OPEN Project

STREP Project FP7-ICT-2007-1 N.216552

Title of Document:	Solutions for Application Logic Reconfiguration
Editor(s):	H. Klus, D. Niebuhr, B. Schindler, M. Deynet, C. Deiters
Affiliation(s):	Clausthal University of Technology
Contributor(s):	Agnese Grasselli, Stefano Marzorati
Affiliation(s):	Vodafone Omnitel NV
Date of Document:	31.01.2009
OPEN Document:	WP 4, D4.1
Distribution:	Public
Keyword List:	Adaptive Systems, System Reconfiguration, Migration, Context-Awareness
Version:	Final

OPEN Partners:

CNR-ISTI (Italy)
Aalborg University (Denmark)
Arcadia Design (Italy)
NEC (United Kingdom)
SAP AG (Germany)
Vodafone Omnitel NV (Italy)
Clausthal University (Germany)

Title: Solutions for Application Logic Reconfiguration	**Id Number:** WP 4, D4.1

Abstract

This document introduces aspects and solutions for the reconfiguration and adaptation of services during their lifecycle. During the migration of one or more services, their look and behaviour has to be adapted to the target device. Context information is for example one important aspect which may influence how to adapt services, like residence of the service user or current battery power and CPU frequency of the devices in use. In this document we will introduce an application scenario which demonstrates various kinds of adaptation and their triggers and according parameters which influence the way services are adapted. Furthermore we will present current solutions for application logic reconfiguration. Finally we will show architectural solutions for the different reconfiguration approaches.

Table of Contents

1 INTRODUCTION 2

2 SCENARIO FOR APPLICATION LOGIC RECONFIGURATION 4

2.1 THE ORIGINAL PACMAN GAME 4
2.2 MIGRATION AND ADAPTATION SCENARIOS 4
2.3 BASIC SOFTWARE ARCHITECTURE OF THE GAME 7
2.3.1 Architecture of the Single-Player PacMan Game 7
2.3.2 Architecture of the Multi-Player Game 9

3 SOLUTIONS FOR APPLICATION LOGIC RECONFIGURATION 10

3.1 DIFFERENTIATION OF WIRING AND ORCHESTRATING SERVICES 10
3.2 WIRING APPROACH 11
3.2.1 Service Usage Adaptation 11
3.2.2 Service Behaviour Adaptation 13
3.3 ORCHESTRATION APPROACH 15
3.3.1 Workflow Patterns 15
3.3.2 Example of Workflow Patterns 16
3.3.3 Evaluation of State of the Art Technologies 18
3.3.4 jBPM 19
3.3.5 YAWL 21
3.3.6 Windows Workflow Foundation 23
3.3.7 OWL-S 24
3.3.8 Orchestration Tools Comparison 25
The features of the tools that have been evaluated are summarized in the following. 25
3.4 COMPARISON OF WIRING AND ORCHESTRATION APPROACH 26

4 ARCHITECTURAL SOLUTIONS FOR APPLICATION LOGIC RECONFIGURATION 28

4.1 ARCHITECTURE OF WIRING APPROACH 28
4.2 ARCHITECTURE FOR ORCHESTRATION APPROACH 30
4.3 ARCHITECTURE OF COMBINED APPROACH 32

5 COMMUNICATION ASPECTS 33

6 CONCLUSION AND NEXT STEPS 36

7 REFERENCES 37

1 Introduction

In these days the trend towards "everything, every time, everywhere" becomes more and more apparent. Electronic assistants, so called "information appliances", like network enabled PDAs, Internet capable mobile phones and electronic books or tourist guides are well known. The continuing progress of all IT sectors towards "smaller, cheaper, and more powerful" mainly enables this trend.

IT components are embedded in nearly every industrial or everyday life object. This trend is driven by new developments in the field of materials science like midget sensors, organic light emitting devices or electronic ink and the evolution in communications technology, especially in the wireless sector. As consequence of this trend, nearly everyone has small, nearly invisible devices in his adjacencies, e.g. mobile phones, PDAs, or music players. Furthermore, network technologies like (W)LAN or Bluetooth moved mainstream. This facilitates the connection and combined usage of those devices.

The mobility of users and their devices leads to the need of customizable applications that adapt dynamically to their specific needs in constantly changing situations. Services and devices for example can appear or disappear at any time, the physical environment of the user may change, or the user's preferences.

In OPEN we consider applications which migrate from one device to another. Hereby, the application has to adapt to the new environment during migration. For this the application has to consider available resources like battery power or CPU rate, and display size and resolution for example. Furthermore, not always the whole application migrates, but sometimes only parts of it.

Therefore, we distinguish two parts of which an application typically consists of, namely the user interface part, and the application logic part as illustrated in Figure 1.

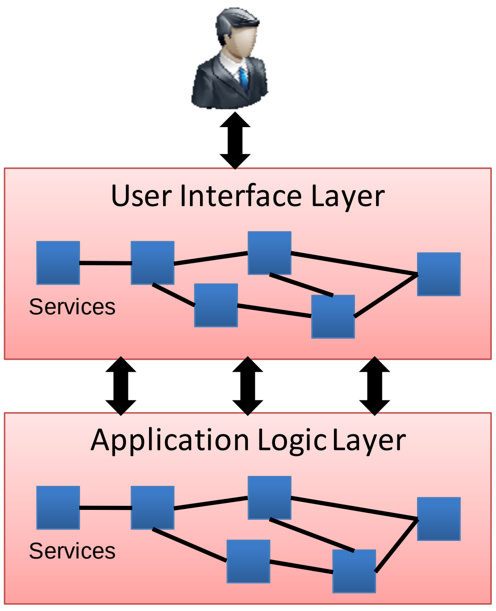


Figure 1: Two layers most applications consist of, namely the user interface layer and the application logic layer.

The user interface layer is responsible for the interaction between the user and the application. It retrieves user input and provides also feedback to the user. The user interface itself can be realized by one or more services which interact with each other or with the user. The application logic layer on the other hand is responsible for computing the reaction to user input. To do this, the services within the user interface layer interact with services within the application logic layer. In the next section we will present an example application and according services at the user interface and application logic layer.

Adaptation is required in the user interface part, as well as in the application logic part. The user interface part for example has to adapt to the new screen size while migrating from a PC to a PDA. The application logic part has among others to adapt to the change of resources for example by replacing a resource-consuming service by another one which might not offers the full functionality, but which is less resource-consuming.

In this deliverable we will describe how adaptation of the application logic can take place. We will describe triggers for adaptation, how context can be considered during adaptation, and which technologies are available at the moment. In order to introduce the relevance of adaptation for the user, we will first present an application scenario. The mentioned topics will then be explained using this example.

2 Scenario for Application Logic Reconfiguration

In this section we will introduce an application and scenarios from the user's point of view where adaptation of the application logic is relevant. After that we will introduce two main approaches of how applications can be structured, namely the wiring approach and the orchestration approach.

2.1 The Original PacMan Game

PacMan is a game where a character called PacMan, which is steered by the user, has to collect dots in a maze, like shown in Figure 2.

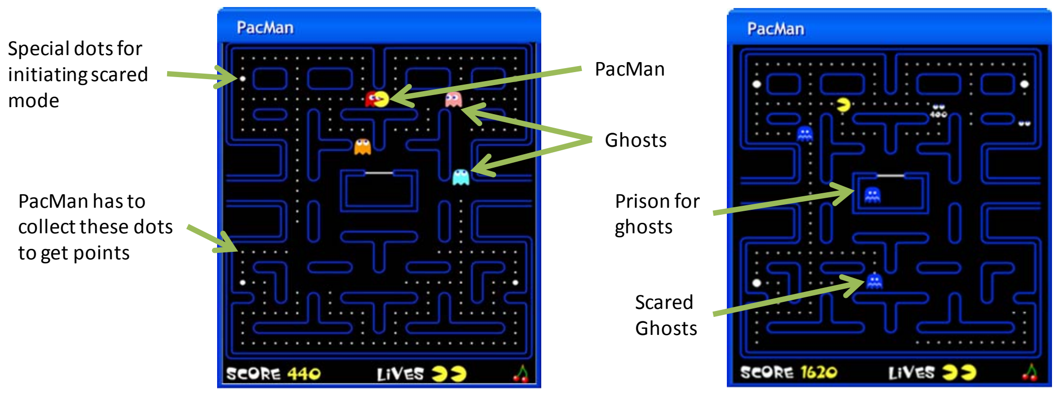

Special dots for initiating scared mode
PacMan has to collect these dots to get points
PacMan
PacMan
Ghosts
Prison for ghosts
Scared Ghosts
SCORE 440
LIVES
PacMan
SCORE 1620
LIVES

Figure 2: The two modes of a single PacMan game. On the left hand side the game is in normal mode where the ghosts try to catch the PacMan. On the right side the game is in scared mode where the PacMan can catch the ghosts.

Ghosts, who are controlled by the computer, are running around with the goal to catch the PacMan. If the PacMan collects special dots, ghosts and PacMan change roles for some seconds like depicted in Figure 2 on the right hand side. That means that the PacMan now can catch ghosts and that the ghosts try to run away. Caught ghosts will be imprisoned in the middle of the maze for some seconds. After some seconds the roles change back again. The goal for the player is to get as much scores as possible by collecting dots and catching ghosts.

2.2 Migration and Adaptation Scenarios

Within OPEN we defined scenarios where migration, and therefore adaptation takes place (Faatz et al., 2008). Assume that the user starts playing PacMan on her PC at home using her keyboard to steer the PacMan. Then she remembers that she has a PDA with an accelerometer. Therefore, she wants to use the PDA to steer the ghost while keeping the GUI on the large screen of her desktop PC. To do this, she simply switches on her PDA and the input is automatically migrated from PC to the PDA like shown in Figure 3.

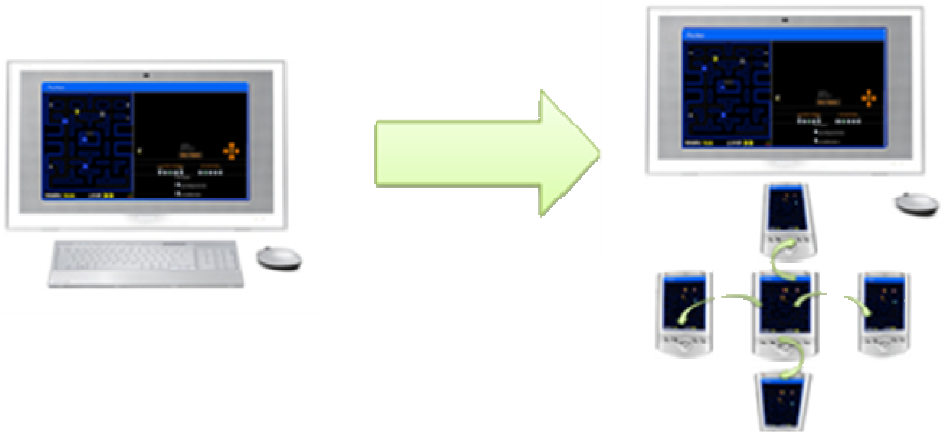


Figure 3: The control of the PacMan is migrated from the keyboard of the PC to a PDA with an accelerometer. After migration the PacMan can be steered by tilting the PDA. The GUI stays at the display of the PC.

After migration the PacMan can be steered by tilting the PDA into the desired direction. In this case, a service from the user interface layer is migrated and adapted to the target platform. However, this may also trigger a reconfiguration of services of the application logic layer. As it is now much more difficult for the player to steer the PacMan, the speed of the ghosts could for example be adapted.

After some time she leaves to meet some friends. Therefore, she would like to continue playing the current game on her PDA during the bus trip. To do this, a migration like depicted in Figure 4 is needed.

