Rheinische Friedrich-Wilhelms-Universität

Institut für Informatik III
Prof. Dr. Armin B. Cremers

Design and Implementation of a Workflow Engine

Diplomarbeit von

Sebastian Bergmann
Aulgasse 14
53721 Siegburg

E-Mail: sb@sebastian-bergmann.de

Matrikelnummer: 1247261

1. Gutachter: Prof. Dr. Armin B. Cremers
2. Gutachter: Prof. Dr. Rainer Manthey

Tag der Abgabe: 13. Februar 2007
Letzte Aktualisierung: 19. August 2009

Manifesto

Except where indicated otherwise, this thesis is my own original work.

Sebastian Bergmann

Abstract

This thesis discusses the design and implementation of a software component that faciliates the specification, management, and execution of so-called workflows.
The discussion of this component's design includes the semantics and syntax of the underlying workflow model as well as the actual software design. The former builds upon the *Workflow Patterns* [BK03] terminology, the latter on the concepts of a *Workflow Virtual Machine* [SF04] and the idea that a workflow system should be comprised of loosely coupled components [DAM01, DG95, PM99].

Diese Diplomarbeit behandelt den Entwurf und die Implementierung einer Softwarekomponente für die Definition, Verwaltung und Ausführung von Spezifikationen so genannter Workflows.
Die Diskussion des Entwurfs dieser Komponente behandelt Semantik und Syntax des zugrunde liegenden Workflow-Modells ebenso wie das eigentliche Software-Design. Ersteres baut auf der Terminologie der *Workflow Patterns* [BK03] auf, letzteres auf dem Konzept einer *Workflow Virtual Machine* [SF04] und der Idee, dass ein Workflow-System aus lose gekoppelten Komponenten aufgebaut sein sollte [DAM01, DG95, PM99].

Diese Diplomarbeit wurde in englischer Sprache verfasst.

Contents

List of Figures

Introduction

Problem Statement and Goal

This thesis has the design and implementation of a workflow engine as its goal. This goal has motivations from both academia and industry that were represented by the two supervising institutions, the Institute of Computer Science of the University of Bonn, Germany and eZ Systems AS, respectively.

The topic of this diploma thesis was set up by eZ Systems AS in Skien, Norway. The company is the creator of eZ Publish, an Open Source Enterprise Content Management System, and eZ Components, a components library for PHP 5. As we will see in Chapter 4, eZ Systems AS is in need of a flexible and reusable workflow engine component, written in the PHP programming language, that can be used in the development of the next version of their eZ Publish ECMS. Of academic interest is how research such as [BK03, DAM01, PM99, SF04] can be put to use for the design and implementation of such a software component.

The goal of this thesis is therefore to *review the relevant literature*, to *find a suitable workflow model* as the foundation for the *design and implementation of a workflow engine*, and to *evaluate the resulting software component* with regard to the industry requirements set up by eZ Systems AS.

Structure

Chapter 1 gives an introduction to the problem domain of Enterprise Content Management and Workflow Management. Chapter 2 presents Petri nets as a formal way and workflow patterns as a more pragmatic way to define the semantics of a workflow model.

Chapter 3 gives an introduction to the technology stack (PHP, eZ Publish, eZ Components) that is relevant to and used by the software that has been implemented as part of this thesis. Chapter 4 discusses the requirements that lead to the development of this software.

Chapter 5 presents the semantics and syntax of the workflow model that is the foundation for the software. Chapter 6 discusses the design and implementation of the software.

This paper concludes with an evaluation of the software (Chapter 7), a comparison to related work, and an outlook on future work (Chapter 8).

Acknowledgements

I would like to thank Prof. Dr. Armin B. Cremers for making it possible that I could do my thesis in cooperation with eZ Systems AS and Dr. Stefan Lüttringhaus-Kappel for being my thesis advisor. I would like to thank everyone at eZ Systems AS for the great time I had in Norway.

Finally, I would like to express my appreciation for the people involved in the development of the free software that is discussed in this paper (PHP, eZ Publish, eZ Components) and was used to typeset (LaTeX, Dia, Doxygen, GraphViz, KOMA-Script, PGF, Ti*k*Z) this paper.

Chapter 1.

Problem Domain

This chapter gives an introduction to the problem domain of Enterprise Content Management and Workflow Management.

1.1. Enterprise Content Management

The meaning of the term ”Enterprise Content Management” can be approached gradually by looking at the three words that make it up:

- **Enterprise** refers to the employees of an enterprise with access and editing rights.
- **Content** refers to arbitrary content stored in the electronic systems of an enterprise.
- **Management** refers to a software system for the administration, control, and processing of content in an enterprise, both internally (in an intranet, for example) and externally (on the internet, for example).

The Association for Information and Image Management (AIIM) defines Enterprise Content Management (ECM) as

> *the technologies used to capture, manage, store, preserve, and deliver content and documents related to organizational processes. ECM tools and strategies allow the management of an organization's unstructured information, wherever that information exists* [AIIM].

The usage scenarios of eZ Publish (see Section 3.2), for example, range from blogs and personal websites to community portals, company websites, webshops, business process management, enterprise resource planning, and document management in both governmental institutions and corporate environments.

1.2. Workflow Management

The Workflow Management Coalition (WfMC) describes workflow management as

> *the automation of a business process, in whole or parts, where documents, information or tasks are passed from one participant to another to be processed, according to a set of procedural rules* [RA01].

Georgakopoulos et. al. define workflow management as a

> *technology supporting the reengineering of business and information processes. It involves: (1) defining* workflows*, i.e., describing those aspects of a process that are relevant to controlling and coordinating the execution of its tasks [...], and (2) providing for fast (re)design and (re)implementation of the processes as business needs and information systems change* [DG95].

Workflow management systems are software systems that enable workflow management.

There are two kinds of workflow management systems: those that are *activity-based* and those that are *entity-based*. The former have their focus on the activities that are to be completed throughout the workflow, the latter focus on entities, such as documents, that are processed by a workflow [FG02].

The documentation of the OpenFlow workflow management system [OPENFLOW] summarizes the purpose of an activity-based workflow management system as *answering the question "who must do what, when and how"*:

- The workflow definition (or *workflow schema*) defines the sequence of activities that are to be carried out. It specifies *what* should be done and *when* by the definition of activities (represented by the *nodes* of a directed graph) and transitions (represented by the *edges* of a directed graph).

- An activity (the *what* part of the issue) represents something to be done: reviewing a document, publishing a document, placing an order, sending an e-mail, and so on.
- Transitions define the appropriate sequence of activities for a process (the *when* part of the issue).
- Each activity will have an associated application designed to carry out the job: the *how* part.
- The *who* part is generally the user or system assigned to carry out the activity, through its application.

Figure 1.1 shows a sample workflow that illustrates this: The green nodes represent the activities that are to be completed throughout the workflow. The red edges between the nodes represent the control flow. Depending on the input that is provided by a user with the appropriate access rights (blue), the *branch* nodes chooses one of two possible actions that are encapsulated by so-called *service objects* (yellow). After one of those two possible actions has been performed, the *merge* nodes merges the control flow again.

The interaction with the user (to receive input, for instance) is performed through a so-called *worklist* interface. The software system into which the workflow management system is integrated queries the workflow system whether a workflow instance is waiting for input that can be provided by the current user. The user can then provide the input through the worklist interface.

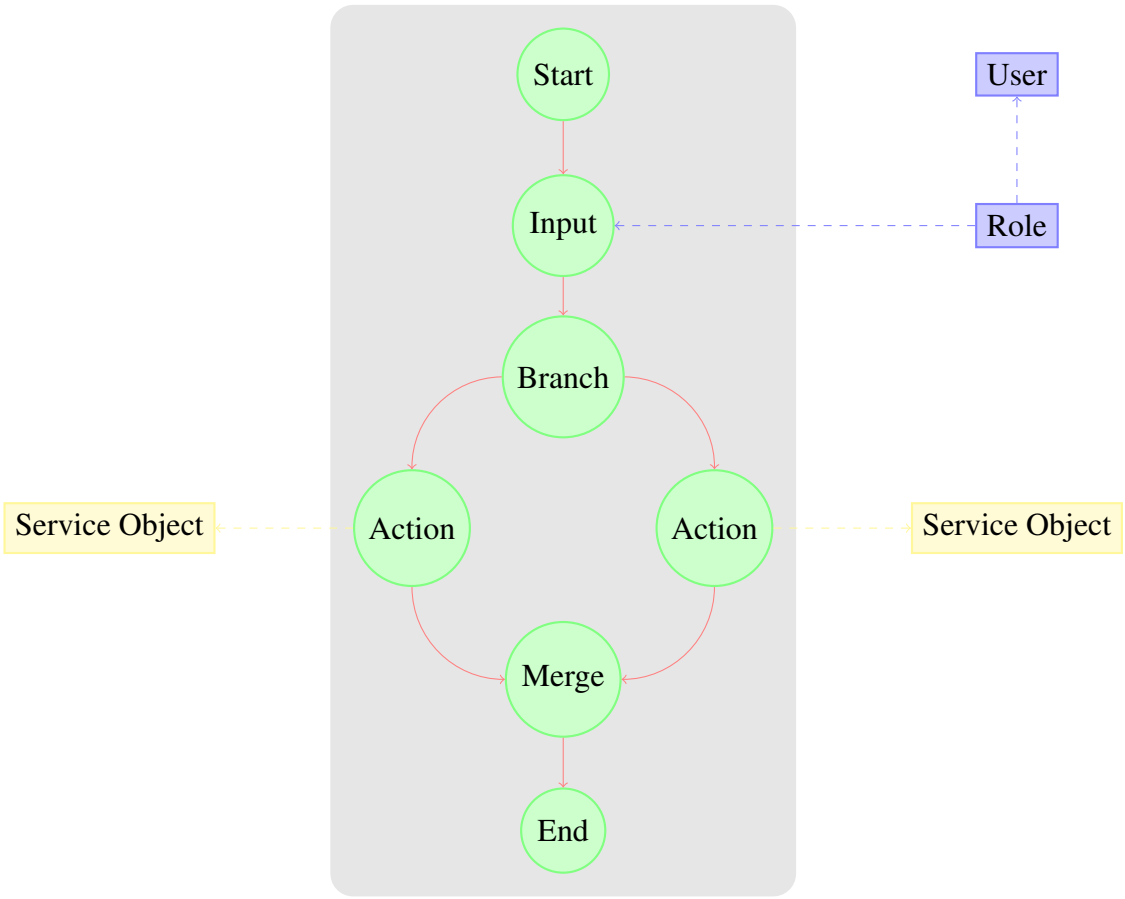

Figure 1.1.: Who must do what when and how?

1.3. Summary

Business enterprises need to reduce the cost of doing business and continually develop new services and products. Enterprise Content Management, as well as the related practices of Document Management and Knowlege Management, helps with storing business-critical content (customer data, documents, etc.) in a central repository and in a unified way. Business Process Management and Workflow Management provide the methodologies and software that help with organizing the processes that operate on this content inside an organization.

Chapter 2.

Workflow Semantics

This chapter presents *Petri nets* as a formal way and *workflow patterns* as a more pragmatic way to define the semantics of a workflow model.

2.1. Petri Nets

A Petri net is a mathematical representation for discrete distributed systems. It is a 5-tuple (S, T, F, M_0, W), where [JD01]

- S is a set of *places*.
- T is a set of *transitions*.
- F is a set of *arcs* known as a *flow relation*. The set F is subject to the constraint that no arc may connect two places or two transitions, or more formally: $F \subseteq (S \times T) \bigcup (T \times S)$.
- $M_0 : S \to \mathbb{N}$ is an *initial marking*, where for each place $s \in S$, there are $n_s \in \mathbb{N}$ tokens.
- $W : F \to \mathbb{N}$ is a set of *arc weights*, which assigns to each arc $f \in F$ some $n \in \mathbb{N}^+$ denoting how many tokens are consumed from a place by a transition, or alternatively, how many tokens are produced by a transition and put into each place.

A place that has an arc to a transition is called an *input place*, a place that has an arc from a transition is called an *output place*. A place may contain any number of tokens. A distribution of tokens over the places of a net is called a *marking*. When a transition is activated (or *fired*), it consumes the tokens from its input places, performs some form of processing, and places a specified number of tokens in its output places. These three steps are performed atomically.

The activation of transitions is non-deterministic. This makes Petri nets well suited for the modelling of concurrent behaviour in distributed systems.

Petri nets have been proposed for modelling workflows by van der Aalst [WA96], for instance, because they provide *formal semantics despite the graphical nature* and an *abundance of analysis techniques* exists. Figures 2.1 to 2.6 show how the workflow primitives of the workflow reference model [WfMC95] can be expressed using Petri nets.

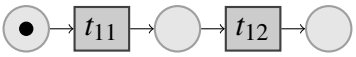

Figure 2.1.: Causality

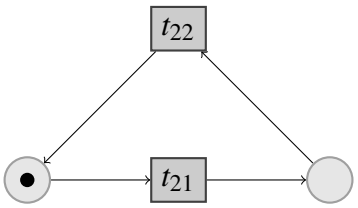

Figure 2.2.: Iteration

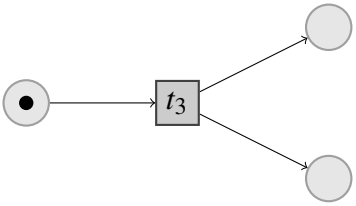

Figure 2.3.: AND-Split

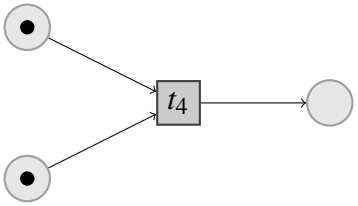

Figure 2.4.: AND-Join

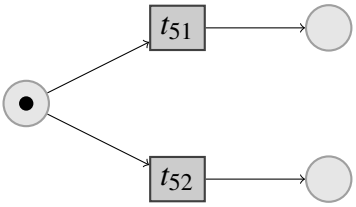

Figure 2.5.: OR-Split

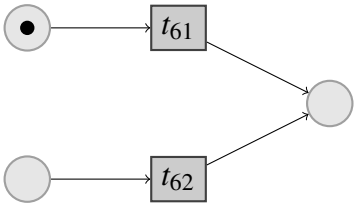

Figure 2.6.: OR-Join

Finite Automata are another formalism that can be used to describe workflows [ARK03].

2.2. Workflow Patterns

In Chapter 3 of his PhD thesis [BK03], Kiepuszewski lists *requirements for workflow languages through workflow patterns*.

Much like the software design patterns [GoF94], these workflow patterns describe recurring solutions to common problems. They are relevant to both the implementor and the user of a workflow management system. The former uses the workflow patterns as a common vocabulary for workflow description language constructs and to define the semantics of a workflow model (see Chapter 5) whereas the latter uses them as a guide while formulating his workflow in the workflow system's description language. The workflow patterns also faciliate the comparison with other workflow systems with regard to expressiveness and power.

In Chapter 4 of his PhD thesis [BK03], Kiepuszewski maps the workflow patterns that he identified to Petri nets to provide a formal foundation for this more pragmatic approach to defining workflow semantics.

