand, we must also consider existing applications, technical resources, and off-the-shelf vendor products, when designing the services. We need a clear decision what services must be designed and implemented, considering the constant evolution of services.

In the service-oriented architecture (SOA) context it is rather difficult to decide what services are actually required to satisfy the business goals. Actually, it is even worse as the problem starts one step earlier: It is necessary to break down high-level strategic business goals into more fine-grained goals that must then be transformed into requirements for the services. For this reason, the problem is at first hand to decide on a de-composition of business goals and provide a rationale for this structure.

On the other hand, we are usually dealing with an existing IT infrastructure, and it is necessary to offer available application functionality as services. These applications imply limitations on the functionality, as an application will deliver only a certain set of functions that can be offered as services. The existing functionality must also be related to the strategic business goals to have a basis for the decision what existing functionality is ready to use. That means a priori it is not obvious whether this existing functionality is sufficient to support the business goals or whether additional functionality must be developed within single applications. It must be decided what functionality of what application must be extended and whether this is possible at all with an acceptable effort.

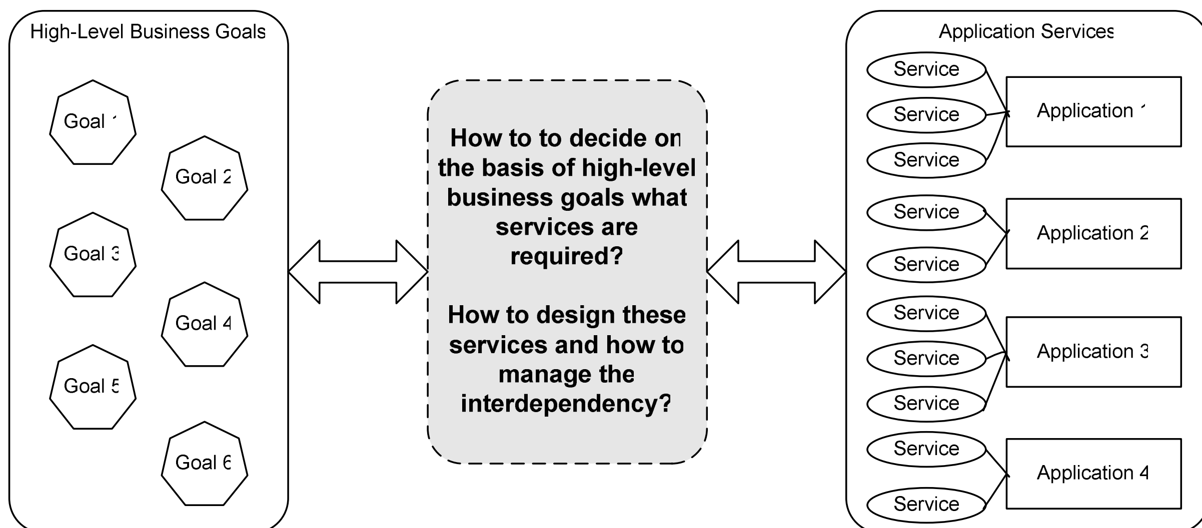


Figure 27: How to design services of applications that relate to business goals?

As far as off-the-shelf vendor products are concerned these issues consequently influence evaluation of vendor products and also the decision what product best suits the requirements, i.e., the selection of right products within the IT strategy context. The best strategic decision

might be to replace some of the existing applications, migrate to new versions, or leave some of the existing systems in their current state of functionality.

The decision process to decide on the requirements of services that need to be defined is a non-trivial process that has to consider evolutionary aspects of the business. An engineering approach is required to define and break down business goals, and map them to requirements of services. The approach should further allow for planning the definition and design of these services. It should also consider the evolution of the business, as the business goals might change over time.

Design BUSINESS-DRIVEN SERVICES that are defined according to a convergent top-down and bottom-up engineering approach, where high-level business goals are mapped to to-be macroflow business process models that fulfil these high-level business goals and where more fine grained business goals are mapped to activities within these processes. Those macroflow business process activities are further refined by engineered microflows that merge bottom-up and top-down engineered application services.

The high-level business goals being defined by strategic management can be mapped to to-be models of business processes. Following the MACRO-MICROFLOW pattern as a conceptual foundation to structure business processes, it is possible to design the to-be macroflow business processes that represent these high-level business goals. After that it is possible to derive MACROFLOW INTEGRATION SERVICES that need to be refined by microflow models. The design of these microflows is done by mapping existing application functions to services as follows:

- Existing services are modified, or
- new services that are missing are designed to fulfil the requirements of the MACROFLOW INTEGRATION SERVICES, or
- existing services are orchestrated in microflow models to realize a composed service that fulfils the requirements.

The result is a set of BUSINESS-DRIVEN SERVICES that are invoked and orchestrated in microflows, which fulfil the requirements of MACROFLOW INTEGRATION SERVICES.

It might turn out that the implementation of the to-be macroflow business processes implies unacceptable effort or exceeds the limits of the project budget and/or timeframe. A typical reason is that too many difficult modifications to existing services are necessary. Also, too much effort might be required for implementing new BUSINESS-DRIVEN SERVICES. In such cases, it might be necessary to adapt the to-be macroflows and thus the high-level business goals to stay within acceptable limits of the project budget and timeframe. Then a migration plan needs to be developed to achieve the original business goals and evolve the originally desired to-be macroflow business processes via several stages and/or projects.

That means, one should possibly plan for a few iterations until all interdependent elements of the various models match: the high-level business goals, the to-be macroflow business process models, the MACROFLOW INTEGRATION SERVICES, the microflows, and the BUSINESS-DRIVEN SERVICES. As far as architecture management is concerned, procedures and tools are very useful to manage these dependencies and the changes being initiated.

A BUSINESS-DRIVEN SERVICE in a microflow might be a composed service, where the BUSINESS-DRIVEN SERVICE invokes a set of more fine grained services (that possibly invoke even more fine grained services). This de-compositional structure should be considered as rather static and not subject to regular changes. If the orchestration of the more fine grained services needs to

be configurable, the orchestration will usually be modelled as a sub-microflow. Configurability is thus the design criterion to decide whether to have a static de-composition or a configurable de-composition via sub-microflows.

The MICROFLOW EXECUTION SERVICE exposes a microflow as a service and is thus composed of BUSINESS-DRIVEN SERVICES that are orchestrated in the microflow.

Figure 28 summarizes this interdependent and convergent top-down and bottom-up driven approach to identifying and designing BUSINESS-DRIVEN SERVICES.