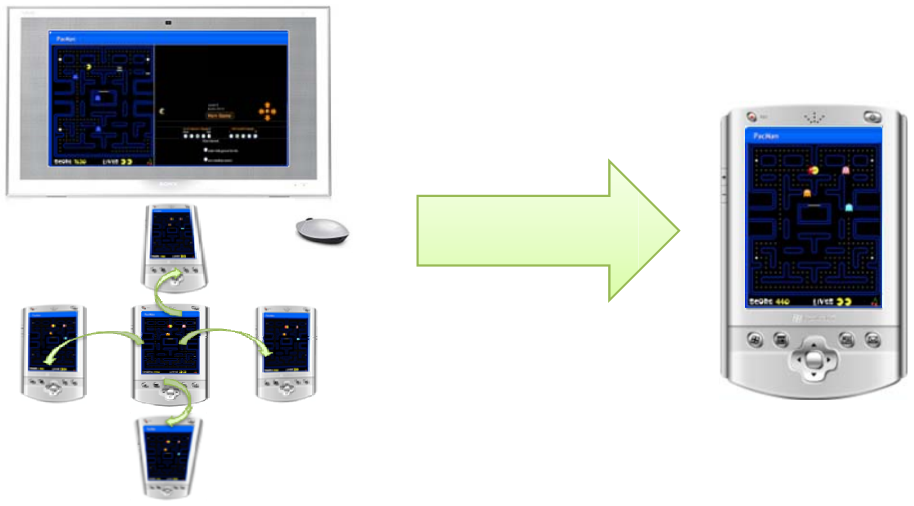


Figure 4: Migration of PacMan from PC with large screen to PDA with small screen and limited resources.

The OPEN platform offers a context menu with the option to migrate the game. By clicking on the migration button, the game will pause first. Afterwards the user can select an available target device and confirm the migration. The game will now appear on his PDA. When the user gets into the bus and takes a seat, she can resume the game by clicking on the resume button of the OPEN platform context menu on her PDA.

In this case, not only services of the user interface are migrated, but the whole application including the services of the application logic layer. Because of the limited screen size and resources of her PDA, the game has to adapt to the target platform. Since there is no place to display all information at once on the PDA, some options are now no longer accessible directly, but moved into an extra dialog. Thus, services on the user interface layer have to be adapted again, this aspect is described in depth in the D2.2 deliverable. Furthermore, the number of dots within the maze is less than on the PC because of limited screen size which again results in an adaptation of services within the user interface layer. But also the application logic has to be adapted accordingly. If for example fewer dots are available on the screen, the game now has to adapt in a way that the player gets more points for collecting a single dot. As it is more difficult to steer the PacMan on such a small screen, the speed of the ghosts is also reduced. Depending on the current game level, the artificial intelligence of the ghost is also adapted. All these kinds of adaptation result in the adaptation of services within the application logic layer.

Arriving at her friend's home, she and her friend want to play together. Both want to play on their own PDA. At this, a new variant of the PacMan game enables it. Her friend starts the PacMan game on the PDA, too. Now the friend gets the option to join the game. She decides to do so and both start their game. If she catches a ghost then that ghost will now appear in the game of her friend and vice versa instead of appearing in her own game like depicted in Figure 5. That means they play against each other by sending ghosts to each other.

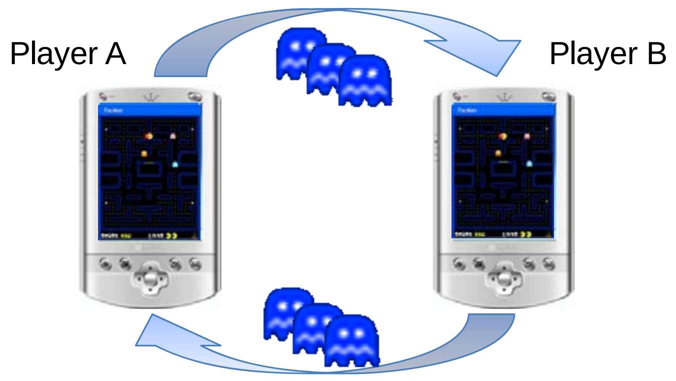


Figure 5: Two players play against each other by sending ghosts to each other.

Finally, a third friend visits them and also joins the game using again her own PDA like depicted in Figure 6.

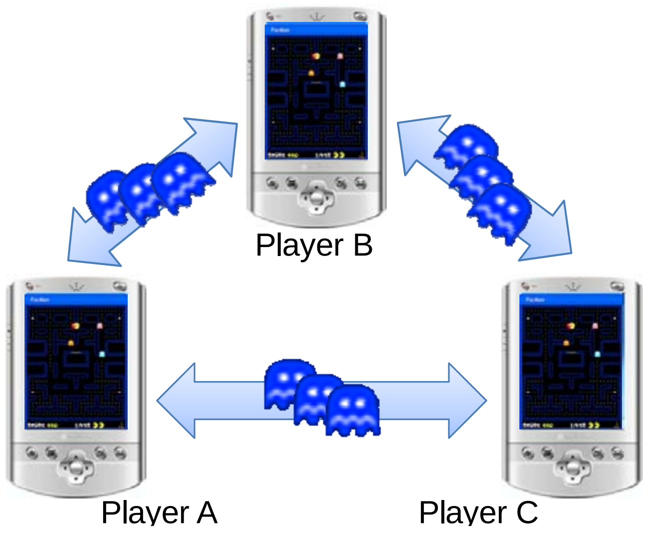


Figure 6: Special rules define where to send the caught ghosts if more than two players have joined the game.

Now special rules define where to send a caught ghost. One rule could be to send the ghost to the closest player. Another possible rule is to send them to the player with the highest or lowest score for example. These kinds of rules are also part of the application layer and thus, the orchestration of the application logic services has to be adapted in this case, too.

2.3 Basic Software Architecture of the Game

The game can be divided into two main building blocks, namely the single PacMan game running on the individual devices, and the multiplayer functionality. In addition, we distinguish two application layers, namely the user interface layer, and the application logic layer. We will now sketch a possible architecture for both building blocks and including the relevant services, their interaction, and their belonging to one of the two application layers.

2.3.1 Architecture of the Single-Player PacMan Game

The single PacMan game can be divided into the following components, where a component may implement one or more services, and in addition may require other services in order to be executable:

- **OutputPacman:** A component that displays the ghosts and the PacMan, the maze including dots, the score, the left lives of the PacMan, and so on. This component implements services which belong to the user interface layer.
- **InputPacman:** A component which is responsible for accepting user input and forwards it to services which need information about what the user did, as for example a service within the application layer. The component itself belongs to the user interface layer. Possible user instructions could be to start/stop the game and steering directions of the PacMan.

- **GameLogic:** This component is responsible for steering the ghosts, computing the position of the PacMan based on user input coming from InputPacman, computing the score of the player, and so on. This is a typical example of a component which implements services within the application logic layer.
- **GameState:** This component encapsulates the state of the game as described in the previous bullet point. In many systems, those kinds of components also belong to the application layer.

These components are wired in order to build the single PacMan game. Figure 7 shows a possible wiring between these components. The structure is derived from the Model-View-Controller pattern described in (Buschmann et al., 1996). In this example, the *GameState* takes the role of the Model, the *GameLogic* the role of the Controller, and the *GUI* takes the role of the View.

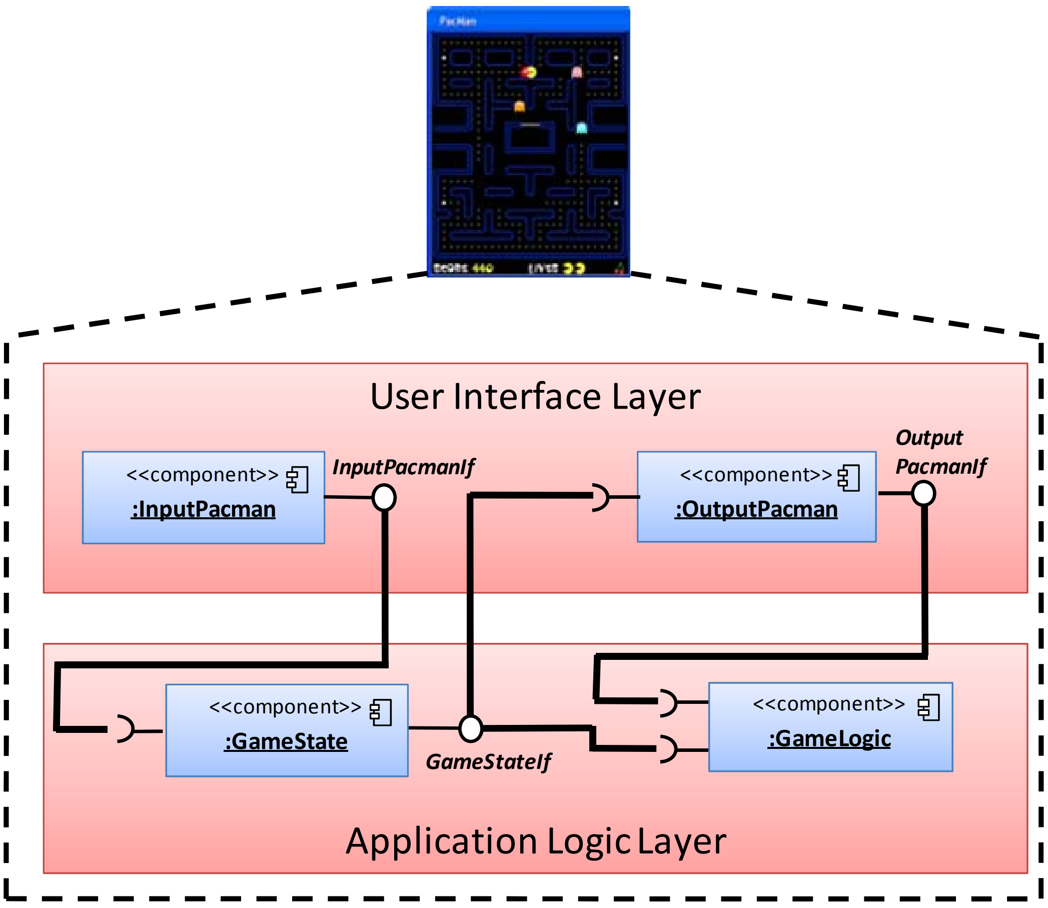


Figure 7: White-Box-view of a single PacMan game.

The figure shows one possible wiring of these components. The illustration is similar to the UML 2.0 notation for component diagrams. The blue boxes represent components, the circles attached to components represent provided services, and the semi-circles, also attached to components, represent required services. In the figure, the game state is manipulated by *InputPacman* and

GameLogic via the service called *GameStateIf*. The *OutputPacman* accesses the game state via the service called *GameStateIf* in order to display the various items of the game like dots, score of the player, or information like speed of ghosts and PacMan. In summary, the game consists of two services on the user interface layer, called *InputPacmanIf* and *OutputPacmanIf*, and of one service within the application logic layer called *GameStateIf*. Later in this section, we will extend the game by further services in order to demonstrate reconfiguration within the application logic layer.

2.3.2 Architecture of the Multi-Player Game

In the multiplayer scenario, multiple single PacMan games like described before have to cooperate in a way that they are able to send ghosts to another game. Therefore, the different instances of the PacMan games have to be orchestrated like shown in Figure 8. At this, a central orchestrator is responsible for orchestrating the different PacMan games to build the multiplayer PacMan. The main difference between this orchestration approach and the wiring of services described in the previous section is that components in the orchestration approach are not communicating directly with each other, but through the orchestrator. To do this, the orchestrator decides which method to call at which service. Furthermore, the orchestrator is responsible to manage the data flow.