In this section we discuss the subset of the workflow patterns identified by Kiepuszewski that is directly supported by the software that has been implemented as part of this thesis.

2.2.1. Basic Control Flow Patterns

The workflow patterns for basic control flow *capture elementary aspects of process control* and *closely match the definitions of elementary control flow concepts provided by the Workflow Management Coalition in [WfMC95, WfMC99].*

Sequence

The *Sequence* workflow pattern represents linear execution of workflow steps: one action of a workflow is activated unconditionally (for example *B* in Figure 2.7) after another (for example *A* in Figure 2.7) finished executing.

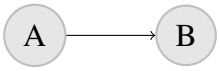

Figure 2.7.: Sequence

Use Case Example: After an order is placed, the credit card specified by the customer is charged.

Parallel Split (AND-Split)

The *Parallel Split* workflow pattern divides one thread of execution (for example the one that activates *A* in Figure 2.8) unconditionally into multiple parallel threads of execution (for example the ones that start in *B*, *C*, and *D* in Figure 2.8).

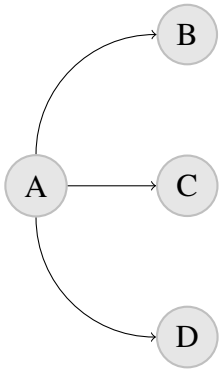

Figure 2.8.: Parallel Split

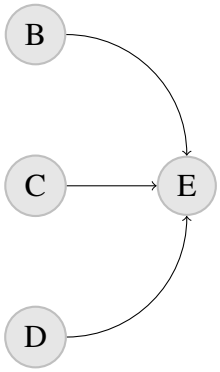

Figure 2.9.: Synchronization

Use Case Example: After the credit card specified by the customer has been successfully charged, the activities of sending a confirmation email and starting the shipping process can be executed in parallel.

Synchronization (AND-Join)

The *Synchronization* workflow pattern synchronizes multiple parallel threads of execution (for example the ones that end in *B*, *C*, and *D* in Figure 2.9) into a single thread of execution (for example the one that starts in *E* in Figure 2.9).

Workflow execution continues once all threads of execution that are to be synchronized have finished executing (exactly once).

Use Case Example: After the confirmation email has been sent and the shipping process has been completed, the order can be archived.

The workflow patterns that have been discussed so far handle the *unconditional routing* of control flow. We will now take a look at the workflow patterns for *conditional routing*.

Exclusive Choice (XOR-Split)

The *Exclusive Choice* workflow pattern defines multiple possible paths (for example the ones that start in *B*, *C*, and *D* in Figure 2.10) for the workflow of which exactly one is chosen (for example the one that starts in *C* in Figure 2.10).

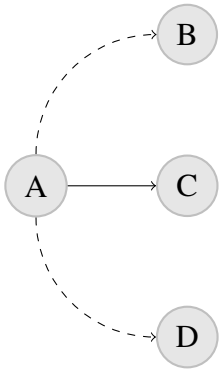

Figure 2.10.: Exclusive Choice

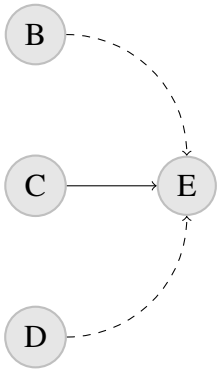

Figure 2.11.: Simple Merge

Use Case Example: After an order has been received, the payment can be performed by credit card or bank transfer.

Simple Merge (XOR-Join)

The *Simple Merge* workflow pattern is to be used to merge the possible paths that are defined by a preceding *Exclusive Choice*. It is assumed that of these possible paths exactly one is taken (for example *C* in Figure 2.11) and no synchronization takes place.

Use Case Example: After the payment has been performed by either credit card or bank transfer, the order can be processed further.

2.2.2. Advanced Branching and Synchronization

The workflow patterns for advanced branching and synchronization *do not have straightforward support in most [of the] workflow engines [that Kiepuszewski evaluated]* (see Table 7.1). *Nevertheless, they are quite common in real-life business scenarios.*

Multi-Choice (OR-Split)

The *Multi-Choice* workflow pattern defines multiple possible paths (for example the ones that start in *B*, *C*, and *D* in Figure 2.12) for the workflow of which one or more are chosen (for example the ones that start in *B* and *D* in Figure 2.12). It is a generalization of the Parallel Split and Exclusive Choice workflow patterns.

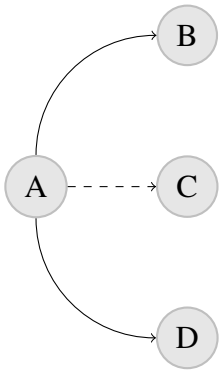

Figure 2.12.: Multi-Choice

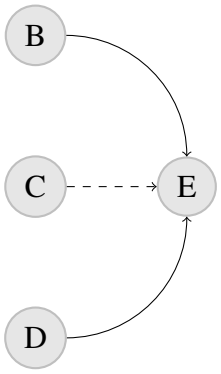

Figure 2.13.: Synchronizing Merge

Synchronizing Merge

The *Synchronizing Merge* workflow pattern is to be used to synchronize multiple parallel threads of execution that are activated by a preceding *Multi-Choice* (for example the ones that end in *B* and *D* in Figure 2.13).

Discriminator

The *Discriminator* workflow pattern can be applied when the assumption we made for the *Simple Merge* workflow pattern does not hold. It can deal with merge situations where multiple incoming branches may run in parallel.

It activates its outgoing node after being activated by the first incoming branch and then waits for all remaining branches to complete before it resets itself. After the reset the *Discriminator* can be triggered again.

Use Case Example: To improve response time, an action is delegated to several distributed servers. The first response proceeds the flow, the other responses are ignored.

2.2.3. Structural Patterns

The structural workflow patterns deal with restrictions that different workflow models can impose.

Arbitrary Cycles

A common restriction workflow models impose is that arbitrary cycles, ie. one or more activities are done repeatedly, are not supported. As an alternative, special loop constructs that mark the start and end point of a structured cycle are offered.

Implicit Termination

The execution of the workflow is (successfully) terminated when there are no activated activities left and no other activity can be activated. This implicit termination of workflow execution can be used in addition to explicit end activities.

2.2.4. Cancellation Patterns

Cancel Case

The execution of a workflow instance is cancelled.

Use Case Example: An order is cancelled.

2.3. Summary

This chapter presented Petri nets as a general, well understood and well researched, theory for concurrency and the workflow patterns as an pragmatic approach to describe the semantics of workflow routing constructs.

We will use the workflow patterns in Chapter 4 during the requirements analysis and in Chapter 5 to define the semantics of the workflow model for the software that has been developed as part of this thesis.

Chapter 3.

Technology

This chapter gives an introduction to the technology stack (PHP, eZ Publish, eZ Components) that is relevant to and used by the software that has been implemented as part of this thesis.

3.1. PHP

PHP is a widely-used scripting language. Initially designed for Web programming, it has matured to a general-purpose programming language that supports both procedural and object-oriented programming. PHP makes strong inroads into large-scale, business-critical Web systems. For instance, financial institutions develop and maintain BASEL II credit and insurance rating tools using PHP and Yahoo! runs all its business on PHP (except the core of the search engine). As of version 5, released July 2004, the PHP language features an object model that is similar to the ones of Java and C# and integrates ideas from other programming languages. The key technical contributor to PHP's success is its simplicity, which translates into shorter development cycles, easier maintenance, and lower training costs. The second one is social – the very large and vibrant community around it, which develops not only PHP itself but also thousands of open source applications that can be used off-the-shelf or as references for new applications.

[SB05] gives an introduction to object-oriented programming with PHP 5.

3.2. eZ Publish

eZ Publish is the Enterprise Content Management System developed by eZ Systems AS. With its framework architechture, it is both an out-of-the-box solution as well as a platform that can be customized and extended to suit the specific requirements of a customer.

Figure 3.1 shows an overview of the architecture of eZ Publish in its current version.

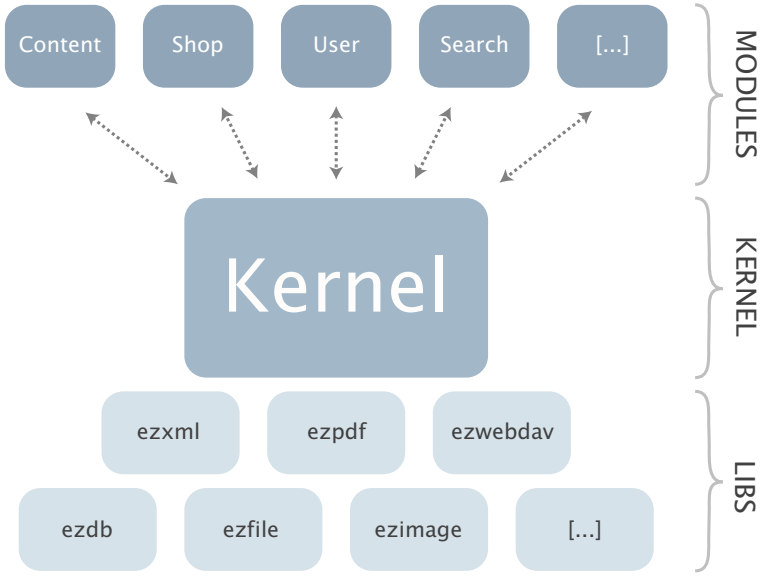

Figure 3.1.: The Architecture of eZ Publish 3

The *libraries* are the main building blocks of the system and are designed as reuseable general purpose PHP classes. Although they are not dependant on other parts of the system, some of them are not usable outside the scope of eZ Publish as the functionality they provide is tightly coupled to eZ Publish.

The *kernel* makes up the system's core and is responsible, among other things, for content handling and versioning, access control, and workflows.

Whereas the kernel and the libraries provide rather low-level functionality, the *modules* implement the higher-level functionality of the system. They provide web-based interfaces to functionality that is, for instance, part of the kernel. For example, the content module provides an interface that makes it possible to use a web browser to manage content. A module consists of two components: at least one *view* and one or more optional *fetch functions*. The former

implement the actual web interface of the module. The latter allow for the extraction of data through a module from within a *template*.

eZ Publish uses an object-oriented approach to organize and store content and allows for the creation of custom structures that fit the needs of the customer. The system offers a selection of fundamental building blocks and mechanisms that together provide a flexible content management solution. An actual data structure is described using something called a *content class*. A content class is built up of attributes. An *attribute* can be thought of as a field, for example the ”title” field in a structure designed to store news articles. The description of the entire structure would be refferred to as the ”article class”. The characteristics of an attribute inside the class are determined by the *datatype* that was chosen to represent that attribute.

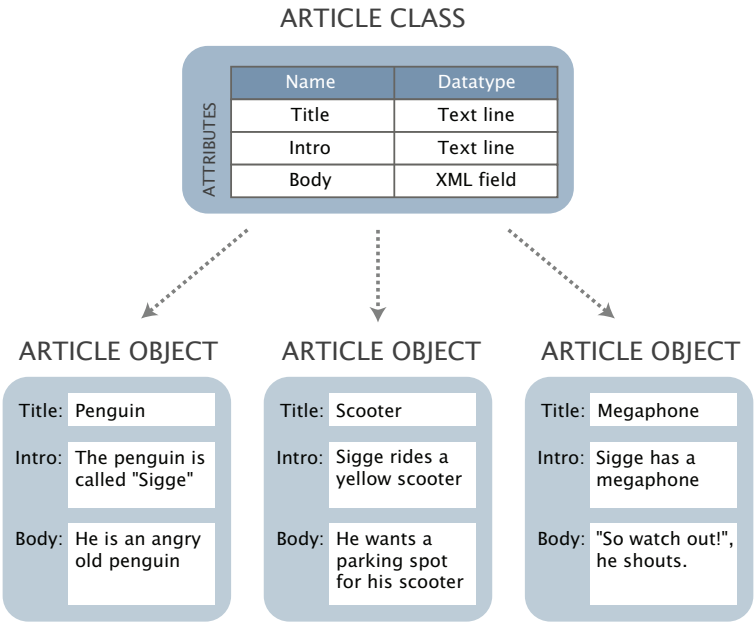

Figure 3.2.: The *Content Object* abstraction of eZ Publish

A *content object* is an instance of a content class. While the class only defines the data structure, it is the content objects themselves that contain actual data. Once a content class is defined, several instances of that class can be created. Figure 3.2 illustrates this relationship of datatypes, attributes, content classes, and objects.

3.3. eZ Components

As part of the development on eZ Publish Telemark, the next major version of its eZ Publish Enterprise Content Management System software, eZ Systems AS has begun refactoring of core functionality from eZ Publish itself into a library of reusable PHP components that provide low-level functionality such as database abstraction, object persistence and caching. This library is called eZ Components.

Figure 3.3 shows an overview of the eZ Components library. One of the design goals of the library is to minimize the number of dependencies between its various components. A component may only depend on the `Base` component. Optional dependencies are handled through so-called *tie-in* components that tie two components together.

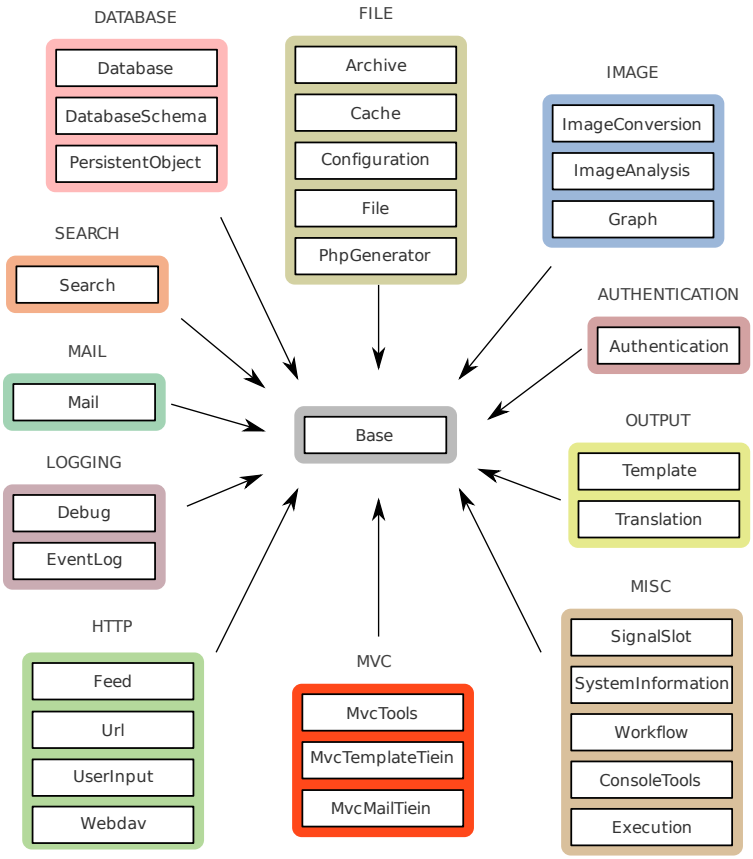

Figure 3.3.: The eZ Components library for PHP 5.