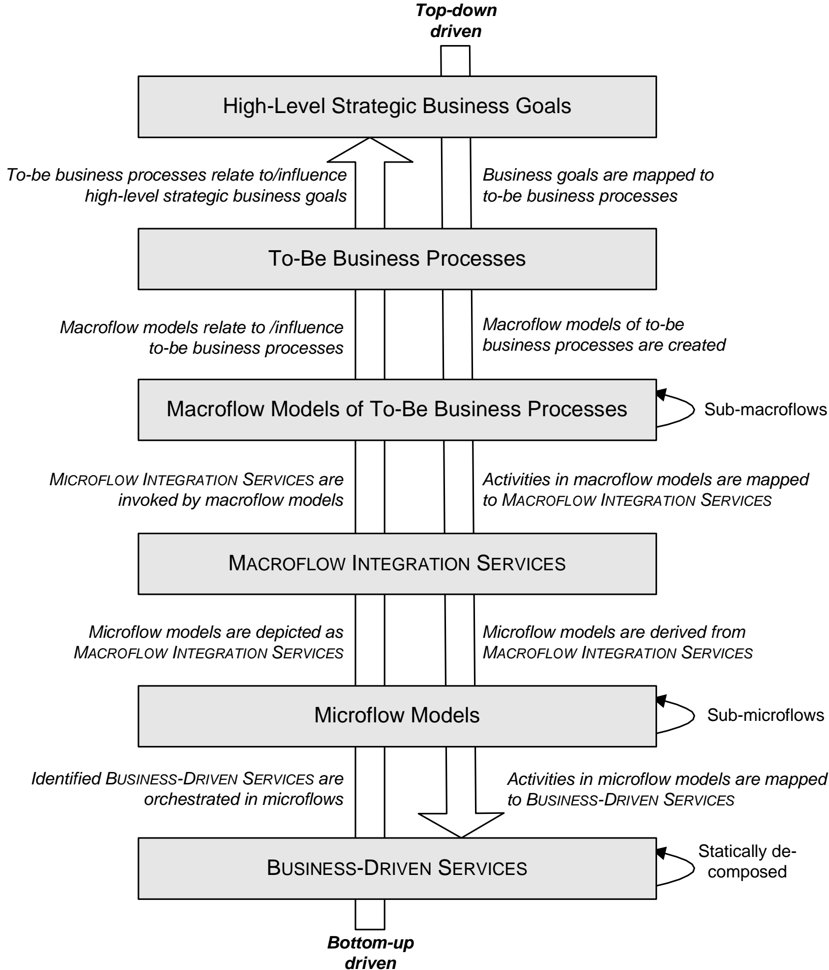


Figure 28: Convergent approach to designing services

The BUSINESS-DRIVEN SERVICE pattern has the following benefits: The pattern links services of business applications to high-level business goals. Identification and design of services is guided by a traceable convergent engineering approach – the pattern implies a methodology to design the services. Decisions on configurability of services are already included in the very early phase of service identification. Limitations of project budget and timeline are considered when creating a service model by considering the efforts to realize the services. Existing application functionality is considered to be offered as services and can be related to business processes.

The BUSINESS-DRIVEN SERVICE pattern also has the drawback that the convergent approach may imply several iterations that need to be planned in order to match the macroflow and microflow process models with the service model.

Some known uses of the pattern are:

- In IBM's WebSphere technology, the BUSINESS-DRIVEN SERVICES to be invoked during microflows that run on WebSphere Process Server are usually implemented as Web Services. However, service realisations using MQ messages or other supported technologies are also possible, if required.
- BOC [BOC 2007] offers an integration of its business modelling tool Adonis with Oracle's BPEL process manager, following an approach as advocated by the BUSINESS-DRIVEN SERVICES pattern. In this approach, Adonis provides modelling support for business processes and organizational structures, mainly from a pure business perspective. The BPEL process manager provides the engine for implementing the business processes and transactions. The process for transferring the design from the business to the BPEL process manager technology is automated. Transferring the technical representation back into the business representation (bottom-up) must be done manually. Also, lower-level microflow aspects ("below the BPEL process models") are not covered by BOC's approach.
- In IBM's Service-Oriented Architecture and Modelling (SOMA) method [Arsanjani 2004], the pattern is applied postulating a convergent top-down and bottom-up driven service modelling style.
- In a large programme in an insurance company the pattern has been applied to design a methodology for process-centric services analysis and design based on ARIS [IDS 2006].

Example of a Java Implementation of the Pattern Language

In following example, the patterns have been implemented in the context of a Java backend framework for process based services integration. Subject of this framework is the execution of automatic activities that integrate services. The framework consists of four elements which will be implemented in this example as architectural components:

- *Job Generator Engine* – transforms messages sent by process engines (MQ Workflow UPES messages in this case) into an enterprise wide standard job-definition format. The job-generator decouples the process engines (in this case MQ Workflow) from actual activity implementations. This component implements the PROCESS INTEGRATION ADAPTER and MACROFLOW INTEGRATION SERVICE patterns. WebSphere MQ Workflow implements the MACROFLOW ENGINE pattern.
- *Job Dispatcher Engine* – is responsible for prioritised distribution of generated jobs to different distributed components that are responsible for executing the jobs. Moreover, the dispatcher is responsible for sending the job results delivered from those components back to the job-generator engine. This component implements the RULE-BASED DISPATCHER.
- *Activity Execution Engine* – is responsible for executing generated jobs that represent activity implementations related to business services integration. This component implements the MICROFLOW ENGINE and the MICROFLOW EXECUTION SERVICE patterns.
- *Business Service Interface Repository* – offers standardised interfaces to integrated business services. This component implements the CONFIGURABLE ADAPTER REPOSITORY and the CONFIGURABLE ADAPTER patterns. The actual services have been designed according to the BUSINESS-DRIVEN SERVICE pattern.

The integration mechanism for the MQ Workflow will be the UPES concept. All four architectural components realize the UPES implementation in the context of MQ workflow.

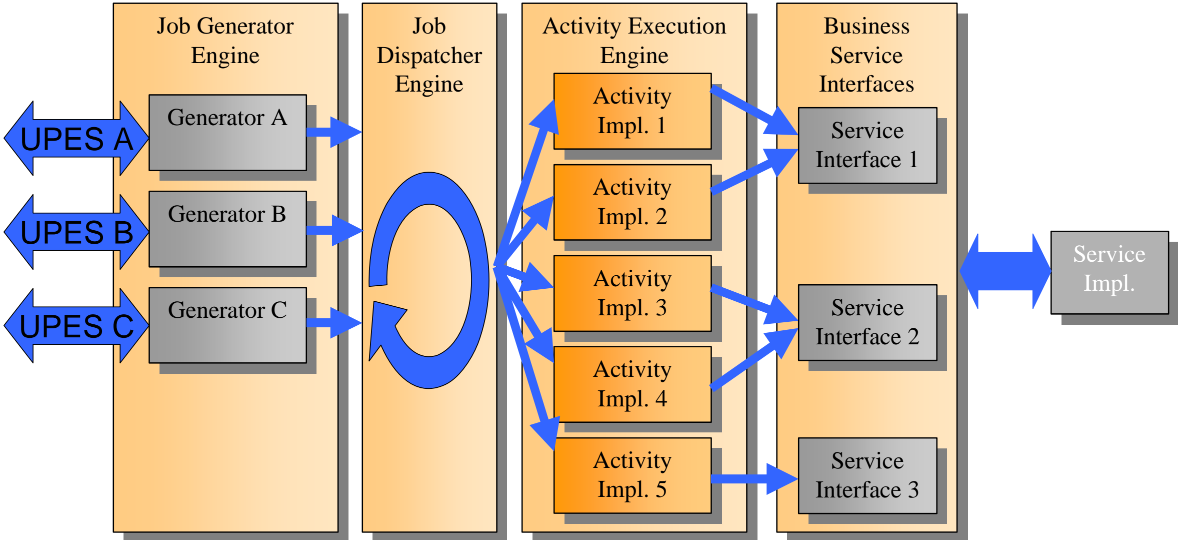


Figure 29 : Process based business services integration overview

The components are connected via asynchronous messaging mechanisms. However, as far as business services invocation and integration is concerned, the architecture considers both, synchronous and asynchronous business services. In order for the architecture to be scalable and flexible, there will be a one-to-many relationship between all five components (the four service integration components plus MQ Workflow).