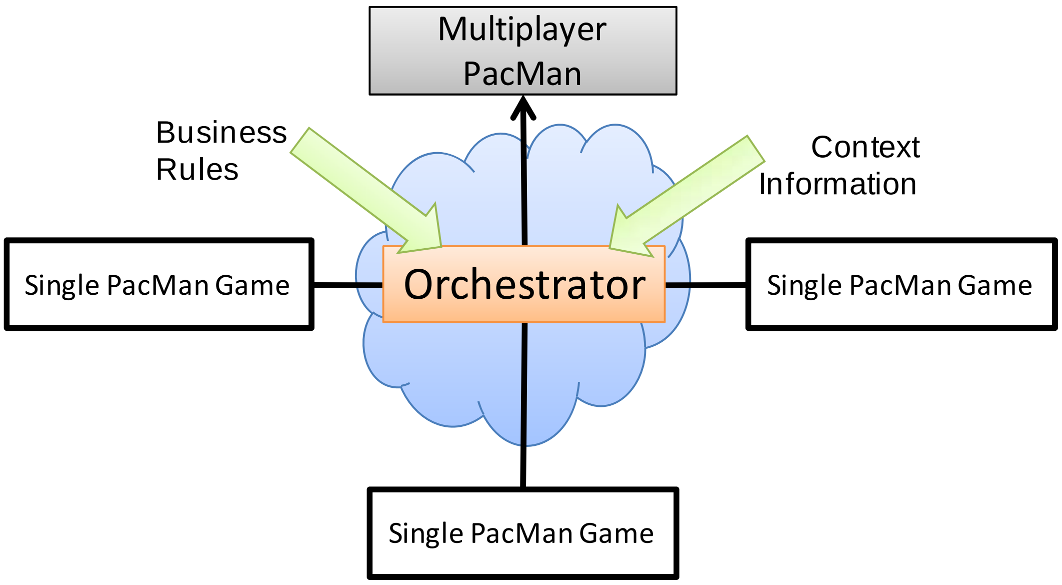


Figure 8: Orchestration of single PacMan games considering business rules and Context Information in order to build a new application called Multiplayer PacMan.

Business rules can be used to define how the orchestration should be performed, which components should interact, and under which circumstances they should interact. The outcome is a new functionality, which is in this case the multiplayer PacMan game. The rules have also to consider context information like location of persons and devices, as well as information like battery power for example.

In the next sections, we will describe techniques for realizing these different kinds of adaptation using the example scenario described above.

3 Solutions for Application Logic Reconfiguration

In this section, we will first describe different types of adaptation using the PacMan game introduced above. The adaptation approaches can be distinguished in what we call wiring approach and orchestration approach. The main differences between the wiring approach and the orchestration approach and how they can be combined to allow a large diversity of supported applications will be described next. Afterwards we will describe both approaches in detail including techniques for application logic reconfiguration.

3.1 Differentiation of Wiring and Orchestrating Services

In service-oriented applications, we can distinguish the wiring of services and the orchestration of services, no matter if these are services within the application logic layer or the user interface layer. In Figure 9 the wiring approach (left) is opposed to the orchestration approach (right).

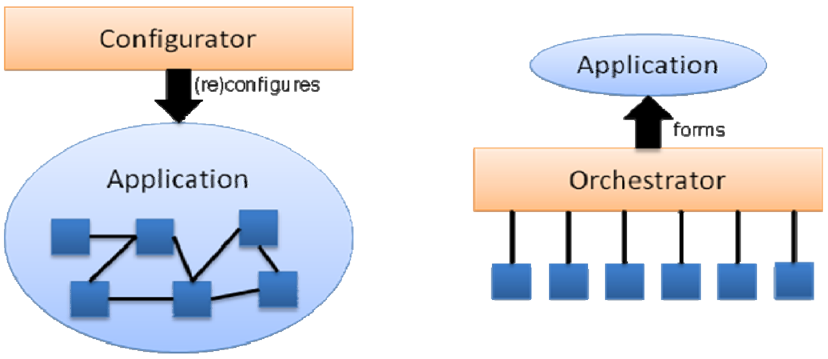


Figure 9: Illustration of the wiring approach on the left and the orchestration approach on the right.

In the wiring approach, the application is built out of services, which communicate directly with each other. A Configurator is responsible to build the application by wiring the different services, starting them and if required adapts the application. In the next section, we will describe in detail which kind of adaptation can be considered.

In the orchestration approach on the other hand, the various services are not connected with each other directly. To build an application, a central orchestrator accesses required services and manages the data flow between the various services. Out of that, a new application is built like shown in Figure 9 on the right hand side.

Due to the fact that an application itself again can be a service, both approaches can be combined in at least two ways like illustrated in Figure 10.

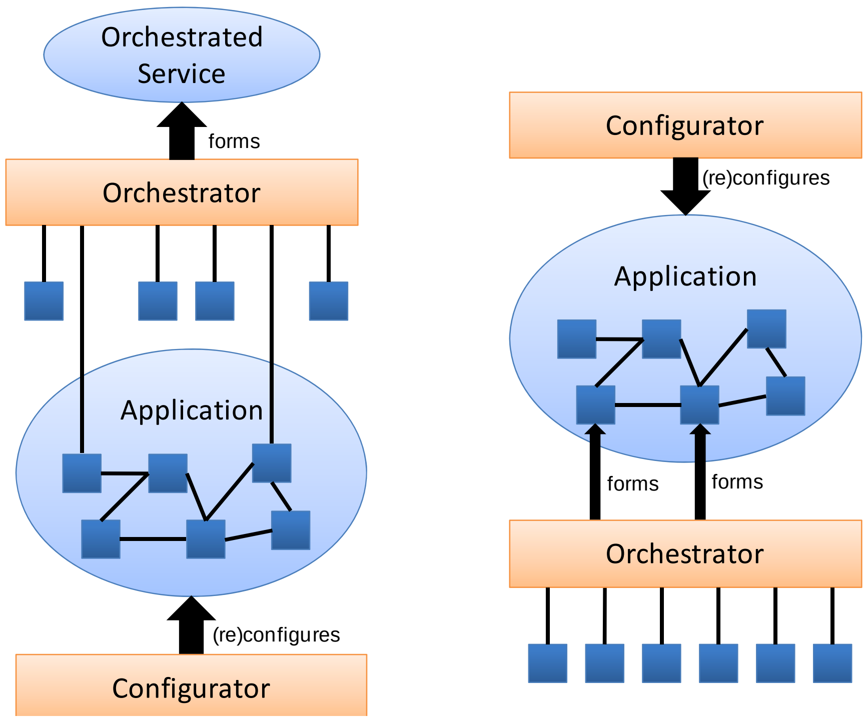


Figure 10: Two possible ways of combining the wiring approach and the orchestration approach.

On the left hand side, an application is built out of single services. In addition, some of these services are also used within the realization of an orchestrated service. On the right hand side of Figure 10, the Orchestrator forms a service, which in turn is used within an application where services are wired by a Configurator.

In fact, if both approaches are combined, the behaviour of a wired application and an orchestrated application may influence each other. If for example the Configurator decides to remove a certain component, which in turn is part of an orchestrated service, the orchestrated service has to adapt itself as well. If on the other hand the orchestrator changes the behaviour of an orchestrated service, the Configurator may have to adapt the wired service as well.

In the following, we will finally describe both approaches in detail including solutions for reconfiguration.

3.2 Wiring Approach

We mainly distinguish two types of adaptation within the wiring approach, which will be described in the following sections. These types of adaptation have already been described in (Niebuhr et al., 2007). In addition, we will describe how the different adaptation types can be realized and how they can be applied to the PacMan game.

3.2.1 Service Usage Adaptation

As we already described before, services usually interact with other services like shown in the single-player version of the PacMan game. In the following we are considering how to realize the adaptation scenario depicted in Figure 3 where the control of the PacMan migrates from the keyboard of the PC to the accelerometer of the PDA. As already described before, this migration leads not only to adaptation of user interface services, but also to adaptation of application logic services. Figure 11 shows the deployment and the configuration before and after migration of the *InputPacman* component from the PC to a PDA.

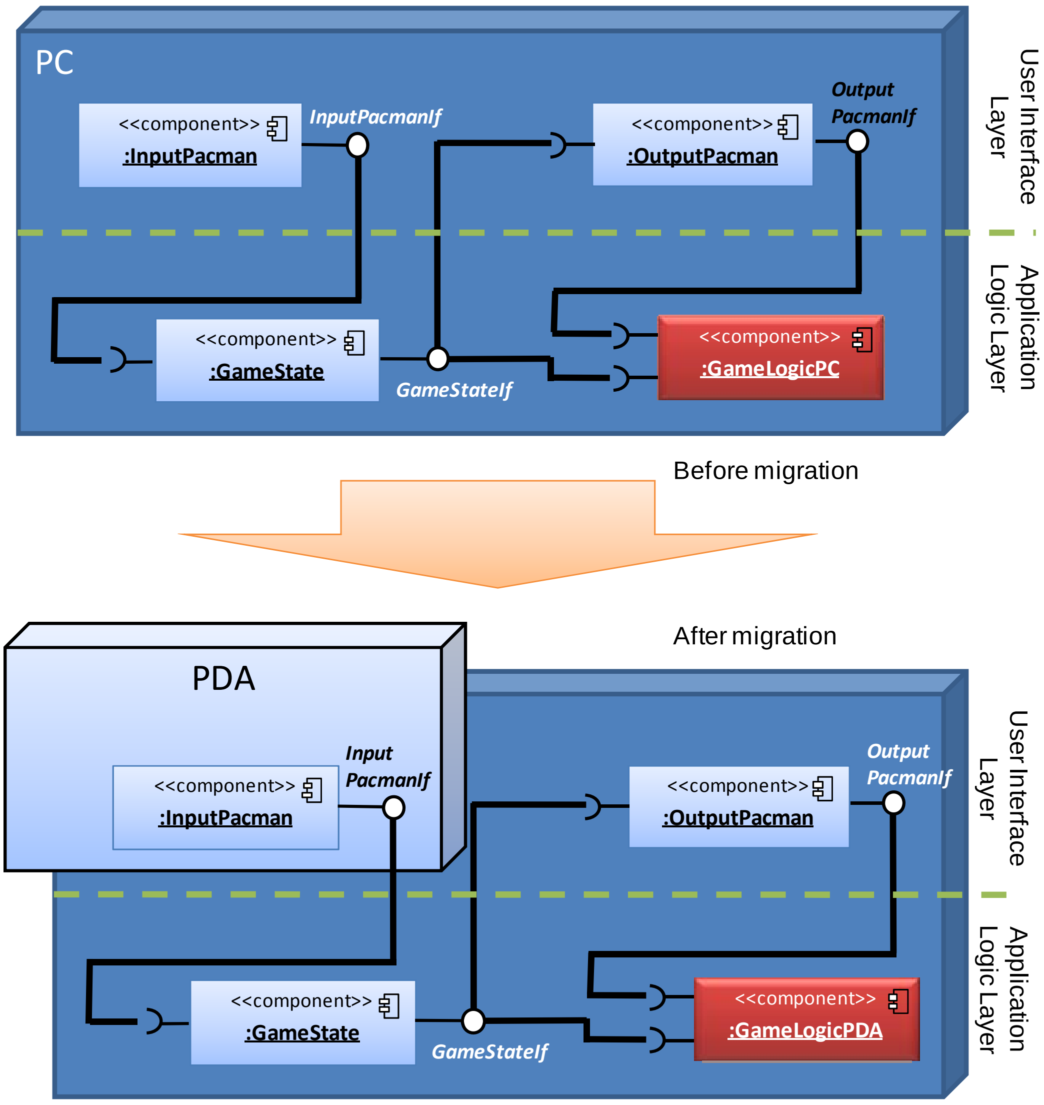


Figure 11: Migration of the *InputPacman* component and the resulting change within the application logic layer.