The workflow engine that has been developed as part of this thesis is released under the New BSD License as part of the eZ Components library and utilizes the `Database` and

`DatabaseSchema` components for database abstraction and the `EventLog` component for logging abstraction.

Chapter 4.

Requirements

This chapter is divided into two sections. Section 4.1 presents the workflow mechanism that is part of eZ Publish 3, the current version of the Enterprise Content Management System by eZ Systems AS. Section 4.2 discusses the requirements that the workflow engine for eZ Publish Telemark, the next version of eZ Publish, needs to fulfill.

4.1. eZ Publish 3

eZ Publish 3 comes with an integrated workflow mechanism that makes it possible to perform different tasks with or without user interaction.

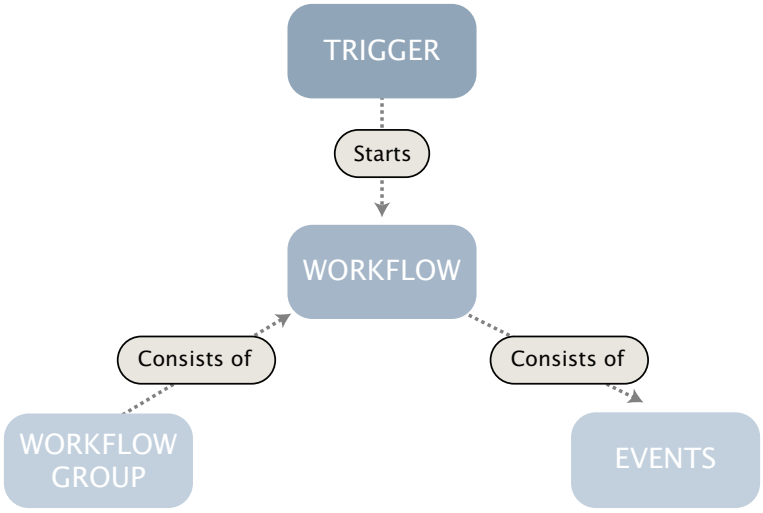

Figure 4.1.: The workflow system in eZ Publish 3

Figure 4.1 shows the components that comprise this mechanism.

An *event* performs a specific task. eZ Publish 3 ships with a library of events and custom events can be implemented in PHP.

A *workflow* defines an ordered sequence in which workflow events are executed and is initiated by a *trigger* that is associated with a function of a module (see Section 3.2). It will start the specified workflow either before or after that function has finished executing.

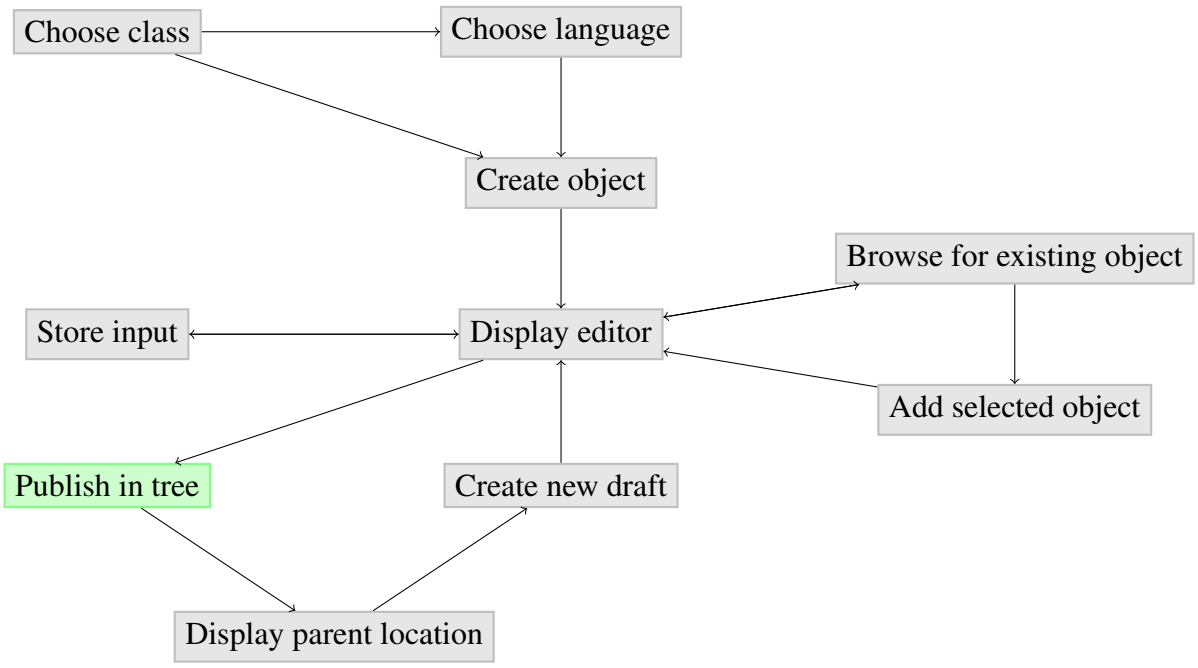

Figure 4.2.: Workflow for publishing a content object in eZ Publish 3

Figure 4.2 shows the built-in workflow for publishing acontent object in eZ Publish 3. This workflow can only be customized at the *Publish in tree* activity. This activity serves as the trigger for a custom workflow that can be executed either before or after the activity was executed.

4.2. eZ Publish Telemark

Both the architecture of the current eZ Publish version as well as its workflow feature have shortcomings that are to be overcome:

- Only some operations are workflows. This inconsistency has a negative effect on the maintainability of the software as a whole.
- It is not easy to configure (hook in) the (internal) workflows. This makes extending the software hard.
- Support for checking the state of executing workflows and control over them is limited.
- Support for conditions is limited.

Eventually, the workflow component should become an important part of the overall solution. However, it must not be tightly integrated or too much dependent on other parts of the system (and vice versa). This means that the workflow component must be flexible and provide good interfaces which allow it to co-exist and plug into the software.

Georgakopoulos et. al. list general requirements for workflow management systems:

> *To effectively support [workflow management], organizations must evolve their existing computing environments to a new distributed environment that: is* component-oriented*, i.e. supports integration and interoperability among loosely-coupled components [...], supports* workflow applications *corresponding to business or information process implementations [...], ensures the correctness and reliability of applications in the presence of concurrency and failures, and supports the evolution, replacement, and addition of workflow applications and component systems as processes are reengineered* [DG95].

Following are the requirements set up by eZ Systems AS. We start with the requirements that are relevant to the underlying workflow model of the workflow component that is to be implemented:

- The workflow component should provide good support for expressing control flow using the workflow patterns (see Chapter 5).
- Any non-trivial operation in eZ Publish, for instance the publishing, removal, and modification of content objects, should be a expressable through workflows.
- Workflows should be composable through a concept of sub-workflows.

Now we come to the requirements that relate to the actual software implementation:

- The workflow component has to be implemented using version 5 of the PHP programming language.

- It should be possible to integrate workflows with the background processes of eZ Publish (run workflow as background process, interact with a background process).
- The workflow component should be customizable and extendable.
- The data storage (for workflow schemas and the persistence of workflow instances) should be abstracted, relational databases must be supported as one backend.
- Versioning of workflow schemas should be supported.
- It should be possible to get information on the workflow instances that are currently executing.
- It should be possible to manually control the workflow instances that are currently executing.
- Simulation of workflow execution for debugging and testing purposes should be possible.

4.2.1. Use Cases

Here are two use cases that should be supported by the workflow engine component that is to be implemented for eZ publish Telemark as part of this thesis. They are currently implemented using custom extensions for eZ publish 3.

Multiple Approval, ISO Certification

This scenario is from a current customer of the eZ Publish ECMS providing quality assurance for dairy products. The customer has information about the dairy products stored in eZ Publish. When they update any content there is a strict ISO-governed process to follow. This process basically consists of a *five-level approval*:

- Bertrand produces an article.
- Approver Level 1: Bård decides who the next four approvers are.
  He can also edit the article and send it back to its creator.
- Approver Level 2: Melissa reviews the article for political correctness.
  She can edit the article and send it back one level.

- Approver Level 3: Vidar reviews the article for sales arguments.
  He can edit the article and and send it back one level.
- Approver Level 4: Jennifer does grammar checks on the article.
  She can edit the article and and send it back one level.
- Publisher: Markus approves the final article and chooses the time and location for publication.

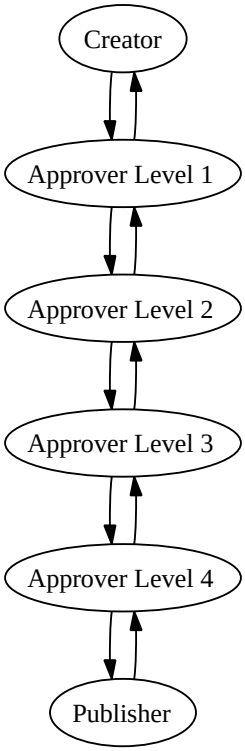

Figure 4.3.: The *Multiple Approval* workflow

It is possible to see all on-going processes for an administrator. He or she can see each article as well as its state and which person currently handles it.

Employment Process

This scenario is from the intranet of a current customer of the eZ Publish ECMS and is used when a new employee is hired.

- One person creates an Employee object (including name, address, email, etc.).

- An e-mail with a link for final approval of the employment is sent to the CEO.
- Once the CEO has approved the new employment three parallel activities are started:
  - An e-mail to the system administrator is sent with the request to create e-mail and other accounts.
    The e-mail contains a link for the system administrator to click when he is done.
  - An automatic process is started to set up accounts on different systems.
  - An e-mail to the administration is sent with the request to buy new hardware for the new employee.
- Once these three activities have been completed, the workflow continues.
- The Employee object is published.
- An e-mail with detailed information is sent to the new employee.

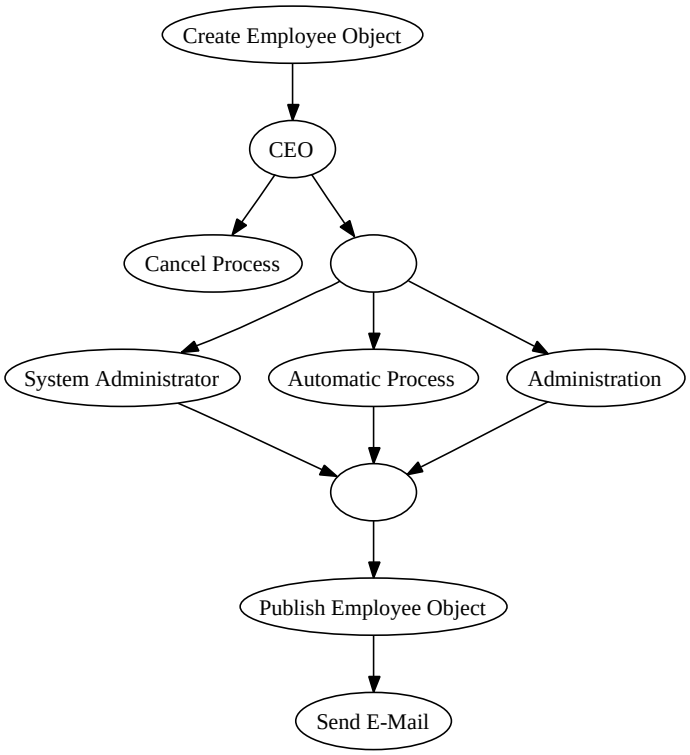

Figure 4.4.: The *Employment Process* workflow

The on-going status for all employment processes at any time is available to anyone with the appropriate permissions.

4.3. Summary

This chapter discussed the requirements for the software that has been developed as part of this thesis. This software will replace the workflow engine of eZ Publish 3 which has severe limitations with regard to the underlying workflow model (only one directly supported workflow pattern) and the software implementation (not easily extendable).

Chapter 5.

Workflow Model

This chapter presents the semantics and syntax of the workflow model that is the foundation for the software that has been developed as part of this thesis.

5.1. Semantics

5.1.1. Activities and Transitions

The workflow model is activity-based. The activities that are to be completed throughout the workflow and the transitions between them are mapped to the nodes and edges of a directed graph. This choice was made to faciliate the application of the Graph-Oriented Programming paradigm for the implementation of the software component that is discussed in Chapter 6. Using a directed graph as the foundation for the workflow model makes it possible to define the syntax of the workflow description language using the formalism of graph grammars (see Section 5.2).

Graph Traversal and Execution Strategy

The execution of a workflow starts with the graph's only *Start* node. A graph may have one or more *End* nodes that explicitly terminate the workflow execution.

After a node has finished executing, it can activate one or more of its possible outgoing nodes. Activation adds a node to a set of nodes that are waiting for execution. During each execution

step, a node from this set is executed. When the execution of a node has been completed, the node is removed from the set.

The workflow execution is implicitly terminated when no nodes are activated and no more nodes can be activated (see the *Implicit Termination* workflow pattern from [BK03] that was discussed in Section 2.2).

5.1.2. State and Workflow Variables

The workflow model supports state through the concept of workflow variables. Such a variable can either be requested as user input (from an *Input* node) or be set and manipulated through the *VariableSet*, *VariableAdd*, *VariableSub*, *VariableMul*, *VariableDiv*, *VariableIncrement*, and *VariableDecrement* nodes.

While a *VariableSet* node may set the value of a workflow variable to any type that is supported by the underlying programming language, the other variable manipulation nodes only operate on numbers.

Variables are bound to the scope of the thread in which they were defined. This allows parallel threads of execution to use variables of the same name without side effects.

Wait States

When the execution of a workflow reaches an *Input* node (see above), the execution is suspended until such time when the user input has been provided and the execution can be resumed.

5.1.3. Control Flow

The control flow semantics of the workflow model draws upon the workflow patterns from [BK03] that were discussed in Section 2.2. The *Sequence*, *Parallel Split*, *Synchronization*, *Exclusive Choice*, *Simple Merge*, *Multi-Choice*, *Synchronizing Merge*, and *Discriminator* workflow patterns are all directly supported by the workflow model.

Exclusive Choice and *Multi-Choice* nodes have branching conditions attached to them that operate on workflow variables to make their control flow decisions.

5.1.4. Action Nodes and Service Objects

So far we have only discussed nodes that control the flow and that can manipulate workflow variables. We are still missing a type of nodes that actually performs an activity. This is where the *Action* node comes into play.

When the execution of a workflow reaches an *Action* node, the business logic of the attached *service object* is executed. Service Objects ”live” in the domain of the application into which the workflow engine is embedded. They have read and write access to the workflow variables to interact with the rest of the workflow.

5.1.5. Sub-Workflows

The workflow model supports sub-workflows to break down a complex workflow into parts that are easier to conceive, understand, maintain, and which can be reused.

A sub-workflow is started when the respective *Sub-Workflow* node is reached during workflow execution. The execution of the parent workflow is suspended while the sub-workflow is executing. It is resumed once the execution of the sub-workflow has ended.