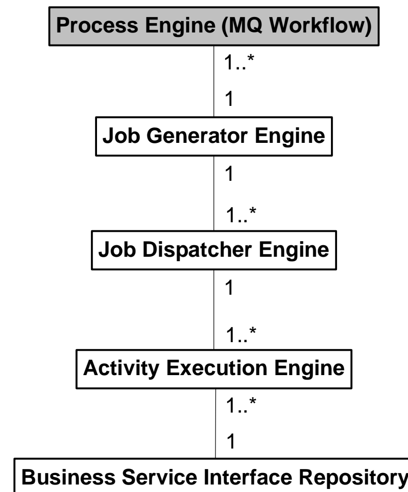


Figure 30: Relationships of components

The Job Generator Engine

The UPES mechanism realizes among other things the ASYNCHRONOUS COMPLETION TOKEN [Schmidt et al. 2000] pattern. The CORRELATION IDENTIFIER [Hohpe et al. 2003] pattern describes basically the same concept in the asynchronous messaging context. The MQ Workflow specific Activity Implementation Correlation ID, which is an element of all exchanged UPES XML messages, is the completion token that is used to correlate a return message back to an activity instance. Moreover, the completion token is used to connect all architectural components, i.e. Job Generator Engine, Job Dispatcher Engine, Activity Execution Engine, and the Business Service Interface Repository (only in case of asynchronous business services), as the tokens will be passed from one component to the next one and will also be returned back until they arrive again at the process engine.

A UPES is basically just a message queue definition from the point of view of MQ Workflow. An activity instance will send a request message to that message queue, and the message can thus be processed by an external component. The return message will be sent by that external component to a defined input queue. MQ Workflow takes the messages out of that input queue and relates the return message back to an activity instance by the Activity Implementation Correlation ID. That is, the completion token and the process flow can carry on to the next activity, as defined by the process model.

The Job Generator Engine is the backend interface to the process engine (MQ Workflow) and thus implements the PROCESS INTEGRATION ADAPTER and MACROFLOW INTEGRATION SERVICE patterns. The task of the Job Generator Engine is to decouple the message format of the process engine from a standardised message format used for job definitions. Thus, an incoming request in a UPES queue will be transformed into a standardised job definition message format and will be forwarded to a Job Dispatcher Engine, representing a RULE-BASED DISPATCHER, which is responsible for further processing the jobs. For this reason, the Job Generator Engine decouples the product specific process integration logic from job-based

integration logic. The Job Generator Engine contains a set of Job Generators, implementing MACROFLOW INTEGRATION SERVICES, being based on an MQ Workflow PROCESS INTEGRATION ADAPTER, which listen to input queues (UPES). The job generators are responsible for taking the messages out of the queues, transforming them into the job definition format, completing the message with some additional information, and forwarding the jobs to the dispatcher.

Thus, parallel processing of messages in different queues is possible. If a job has been processed, the result will be reported back to the Job Generator by the Job Dispatcher Engine in a corresponding job-result queue. The Job Generator will transform the job-result into a return message for the process engine and will put the message in the input queue of the process engine. Eventually, each Job Generator listens to two queues and writes to two other queues. The following UML model illustrates the structure of the Job Generator Engine.

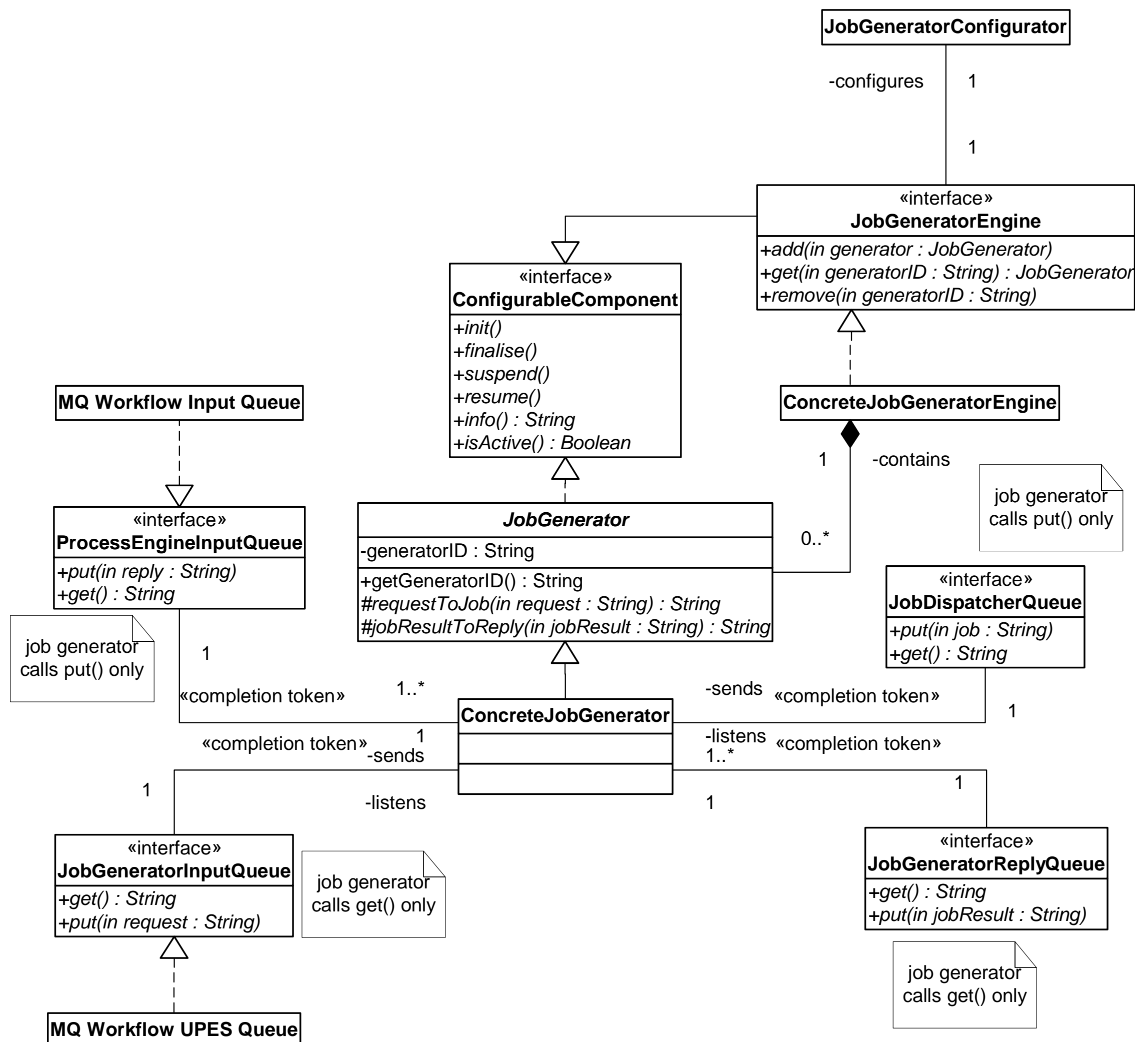


Figure 31: Job generator engine

Basically, the Job Generator Engine is a repository of Job Generators. Once a Job Generator is started by calling its *init* method, it will start to listen to queues and thus to process the incoming messages. The forward and backward transformation of messages will be implemented by a Concrete Job Generator in the methods *requestToJob* (forward transformation into the job definition format) and *jobResultToReply* (backward transformation from job definition format to

UPES XML format). Most of the development projects that have used this architectural framework implementation within the overall programme have used XML definitions and corresponding messages for the job definition format. The following sequence diagrams illustrate the forward and backward transformation of messages.