As already described in Section 2.2, the application logic has to adapt to the new situation for example by slowing down the speed of the ghost as it is now more difficult to steer the ghost. One possible solution is, to replace the *GameLogic* component by another one which implements the appropriate behaviour. In this case, the component *GameLogicPC* is replaced by a component called *GameLogicPDA*. This example shows how changes in the user interface services result in adaptation of the application logic.

Service Usage Adaptation may also occur during migration of the whole game from the PC to the PDA. Also other components may have to be replaced in some situations. The game state component for example could be replaced by another one, if the user interface layer offers new kind of state information. However, the replacement of components during runtime is a kind of adaptation which may occur in many applications and situations.

We call this type of adaptation *Service Usage Adaptation*, because the service in use is changed based on information like context, user preferences or other information. A middleware could perform this kind of adaptation automatically like introduced in (Niebuhr et al., 2007).

3.2.2 Service Behaviour Adaptation

This type of adaptation considers the adaptation of the behaviour of single services without replacing them. Adaptation means that the behaviour of single services changes with respect to context information, user preferences, or with respect to the availability of other services.

To illustrate *Service Behaviour Adaptation* we will now enrich the *GameLogic* component by two so called *Component Configurations*. Each component configuration is attached with required services (semi-circles). If all required services of one configuration are available, the configuration can become active and will now offer all attached provided services. But not only new services may become available; also the behaviour of a single service may change according to the currently active component configuration.

Figure 12 shows the enriched *GameLogic* component. The component configuration called “Standard” implements the same behaviour as described in the previous section. It requires the services *GameStateIf* and *OutputPacmanIf* to be present in order to become active. As both services are available, the component configuration called “Standard” becomes active.

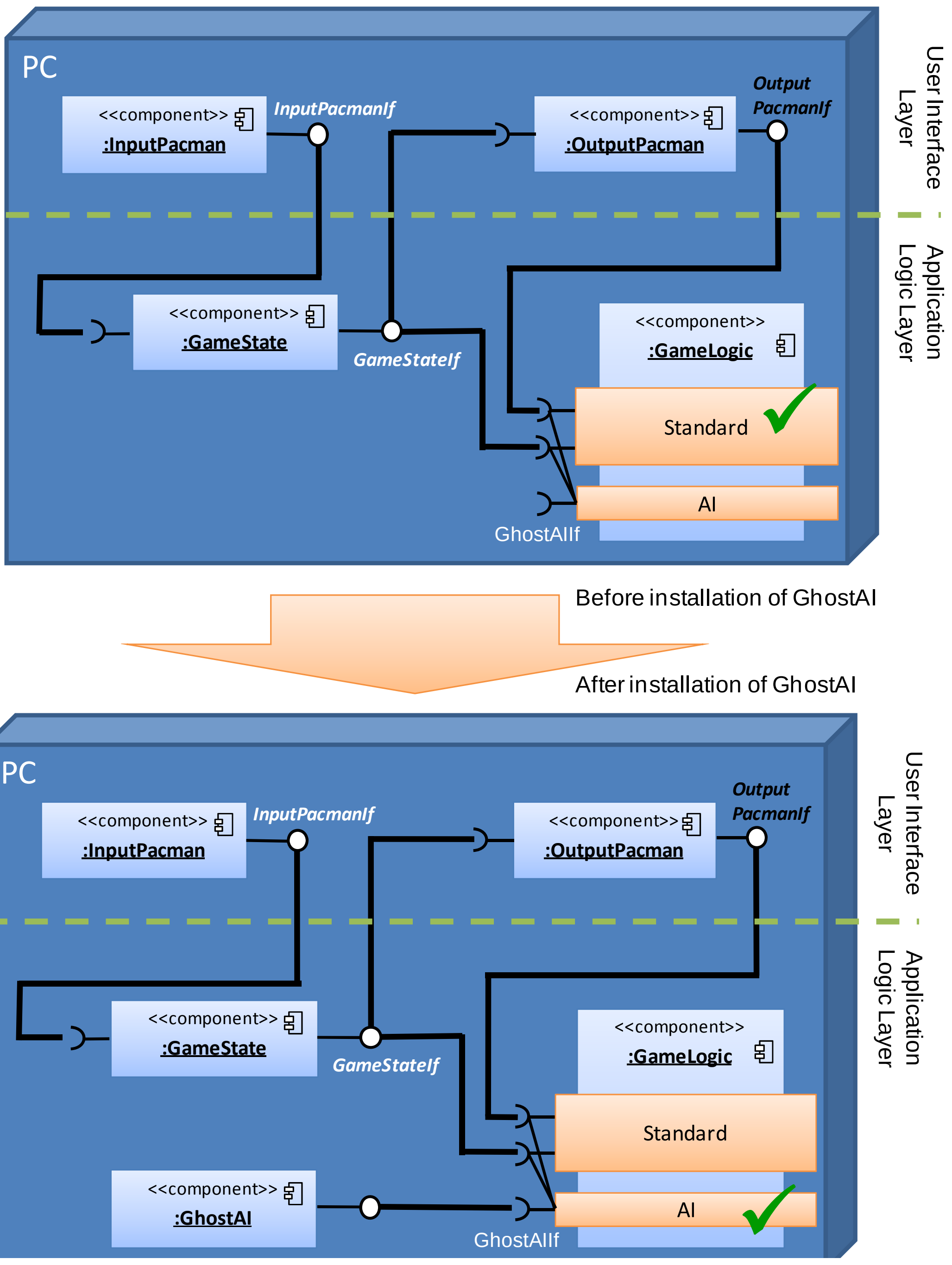


Figure 12: Example which illustrates the concept of component configurations. The *GameLogic* component is therefore enriched with two component configurations.

If now a service *GhostAIIf* becomes available, the component configuration of *GameLogic* switches from “Standard” to “AI”. That may result in a higher difficulty for the PacMan while running away from the ghosts as the ghosts now behave according to some kind of artificial intelligence.

Component Configurations may not only change on behalf of available services, but also with respect to the change of context information like battery power of the device hosting the component.

Again, this kind of adaptation can be performed automatically by an appropriate middleware. Doing so, an unobtrusive way to perform adaptation can be realized.

3.3 Orchestration Approach

The orchestration approach enables the application developer and/or the application provider to define a workflow, which provides an overall application logic description and decides how and when the different services interact. The orchestration approach is mainly used for business processes: a business process can be modeled as a sequence of services, e.g. web services, with a specific language, as BPEL (Business Process Execution Language) (Alves et al., 2007). This model is used by an orchestration engine, as ActiveBPEL (Active Endpoints, 2008), which creates an instance of the process. The engine calls the different services involved in the process, maintaining the control of the process during all the time in which it is running.

The main components of the orchestration approach are:

- The workflow languages or the business process modeling language: these languages define the grammar for connecting services or tasks to produce an application logic description.
- The available tools for specifying the application logic: different tools are available for supporting the application logic description, as for example graphical tools which enable the designer to describe the application logic as a workflow diagram.
- The orchestration engine: it takes as input the application logic description and creates an instance of the process. The engine calls the different services involved in the process, maintaining the control of the process during all the time in which it is running.

In the next paragraph, the evaluation of state of art technologies will be addressed, using a list of criteria to compare and to evaluate the existing languages and workflow engines.

3.3.1 Workflow Patterns

We refer to Workflow Patterns (Aalst et al., 2004; Aalst et al., 2007), which provide a thorough examination of the various perspectives that need to be supported by a workflow language or a business process modelling language. Workflow Patterns are widely used for examining the suitability of a particular process language or workflow system for a particular project, assessing relative strengths and weaknesses of various approaches to process specification, implementing certain business requirements in a particular process-aware information system, and as a basis for language and tool development.

In process-aware information systems, various perspectives can be distinguished.

- The control-flow perspective captures aspects related to control-flow dependencies between various tasks (e.g. parallelism, choice, synchronization etc). Originally, the Workflow Pattern Initiative proposes twenty patterns for this perspective, but in the latest iteration this has grown to over forty patterns.
- The data perspective deals with the passing of information, scoping of variables, etc.
- The resource perspective deals with resource to task allocation, delegation, etc.

The exception handling perspective deals with the various causes of exceptions and the various actions that need to be taken as a result of exceptions occurring.

3.3.2 Example of Workflow Patterns

In this paragraph, some examples taken from the control-flow patterns are given. The control-flow perspective captures aspects related to control-flow dependencies between various tasks (e.g. parallelism, choice, synchronization etc) (Russell et al., 2006) and are more intuitive respect to the other patterns. Simple examples of control flow patterns are illustrated using the Colored Petri-Net (CPN) formalism. This allows providing a precise description of each pattern that is both deterministic and executable.

A Petri net is a directed graph, in which:

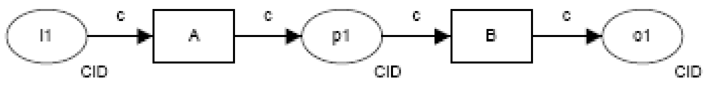
- the nodes represent transitions (i.e. discrete events that may occur) and places (i.e. conditions)
- the directed arcs describe which places are pre- and/or post-conditions for which transitions.

Arcs run between places and transitions, never between places or between transitions. The places from which an arc runs to a transition are called the input places of the transition; the places to which arcs run from a transition are called the output places of the transition. Places may contain any non-negative number of tokens. A distribution of tokens over the places of a net is called a marking. A transition of a Petri net may fire whenever there is a token at the end of all input arcs; when it fires, it consumes these tokens, and places tokens at the end of all output arcs. A firing is atomic, i.e., a single non-interruptible step.

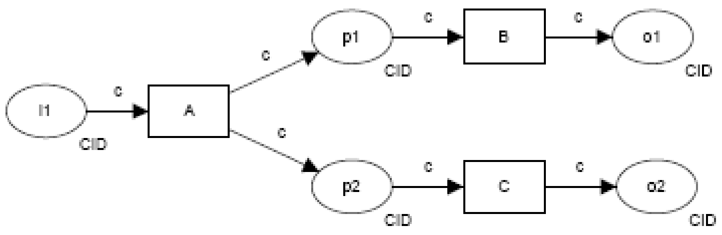
There are some blanket assumptions that apply to all of the CPN models used in this document. For each of them, we adopt a notation in which input places are labelled i1...in, output places are labelled o1...on, internal places are labelled p1...pn and transitions are labelled A...Z, tokens are represented by the “c”. In general, transitions are intended to represent tasks or activities in processes, and places are the preceding and subsequent states which describe when the activity can be enabled and what the consequences of its completion are. We assume that the tokens flowing through a CPN model are typed CID (short for “Case ID”) and that each executing case (i.e. process instance) has a distinct case identifier.

Examples of Basic Control Flow patterns capturing elementary aspects of process control are listed below.

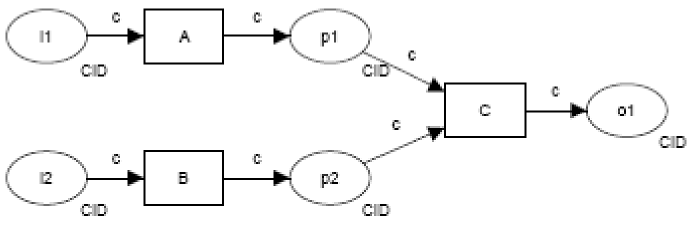
- **Sequence:** a task in a process enabled after the completion of a preceding task in the same process.



- **Parallel Split:** the divergence of a branch into two or more parallel branches each of which execute concurrently.

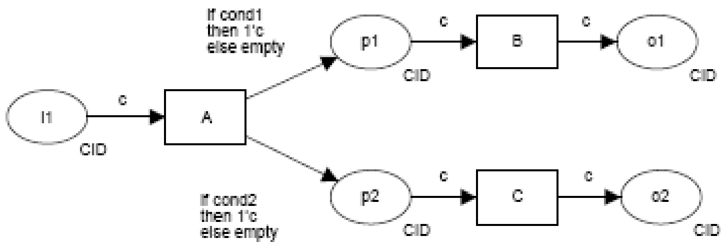


- **Synchronization:** the convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled.