5.2. Syntax

5.2.1. Graph Structure

Graph Grammars are a formalism for the specification of visual languages. In the following, we will use the *reserved graph grammar* variant presented by Zhang et al. in [DQZ01]. It *allows left-hand and right-hand graphs of a production to have an arbitrary number of nodes and edges*. This feature makes the graph grammars more expressive. A node in these graphs is a two-level structure: a so-called *super vertex* contains named vertices, *T (top)*, *B (bottom)*, *L (left)*, *R (right)*. The names correspond to the position of the vertex in the super vertex.

Figures 5.1 to 5.6 show the graph rewriting rules (productions) that make up the grammar for our workflow model.

The *Axiom* grammar rule shown in Figure 5.1 expresses that an empty graph (left-hand side) can be transformed into a graph that contains three nodes: a *Start* node that has a *Statement* node as its only outgoing node, which in turn has an *End* node as its only outgoing node.

The *Reduction* grammar rule shown in Figure 5.2 expresses that a *Statement* node can be added to another *Statement* node.

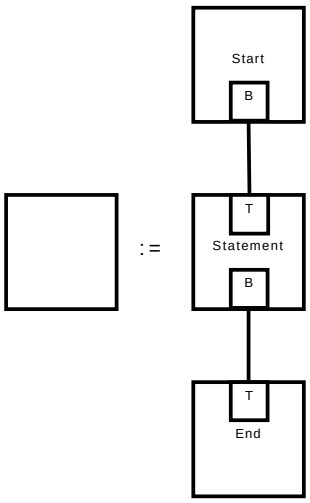

Figure 5.1.: Axiom

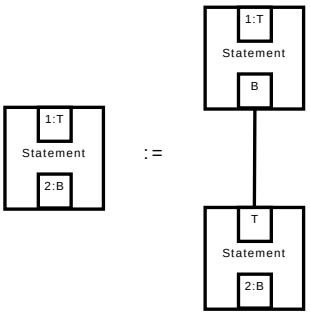

Figure 5.2.: Reduction

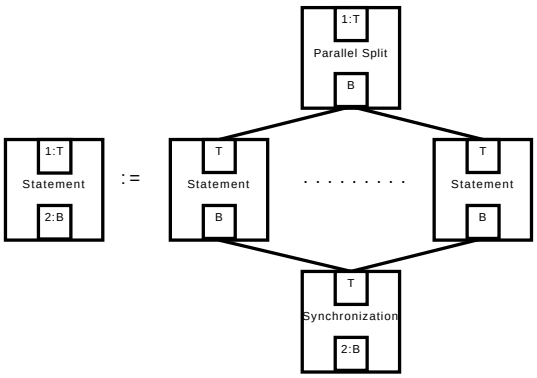

Figure 5.3.: AND

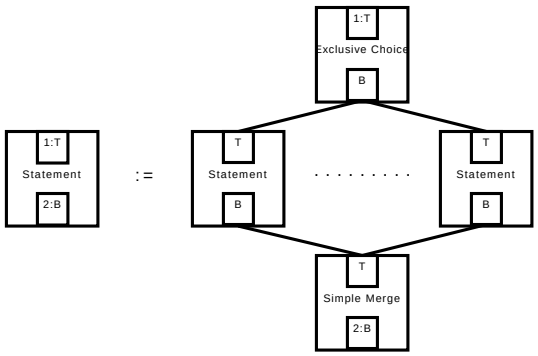

Figure 5.4.: XOR

A *Statement* node can either be replaced by applying the grammar rules shown in Figure 5.3 to 5.6 or by replacing it with a node of type *Action*, *End*, *Input*, *Sub-Workflow*, *VariableSet*, *VariableAdd*, *VariableSub*, *VariableMul*, *VariableDiv*, *VariableIncrement*, and *VariableDecrement*.

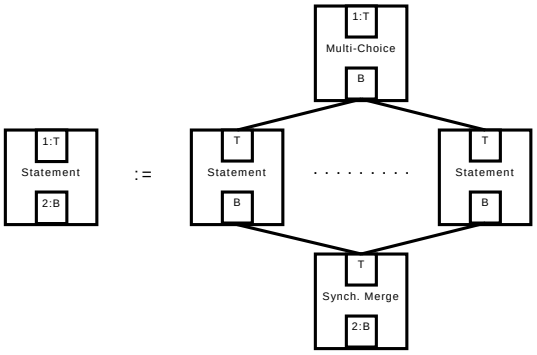

Figure 5.5.: OR

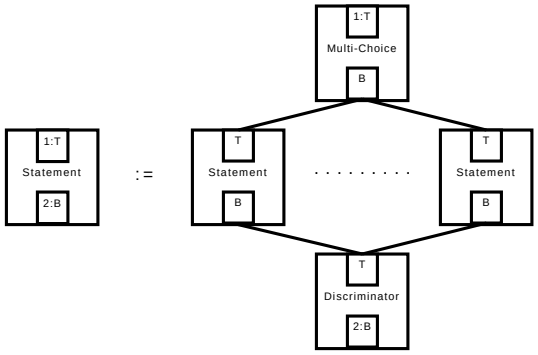

Figure 5.6.: Discriminator

5.2.2. Conditions

The conditions that can be used with branching and input nodes are expressions built using the following constructs: *Not*, *And*, *Or*, *Xor*, *IsAnything*, *IsArray*, *IsBool*, *IsTrue*, *IsFalse*, *IsFloat*, *IsInteger*, *IsEqual*, *IsNotEqual*, *IsGreaterThan*, *IsEqualOrGreaterThan*, *IsLessThan*, *IsEqualOrLessThan*, *IsObject*, and *IsString*.

The examples in Appendix B.2 show the syntax using which these constructs can be combined to form condition expressions.

5.3. Summary

This chapter used the workflow patterns to describe the semantics and a graph grammar to define the syntax of the workflow model that is the foundation for the software that has been developed as part of this thesis.

The workflow model can be extended, for instance, with support for more workflow patterns, by adding the respective node types.

Chapter 6.

Design and Implementation

This chapter discusses the design and implementation of the software that has been developed as part of this thesis.

6.1. Architecture

The workflow engine that has been developed as part of this thesis has been designed and implemented as three loosely coupled components. The `Workflow` component provides an object-oriented framework to define workflows and an execution engine to execute them. The `WorkflowDatabaseTiein` and `WorkflowEventLogTiein` components tie the `Database` and `EventLog` components from the eZ Components library into the main `Workflow` component for persistence and monitoring, respectively.

A workflow can be defined programmatically by creating and connecting objects (see Section 6.5) that represent control flow constructs. The classes for these objects are provided by the *Workflow Definition API* (see Appendix B for a reference). This API also provides the functionality to save workflow definitions (ie. object graphs) to and load workflow definitions from a data storage. Two data storage backends have been implemented, one for relational database systems and another for XML files. Through the *Workflow Execution API* the execution of a workflow definition can be started (and resumed). Figure 6.1 shows the conceptual architecture for the workflow engine.

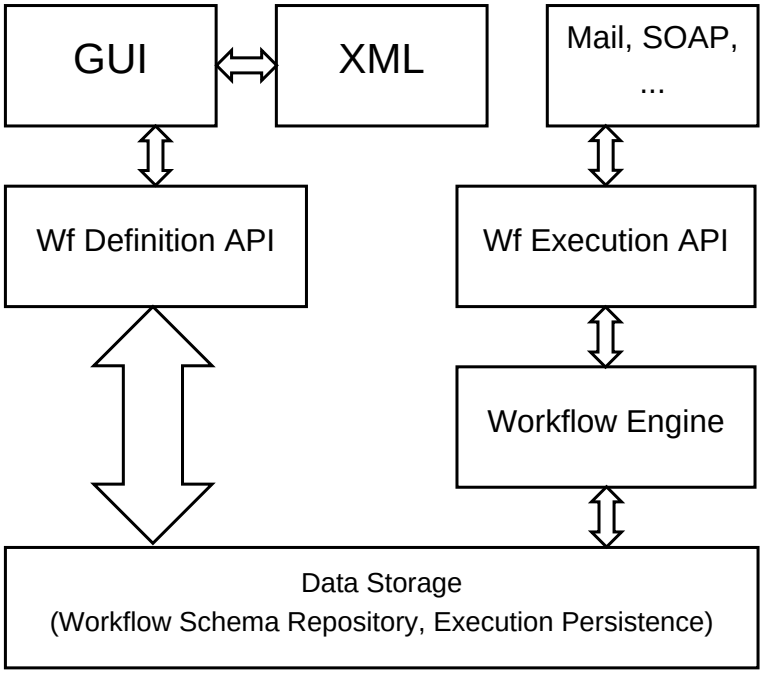

Figure 6.1.: Conceptual architecture for the workflow engine

The idea that a workflow system should be comprised of loosely coupled components is discussed, for instance, in [DAM01, DG95, PM99]. Manolescu states that

> *an object-oriented workflow architecture must* provide abstractions *that enable software developers to define and enact how the work flows through the system* [DAM01].

The component-based workflow architecture Micro-Workflow *encapsulates workflow features in separate components*. This architecture follows the *Microkernel* pattern which

> *applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration* [FB96].

The minimalistic core of Micro-Workflow is comprised of three components that provide basic workflow functionality:

- The *process component* implements an activity-based workflow model and provides the abstractions required to build workflows.

- The *execution component* implements the functionality to execute workflows.
- The *synchronization component* allows developers to define dependencies within the workflow domain.

On top of these core components other components, for instance for persistence (suspending and resuming workflow execution), monitoring (status of running workflows), history (history of executed workflows), and worklist management (human-computer interface), can be implemented. Each of these components *encapsulates a design decision* and can be customized or replaced.

6.2. Workflow Virtual Machine

This section proposes a so-called *workflow virtual machine* as the executing component of a component-based workflow architecture.

Given the fact that *standardization efforts, e.g. XPDL [WfMC05] proposed by the WfMC, have essentially failed to gain universal acceptance* [WA04], the *problem of developing a [workflow system] that supports changes in the [workflow description language]* needs to be addressed.

Fernandes et. al. propose to

> *split the [workflow system] into two layers: (1) a layer implementing a* Workflow Virtual Machine*, which is responsible for most of the [workflow system] activities; and (2) a layer where the different [workflow description languages] are handled, which is responsible for making the mapping between each [workflow description language] and the Workflow Virtual Machine* [SF04].

A workflow virtual machine isolates the executing part of a workflow management system, the *backend*, from the parts that users interact with, the *frontend*. This isolation allows for the definition of a *backend language* to describe exactly the workflows that are supported by the executer and its underlying workflow model. This backend language is not the workflow description language users use to define their workflows. They use *frontend languages* that can be mapped to the system's backend language.

6.3. Graph-Oriented Programming

The manual of JBoss jBPM [JBOSS], a platform for multiple process languages supporting workflow, business process management, and process orchestration, introduces *Graph-Oriented Programming [as a] new implementation technique that serves as a basis for all graph-based process languages*.

Graph-Oriented Programming implements the *graphical representation* and the *wait states* of a process language in an object-oriented programming language. The former can be achieved by providing a framework of node classes. Objects of these classes represent the nodes in the process graph, relations between these objects represent the edges. Such an object graph can then be traversed for execution. These executions need to be persistable, for instance in a relational database, to support the wait states.

The aforementioned node classes implement the *Command* design pattern [GoF94] and encapsulate an action and its parameters.

The executing part of the workflow engine is implemented in an `Execution` class. An object of this class represents a workflow in execution. The execution object has a reference to the current node. When the execution of a workflow is started, a new execution object is created and the current node is set to the workflow's start node. The `execute()` method that is to be provided by the node classes is not only responsible for executing the node's action, but also for propagating the execution: *a node can pass the execution that arrived in the node [to] one of its leaving transitions to the next node*.

Like Fowler in [MF05], the authors of the JBoss jBPM manual acknowledge the fact that *current software development relies more and more on domain specific languages*. They see Graph-Oriented Programming as a means to implement domain specific languages *that describe how graphs can be defined and executed* on top of an object-oriented programming language.

In this context, a process language (like a workflow description language) is *nothing more than a set of `Node` implementations*. The semantics of each node are defined by the implementation of the `execute()` method in the respective node class. This language can be used as the back-end language of a Workflow Virtual Machine (see Section 6.2). In this lanugage, the workflow is represented as a graph of command objects. The workflow patterns (see Section 2.2) make up the requirements for and can be mapped to the respective classes.

One of the advantages of using a domain specific language that Fowler gives in [MF05] regards the *involvement of lay programmers: domain experts who are not professional programmers but program in domain specific languages as part of the development effort*. In essence this means that a software system that provides a domain specific language can be customized and extended without knowledge of the underlying programming language that was used to implement it.

6.4. Implementation Details

The workflow engine maintains a set of activated nodes. At the beginning of the execution, the workflow's start node is activated.

Listing 6.1 shows the main execution loop of the workflow virtual machine. As long as the workflow execution has not explicitly ended (by reaching an *End* node), the next activated node is executed. After the node has been successfully executed it is removed from the set of activated nodes. In a situation where the set of activated nodes is not empty but none of the activated nodes can be completed (because they are waiting for user input, for instance), the workflow execution is suspended. When the set of activated nodes is empty, the execution of the workflow ends.

During its execution, a node can activate an arbitrary amount of its outgoing nodes. *Synchronization*, *Synchronizing Merge*, and *Discriminator* nodes, for instance, need to be activated several times before they can complete their execution.

Parallel threads of execution that are branched by the *Parallel Split* and *Multi-Choice* (and merged by the *Synchronization*, *Synchronizing Merge*, and *Discriminator*) workflow patterns are executed serialized.

Listing 6.1: The workflow engine's main execution loop

```
abstract class ezcWorkflowExecution
{
    // ...

    protected function execute()
    {
        do
        {
            $executed = false;

            foreach ( $this->activatedNodes as $key => $node )
            {
                if ( $this->cancelled && $this->ended )
                {
                    break;
                }

                if ( $node  instanceof ezcWorkflowNodeEnd &&
                     !$node instanceof ezcWorkflowNodeCancel &&
                     $this->numActivatedNodes != $this->
                         numActivatedEndNodes )
                {
                    continue;
                }

                if ( $node->execute( $this ) )
                {
                    unset( $this->activatedNodes[$key] );
                    $this->numActivatedNodes--;

                    if ( !$this->cancelled && !$this->ended )
                    {
                        foreach ( $this->plugins as $plugin )
                        {
                            $plugin->afterNodeExecuted( $this, $node );
                        }
                    }

                    $executed = true;
                }
            }
        }
        while ( !empty( $this->activatedNodes ) && $executed );

        if ( !$this->cancelled && !$this->ended )
        {
            $this->suspend();
        }
    }

    // ...
}
```

6.5. Example

Listing 6.2 shows an example of PHP code that programmatically creates an object graph for a workflow.

The workflow in this example consists of seven nodes:

1. A *Start* node (line 10).
2. An *Input* node (lines 13–17) that requests a boolean input variable.
3. An *Exclusive Choice* node (line 21) that conditionally branches (lines 27–39) based upon the value of the previously supplied input variable.
4. An *Action* node that has a `PrintTrue` service object attached to it (line 24).
5. An *Action* node that has a `PrintFalse` service object attached to it (line 25).
6. A *Simple Merge* node (lines 41–44).
7. An *End* node (line 11).