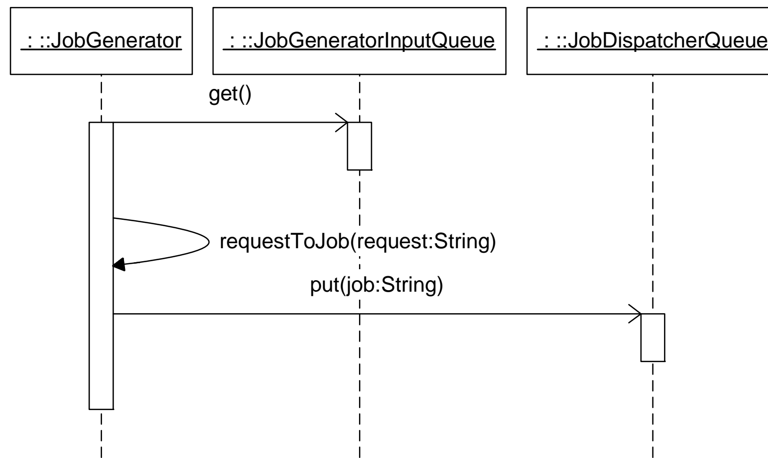


Figure 32: Request forward transformation into job definition format

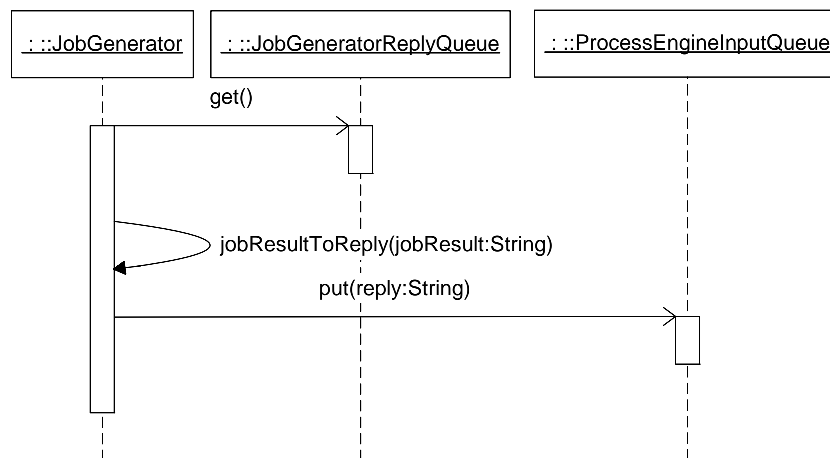


Figure 33: Job result backward transformation into a reply

The *get* operation, the message transformation, and the *put* operation must be understood as one transaction. As far as this aspect is concerned, WebSphere MQ (MQ Series) can be used as the transaction coordinator. Thus, transaction security is ensured via WebSphere MQ.

The Job Dispatcher Engine

To dispatch and finally execute a job, the job definition must at least contain two additional pieces of information (compared to the original request from the process engine) that must be added by the Job Generator: the Activity Execution Engine that is responsible for processing the job and the Activity Implementation that actually executes the job within the Activity Execution Engine. The Job Generator will add this information to the job definition by putting the input queue name of the corresponding Activity Execution Engine that implements a MICROFLOW ENGINE, and the identifier of the Activity Implementation into the job definition message. The

Job Dispatcher Engine represents a RULE-BASED DISPATCHER and will thus simply read the jobs from the input queue and forward them to the Activity Execution Engine that is defined in the job. Additionally, the dispatcher will add the name of its reply queue to the job definition before forwarding the message.

If an Activity Execution Engine has finished execution of a job, it will send a job-result message back to the dispatcher. The reply queue of the dispatcher has previously been dynamically added to the job definition by the dispatcher. The dispatcher will take that response message and forward the reply to the Job Generator that is defined in the job definition.

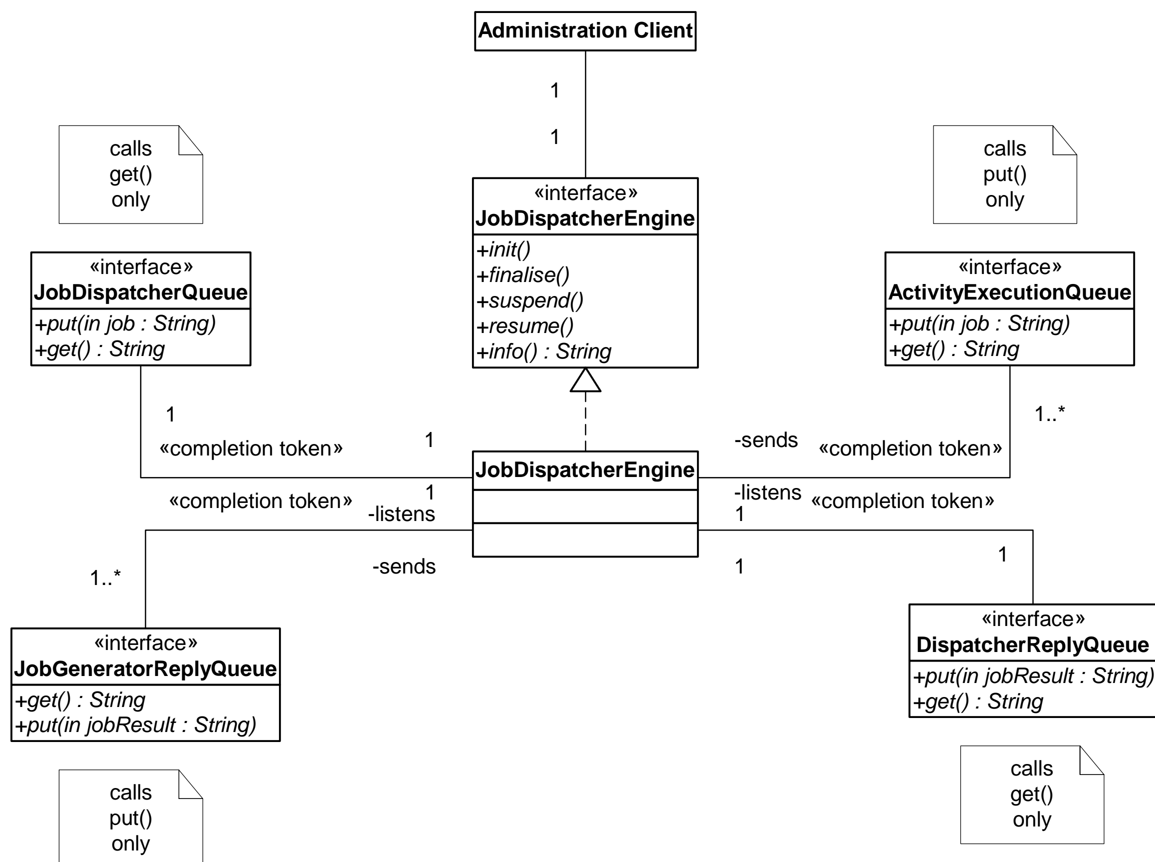


Figure 34: Job dispatcher engine

The following sequence diagrams illustrate how the Job Dispatcher Engine distributes the jobs to different Activity Execution Engines and how the job-results are re-dispatched back to the Job Generator Engine. Obviously, the basic functionality is principally very similar to the Job Generator Engine. Analogous to the Job Generator Engine, the interdependent *get* and *put* operations are implemented as transactions.