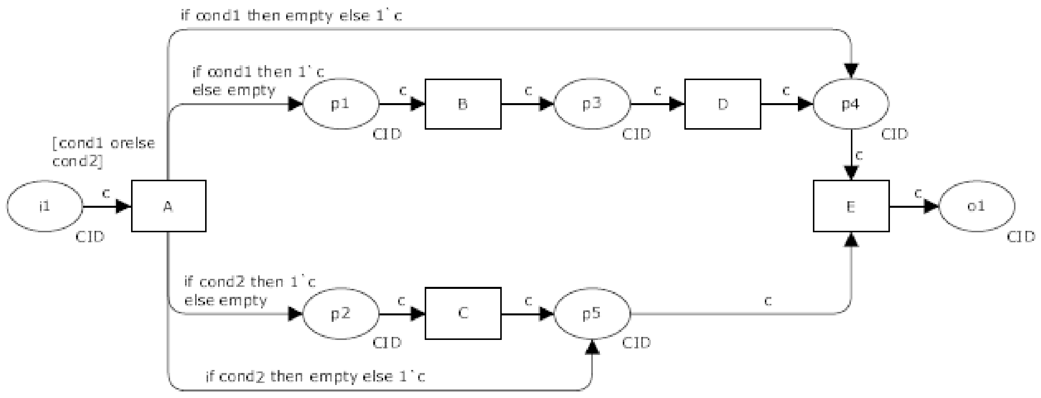


More complex patterns are defined in order to design more elaborate workflow, as for example the Advanced Branching and Synchronization Patterns, used in order to characterise more complex branching and merging concepts. Some pattern examples are:

- **Multi-Choice:** The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches.



- **Structured Synchronizing Merge:** The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. The *Structured Synchronizing Merge* occurs in a structured context, i.e. there must be a single *Multi-Choice* construct earlier in the process model with which the *Structured Synchronizing Merge* is associated and it must merge all of the branches emanating from the *Multi-Choice.* These branches must either flow from the *Structured Synchronizing Merge* without any splits or joins or they must be structured in form (i.e. balanced splits and joins).



The exhaustive description of the workflow patterns is out of scope for this document, a complete description can be found in (Aalst et al., 2007).

3.3.3 Evaluation of State of the Art Technologies

The evaluation of existing service orchestration and workflow engine tools is based on the parameters listed in the Table 1. This is a first round of evaluation of the state of art technologies, if the orchestration approach will be chosen for the OPEN platform, a second round of evaluation should be performed, in order to select the proper system to adopt.

Table 1: Evaluation parameters.

Criteria	Description	Values
Workflow	The Workflow Patterns are a well known method to	0: limited number of

patterns	analyze the representation capacity of a workflow modeling language. The complete description of the workflow patterns is available on (Aalst et al., 2007)	workflow patterns represented 1: able to represent crucial workflow patterns 2: able to represent a great number of workflow patterns
Semantic Support	There is semantic support to semantically communicate the processes in the proposed model?	0: No semantic support 1: Semantic support
Efficiency	There is an engine available that implements the proposed model? Is this engine efficient?	0: No available Engine 1: Engine available, but slow 2: Engine available and efficient
Easiness to understand and design	Is the proposed model easy to understand even for process design people? There are designing tools available to build processes choreography?	0: Model difficult to understand & design 1: Model understandable, but design by coding 2: Model understandable, with native authoring tool
Connectivity	Is it possible to connect the system with external systems? The connectivity is made only by Web Services or it is possible to create native process coding?	0: Unable to connect with external systems 1: Interoperable with external systems 2: Able to be integrated with external system
Time modeling	Is it possible to define time driven processes in the model (i.e. wait 2 hours, or begin at 15:30)?	0: Unable to model the time 1: The time model can be simulated 2: Able to odel the time
Extensibility	Is it possible to derive custom patterns from the proposed model?	0: Unable to be modified 1: Open Source, but difficult to modify 2: Able to create custom patterns

3.3.4 jBPM

JBoss jBPM (JBoss, 2009) is a framework enabling the user to create and automate business processes that coordinate between people, applications, and services. JBoss jBPM provides both the tools for an easy development of workflow applications and a process execution engine to integrate services.

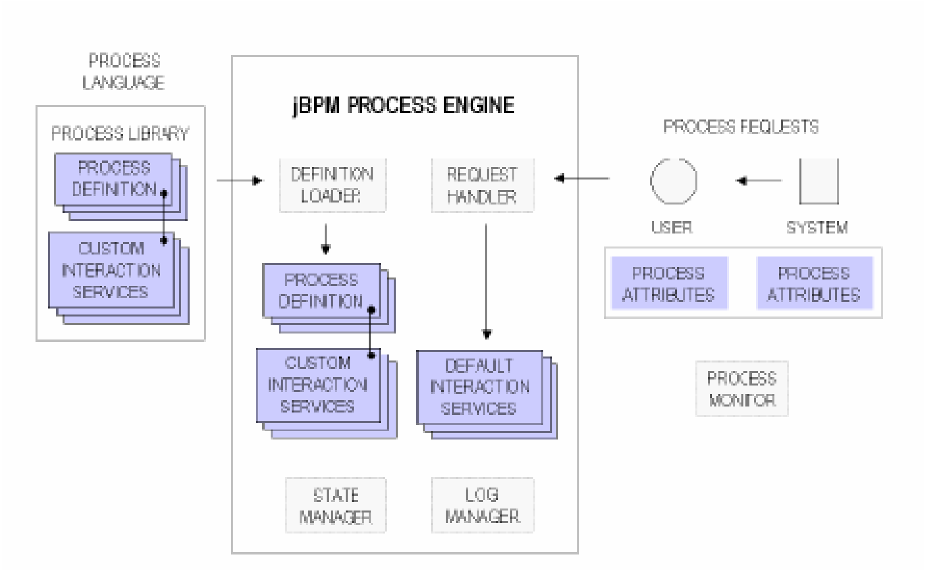


Figure 13: jBoss jBPM system.

The system contains the following main components:

- The jBPM process engine: takes care of the execution of process instances. JBoss jBPM process engine provides a powerful foundation for orchestrating interactions between applications and services. It is suited to service-oriented architectures and is interoperable with all of the J2EE-based integration technologies including Web Services, Java Messaging, J2EE Connectors, JDBC, and EJBs. The process engine automatically handles state, variable, and task management as well as process timers.
- jPDL (Process Definition Language): process oriented programming model.
- JBoss jBPM GPD (Graphical Process Designer): provides support for defining processes in jPDL. This tool is a plugin to Eclipse. A screenshot of the tool is depicted in Figure 14.
- JBoss jBPM console web application: it is a web based workflow client whereby, in Home mode, users can initiate and execute processes. It is also an administration and monitoring tool, which offers a Monitoring mode where users can observe and intervene in executing process instances.
- JBoss jBPM identity component, which will take care of the definition of organizational information, such as users, groups and roles to which different tasks can be assigned. Currently the definition of all these information is done through standard SQL insert statements directly in the workflow database.

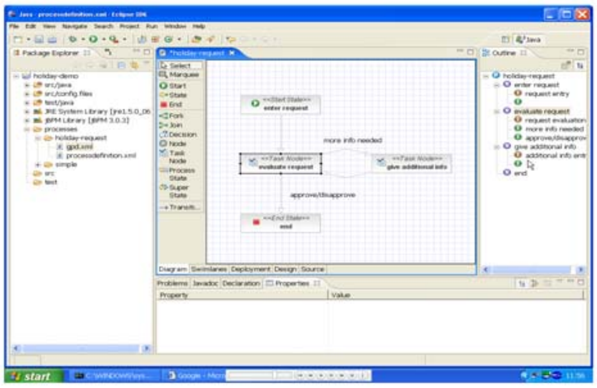


Figure 14: jBoss jBPM GPD.

In Table 2, the evaluation of jBoss jBPM is provided.

Table 2: Evaluation of jBoss jBPM.

Criteria	Description	Values
Workflow patterns	Not able to represent some patterns, however these can be simulated using arbitrary cycles.	2: able to represent a great number of workflow patterns
Semantic Support	jBPM does not have semantic support.	0: No semantic support
Efficiency	jBPM is a heavy engine that requires a powerful server.	1: Engine available, but slow
Easiness to understand and design	If the process designer is intimately familiar with Java, jBPM may be a good choice, while if this is not the case, choosing jBPM is less advisable.	1: Model understandable, but design by coding
Connectivity	jBPM is able to be integrated with external system using Java coding.	2: Able to be integrated with external system
Time modeling	Time can be simulated; jBPM has a calendar class and other time tools to implement this.	1: The time model can be simulated
Extensibility	It's not possible to derive custom patterns but we can design custom activities.	0: Unable to be modified

3.3.5 YAWL

YAWL (Yet Another Workflow Language) (Hofstede et al., 2008) is a workflow language defined by the authors of the reference articles on workflow patterns (Aalst et al., 2007). YAWL is supported by a software system that includes an execution engine and a graphical editor. The system is open source, distributed under the LGPL license. In Figure 15 a screenshot of the YAWL editor is provided.

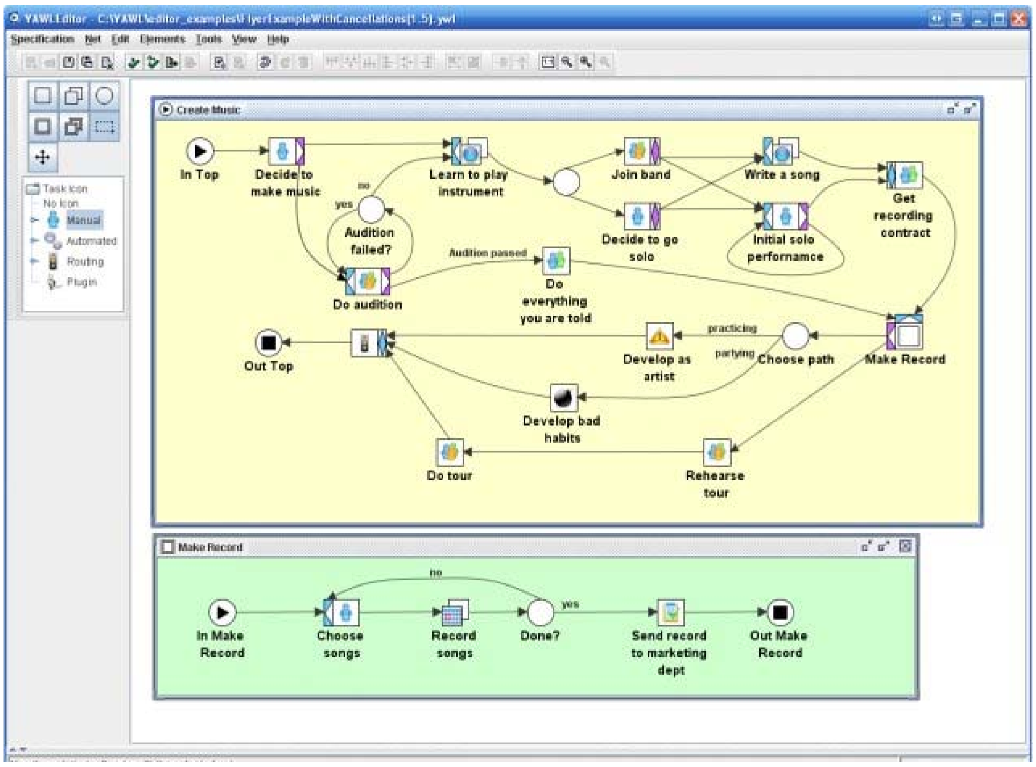


Figure 15: screenshot of YAWL editor.