Listing 6.3 shows the object graph created by this PHP code serialized to XML, Appendix A elaborates on this example and explains it in detail.

Listing 6.2: Creating an object graph using the *Workflow Definition API*

```
<?php
require_once 'Base/base.php';

function __autoload( $className )
{
    ezcBase::autoload( $className );
}

$workflow = new ezcWorkflow( 'Test' );

$input = new ezcWorkflowNodeInput(
  array(
    'choice' => 'boolean'
  )
);

$workflow->startNode->addOutNode( $input );

$branch = new ezcWorkflowNodeExclusiveChoice;
$branch->addInNode( $input );

$true  = new ezcWorkflowNodeAction( 'PrintTrue' );
$false = new ezcWorkflowNodeAction( 'PrintFalse' );

$branch->addConditionalOutNode(
  new ezcWorkflowConditionIsTrue(
    'choice'
  ),
  $true
);

$branch->addConditionalOutNode(
  new ezcWorkflowConditionIsFalse(
    'choice'
  ),
  $false
);

$merge = new ezcWorkflowNodeSimpleMerge;
$merge->addInNode( $true )
      ->addInNode( $false )
      ->addOutNode( $workflow->endNode );
?>
```

Listing 6.3: Workflow specification in XML markup

```
<?xml version="1.0" encoding="UTF-8"?>

<workflow name="Test" version="1">
  <node id="1" type="Start">
    <outNode id="3"/>
  </node>

  <node id="2" type="End"/>

  <node id="3" type="Input">
    <variable name="choice" constraint="boolean"/>
    <outNode id="4"/>
  </node>

  <node id="4" type="ExclusiveChoice">
    <condition type="IsTrue" variable="choice">
      <outNode id="5"/>
    </condition>

    <condition type="IsFalse" variable="choice">
      <outNode id="6"/>
    </condition>
  </node>

  <node id="5" type="Action" serviceObjectClass="PrintTrue">
    <outNode id="7"/>
  </node>

  <node id="6" type="Action" serviceObjectClass="PrintFalse">
    <outNode id="7"/>
  </node>

  <node id="7" type="SimpleMerge">
    <outNode id="2"/>
  </node>
</workflow>
```

6.6. Summary

The core of the workflow engine that has been developed as part of this thesis is a virtual machine that executes workflows represented through object graphs. These object graphs can be created programmatically through the software component's Workflow Definition API. Alternatively, a workflow definition can be loaded from an XML file. Object graph and XML file are two different representations of a workflow definition that uses the so-called backend language of the workflow engine's core. Arbitrary frontend languages such as the XML Process Definition Language (XPDL) [WfMC05], for instance, can be mapped to the workflow engine's backend language.

Chapter 7.

Evaluation and Related Work

This chapter evaluates the workflow model (see Chapter 5) and the software (see Chapter 6) that have been developed as part of this with regard to the requirements (see Chapter 4) and compares it related work.

7.1. Evaluation

7.1.1. Workflow Model

The workflow model (see Chapter 5) that is the basis of the software that has been developed as part of this thesis meets the requirements set up by eZ Systems AS (see Section 4.2). It provides good support for expressing control flow with its direct support of the basic control flow patterns (see Section 2.2.1) and the workflow patterns for advanced branching and synchronization (see Section 2.2.2). This allows the expression of operations such as the publishing, removal, and modification of content objects in eZ Publish to be expressed through workflows. The support of sub-workflows allows the decomposition of these workflows into manageable and reusable parts.

Table 7.1 compares the expressiveness of the `ezcWorkflow` components's *backend language* with regard to the directly supported workflow patterns to other workflow systems. The comparison data is partially taken from [BK03].

Workflow Pattern	ezcWorkflow	YAWL	eZ Publish 3	Galaxia	Radicore	Visual WorkFlo	Verve Workflow	Staffware	MQSeries Workflow	Forté Conductor	HP ChangeEngine	Fujitsu i-Flow	SAP R/3 Workflow
Sequence	√	√	√	√	√	√	√	√	√	√	√	√	√
Parallel Split	√	√	(√)	√	√	√	√	√	√	√	√	√	√
Synchronization	√	√		√	√	√	√	√	√	√	√	√	√
Exclusive Choice	√	√		√	√	√	(√)	√	(√)	√	√	√	√
Simple Merge	√	√		(√)	√	√	√	√	√	√	√	√	√
Multi-Choice	√	√				(√)	√	(√)	√	√	√	(√)	(√)
Synchronizing Merge	√	√			√			√					
Multi-Merge		√					√			√			
Discriminator	√	√					√			(√)	√		√
Arbitrary Cycles	√	√			√		√	√		√	√	√	
Implicit Termination	√							√	√				
Multiple Instances without Synchronization		√				√	√			√		√	
Multiple Instances with A Priori Design Time Knowledge		√				√	√	√	√	√	√	√	√
Multiple Instances with A Priori Runtime Knowledge		√											(√)
Multiple Instances without A Priori Runtime Knowledge		√											
Deferred Choice		√						(√)					
Interleaved Parallel Routing		√											
Milestone		√											
Cancel Activity		√						√					√
Cancel Case	√	√				√	√			√	√		√

Table 7.1.: Comparison of Workflow Systems

7.1.2. Implementation

The software that has been developed as part of this thesis meets the requirements set up by eZ Systems AS (see Section 4.2). It has been implemented using version 5 of the PHP programming language and is customizable and extendable. Its architecture allows the addition and customization of components for workflow execution, persistence, history, monitoring, and worklist management, for instance. The backend language that is understood by its virtual machine can be extended by implementing new node classes. The data storage for workflow schemas and the persistence of workflow instances has been abstracted, reference implementations for relational databases (workflow schemas and persistence) and XML files (workflow schemas only) are available. The workflow schemas are stored in such a way that they are versioned, old and new versions of a workflow can be executed at the same time. The *Workflow Execution API* provides access to information on the workflow instances that are currently executing and it is possible to manually control the workflow instances that are currently executing. Through PHP's native SOAP support, the *Workflow Execution API* can be exposed as a web service, thus faciliating a distributed and federated workflow environment where one workflow on one server can start another workflow on another server, for instance. A special purpose implementation of the workflow virtual machine, `ezcWorkflowTestExecution`, allows for the simulation of workflow execution for debugging and testing.

7.2. Related Work

7.2.1. Research

Micro-Workflow

Manolescu proposes *a new workflow architecture that bridges the gap between the type of functionality provided by current workflow systems and the type of workflow functionality required in object-oriented applications*. In his PhD thesis [DAM01], he discusses the design and implementation of Micro-Workflow, an object-oriented framework that is built using this architecture. One of Manolescu's key findings is that more advanced workflow features can be added to light-weight workflow core through composition.

SWAMP

In his Diploma thesis [TS04], Schmidt discusses the design and implementation of the SWAMP workflow system (SuSE Workflow and Management Platform). The goal of Schmidt's thesis is to replace an inhomogeneous legacy system with a unified workflow system that is easier to maintain and that can be easily customized and extended. The motivations behind and the requirements for the SWAMP workflow system have some similarities with the needs for the software that has been developed as part of this thesis.

YAWL

In her Master thesis [SH05], Heijens discusses the design and implementation of YAWL. YAWL is both a workflow language (Yet Another Workflow Language) and a workflow system. The workflow model of YAWL is formally based on Petri nets and supports all the workflow patterns with the exception of *Implicit Termination*.

7.2.2. Workflow Systems for PHP

eZ Publish 3

From the workflow patterns that were discussed in Section 2.2, the workflow system of eZ Publish 3 only supports the *Sequence* workflow pattern directly. Through its *Multiplexer* workflow event, which starts another workflow from within a workflow, it indirectly supports the *Parallel Split* workflow pattern.

The technical limitations of the workflow system in eZ Publish 3 (see Section 4.1) are representative for other content management systems for the PHP platform (and most likely for those for other platforms as well). The tight integration with the application into which the workflow system is embedded makes the independent usage of the workflow system impossible.

Galaxia Workflow Engine

The Galaxia Workflow Engine [GF03] is an activity-based workflow engine for PHP that is loosely based on OpenFlow [OPENFLOW].

The graphical workflow description language supported by the Galaxia workflow engine consists of six *activity types*:

1. *Start* represents the beginning of a workflow.
2. *End* represents the end of a workflow.
3. *Activity* represents an activity that is to be performed.
4. *Switch* represents a point of decision in the workflow and can be compared to the *Exclusive Choice* workflow pattern.
5. *Split* is equivalent to the *Parallel Split* workflow pattern.
6. *Join* is equivalent to the *Synchronization* workflow pattern.

Galaxia does not have an explicit *Simple Merge* construct to merge the multiple possible threads of a *Switch* construct. Instead, an *Activity* construct implicitly merges its incoming threads.

Radicore

The Radicore toolkit for PHP features an activity-based workflow engine that is based on Petri nets [TM04].

It supports

- Sequential Routing.
- Parallel Routing through *AND-Split* and *AND-Join* constructs.
- Conditional Routing through explicit and implicit *OR-Split* and *OR-Join* constructs.
- Iterative Routing using the *OR-Split* construct.

These constructs correspond to the *Sequence*, *Parallel Split*, *Synchronization*, *Exclusive Choice*, and *Simple Merge* workflow patterns (see Section 2.2).

7.3. Summary

Workflow systems such as Micro-Workflow [DAM01] and YAWL [SH05], for instance, that have been implemented as part of academic research often excel only in the aspect that is specific to the research while neglecting other aspects that are relevant to a workflow system. One of the goals of this thesis was to combine aspects such as component-based workflow architecture, workflow virtual machine, and workflow patterns to create a workflow system that meets industry requirements.

The existing workflow systems for PHP do not lend themselves well to customization and extension with regard to the requirements set up by eZ Systems AS. This fact, together with eZ Systems' requirement for clear intellectual property, lead to the development of a new workflow engine instead of starting with an existing one.

Chapter 8.

Conclusion and Future Work

8.1. Conclusion

This thesis reviewed previous research such as [BK03, DAM01, PM99, SF04] and combined it for the first time in an effort to design and implement a workflow system that meets industry requirements.

The pragmatic approach to describe the semantics of workflow routing constructs through *Workflow Patterns* [BK03] provides a good foundation for the *Backend Language* of a *Workflow Virtual Machine* [SF04] that executes workflow definitions represented through object graphs in a component-based workflow architecture [DAM01].

The software that has been developed as part of this thesis is a contribution to the PHP community. It provides an extendable framework to define workflows and a virtual machine for the execution of these definitions that can be embedded into a PHP application, thus extending it with workflow capabilities. This workflow system can be customized and extended through the composition of components, its workflow model can be customized and extended through the classes that define the control flow constructs. It is neither bound to a specific application into which it is embedded nor to a specific workflow description language, thus providing more degrees of freedom with regard to use – and re-use – of the workflow engine.

8.2. Future Work

8.2.1. Analysis and Verification of Workflows

The current implementation of the software that has been developed as part of this thesis has basic support for the analysis and verification of workflow specifications. Future versions of the software can implement more advanced verification tools based upon the *abundance of analysis techniques* that exists for Petri nets [WA96].

8.2.2. Workflow Model

The workflow model can be extended, for instance, with support for more workflow patterns, by adding the respective node types.

8.2.3. Aspect-Oriented Programming

Aspect-Oriented Programming *allow[s] programming by making quantified programmatic assertions over programs written by programmers oblivious to such assertions* [RF00]. These assertions make *quantified statements about which code is to execute in which circumstances*.

Section 2 of [SB06] presents an overview of the various implementations of AOP for PHP that support AOP by extending the base programming language. The combination of Graph-Oriented Programming with Aspect-Oriented Programming would add yet another possibility to faciliate AOP with the PHP platform, but without the need to change or extend the base programming language. The workflow model discussed in Chapter 5 would serve as the Joinpoint Model of an AOP system that can be implemented as an additional component for the software that was presented in Chapter 6. Pointcuts such as *node of type X is executed* could then be used to express when additional code is to be run during workflow execution. Compared to the implicit callgraph structure on which language-level AOP systems operate, the explicit graph structure of workflows makes the idea of AOP intuitively clear.

This combination of Aspect-Oriented Programming with Graph-Oriented Programming could then be compared to Aspect-Oriented Programming in general as well as to Adaptive Programming which Lieberherr describes as *the special case of Aspect-Oriented Programming (AOP) where some of the building blocks are expressible in terms of graphs* [KL97].

8.2.4. Compilation of Workflows

Model-Driven Architecture (MDA) *separates business and application logic from underlying platform technology* [JM01] and supports the Model-Driven Engineering (MDE) of software systems. It offers a *promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively* [DS06].

In this context the possibility could be evaluated whether the software that has been developed as part of this thesis can be extended with a code generator component that can compile a workflow specification into a ready-to-use application.

Appendix A.

Tutorial

This appendix serves as an introduction to the *Workflow Definition API* and *Workflow Execution API* (see Figure 6.1).

A.1. Workflow Definition API

A.1.1. Defining a New Workflow

In this subsection, we define a new workflow by creating an object graph of `ezcWorkflowNode` objects. Once we learned how to define and store such a workflow definition, we we will look at loading and editing an existing workflow definition.

Creating the Object Graph

First, we load the `ezcBase` component (line 2) and set up its classloader that is based on PHP's `__autoload` interceptor (lines 4–7).

```
1 <?php
2 require_once 'Base/base.php';
3
4 function __autoload( $className )
5 {
6     ezcBase::autoload( $className );
7 }
```

For the following code listings, lines 1–7 will always be the same as in the listing above.

We create a new object of the `ezcWorkflow` class (line 8). This object represents the workflow that we are about to define. The constructor of the class expects a string with a name for the workflow. This name is unique for the schema repository to which the workflow may be saved later.

```
8 $workflow = new ezcWorkflow( 'Test' );
```

We define an *Input* node by creating an object of the `ezcWorkflowNodeInput` class. The constructor expects an associative array where the key stands for the name of an input variable and the value may hold arbitrary expectations for this variable. These expectations are evaluated and checked by the application that uses the workflow component.

Then we set up the *Input* node as an outgoing node of the *Start* node.

```
9  $input = new ezcWorkflowNodeInput(
10   array(
11     'choice' => new ezcWorkflowConditionIsBool
12   )
13 );
14
15 $workflow->startNode->addOutNode( $input );
```

For the purposes of this example we assume that `'choice' => 'boolean'` means that a boolean input variable of name "choice" is expected.

We now define an *Exclusive Choice* node that uses the value of the input variable to activate one of two possible paths. Before we set up its outgoing nodes (and the related conditions), we set up the *Exclusive Choice* as an outgoing node of the *Input* node (lines 16–17).

```
16 $branch = new ezcWorkflowNodeExclusiveChoice;
17 $branch->addInNode( $input );
```

In the next step, we create two objects, `$true` and `$false` of the `ezcWorkflowNodeAction` class (lines 18–19). The constructor expects the name of a class that implements the `ezcWorkflowServiceObject` interface. Such a class encapsulates the business logic that is associated with the *Action* node.

```
18 $true  = new ezcWorkflowNodeAction( 'PrintTrue' );
19 $false = new ezcWorkflowNodeAction( 'PrintFalse' );
```

Let us take a look at what an implementation of the `ezcWorkflowServiceObject` interface looks like:

```
<?php
class PrintTrue implements ezcWorkflowServiceObject
{
    public function execute( ezcWorkflowExecution $e )
    {
        print "TRUE\n";
    }

    public function __toString()
    {
        return 'PrintTrue';
    }
}
?>
```

The `ezcWorkflowServiceObject` interface requires two methods, `execute()` and `__toString()`. The former implements the business logic of the service object and is passed the execution context as its only argument. The latter provides a textual representation of the service object.