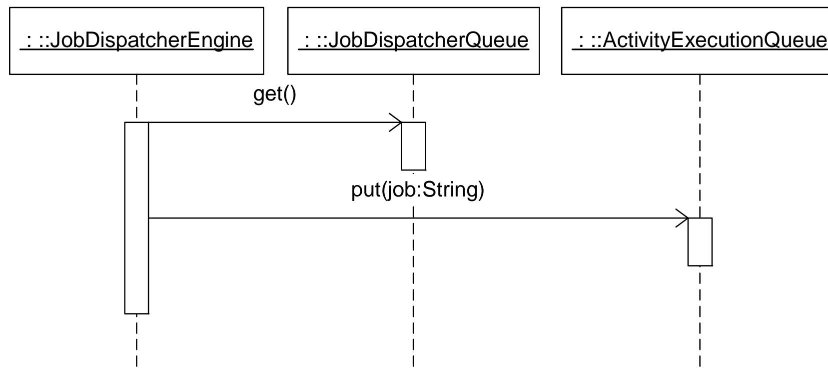


Figure 35: Forwarding a job

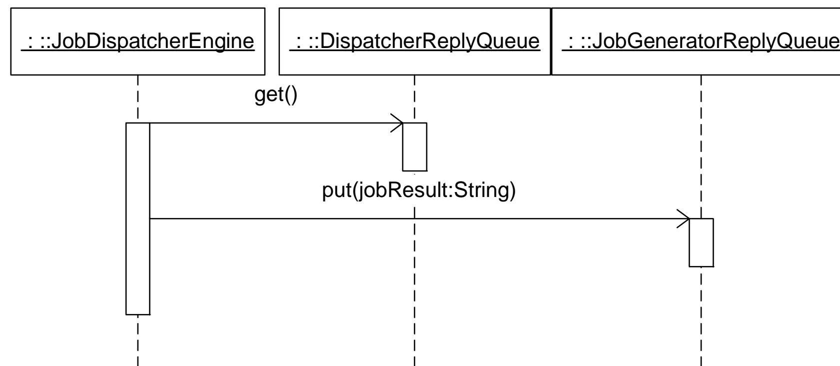


Figure 36: Forwarding a job-result back

The Activity Execution Engine

The Activity Execution Engine implements a MICROFLOW ENGINE and is responsible for executing and managing activity implementations, i.e. business logic that is related to automatic activities in business process models. Thus, a concrete Activity Implementation that actually represents a MICROFLOW EXECUTION SERVICE will call business services and will set the process control data as specified by the process logic (the referenced process-activity), depending on the results of the service calls. The results of the called business services will be reported back to the process instance via the process control data that will be sent in the job-result. This reporting procedure of the job-result goes backwards via the Job Dispatcher Engine, to the Job Generator Engine, and finally to the process engine (MQ Workflow). Thus job-results move backwards the same path that they have arrived at an Activity Execution Engine but they move in reverse order.

An Activity Execution Engine has an input queue where the jobs are delivered to be executed by the engine. The engine then retrieves the Activity Implementation that is associated to the job from a repository (the job definition contains the identifier of the Activity Implementation). Once an Activity Implementation is retrieved from the repository, the input data that are also included in the job definition will be passed to the Activity Implementation. Ultimately, the Activity Implementation will be executed and its execution will be controlled. If an Activity Implementation has finished execution of a job, the job-result that is delivered as output by an Activity Implementation will be put in the reply queue of the corresponding dispatcher engine (the queue identifier is also included in the job definition). Conclusively, many Activity

Implementations will be executed in parallel by the Activity Execution Engine. For this reason, an Activity Implementation is executable as a thread in the Activity Execution Engine.

A concrete Activity Implementation will invoke certain business services. It might be necessary to access business objects associated to the process instance in order to do this, as those objects may provide the necessary data for invoking a service, e.g. customer details like name, address, account numbers, etc. As an Activity Implementation has received the process control data as input, it will have access to references of the corresponding business objects of the process instance.

The implementations of interfaces to business services will be kept in repositories in order to decouple them from the Activity Execution Engine. A concrete Activity Implementation will thus get access to the required business service interfaces via access to a Business Service Interface Repository.