Designers of YAWL decided to take Petri nets as a starting point and to extend this formalism with three main constructs, namely or-join, cancellation sets, and multi-instance activities. These three concepts are aimed at supporting five of the Workflow Patterns that were not directly supported in Petri nets, namely synchronizing merge, discriminator, N-out-of-M join, multiple instance with no a-priori runtime knowledge and cancel case. In addition, YAWL adds some syntactical elements to Petri nets in order to intuitively capture other workflow patterns such as simple choice (xor-split), simple merge (xor-join), and multiple choice (or-split). During the design of the language, it turned out that some of the extensions that were added to Petri nets were difficult or even impossible to re-encode back into plain Petri nets. As a result, the original formal semantics of YAWL is defined as a Labeled Transition System and not in terms of Petri nets.

The evaluation of YAWL system is provided in Table 3.

Table 3: Evaluation of YAWL system.

Criteria	Description	Values
Workflow patterns	Although YAWL was considered for the purposes of the patterns-based evaluation, some patterns needed structured loop to be represented. Nevertheless, it is possible to simulate it using arbitrary loops.	1: Able to represent crucial workflow patterns
Semantic Support	YAWL does not have semantic support.	0: No semantic Support
Efficiency	YAWL has workflow engine but still is a beta, its	1: Engine available, but

	engine is slow and unstable.	slow
Easiness to understand and design	YAWL has a workflow Editor written in Java.	2: Model understandable, with native authoring tool
Connectivity	YAWL is able to be integrated with external system using Java.	2: Able to be integrated with external system
Time modeling	It is possible to define time driven processes with timers using the YAWL editor and it is possible to control processes life cycle with the workflow engine	1: The time model can be simulated
Extensibility	YAWL is open source.	1: Open Source, but difficult to modify

3.3.6 Windows Workflow Foundation

Windows Workflow Foundation (Microsoft, 2009) is a framework that enables users to create workflows. Windows Workflow Foundation comes with a programming model, a hostable and customizable workflow engine, and tools for quickly building workflow-enabled applications on Windows.

Windows Workflow Foundation supports the following authoring modes for workflow implementation:

- Code-only. This is the default authoring mode for Windows Workflow Foundation. It enables you to use C# or Visual Basic code to specify a workflow using the Windows Workflow Foundation API set. In the code-only workflow, the workflow definition uses C# or Visual Basic code to declare the workflow structure. A code-only workflow must be compiled.
- Code-separation. This mode enables you to define workflows by using workflow markup and combining it with C# or Visual Basic code. Unlike the no-code authoring mode, code-separated workflows must be compiled and do not have the option of being loaded directly into the workflow runtime engine.
- No-code. This mode enables you to create a workflow by using workflow markup. You can then compile the workflow with the Windows Workflow Foundation command-line workflow compiler, or you can load the workflow markup file into the workflow runtime engine through a host application. Windows Workflow Foundation gives designers and developers a declarative way to create workflows by using eXtensible Application Markup Language (XAML) to create markup source files.

Every running workflow instance is created and maintained by an in-process runtime engine that is commonly referred to as the workflow runtime engine. When a workflow model is compiled, it can be executed inside any Windows process including console applications, forms-based applications, Windows Services, ASP.NET Web sites, and Web services. Because a workflow is hosted in process, a workflow can easily communicate with its host application.

In Table 4 the evaluation of the windows workflow foundation is provided.

Table 4: Evaluation of WF.

Criteria	Description	Values
Workflow patterns	Although WF is not based in standard patterns, it is able to represent all recommended workflow patterns (Design can be very complex in some cases)	2: Able to represent a great number of workflow patterns
Semantic Support	WF does not have semantic support.	0: No semantic support
Efficiency	Engine is heavy.	1: Engine available, but slow
Easiness to understand and design	The proposed model is easy to understand although workflows are addressing to code.	1: Model understandable, but design by coding
Connectivity	The connectivity is made only by Web Services in ASP.NET and the connectivity with external systems can be made using C#.	2: Able to be integrated with external system
Time modeling	WF has a hosting layer with runtime services, one of them is the timer runtime service. At workflow model level we can design workflows with delay activities.	2: Able to Model the time
Extensibility	It is not possible to derive custom patterns but we can design custom activities.	0: Unable to be modified

3.3.7 OWL-S

OWL-S (Martin et al., 2004) is based on the OWL (Ontology Web Based) Recommendation and supplies a core set of markup language constructs to describe Web services in an unambiguous, computer-interpretable form. To make use of a Web service, a software agent needs a computer-interpretable description of the service, and the means by which it is accessed. In this context, an important goal for markup languages is to establish a framework within which these descriptions are made and shared. Web sites should be able to employ a standard ontology, consisting of a set of basic classes and properties, for declaring and describing services, and the ontology structuring mechanisms of OWL provides an appropriate, Web-compatible representation language framework within which to do this. OWL-S enables the creation of ontologies for any domain and the instantiation of these ontologies in the description of specific Web sites. Tasks that OWL-S is expected to enable are:

- Automatic Web service discovery: automated location of web services (WSs) that provide a particular service and adhere to requested constraints
- Automatic Web service invocation: automated execution of an identified WS by a computer program or agent
- Automatic Web service composition and interoperation: automatic selection, composition and interoperation of WSs to perform some tasks
- Automatic Web service execution monitoring: individual services and composite services generally require some time to execute completely; it is useful to know the state of execution of services

The OWL-S ontology has three main parts: the service profile, the process model and the grounding.

- The service profile is used to describe what the service does. This information is primary meant for human reading, and includes the service name and description, limitations on applicability and quality of service, publisher and contact information.
- The process model describes how a client can interact with the service. This description includes the sets of inputs, outputs, pre-conditions and results of the service execution.
- The service grounding specifies the details that a client need to interact with the service, as communication protocols, message formats, port numbers, etc.

The following table provides the evaluation of OWL-S.

Table 5: Evaluation of OWL-S.

Criteria	Description	Values
Workflow patterns	OWL-S offers many control constructs.	1: Able to represent crucial workflow patterns
Semantic Support	There is semantic support to semantically communicate the processes in the proposed model (ontology).	1: Semantic support
Efficiency	There is no engine available.	0: No available Engine
Easiness to understand and design	The model is quite easy to understand. The OWL-S environment exists in the form of a loose collection of individual tools that focus on different specific aspects of its conceptual model.	2: Model understandable, with native authoring tool
Connectivity	The connectivity is given in the "grounding". The grounding provides the needed details about transport protocols, allowing for automatic invocation of services.	1: Interoperable with external systems
Time modeling	From the specification it seems that it is not possible to define time driven processes.	0: Unable to model the time
Extensibility	Is should be possible to derive custom patterns from the proposed model.	2: Able to create custom patterns

3.3.8 Orchestration Tools Comparison

The features of the tools that have been evaluated are summarized in the following.

jBPM has a wide community and it is always in a continuous developing and bug-fixing processes. jBPM includes an Eclipse plug-in to model business processes. The developer can specify a workflow using a drag and drop interface, adding nodes and transitions in a very seamless way. Eclipse automatically creates an XML file that defines the process. The developer can deploy this file and jBPM

workflow engine controls its execution. jBPM is a Java-based solution, also supporting OSGi.
The YAWL initiative comes from the workflow research community, it supports almost all workflow patterns and is available a graphical editor. YAWL is an open source and is able to be integrated with external system using Java. The major drawback of the YAWL engine is quite inefficient and far from being stable.
Windows Workflow Foundation (WF) is a Microsoft technology for defining, executing, and managing workflows. This technology is part of .NET Framework 3.0 which is available natively in the Windows Vista operating system. WF has many advantages, e.g. typical Visual Studio‘s “Drag and Drop” system design applied to flowcharts, similar to Visio or other drawing tools. When a workflow model is compiled, it can be executed inside any Windows process including console applications, forms-based applications, Windows Services, ASP.NET Web sites, and Web services. The main disadvantage of WF is the dependence of Windows platform.
OWL-S supplies a core set of markup language constructs to describe Web services in an unambiguous, computer-interpretable form. For OWL-S there is no engine available, however it can be used to establish a framework within which the services descriptions are made and shared in an environment in which other workflow engine are implemented.

3.4 Comparison of Wiring and Orchestration Approach

As introduced before, both approaches can be applied for specific application scenarios. In addition, they can be combined in order to build applications like the multiplayer PacMan for example as shown in Section 3.1. However, both approaches have their assets and drawbacks, which are summarized in Table 6.

Table 6: Summarized comparison between wiring and orchestration approach.

	Orchestration approach	Wiring approach
Application logic definition and maintenance	PRO: People without IT knowlege can design a workflow using available tools.	CON: Application logic is embedded in the code and therefore IT knowledge is required.
Adaptation	CON: High-level adaptation and therefore types of adaptation limited.	PRO: Fine-granular realization of adaptation and therefore manifold types of adaptation possible.
Available tools	PRO: Graphical tools for the workflow design are available.	CON: No graphical tools for specifying the wiring is available at the moment.
Application logic legibility	PRO: Workflows have a high legibility.	CON: application logic embedded in the code and therefore legibility may suffer. PRO: Framework support can alleviate definition of application logic.
Users	PRO: possible users are: -application developers -service providers	CON: Developers are main users.
Workflow monitoring	PRO: The orchestrator creates an instance of the workflow. The activities in a workflow instance can be monitored.	CON: The components know which method of which component to call, no easy monitoring can be performed. PRO: Graphical tool to monitor current system configuration is available.
Architecture supported	CON: Only a centralized architecture can be supported.	PRO: Both, centralized and distributed approach can be supported.
Service Coupling	PRO: Loosly coupled service enable independent service development CON: Performance of communication may suffer.	PRO: Tighter coupled services enable satisfactorily communication performance between services even in the game domain.

We identified various criteria which may be relevant in order to decide which approach to use in which use case. The orchestration approach on the one hand can be applied even by people without IT knowledge because of the availability of various graphical tools for designing workflows and service orchestrations. Furthermore, many realizations offer to monitor the execution of the orchestrated service. But the drawbacks are that a central orchestrator is required which in addition must have access to all required services. Therefore, such orchestrator may become a bottleneck. Furthermore, the performance of the communication between services is quite low as some of our experiments have shown. We will discuss this topic on more detail in Section 5.

The wiring approach on the other hand provides high performance in communication between services. But using this approach, the definition of workflows is not provided. However, as already shown in Section 3.1, both approaches can be combined in order to take advantage of both.

4 Architectural Solutions for Application Logic Reconfiguration

In this section we will present roughly three architectural alternatives regarding the realization of application logic reconfiguration.

4.1 Architecture of Wiring Approach

The wiring approach can be realized in two ways, namely a centralized and a decentralized approach. The main assumption of the centralized approach is that there exists a server which is accessible by all clients and vice versa like depicted in Figure 16. On the OPEN Server, the main part of the OPEN middleware is executed, including the *Configurator*. Other middleware components like the *Context* Manager or the *Migration Manager* may also run here. A complete list of middleware parts according descriptions are presented in (Nickelsen et al., 2009). The *Migration Manager* is responsible for controlling and executing the migration of services. The *Context Manager* on the other hand collects and evaluates available context information and makes this information available to other middleware parts. Finally, the *Configurator* performs the various kinds of adaptation, including the reconfiguration of wired services like described in Section 3.2.1 and 3.2.2. Therefore it may access the *Context Manager* in order to get the latest context information to determine the most appropriate kind of adaptation.