Using the `$branch` object's `addConditionalOutNode()` method, we can now set up the *Action* node that is represented by the `$true` object as an conditional outgoing node (lines 20–26). This method expects an `ezcWorkflowCondition` object as its first argument. This object encapsulates the branching condition.

For our example we use the `ezcWorkflowConditionIsTrue` class. The constructor of this class expects the name of a workflow variable that is to be evaluated.

```
20 $branch->addConditionalOutNode(
21   new ezcWorkflowConditionVariable(
22     'choice',
23     new ezcWorkflowConditionIsTrue
24   ),
25   $true
26 );
```

Section B.2 shows the available `ezcWorkflowCondition` implementations.

Analogous, we set up the *Action* node that is represented by the `$false` object as a second conditional outgoing node (lines 27–33).

```
27 $branch->addConditionalOutNode(
28   new ezcWorkflowConditionVariable(
29     'choice',
30     new ezcWorkflowConditionIsFalse
31   ),
32   $false
33 );
```

Finally, we create a new object of the `ezcWorkflowNodeSimpleMerge` class (line 34). We set up the two *Action* nodes as incoming nodes and the *End* node as an outgoing node of this *Simple Merge* node (lines 35–37).

```
34 $merge = new ezcWorkflowNodeSimpleMerge;
35 $merge->addInNode( $true )
36       ->addInNode( $false )
37       ->addOutNode( $workflow->endNode );
```

This concludes the creation of the object graph that represents the workflow specification and can now be stored in a workflow schema repository. Currently two workflow schema repository backends are supported, XML files and relational databases.

Writing the Workflow Schema to an XML File

The following code snippet demonstrates how to serialize the object graph to an XML representation.

```
38 $definition = new ezcWorkflowDefinitionStorageXml;
39 $definition->save( $workflow );
40 ?>
```

The constructor of the `ezcWorkflowDefinitionStorageXml` class accepts an optional argument that specifies the directory in which the XML files are stored.

Listing A.1 shows the resulting XML document that is written to a file named `Test_1.xml`. The filename includes the name of the workflow definition and its version number.

Frontend languages such as the XML Process Definition Language (XPDL) [WfMC05] can be transformed to this format using XSL Transformations (XSLT) [W3C07], for instance.

Listing A.1: Workflow specification in XML markup

```
<?xml version="1.0" encoding="UTF-8"?>

<workflow name="Test" version="1">
  <node id="1" type="Start">
    <outNode id="3"/>
  </node>

  <node id="2" type="End"/>

  <node id="3" type="Input">
    <variable name="choice">
      <condition type="IsBool"/>
    </variable>
    <outNode id="4"/>
  </node>

  <node id="4" type="ExclusiveChoice">
    <condition type="Variable" name="choice">
      <condition type="IsTrue"/>
      <outNode id="5"/>
    </condition>

    <condition type="Variable" name="choice">
      <condition type="IsFalse"/>
      <outNode id="6"/>
    </condition>
  </node>

  <node id="5" type="Action" serviceObjectClass="PrintTrue">
    <outNode id="7"/>
  </node>

  <node id="6" type="Action" serviceObjectClass="PrintFalse">
    <outNode id="7"/>
  </node>

  <node id="7" type="SimpleMerge">
    <outNode id="2"/>
  </node>
</workflow>
```

Saving the Workflow Schema to a Database

The constructor of the `ezcWorkflowDatabaseDefinition` class expects and object of the `ezcDbHandler` class. In line 38 we create such an object and connect to a MySQL database server. Then we can use the `save()` method of the `ezcWorkflowDatabaseDefinition` object to save the object graph to the database.

```
38 $db         = ezcDbFactory::create( 'mysql://test@localhost/test' );
39 $definition = new ezcWorkflowDatabaseDefinition( $db );
40 $definition->save( $workflow );
41 ?>
```

Visualizing a Workflow Graph

The next code snippet shows how to use the `ezcWorkflowVisitorVisualization` class to generate a description of the object graph in the DOT graph description language.

```
38 $visitor = new ezcWorkflowVisitorVisualization;
39 $workflow->accept( $visitor );
40 print $visitor;
41 ?>
```

Listing A.2 shows the resulting DOT graph description, Figure A.1 shows the workflow graph rendered using GraphViz from this graph description.

A.1.2. Loading an Existing Workflow

Loading a Workflow Schema from an XML File

The following code snippet demonstrates how to load an object graph from an XML representation.

```
8 $definition = new ezcWorkflowDefinitionStorageXml;
9 $workflow   = $definition->loadByName( 'Test' );
10 ?>
```

The `loadByName()` method accepts an optional second argument that specifies the version number of the workflow definition that is to be loaded. By default (ie. without the second argument) the newest version of the workflow is loaded.

Listing A.2: Workflow specification in DOT markup

```
digraph Test {
node1 [label="Start"]
node3 [label="Input"]
node4 [label="Exclusive Choice"]
node5 [label="PrintTrue"]
node7 [label="Simple Merge"]
node2 [label="End"]
node6 [label="PrintFalse"]

node1 -> node3
node3 -> node4
node4 -> node5 [label="choice is true"]
node4 -> node6 [label="choice is false"]
node5 -> node7
node7 -> node2
node6 -> node7
}
```

Loading a Workflow Schema from a Database

Loading a workflow schema from a database is analogous to loading fron an XML file:

```
8  $db         = ezcDbFactory::create( 'mysql://test@localhost/test' );
9  $definition = new ezcWorkflowDatabaseDefinition( $db );
10 $workflow   = $definition->loadByName( 'Test' );
```

For the following code listings, lines 8–10 will always be the same as in the listing above.

A.2. Workflow Execution API

The software that has been developed as part of this thesis offers three workflow execution engines: `ezcWorkflowExecutionNonInteractive`, `ezcWorkflowDatabaseExecution`, and `ezcWorkflowTestExecution`. This sections shows how they are used.

A.2.1. Workflow with Wait States

For the execution of a workflow that contains wait states (for example *Input* nodes), an execution engine is required that supports persistence. The `ezcWorkflowDatabaseExecution`

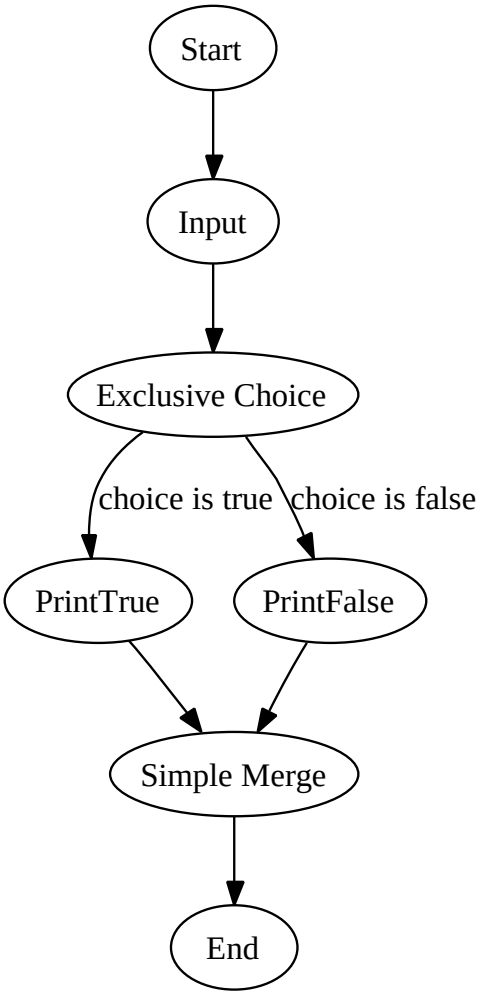

Figure A.1.: Workflow graph rendered using GraphViz

class implements such an execution engine and uses a relational database for persistence storage.

In the following code snippet we start start the execution of a previously loaded workflow.

```
11 $execution = new ezcWorkflowDatabaseExecution( $db );
12 $execution->workflow = $workflow;
13 $executionId = $execution->start();
```

As our workflow contains an *Input* node, the execution will not complete and will be suspended. The `$executionId` uniquely identifies the suspended workflow execution. It can be used, for instance, to resume the workflow execution once the requested input data has been provided:

```
14 $execution = new ezcWorkflowDatabaseExecution( $db );
15 $execution->resume( $executionId, array( 'choice' => true ) );
16 ?>
```

A.2.2. Workflow without Wait States

A workflow that contains no wait states can be executed in one pass. The workflow engine is not required to support persistence for this. The `ezcWorkflowExecutionNonInteractive` class implements a workflow engine without persistence support that can execute such workflow without the overhead of a persistence layer.

```
11 $execution = new ezcWorkflowExecutionNonInteractive;
12 $execution->workflow = $workflow;
13 $execution->start();
14 ?>
```

A.2.3. Simulating Workflow Execution

The workflow engine implemented by the `ezcWorkflowTestExecution` class can be used for testing both the workflow system itself as well as workflow definitions.

```
11 $execution = new ezcWorkflowTestExecution;
12 $execution->workflow = $workflow;
13 $execution->setInputVariable( 'choice', true );
14 $execution->start();
15 ?>
```

The `setInputVariable()` method allows for the mocking of *Input* nodes, thus making it possible to execute and test interactive workflows without interaction.

Appendix B.

API Reference

This appendix provides an API reference for the software that has been developed as part of this thesis.

B.1. Graph Node Classes

Objects of the `ezcWorkflowNode` classes represent the nodes of a workflow.

B.1.1. ezcWorkflowNode

`ezcWorkflowNode` (see Figure B.1) is the abstract base class for all graph node classes.

B.1.2. Start and End Nodes

ezcWorkflowNodeStart

Incoming Nodes: 0
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeStart` class (see Figure B.1) represents the one and only start node of a workflow. The execution of the workflow starts here.

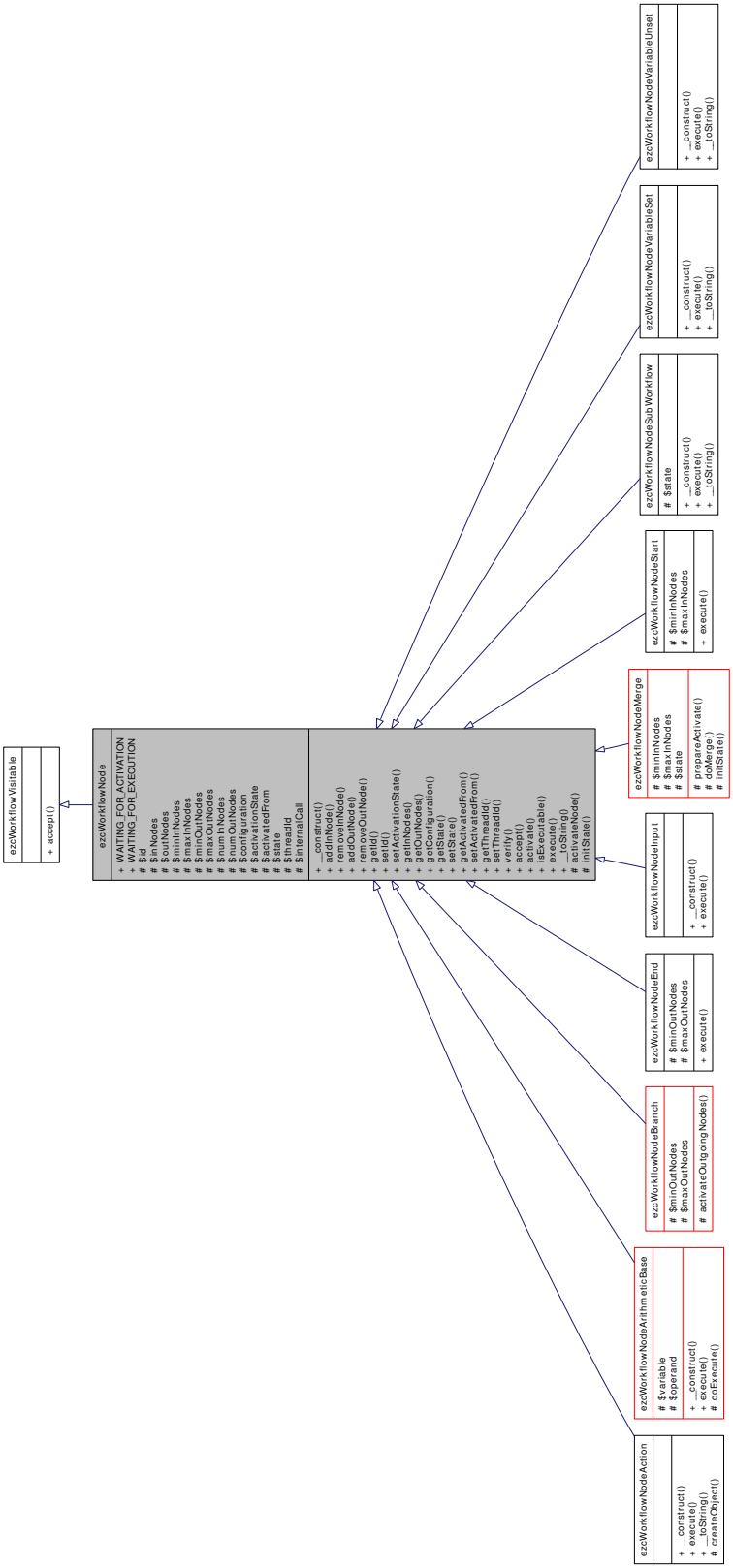

Figure B.1.: The `ezcWorkflowNode` class and its subclasses

Creating an object of the `ezcWorkflow` automatically creates the start node for this new workflow.

```
$workflow = new ezcWorkflow( 'Name' );
$workflow->startNode; // This property holds the ezcWorkflowNodeStart
    object.
```

ezcWorkflowNodeEnd

Incoming Nodes: 1
Outgoing Nodes: 0

An object of the `ezcWorkflowNodeEnd` class (see Figure B.1) represents an end node of a workflow. A workflow must have at least one end node. The execution of the workflow ends when an end node is reached.

Creating an object of the `ezcWorkflow` automatically creates a default end node for this new workflow.

```
$workflow = new ezcWorkflow( 'Name' );
$workflow->endNode; // This property holds an ezcWorkflowNodeEnd object
    .
```

ezcWorkflowNodeCancel

Incoming Nodes: 1
Outgoing Nodes: 0..1

The `ezcWorkflowNodeCancel` class implements the *Cancel Case* workflow pattern.

```
$workflow = new ezcWorkflow( 'Name' );
$workflow->endNode = new ezcWorkflowNodeCancel;
```

As soon as a node of the `ezcWorkflowNodeCancel` type is activated, the complete workflow instance is removed. This includes currently executing nodes, those which may execute at some future time and all parent and sub-workflows. The workflow instance is recorded as having completed unsuccessfully.

ezcWorkflowNodeFinally

Incoming Nodes: 0
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeFinally` class (see Figure B.1) represents the start node of a sequence of final activities that is executed when a workflow execution is cancelled.