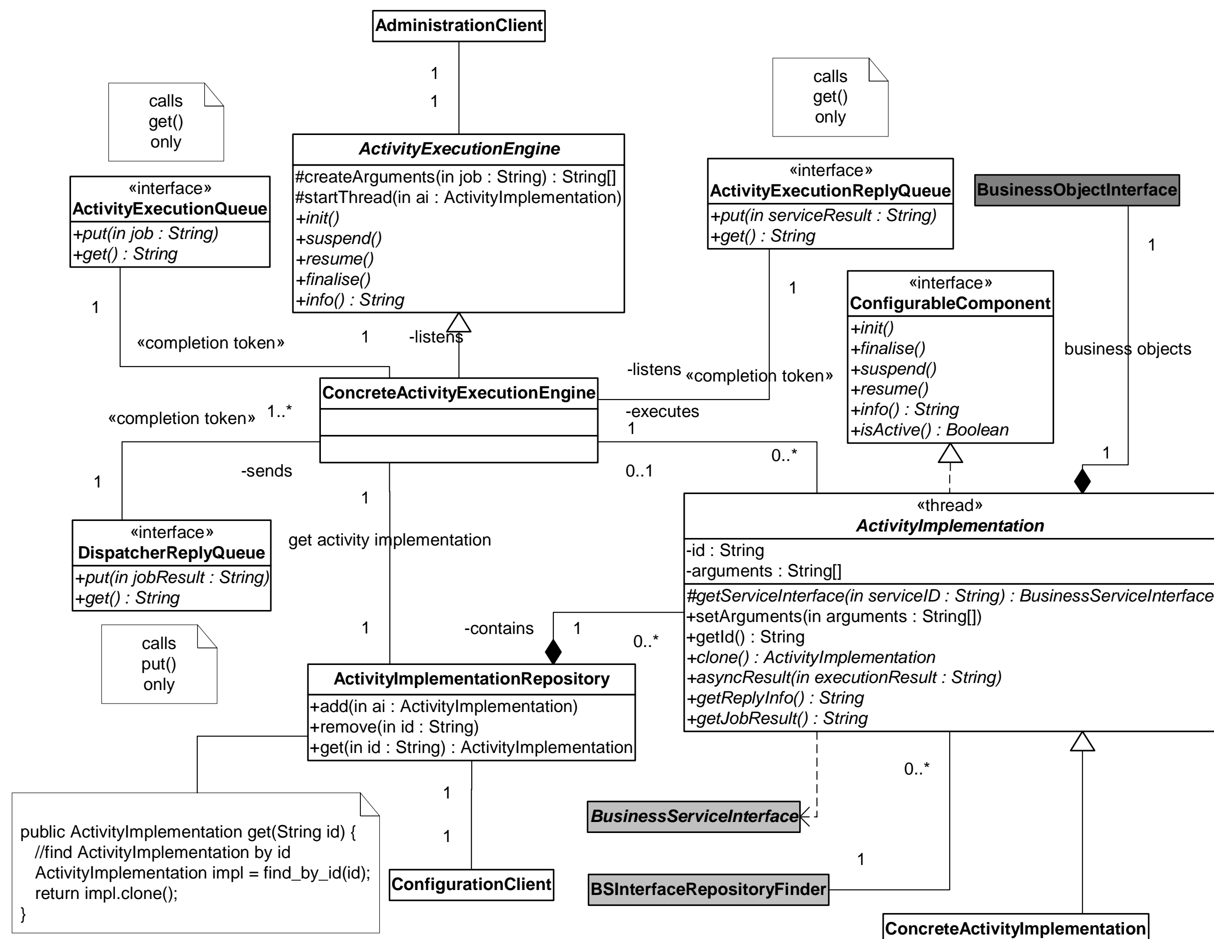


Figure 37: Activity execution engine

A concrete *ActvitiyExecutionEngine* will get the jobs from the *ActivityExecutionQueue* by calling its *get* method. As already mentioned, the job definition contains the *ActivityImplementation* identifier for executing the job. Therefore, the engine will retrieve the appropriate *ActivityImplementation* from the *ActivityImplementationRepository* by calling the *get* method of the repository and by passing the ID of the *ActivityImplementation* as a parameter of the method call. As illustrated in the diagram, this *get* method will search for the *ActivityImplementation* object and will return a clone of the object in the repository. By using this technique, it is easily possible to execute several instances of the same *ActivityImplementation* and to configure objects in the repository in parallel.

The *ActivityExecutionEngine* will generate the arguments for executing the *ActivityImplementation* from the job definition by calling its *createArguments* method. Thereafter, the engine will call the *getReplyInfo* method in order to determine whether the *ActivityImplementation* will receive asynchronous results from business services. The string delivered by *getReplyInfo* provides information how many asynchronous results are expected, as an *ActivityImplementation* may invoke several asynchronous service calls.

In order to set the input data for the *ActivityImplementation,* the method *setArguments* will be called with the previously generated arguments. Finally, the *ActivityExecutionEngine* will start the ActivityImplementation as a thread within the engine by calling its *startThread* method. Asynchronous replies from Business Service Interfaces will be collected by the engine from the *ActivityExecutionReplyQueue.* The following sequence diagram illustrates the process of initialising and starting an *ActivityImplementation.*

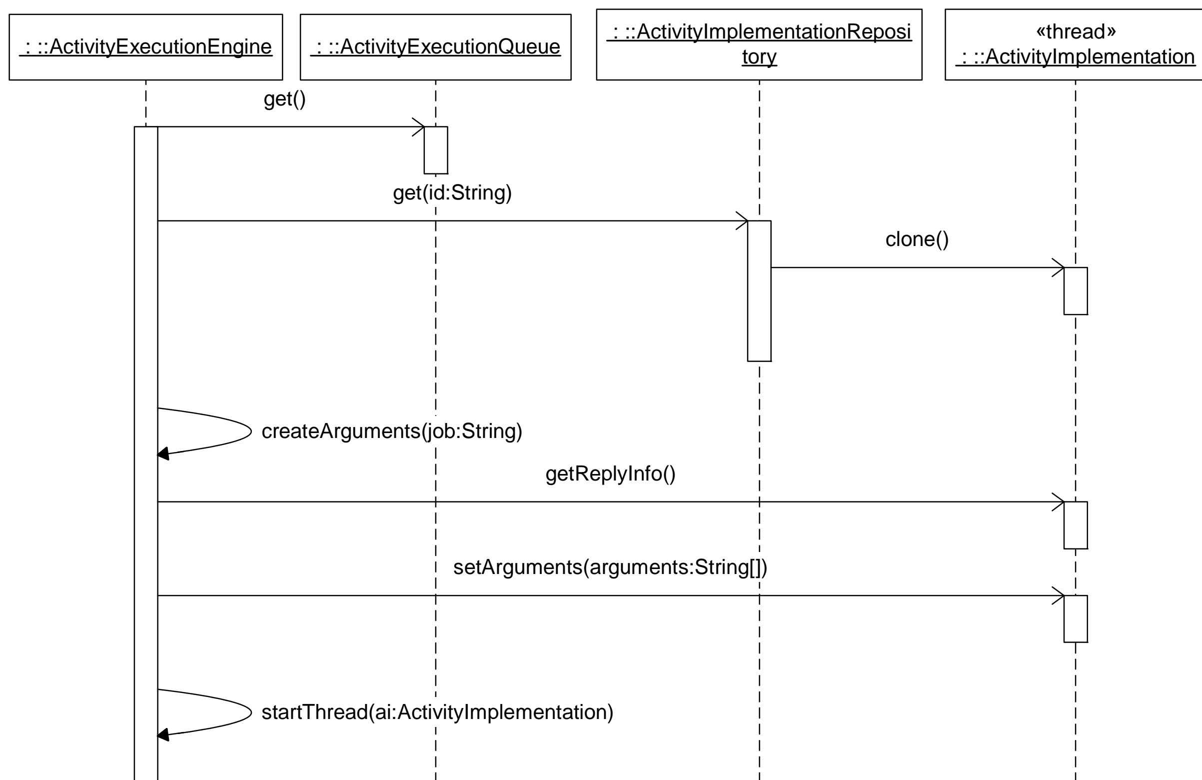


Figure 38: Initialising an activity implementation

After an *ActivityImplementation* has been initialised and started, the subsequent flow of method calls depends whether there are asynchronous replies from business services or not. Ultimately, the engine will collect the job-result from the thread by calling the *getJobResult* method after the thread has finished. The following sequence diagram illustrates the message flow.

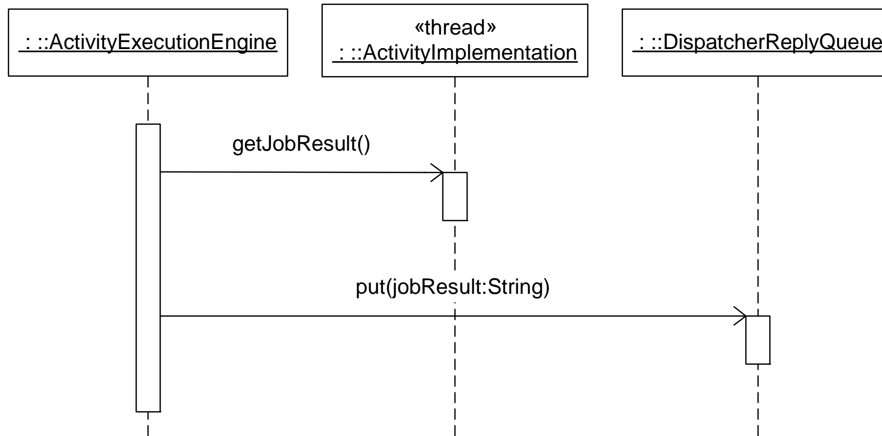


Figure 39: Informing the job dispatcher engine about a job-result

If there are asynchronous replies from business services, the *ActivityExecutionEngine* will be responsible for collecting those results and informing the corresponding *ActivityImplementation* thread that a service result has arrived. The *ActivityImplementation* is responsible for incorporating that service result in the overall job-result. The relationship between an *ActivityImplementation* instance and an asynchronous reply from a business service is achieved by the completion token, which will be part of the service's reply message. Those reply messages are delivered in the *ActivityExecutionReplyQueue*. In case of asynchronous replies, the *ActivityExecutionEngine* must consequently manage these relationships and inform the appropriate thread about an incoming result.