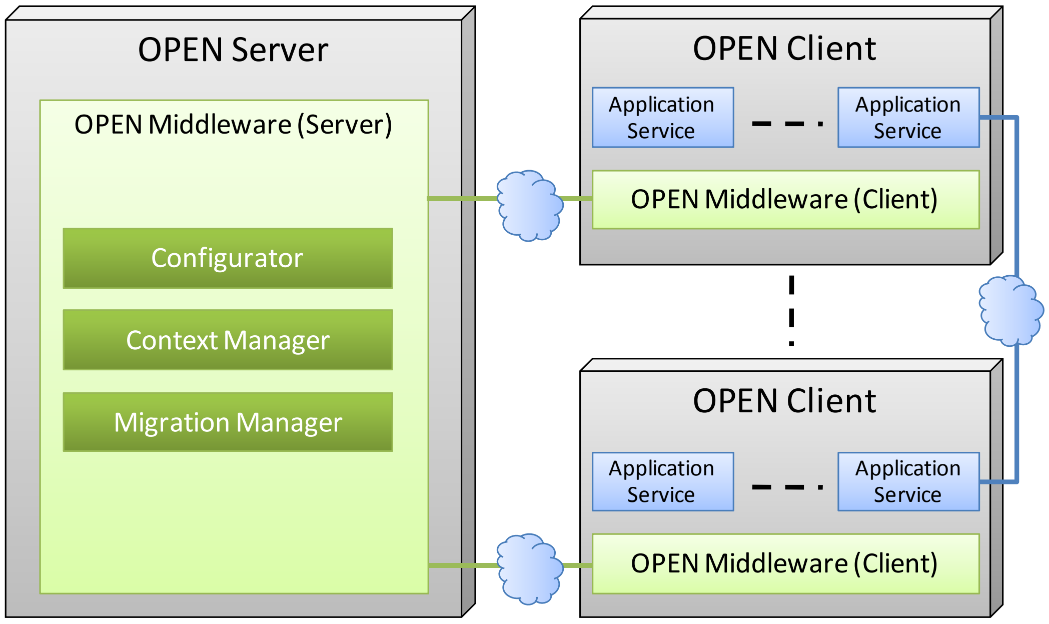


Figure 16: The centralized approach for realizing adaptation of services.

The *Application Services* on the other hand are services which implement an application or parts of it. They are executed on *OPEN Clients*. Furthermore, on each client a client-version of the OPEN middleware is running. Among others, it collects context information which the device provides like for example location, battery power, CPU rate, etc. and sends it to the *Context Manager* of the *OPEN Server*. Furthermore, it may realize reconfigurations initialized by the *Configurator*.

Following this centralized architecture, the deployment of the PacMan game could look like depicted in Figure 17.

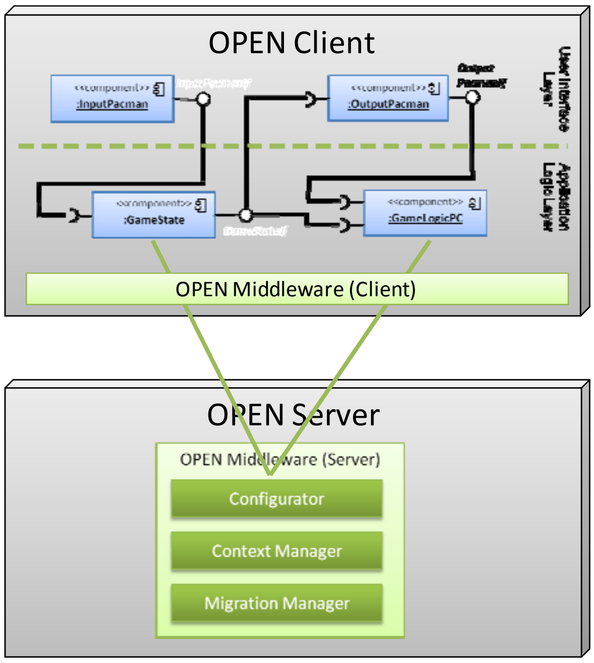


Figure 17: Centralized architecture for the single player PacMan before and after migration of the user input.

All parts of the application run in this case on one device together with the client version of the OPEN middleware. Other deployment alternatives are also possible, like for example running the *GameState* component on one PC, and the *GameLogic* on another. The main part of the OPEN middleware runs on the OPEN Server.

A contrary approach to the centralized approach is to omit the specific *OPEN Server*. Therefore, an application is built out of independent OPEN Clients, which build a federation like shown in Figure 18.

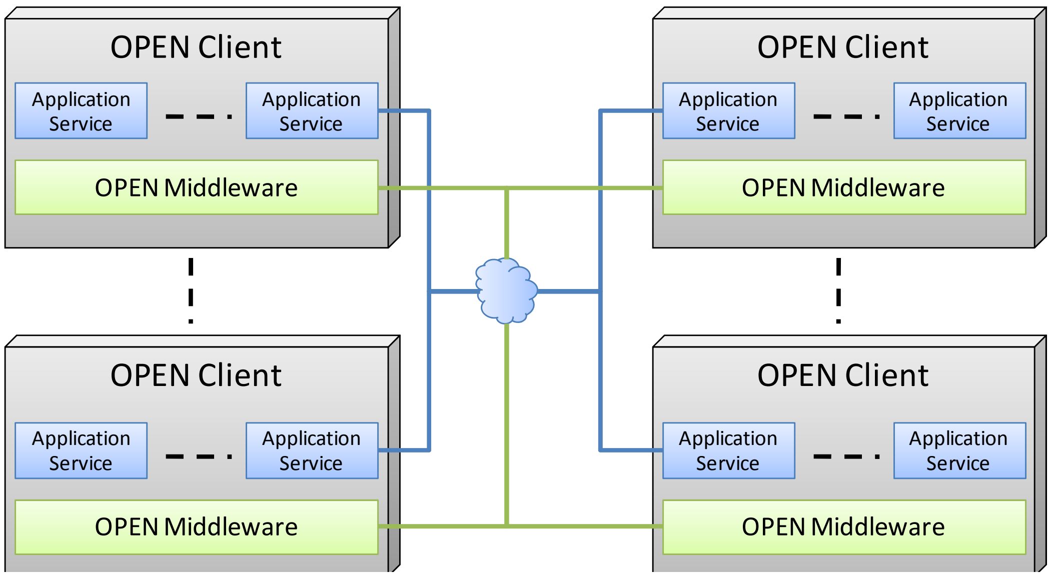


Figure 18: A decentralized architecture realizing adaptability of services.

This approach has some assets and drawbacks compared to the centralized approach. As there is no central server available, service discovery becomes more complex as well as managing context information and performing migration. Furthermore, the realization of service orchestration is not possible. The main advantage of a decentralized architecture is that the clients do not need a connection to a central server anymore. Thus, there is no single point of failure. On the other hand the middleware running on each client will be more complex than in the centralized approach. Adaptation and reconfiguration strategies would have to be established in a distributed way. Furthermore, the middleware may become too resource-consuming in order to be executable on a PDA for example.

4.2 Architecture for Orchestration Approach

The orchestration approach requires a central server which has access to all required services in order to orchestrate them building an application. In Figure 19 a possible deployment of the required software parts is shown.

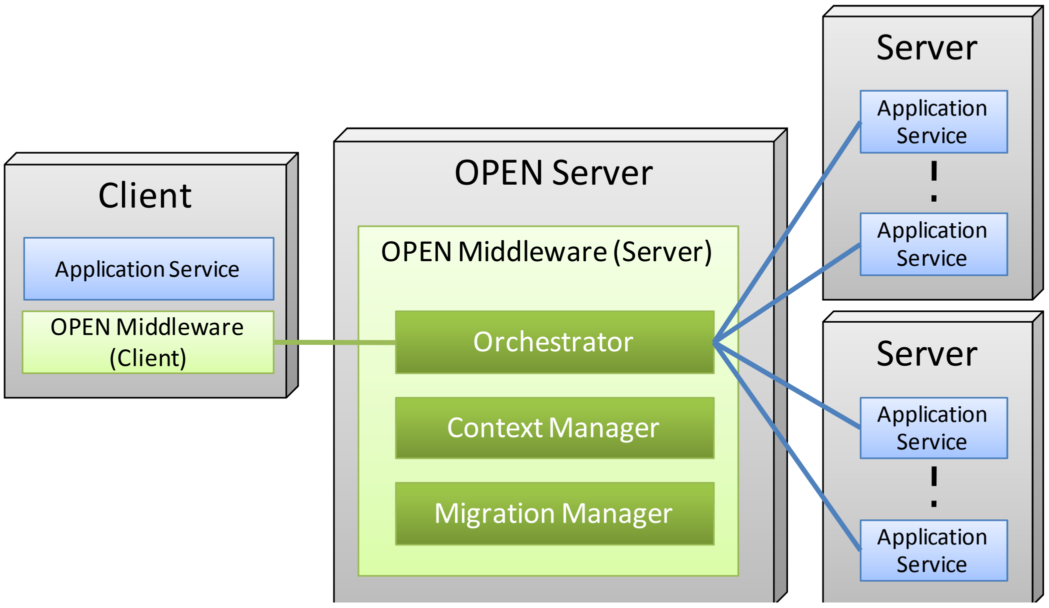


Figure 19: Architecture for the orchestration approach.

The Orchestrator decides which services to include in a workflow. To do this, the Orchestrator takes a workflow description like introduced in Section 3.3 and tries to find the according services within the network. Finally, the orchestrated services may build a new service, which in turn can be used by other services. Figure 20 shows how the deployment of the multi-player PacMan game could look like.

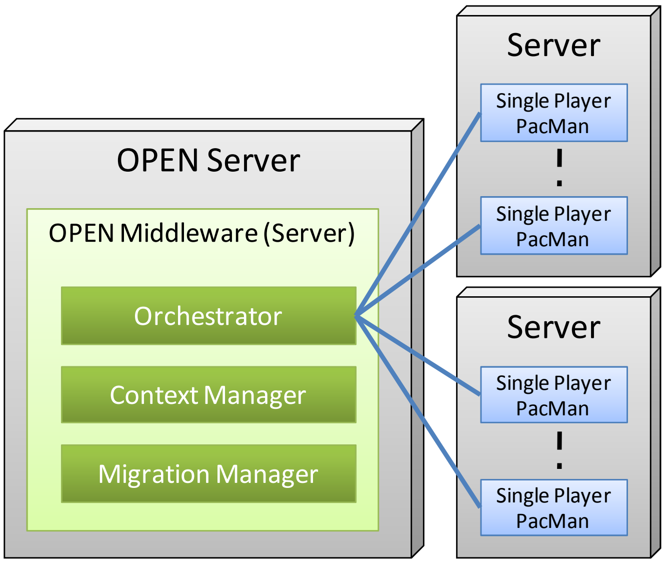


Figure 20: Orchestration approach for realizing the multiplayer PacMan game.

There are several single player PacMan games running on different devices. Each of them offers a service through which the Orchestrator can send and retrieve ghosts in order to realize the multi player PacMan game, like introduced in Section 2.2 and illustrated in Figure 6. The Orchestrator takes rules, which describe among others where to send ghosts or how to adapt the speed of the ghosts. To do this, the Orchestrator takes information from the Context Manager in order to decide how to adapt the orchestration.

4.3 Architecture of Combined Approach

An architectural solution for combining the wiring and the orchestration approach could look like is depicted in Figure 21.