Creating an object of the `ezcWorkflow` class automatically creates the finally node for this new workflow.

```
$workflow = new ezcWorkflow( 'Name' );
$workflow->endNode = new ezcWorkflowNodeCancel;
$workflow->finallyNode->addOutNode( /* ... */ );
```

B.1.3. ezcWorkflowNodeAction

Incoming Nodes: 1
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeAction` class (see Figure B.1) represents an activity node. When the node is reached, the business logic that is implemented by the associated *service object* is executed.

```
class MyAction implements ezcWorkflowServiceObject
{
    public function execute( ezcWorkflowExecution $execution )
    {
        // ...
    }

    public function __toString()
    {
        // ...
    }
}

$action = new ezcWorkflowNodeAction( 'MyAction' );
```

B.1.4. ezcWorkflowNodeSubWorkflow

Incoming Nodes: 1
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeSubWorkflow` class (see Figure B.1) represents a sub-workflow. When the node is reached, the specified sub-workflow is started. The workflow is suspended until the sub-workflow has finished executing.

```
$subWorkflow = new ezcWorkflow( 'Sub-Workflow Name' );
// ...

$subWorkflow = new ezcWorkflowNodeSubWorkflow( 'Sub-Workflow Name' );
```

Workflow variables can be passed from the parent workflow to the child worflow and vice versa. The example below creates a sub-workflow node that passes the parent execution's variable `x` to the variable `y` in the child execution when the sub-workflow is started. When it ends, the child execution's `y` variable is passed to the parent execution as `z`.

```
$subWorkflow = new ezcWorkflow( 'Sub-Workflow Name' );
// ...

$subWorkflow = new ezcWorkflowNodeSubWorkflow(
  array(
    'workflow'  => 'IncrementVariable',
    'variables' => array(
      'in' => array(
        'x' => 'y'
      ),
      'out' => array(
        'y' => 'z'
      )
    )
  )
);
```

B.1.5. Workflow Variables

ezcWorkflowNodeInput

Incoming Nodes: 1
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeInput` class (see Figure B.1) represents an input node. When the node is reached, the workflow engine will suspend the workflow execution if the specified input data is not available (first activation). While the workflow is suspended, the application that embeds the workflow engine may supply the input data and resume the workflow execution (second activation of the input node). Input data is stored in a workflow variable.

ezcWorkflowNodeVariableSet

Incoming Nodes: 1
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeVariableSet` class sets a specified workflow variable to a given value.

```
1 $set = new ezcWorkflowNodeVariableSet(
2   array( 'variable name' => $value )
3 );
```

ezcWorkflowNodeVariableUnset

Incoming Nodes: 1
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeVariableUnset` class unsets a specified workflow variable.

```
1 $unset = new ezcWorkflowNodeVariableUnset( 'variable name' );
```

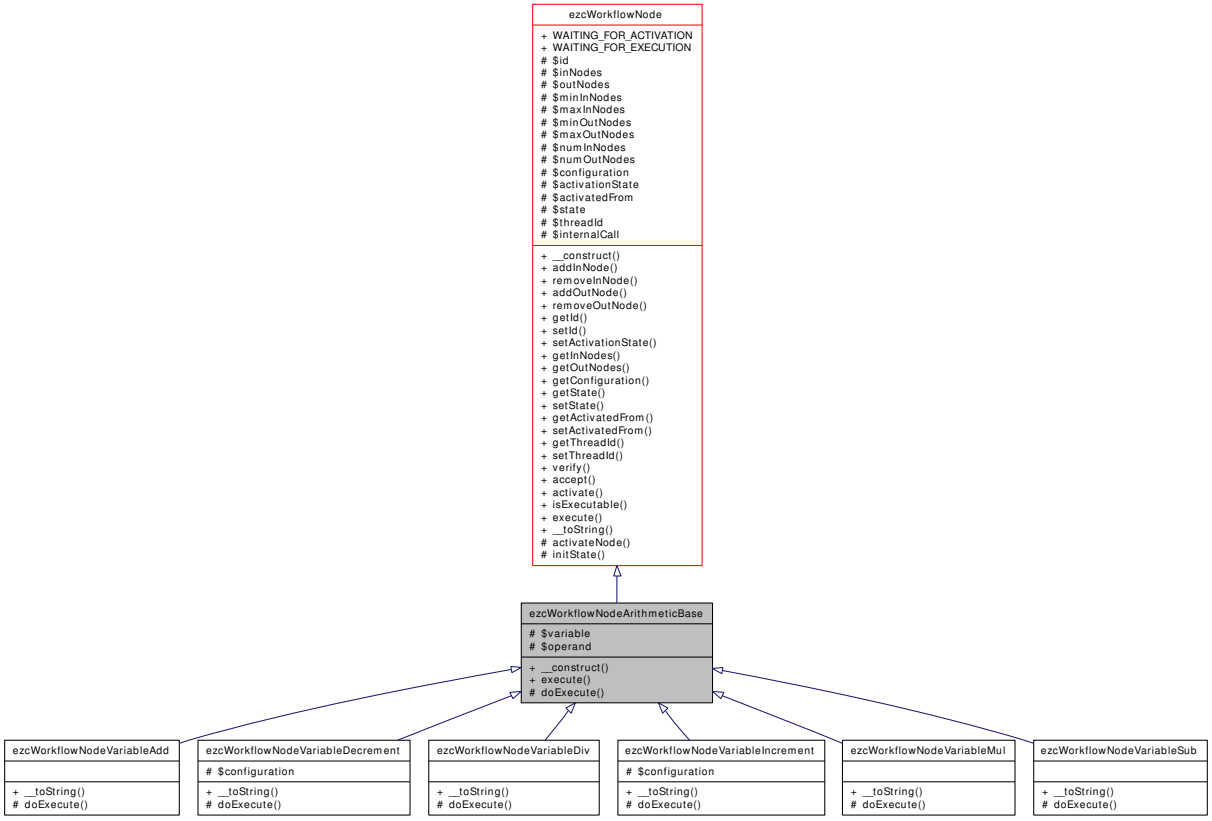

Figure B.2.: The `ezcWorkflowNodeArithmeticBase` class and its subclasses

ezcWorkflowNodeVariableAdd

Incoming Nodes: 1
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeVariableAdd` class adds a given value, either a constant or the value of another workflow variable, to a specified workflow variable.

```
$add = new ezcWorkflowNodeVariableAdd(
  array( 'name' => 'variable name', 'value' => $value )
);
```

When `$value` is a string, the value of the variable identified by that string is used.

ezcWorkflowNodeVariableSub

Incoming Nodes: 1
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeVariableSub` class subtracts a given value, either a constant or the value of another workflow variable, from a specified workflow variable.

```
$sub = new ezcWorkflowNodeVariableSub(
  array( 'name' => 'variable name', 'value' => $value )
);
```

When `$value` is a string, the value of the variable identified by that string is used.

ezcWorkflowNodeVariableMul

Incoming Nodes: 1
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeVariableMul` class multiplies a specified workflow variable with a given value, either a constant or the value of another workflow variable.

```
$mul = new ezcWorkflowNodeVariableMul(
  array( 'name' => 'variable name', 'value' => $value )
);
```

When `$value` is a string, the value of the variable identified by that string is used.

ezcWorkflowNodeVariableDiv

Incoming Nodes: 1
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeVariableDiv` class divides a specified workflow variable by a given value, either a constant or the value of another workflow variable.

```
$div = new ezcWorkflowNodeVariableDiv(
  array( 'name' => 'variable name', 'value' => $value )
);
```

When `$value` is a string, the value of the variable identified by that string is used.

ezcWorkflowNodeVariableIncrement

Incoming Nodes: 1
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeVariableIncrement` class increments the value of a specified workflow variable.

```
$inc = new ezcWorkflowNodeVariableIncrement( 'variable name' );
```

ezcWorkflowNodeVariableDecrement

Incoming Nodes: 1
Outgoing Nodes: 1

An object of the `ezcWorkflowNodeVariableDecrement` class decrements the value of a specified workflow variable.

```
$dec = new ezcWorkflowNodeVariableDecrement( 'variable name' );
```

B.1.6. Workflow Patterns

ezcWorkflowNodeParallelSplit

Incoming Nodes: 1
*Outgoing Nodes: 2 . . . **

The `ezcWorkflowNodeParallelSplit` class implements the *Parallel Split* workflow pattern.

ezcWorkflowNodeSynchronization

*Incoming Nodes: 2 . . . **
Outgoing Nodes: 1

The `ezcWorkflowNodeSynchronization` class implements the *Synchronization* workflow pattern.

ezcWorkflowNodeExclusiveChoice

Incoming Nodes: 1
*Outgoing Nodes: 2 . . . **

The `ezcWorkflowNodeExclusiveChoice` class implements the *Exclusive Choice* workflow pattern.

ezcWorkflowNodeSimpleMerge

*Incoming Nodes: 2 . . . **
Outgoing Nodes: 1

The `ezcWorkflowNodeSimpleMerge` class implements the *Simple Merge* workflow pattern.

ezcWorkflowNodeLoop

Incoming Nodes: 2
Outgoing Nodes: 2

The `ezcWorkflowNodeLoop` class is a specialization of the `ezcWorkflowNodeExclusiveChoice` class and may be used to conveniently express loops.

```
$workflow = new ezcWorkflow( 'IncrementingLoop' );

$set  = new ezcWorkflowNodeVariableSet( array( 'i' => 1 ) );
$step = new ezcWorkflowNodeVariableIncrement( 'i' );

$break = new ezcWorkflowConditionVariable(
  'i', new ezcWorkflowConditionIsEqual( 10 )
);

$continue = new ezcWorkflowConditionVariable(
  'i', new ezcWorkflowConditionIsLessThan( 10 )
);

$workflow->startNode->addOutNode( $set );

$loop = new ezcWorkflowNodeLoop;
$loop->addInNode( $set )
       addInNode( $step )
       addConditionalOutNode( $continue, $step )
       addConditionalOutNode( $break, $workflow->endNode );
```

The code above is equivalent to a `for`-loop that iterates the variable `i` from `1` to `10`.

ezcWorkflowNodeMultiChoice

Incoming Nodes: 1
*Outgoing Nodes: 2 ... **

The `ezcWorkflowNodeMultiChoice` class implements the *Multi-Choice* workflow pattern.

ezcWorkflowNodeSynchronizingMerge

*Incoming Nodes: 2 ... **
Outgoing Nodes: 1

The `ezcWorkflowNodeSynchronizingMerge` class implements the *Synchronizing Merge* workflow pattern.

ezcWorkflowNodeDiscriminator

*Incoming Nodes: 2 . . . **
Outgoing Nodes: 1

The `ezcWorkflowNodeDiscriminator` class implements the *Discriminator* workflow pattern.

B.2. Condition Classes

The `ezcWorkflowCondition` classes can be used to express branch conditions and input validation.

B.2.1. Variable Access

ezcWorkflowConditionVariable

An object of the `ezcWorkflowConditionVariable` class decorates another `ezcWorkflowCondition` object and applies it condition to a workflow variable.

```
$condition = new ezcWorkflowConditionVariable(
  'foo', new ezcWorkflowConditionIsTrue
);
```

ezcWorkflowConditionVariables

An object of the `ezcWorkflowConditionVariables` class decorates an `ezcWorkflowConditionComparison` object and applies it to two workflow variables.

```
$condition = new ezcWorkflowConditionVariables(
  'foo', 'bar', new ezcWorkflowConditionIsEqual
);
```

B.2.2. Boolean Expressions

ezcWorkflowConditionNot

An object of the `ezcWorkflowConditionNot` class decorates an `ezcWorkflowCondition` object and negates its expression.

```
1 $notNondition = new ezcWorkflowConditionNot( $condition );
```

ezcWorkflowConditionAnd

An object of the `ezcWorkflowConditionAnd` class represents a boolean *AND* expression. It can hold an arbitrary number of `ezcWorkflowCondition` objects.

```
1 $and = new ezcWorkflowConditionAnd( array( $condition, ... ) );
```

ezcWorkflowConditionOr

An object of the `ezcWorkflowConditionOr` class represents a boolean *OR* expression. It can hold an arbitrary number of `ezcWorkflowCondition` objects.

```
1 $or = new ezcWorkflowConditionOr( array( $condition, ... ) );
```

ezcWorkflowConditionXor

An object of the `ezcWorkflowConditionXor` class represents a boolean *XOR* expression. It can hold an arbitrary number of `ezcWorkflowCondition` objects.

```
1 $xor = new ezcWorkflowConditionXor( array( $condition, ... ) );
```

B.2.3. Comparisons

ezcWorkflowConditionIsTrue

The condition represented by an `ezcWorkflowConditionIsTrue` object evaluates to `true` when the associated workflow variable has the value `true`.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsTrue
);
```

ezcWorkflowConditionIsFalse

The condition represented by an `ezcWorkflowConditionIsFalse` object evaluates to `true` when the associated workflow variable has the value `false`.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsFalse
);
```

ezcWorkflowConditionIsEqual

The condition represented by an `ezcWorkflowConditionIsEqual` object evaluates to `true` when the associated workflow variable is equal to the comparison value.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsEqual( $comparisonValue )
);
```

ezcWorkflowConditionIsNotEqual

The condition represented by an `ezcWorkflowConditionIsNotEqual` object evaluates to `true` when the associated workflow variable is not equal to the comparison value.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsNotEqual( $comparisonValue )
);
```

ezcWorkflowConditionIsGreaterThan

The condition represented by an `ezcWorkflowConditionIsGreaterThan` object evaluates to `true` when the associated workflow variable is greater than the comparison value.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsGreaterThan( $comparisonValue )
);
```

ezcWorkflowConditionIsEqualOrGreaterThan

The condition represented by an `ezcWorkflowConditionIsEqualOrGreaterThan` object evaluates to `true` when the associated workflow variable is equal or greater than the comparison value.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsEqualOrGreaterThan( $comparisonValue )
);
```

ezcWorkflowConditionIsLessThan

The condition represented by an `ezcWorkflowConditionIsLessThan` object evaluates to `true` when the associated workflow variable is less than the comparison value.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsLessThan( $comparisonValue )
);
```

ezcWorkflowConditionIsEqualOrLessThan

The condition represented by an `ezcWorkflowConditionIsEqualOrLessThan` object evaluates to `true` when the associated workflow variable is equal or less than the comparison value.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsEqualOrLessThan( $comparisonValue )
);
```