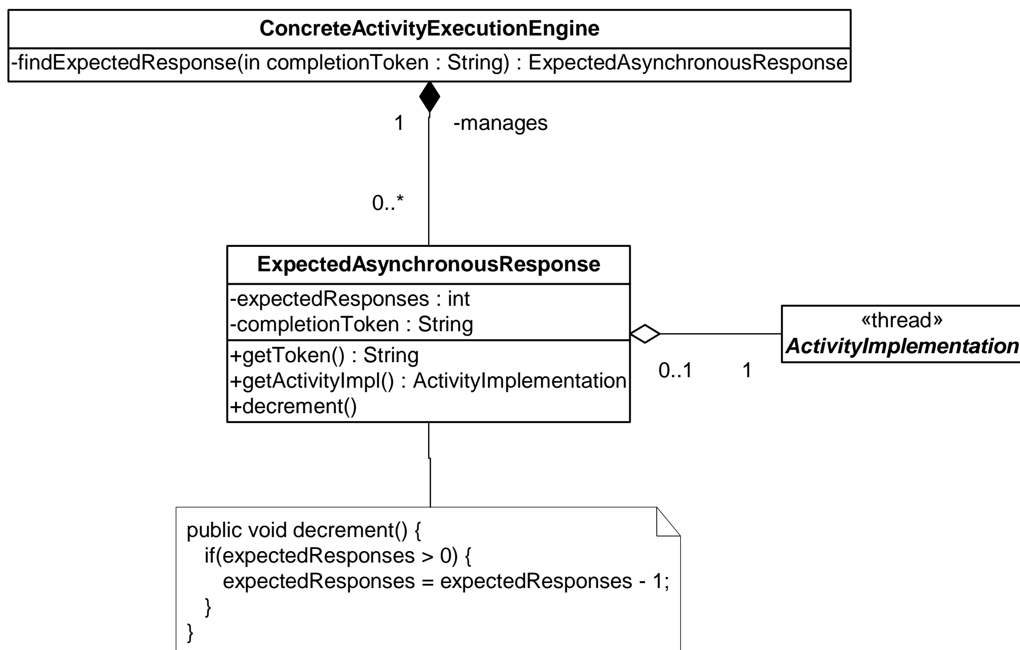


Figure 40: Managing asynchronous responses

Apart from the relationships between completion tokens and *ActivityImplementation* instances, the engine has to remember how many asynchronous responses are expected. This information has been obtained from the *getReplyInfo* method of the *ActivityImplementation* instance. Conclusively, it is possible to manage those incoming responses by keeping the relationships of

the completion tokens, the number of expected responses, and the corresponding *ActivityImplementation* instances in a list. An *ActivityImplementation* can eventually be informed about a service result by calling its *asyncResult* method. Finally, the decrement method must be called in order to register the service result as accepted. The following sequence diagram illustrates the flow of method calls.

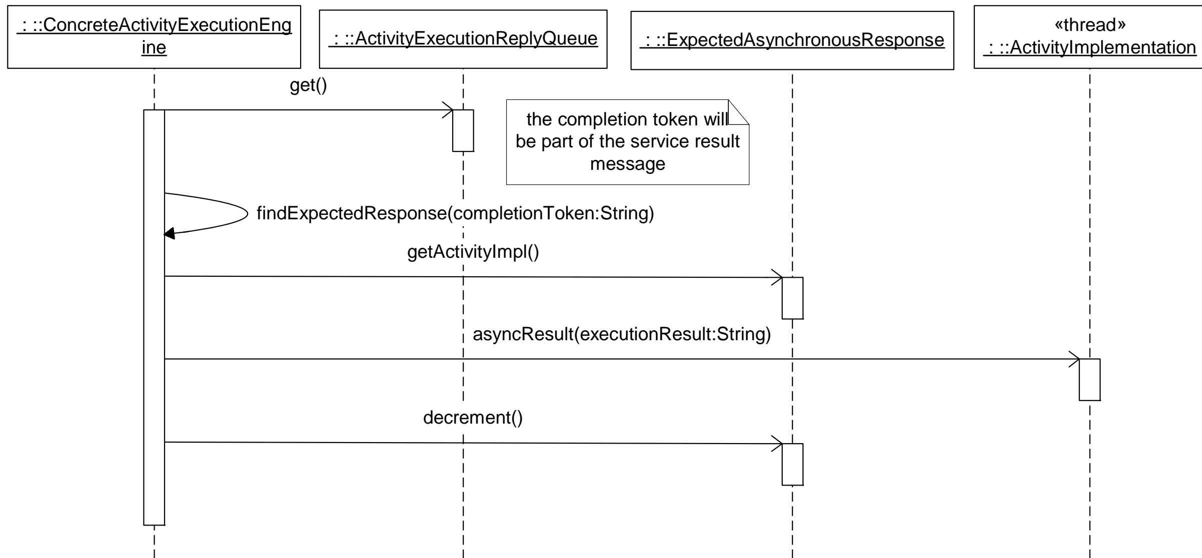


Figure 41: Informing an activity about an asynchronous service result

The necessary initialisation of an *ActivityImplementation*, i.e. retrieving the required Business Service Integration Interfaces from the Business Service Interface Repository and creating a *BusinessObjectInterface* object is implemented in the *init* method of the *ActivityImplementation*. The Business Service Interface Repository will be accessed via a repository finder.

The Business Services Interface Repository

The Business Services Interface Repository component is a CONFIGURABLE ADAPTER REPOSITORY of standardised interface implementations realizing CONFIGURABLE ADAPTERS to various synchronous and asynchronous BUSINESS-DRIVEN SERVICES. The repository will be accessed via a repository finder as the location of the repository might change.

In principle, a Business Service Interface represents a CONFIGURABLE ADAPTER to a BUSINESS-DRIVEN SERVICE and just declares a standardised *invoke* method for a service. A concrete implementation of this interface will actually integrate the business service. If the service is asynchronous the reply will be sent to a defined reply queue, which will be read by the corresponding Activity Execution Engine. The name of the reply queue can be passed as a parameter of the *invoke* method. If a Business Service Interface shall be retrieved from the repository, the repository will return a clone of the interface.