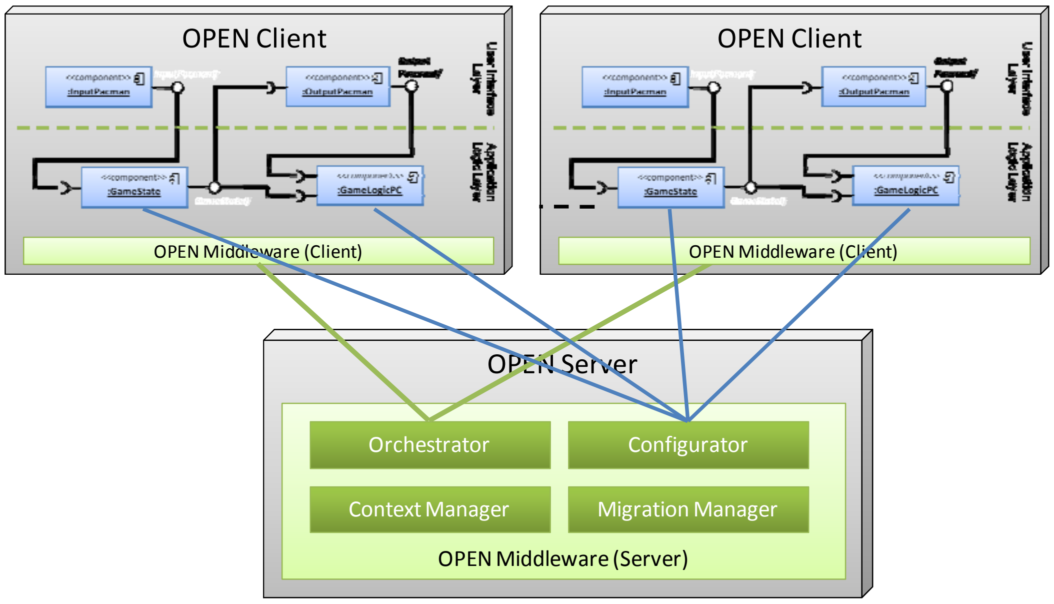


Figure 21: Possible deployment for the PacMan game integrating both, the wiring and the orchestration approach.

In this solution, the Configurator is responsible for adapting the single-player PacMan games while the Orchestrator coordinates the communication between them in order to realize the multi-player PacMan game

5 Communication Aspects

As already explained above, the application logic is mostly only one part of an application. Another important part is the user interface, which builds the bridge between the user interaction and application logic. That means, services within the user interface layer have to communicate with each other, and with services within the application logic layer. Figure 22 illustrates possible communication paths as black lines.

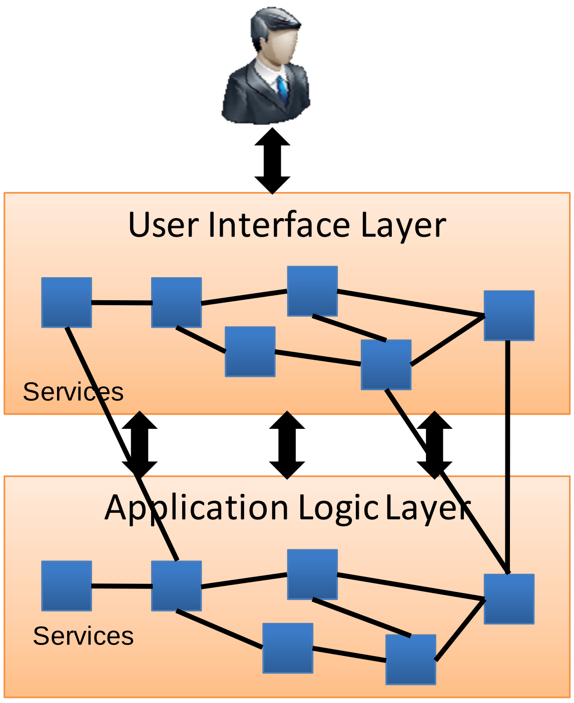


Figure 22: Possible communication paths within typical service-oriented applications.

There exist many technologies to realize the communication between services. TCP/IP for example is a quite low level mechanism to realize the communication. More high-level communication techniques are for example Remote Method Invocation (RMI) and Web Services based on the exchange of SOAP messages. Furthermore there exist more enhanced middleware frameworks like CORBA (Common Object Request Broker Architecture) or OSGi (Open Service Gateway initiative) for example, which bring already additional functionality with them like secure communication, look-up mechanisms, or event-based communication. There are a lot of parameters which influence the decision which technique to use. In the following we will shortly introduce some standard settings and appropriate techniques.

The first setting we consider is to have user interface services and application logic services running on two different machines. The user interface is executed in a browser and implemented with JavaScript while the application logic is written in Java using OSGi and which offers their functionality via Web Services by exchanging SOAP messages.

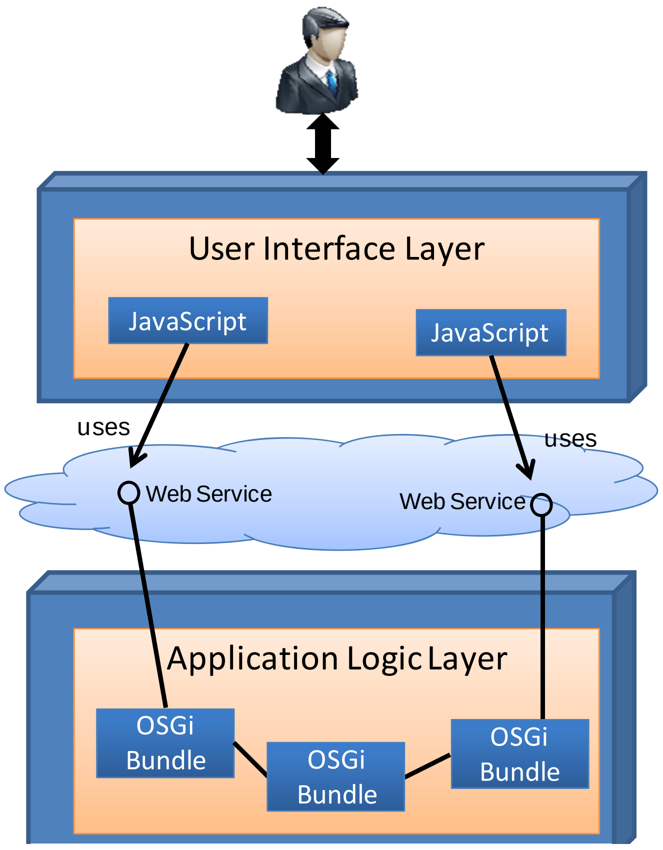


Figure 23: User Interface and application logic deployed on different machines and communicating via web services.

However, internally, the application layer services may communicate via RMI, OSGi, or CORBA for example. This setting is appropriate, if communication between the user interface services and application logic services does not have to be very fast as the transfer time of a single dataset through Web Services can take up to 400 milliseconds. However, the advantage is, that the user interface implementation and application logic implementation are independent from each other regarding the programming language and the operating system. The JavaScript application can for example be executed in a browser on a PDA with Linux running, while the application logic can reside on a PC with Windows running. It is also possible to use CORBA within the application layer instead of OSGi. Also adaptation for the user interface and application logic can be performed independently from each other.

In Figure 24 an alternative setting is shown. Here both, the user interface and the application layer are realized using OSGi.

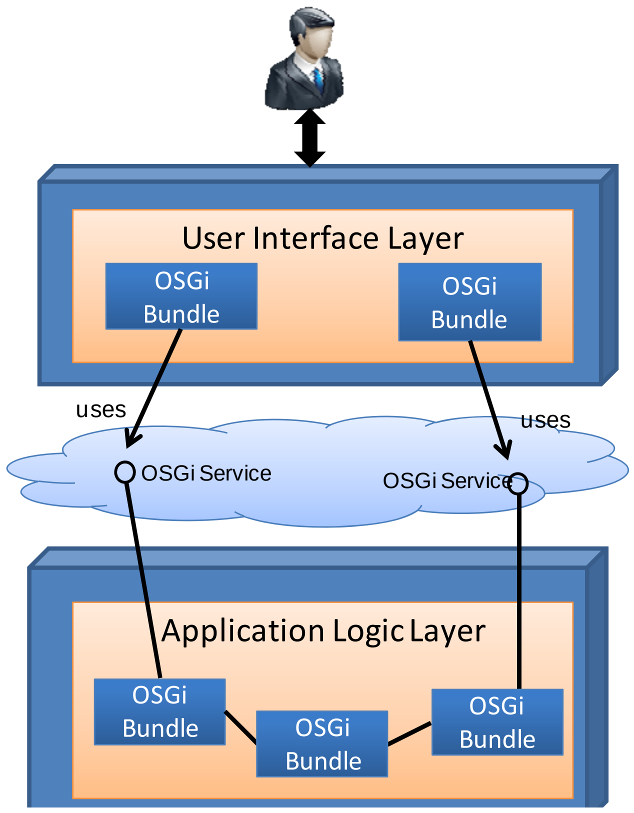


Figure 24: Both, the user interface and the application logic are realize using OSGi.

The advantage is that the same adaptation mechanism can be used to adapt the user interface and the application logic. Furthermore, the communication between network boundaries is much faster than using Web Services. One disadvantage is that an OSGi framework has to be executed on the client side which may be not available for all platforms. However, we already executed OSGi bundles successfully on a Windows Mobile 5 platform. One disadvantage of OSGi is that it is only available for Java applications. Thus it is not possible to implement one part in Java and another in C# for example. However, CORBA provides this functionality but needs an Object Request Broker running on each peer which in fact is not available for all kinds of platforms.

Many other settings are imaginable, but are out of scope of this deliverable. In order to realize application logic reconfiguration using the wiring approach, all presented alternatives are feasible and already partially tested. However, many orchestration approaches require services offered as Web Services with the advantages and drawback already presented.

6 Conclusion and Next Steps

In this deliverable, we introduced two main approaches for realizing application logic reconfiguration, namely the wiring approach and the orchestration approach. Furthermore, we introduced techniques and types of adaptation for both approaches followed by a comparison. Next, two basic architectures have been introduced which show how the several involved middleware services interact and where they could be deployed. Finally, some communication aspects have been discussed.

The next steps will be to decide which approach to use or if an integrated solution is more appropriate depending on target platforms, considered applications and available middleware components. Furthermore, the architecture has to be finalized and the integration of other middleware parts like the Context Manager and the Migration Manager has to be done.

7 References

Aalst, W., Hee, K. Workflow Management, Models, Methods, and Systems. First MIT Press paperback edition, 2004.

Aalst, W., Hofstede, A., Russell, N. Workflow Patterns Initiative. 2007. Online available at: http://www.workflowpatterns.com/patterns/index.php

Active Endpoints. The ActiveBPEL Community Edition Engine. 2008. Online available at http://www.activevos.com/community-open-source.php

Alves, A. et.al. Web Services Business Process Execution Language. Version 2.0. OASIS Standard, 2007. Online available at: http://www.oasis-open.org/specs/

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.. Pattern-Oriented Software Architecture. Wiley, 1996.

Faatz, A., Goertz, Manuel. Requirements for OPEN Service Platform. OPEN Deliverable D1.1. 2008.

Hofstede, A. et al. YAWL – Yet Another Workflow Language. 2008. Online available at: http://yawlfoundation.org/index.html

JBoss. JBoss jBPM. 2009. Online available at: http://www.jboss.com/products/jbpm/

Martin, D. et al. OWL-S: Semantic Markup for Web Services. W3C Member Submission 22 November 2004.

Microsoft. Windows Workflow Foundation. 2009. Online available at: http://msdn.microsoft.com/en-us/library/ms735967.aspx

Nickelsen, A., Olsen, R. L., Schwefel, H.P., Martin, M., Kovacs, E., Ghiani, G., Klus, H., Marzorati, S., Grasselli, A., Piunti, M. Detailed Network Architecture. OPEN Deliverable D3.1. 2009.

Niebuhr, D., Klus, H., Anastasopoulos, M., Koch, J., Weiß, O., Rausch, A. DAiSI – Dynamic Adaptive System Infrastructure. IESE-Report No. 051.07/E, Fraunhofer Institut für Experimentelles Software Engineering, 2007.

Russell, N., Hofstede, A., Aalst, W., Mulyar, N. Workflow Control-Flow Patterns : A Revised View. BPM Center Report BPM-06-22, BPMcenter.org, 2006.