B.2.4. Types

ezcWorkflowConditionIsAnything

The condition represented by an `ezcWorkflowConditionIsAnything` object always evaluates to `true`.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsAnything
);
```

ezcWorkflowConditionIsArray

The condition represented by an `ezcWorkflowConditionIsArray` object evaluates to `true` when the associated workflow variable is an array.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsArray
);
```

ezcWorkflowConditionIsBool

The condition represented by an `ezcWorkflowConditionIsBool` object evaluates to `true` when the associated workflow variable is a boolean.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsBool
);
```

ezcWorkflowConditionIsFloat

The condition represented by an `ezcWorkflowConditionIsFloat` object evaluates to `true` when the associated workflow variable is a float.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsFloat
);
```

ezcWorkflowConditionIsInteger

The condition represented by an `ezcWorkflowConditionIsInteger` object evaluates to `true` when the associated workflow variable is an integer.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsInteger
);
```

ezcWorkflowConditionIsObject

The condition represented by an `ezcWorkflowConditionIsObject` object evaluates to `true` when the associated workflow variable is an object.

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsObject
);
```

ezcWorkflowConditionIsString

The condition represented by an `ezcWorkflowConditionIsString` object evaluates to `true` when the associated workflow variable is a string

```
$condition = new ezcWorkflowConditionVariable(
  'variable name',
  new ezcWorkflowConditionIsString
);
```

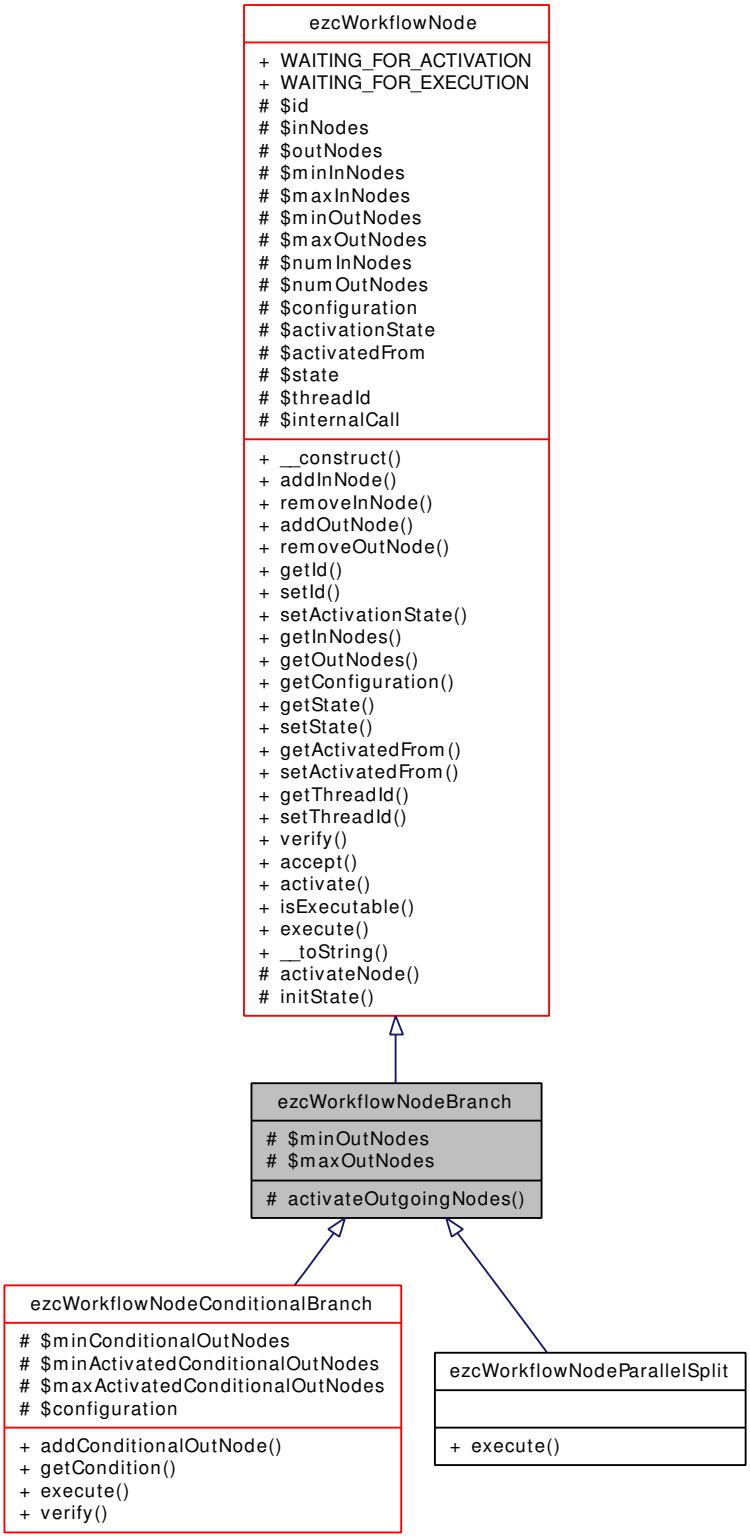

Figure B.3.: The `ezcWorkflowNodeBranch` class and its subclasses

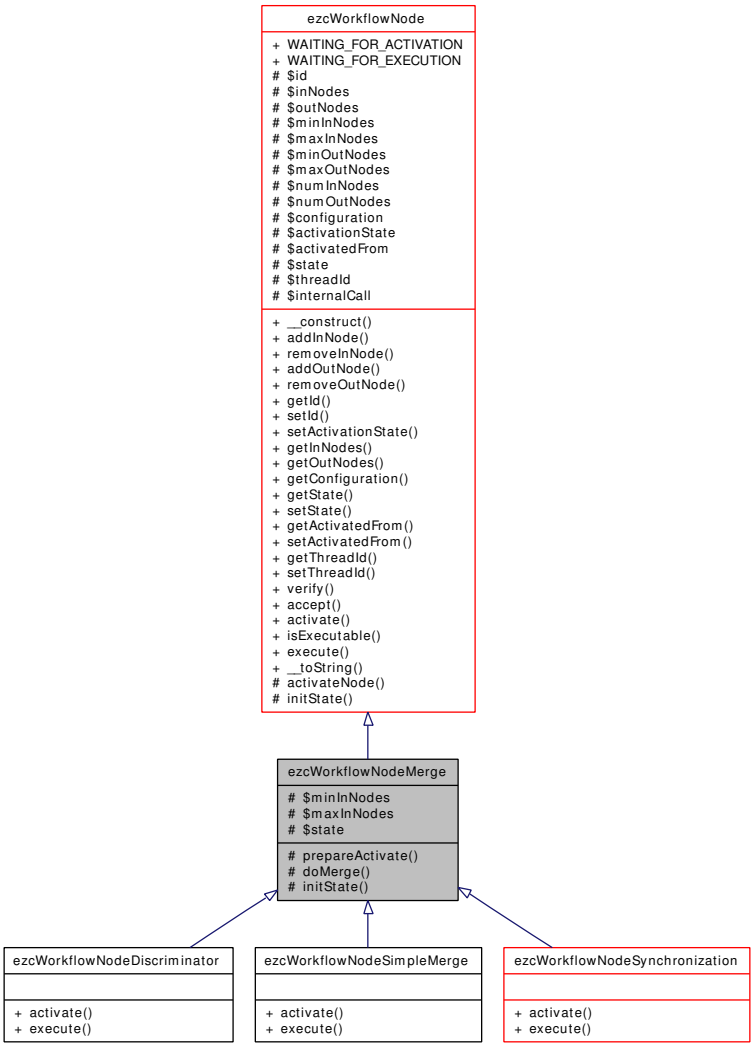

Figure B.4.: The `ezcWorkflowNodeMerge` class and its subclasses

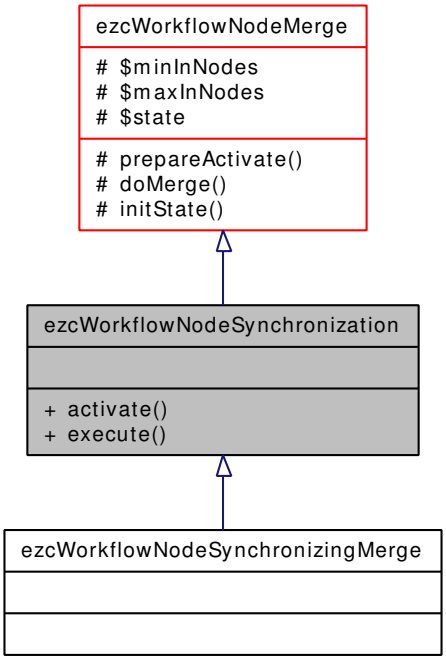

Figure B.5.: The `ezcWorkflowNodeSynchronization` class and its subclasses

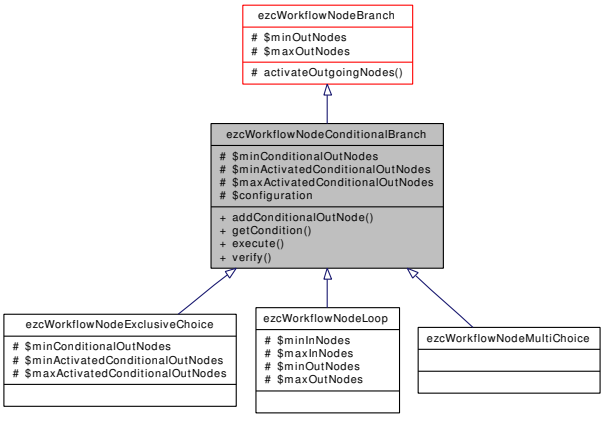

Figure B.6.: The `ezcWorkflowNodeConditionalBranch` class and its subclasses

Appendix C.

Bibliography

[AIIM] Association for Information and Image Management (AIIM). *What is ECM?* `http://www.aiim.org/about-ecm.asp`

[ARK03] Atul Ravi Khemuka. *Workflow Modeling Using Finite Automata*. PhD Thesis, Department of Industrial and Management Engineering, College of Engineering, University of South Florida, USA, 2003.

[BK03] Bartosz Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD Thesis, Faculty of Information Technology, Queensland University of Technology, Australia, 2003.

[DAM01] Dragos-Anton Manolescu. *Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development*. PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, USA, 2001.

[DG95] Dimitrios Georgakopoulos and Mark F. Hornick and Amit P. Sheth. *An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure*. In: Distributed and Parallel Databases, Volume 3, Number 2, Pages 119–153, 1995.

[DQZ01] Da-Qian Zhang and Kang Zhang and Jiannong Cao. *A Context-Sensitive Graph Grammar Formalism for the Specification of Visual Languages*. In: The Computer Journal, Volume 33, Number 3, Pages 186–200, 2001.

[DS06] Douglas C. Schmidt. *Model-Driven Engineering*. In: IEEE Computer, Volume 39, Number 2, Pages 25–31, 2006.

[FB96] Frank Buschmann and Regine Meunier and Hans Rohnert and Peter Sommerlad and Michael Stahl. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons, 1996.

[FG02] Florent Guillaume. *Trying to unify Entity-based and Activity-based workflows*. `http://wiki.zope.org/zope3/TryingToUnifiyWorkflowConcepts`

[GF03] Garland Foster and Richard Moore and Eduardo Polidor. *Galaxia: An Open Source Workflow Engine for Tiki* `http://workflow.tikiwiki.org/tiki-index.php?page=GalaxiaConcepts`

[GoF94] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[JBOSS] The JBoss Project. *JBoss jBPM: Workflow and BPM Made Practical*. `http://docs.jboss.com/jbpm/v3/userguide/graphorientedprogramming.html`

[JD01] Jörg Desel and Gabriel Juhás. *What Is a Petri Net?*. In: Unifying Petri Nets: Advances in Petri Nets, Lecture Notes in Computer Science, Volume 2128/2001, Springer, 2001.

[JM01] Jishnu Mukerji and Joaquin Miller. *MDA Guide*. `http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf`

[KL97] Karl J. Lieberherr. *Connections between Demeter/Adaptive Programming and Aspect-Oriented Programming (AOP)*. College of Computer Science, Northeastern University, Boston, MA, USA, 1997. `http://www.ccs.neu.edu/home/lieber/connection-to-aop.html`

[MF05] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* June, 2005. `http://martinfowler.com/articles/languageWorkbench.html`

[ML05] Markus Löschnigg. *XML in Workflow Management Systems*. Master's Thesis, Graz University of Technology, Austria, 2005.

[OPENFLOW] The OpenFlow Project. *OpenFlow: An Introduction*. `http://www.openflow.it/wwwopenflow/Documentation/documentation/OpenFlowIntroduction/`

[PM99] Peter Muth and Jeanine Weisenfels and Michael Gillmann and Gerhard Weikum. *Integrating Light-Weight Workflow Management Systems within Existing Business Environments*. In: Proceedings of the 15th International Conference on Data Engineering, March 1999, Sydney, Australia.

[RA01] Rob Allen. *Workflow: An Introduction*. In: Workflow Handbook, Workflow Management Coalition, 2001.

[RF00] Robert E. Filman and Daniel P. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*. In: Proceedings of the Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis, USA.

[SB05] Sebastian Bergmann. *Professionelle Softwareentwicklung mit PHP 5*. dpunkt.verlag, 2005.

[SB06] Sebastian Bergmann and Günter Kniesel. *GAP: Generic Aspects for PHP*. Third European Workshop on Aspects in Software, August 2006, University of Twente, Enschede, The Netherlands.

[SF04] Sérgio Fernandes and João Cachopo and António Rito-Silva. *Supporting Evolution in Workflow Definition Languages*. In: Proceedings of the 20th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2004), Springer-Verlag, 2004.

[SH05] Saphira Heijens. *Support for Workflow Administration and Monitoring in the YAWL Environment*. Master Thesis, Vrije Universiteit, Amsterdam, The Netherlands, 2005.

[TM04] Tony Marston. *An Activity-Based Workflow Engine for PHP*. September 2004. `http://www.tonymarston.net/php-mysql/workflow.html`

[TS04] Thomas Schmidt. *Erweiterung und Integration des Open Source Workflow-Systems SWAMP am Beispiel eines Software-Wartungs-Prozesses*. Diploma Thesis, Georg-Simon-Ohm-Fachhochschule, Nürnberg, Germany, 2004.

[W3C07] World Wide Web Consortium. *XSL Transformations (XSLT) Version 2.0*. W3C Recommendation, January 2007. `http://www.w3.org/TR/2007/REC-xslt20-20070123/`

[WA96] W. M. P. van der Aalst. *Petri-net-based Workflow Management Software*. In: Proceedings of the NFS Workshop on Workflow and Process Automation in Information Systems, May 1996, Athens, Georgia, USA.

[WA04] W. M. P. van der Aalst and L. Aldred and M. Dumas and A. H. M. ter Hofstede. *Design and Implementation of the YAWL System*. In: Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE 2004), June 2004, Riga, Latvia.

[WfMC95] Workflow Management Coalition. *The Workflow Reference Model*. Document Number WFMC-TC-1003, 1995.

[WfMC99] Workflow Management Coalition. *Terminology and Glossary*. Document Number WFMC-TC-1011, 1999.

[WfMC05] Workflow Management Coalition. *Workflow Process Definition Interface – XML Process Definition Language (XPDL)*. Document Number WFMC-TC-1025, 2005.