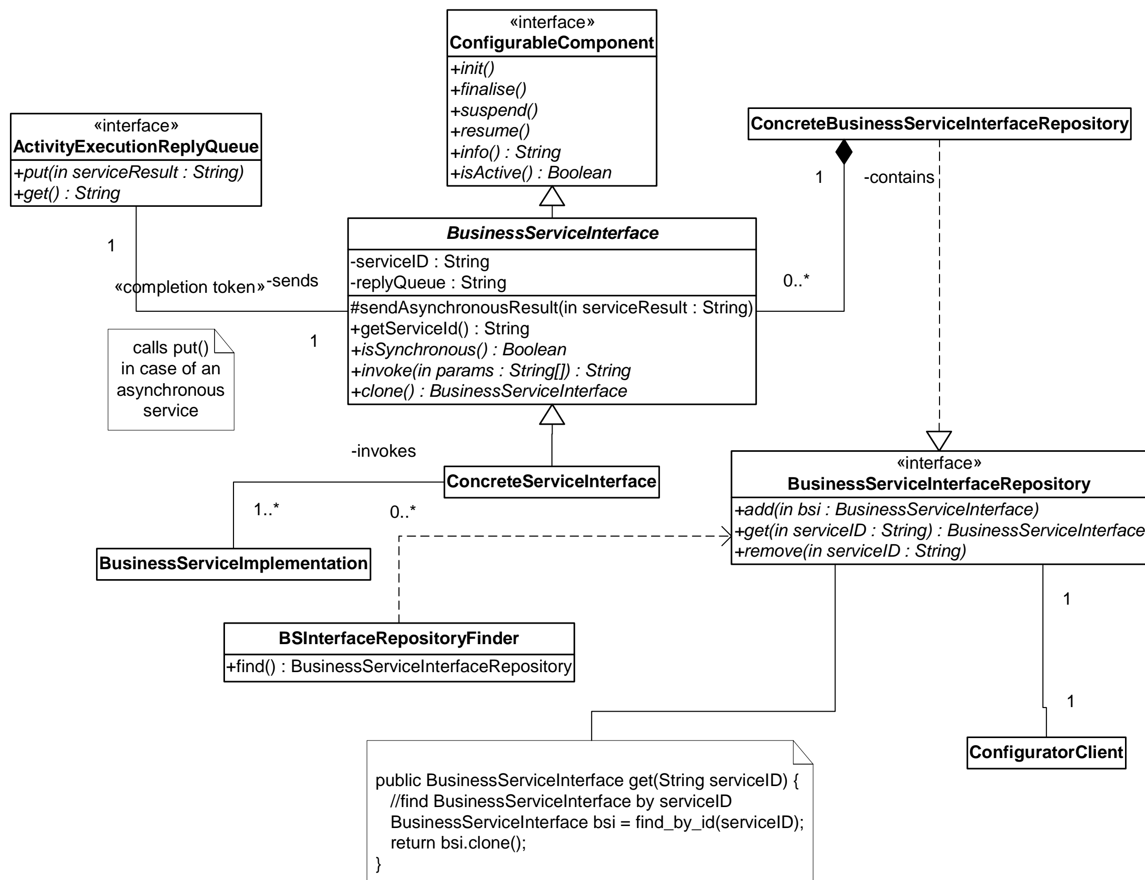


Figure 42: Business service interface repository

The class *BSInterfaceRepositoryFinder* implements a repository finder. Furthermore, the abstract class *BusinessServiceInterface* declares the standard interface for business services including the *invoke* method. If the service interface is synchronous, then the *invoke* method will deliver the service result. If the service interface is asynchronous, the concrete implementation of the Business Service Interface will put the service result in the *ActivityExecutionReplyQueue*. In order to achieve this, the concrete interface implementation generates an internal thread to capture the asynchronous service result and forward it to the *ActivityExecutionReplyQueue*.

Conclusion

In this paper we have documented the fundamental patterns needed for an architecture that composes and orchestrates services at the process level. The individual patterns can be used on their own to address certain concerns in a process-driven SOA design, but the general architecture following the PROCESS-BASED INTEGRATION ARCHITECTURE pattern – in first place – aims at larger architectures. The pattern language as a whole focuses on separating business concerns cleanly from technical concerns, in macroflows and microflows. All integration concerns are handled via services, and macroflows and microflows are used for flexible composition and orchestration of the services.

Acknowledgements

We like to thank Andy Longshaw, our EuroPLoP 2006 shepherd, for his useful comments. We also like to thank the participants of the EuroPLoP 2006 writers' workshop for their valuable feedback.

References

[Active Endpoints 2007] Active Endpoints. ActiveBPEL Open Source Engine. http://www.active-endpoints.com/active-bpel-engine-overview.htm, 2007.

[Arsanjani 2004] A. Arsanjani. Service-oriented modeling and architecture - How to identify, specify, and realize services for your SOA. IBM developerWorks http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design1/ , Nov. 2004.

[Barry 2003] D. K. Barry. Web Services and Service-oriented Architectures, Morgan Kaufmann Publishers, 2003

[BOC 2007] BOC GmbH. ADONIS integration with Oracle BPEL Process Management. http://boc-eu.com/documents/events/bpel_en.pdf, 2007.

[Channabasavaiah 2003 et al.] K. Channabasavaiah, K. Holley, and E.M. Tuggle. Migrating to Service-oriented architecture – part 1, http://www-106.ibm.com/developerworks/webservices/library/ws-migratesoa/, IBM developerWorks, 2003

[Emmerich 2000] W. Emmerich. Engineering Distributed Objects. Wiley & Sons, 2000.

[Gamma et al. 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[GFT 2007] GFT. GFT Inspire Business Process Management. http://www.gft.com/gft_international/en/gft_international/Leistungen_Produkte/Software/Business_Process_Managementsoftware.html, 2007.

[Hentrich 2004] C. Hentrich. Six patterns for process-driven architectures. In Proceedings of the 9th Conference on Pattern Languages of Programs (EuroPLoP 2004), 2004.

[Hohpe et al. 2003] G. Hohpe and B. Woolf. Enterprise Integration Patterns. Addison-Wesley, 2003.

[IDS 2006] IDS Scheer. Aris Platform. http://www.ids-scheer.de/germany/products/53956, 2006.

[iWay 2007a] iWay Software. iWay Adapter Technologies. http://www.iwaysoftware.jp/products/integrationsolution/adapter_manager.html, 2007.

[iWay 2007b] iWay Software. iWay Adapter Manager Technology Brief. http://www.iwaysoftware.jp/products/integrationsolution/adapter_manager.html, 2007.

[JBoss 2007] JBoss. JBoss jBPM. http://www.jboss.com/products/jbpm, 2007.

[Mule 2007] Mule Project. Mule open source ESB (Enterprise Service Bus) and integration platform. http://mule.mulesource.org/, 2007.

[Schmidt et al. 2000] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture. J.Wiley and Sons Ltd., 2000.

[ServiceMix 2007] Apache ServiceMix Project. Apache ServiceMix. http://www.servicemix.org/, 2007.

[webMethods 2007] webMethods. webMethods Fabric 7. http://www.webmethods.com/products/fabric, 2007

[Zdun et al. 2006] U. Zdun, C. Hentrich, and W.M.P. van der Aalst. A Survey of Patterns for Service-Oriented Architectures, Internet Protocol Technology, Inderscience, 2006.

Appendix: 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 a brief introduction to them in form of thumbnails of these patterns. For detailed descriptions of these patterns please refer to the referenced articles.

Pattern	*Problem*	*Solution*
GENERIC PROCESS CONTROL STRUCTURE [Hentrich 2004]	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.
BUSINESS OBJECT REFERENCE [Hentrich 2004]	How can the 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.
ENTERPRISE SERVICE BUS [Zdun et al. 2006]	How is it possible in a large business architecture to integrate various applications and backends in a comprehensive, flexible, and consistent way?	Unify the access to applications and backends using services and service adapters, and use message-oriented, event-driven communication between these services to enable flexible integration.
CORRELATION IDENTIFIER [Hohpe et al. 2003]	How does a requestor that has received a response know to which original request the response is referring?	Each response message should contain a CORRELATION IDENTIFIER, a unique identifier that indicates which request message this response is for.
CANONICAL DATA MODEL [Hohpe et al. 2003]	How to minimize dependencies when integrating applications that use different data formats?	Design a CANONICAL DATA MODEL that is independent from any specific application. Require each application to produce and consume messages in this common format.
COMPONENT CONFIGURATOR [Schmidt et al. 2000]	How to allow an application to link and unlink its component implementations at runtime without having to modify, recompile, or relink the application statically?	Use COMPONENT CONFIGURATORS as central components for reifying the runtime dependencies of configurable components. These configurable components offer an interface to change their configuration at runtime.

Table 2: Thumbnails of referenced patterns