fh HAGENBERG SE
Fachhochschul-Studiengänge OÖ
Software Engineering

Fachhochschul-Diplomstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

Design and Development of a Portal-Based Tailoring System with Slicing Techniques

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Ingenieur (Fachhochschule)

Eingereicht von

Gerhard Dietrichsteiner

Betreuer: DDI Dr. Christoph Steindl, IBM Austria, Linz
Begutachter: DI (FH) Peter Kulzcycki

August 2003

Eidesstattliche Erklärung

Ich versichere, dass ich die Diplomarbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, mich auch sonst keiner unerlaubten Hilfen bedient habe und diese Diplomarbeit bisher weder im In- noch Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Pregarten, August 2003 Gerhard Dietrichsteiner

Abstract

In this thesis we develop a portal-based prototype of a tool that is able to tailor a software development process by using slicing techniques. The tool is developed for IBM's Global Services Method, but its design is flexible enough to make it adaptable to any other process model, provided that the model is based on the OMG's Software Process Engineering Metamodel.

Program slicing is a program analysis technique that has been developed and researched for many years. Applying program slicing to a new domain is an interesting idea. An essential part of our work is to find and define analogies between software development processes and computer programs. After the analogies are defined, a program slicing algorithm is adapted and extended to fit our needs.

We develop an algorithm that computes task and work product dependences in a software development process and use this algorithm in our prototype to supply the user with valuable information about process internal dependences, thus supporting and simplifying the user's decision-making when he or she tailors a software development process.

Another part of this thesis is the discussion of our prototype's design; especially the structure which makes the prototype adaptable to other software development processes and performance-related design decisions are of interest.

Zusammenfassung

In dieser Diplomarbeit wird versucht, bewährte Techniken aus dem *Program Slicing* Bereich auf eine neue Domäne, nämlich die der Softwareentwicklungs-Prozessmodelle, anzuwenden. Diese Idee stammt ursprünglich von C. Steindl, der sich ausgiebig mit der Thematik des Program Slicings auseinander gesetzt hat und mich während meines Praktikums bei IBM betreute.

Die Hauptidee besteht darin, dass Softwareentwicklungs-Prozessmodelle und Computerprogramme offensichtlich Gemeinsamkeiten in ihrer Struktur aufweisen. Beide beschreiben Abläufe auf genau definierte Arten und bedienen sich dabei ähnlicher Elemente, woraus die Idee entstand, dass man Program Slicing Algorithmen auch auf Softwareentwicklungs-Prozessmodelle anwenden könnte.

Program Slicing wird dazu verwendet, Computerprogramme (oder Teile davon) zu analysieren. Es kann den Entwickler in unterschiedlichen Phasen der Entwicklung unterstützen, so z.B. beim Debuggen. Dabei wird etwa ein *Slice* berechnet, welches nur die jeweilig für den Entwickler interessanten Teile des Programmes enthält, da wichtige Zusammenhänge innerhalb des Programmes vom Program Slicing Algorithmus analysiert werden und daher nicht relevante Teile des Programmes einfach entfernt werden können.

Auf ein Softwareentwicklungs-Prozessmodell angewandt, könnten Program Slicing Techniken dabei helfen, Zusammenhänge innerhalb des Prozessmodells aufzuzeigen, was wiederum den Projektleiter beim Anpassen eines Prozessmodelles an ein konkretes Projekt unterstützen würde.

Im Laufe der Diplomarbeit werden zuerst Analogien zwischen Prozessmodellen und Computerprogrammen gefunden und festgehalten, danach wird ein Algorithmus aus dem Program Slicing Bereich an die Bedürfnisse des Prozessmodelles angepasst, und schließlich wird ein Prototyp einer Anwendung entwickelt, die diese Techniken verwendet und damit das Anpassen eines Softwareentwicklungs-Prozessmodells ermöglicht.

Als Prozessmodell wird die IBM eigene *Global Services Method* verwendet. Um

den Sinn dieser Arbeit zu erhöhen, wurde allerdings als generelle Ausgangsbasis das *Software Process Engineering Metamodel* (SPEM) der OMG gewählt. Das Design des Prototypen war so zu wählen, dass die Anwendung später ohne großen Aufwand an verschiedene Prozessmodelle angepasst werden kann, vorausgesetzt die Prozessmodelle basieren am SPEM.

Eine weitere Anforderung an den Prototypen war, dass er am *WebSphere Portal Server* laufen sollte. Die Verwendung des Portal Servers brachte einige Annehmlichkeiten mit sich (integrierte Benutzerverwaltung, einheitliches Design der Benutzeroberfläche, strukturierter Aufbau der einzelnen Seiten, usw.), wirkte sich allerdings auch stark auf das Design des Prototypen aus.

Foreword

Personal Motivation

During my internship at IBM I started to work on a tailoring tool for IBM's Global Services Method. It was really hard to get into the Global Services Method, because of its bigness and complexity, but after this step was mastered things turned out to be pretty interesting and fascinating. So I continued to work within this field and did the thesis work you are currently reading.

Scientific Motivation

My advisor at IBM, Christoph Steindl, worked a lot in the area of program slicing. Since he is *Method Exponent* at IBM, he is also very familiar with the Global Services Method, thus the idea of using program slicing on the Global Services Method was his. The main idea was to find similarities—which apparently existed—between software development processes and conventional programs. Applying program slicing to the new domain of software development processes is an interesting idea and it gave us the necessary scientific motivation to give the whole thing a try.

Acknowledgement

First of all I want to express my gratitude to Christoph Steindl, who had the idea of applying program slicing to software development processes, contributed so much insight and experience concerning both the Global Services Method and program slicing, and was always a friendly and helpful colleague.

Furthermore I want to thank (in no particular order):

My supervisor from the Upper Austria University of Applied Sciences in Hagenberg, Peter Kulczycki, for giving me a lot of valuable hints and tips in the area of writing scientific papers and for proofreading the work.

Roland Ossmann, a true fellow, who helped me to weather through those dark and cold winter days, spent a lot of time with me drinking good coffee—at this point I also have to thank IBM for the really great coffee machine—much too early in the morning, and always was ready for some fun.

Mr. Curtis, for proofreading my work and providing me with lots of excellent English-related tips.

My girlfriend Beatrix, for backing me up and for the joy, fun, and love we had. I know that I did not have enough time for you throughout the last months, but I hope that we can catch up on this in the future. And I wish you the best for completing your own studies next year!

My friends in Hagenberg—especially Stefan Ortner—for occasional variety and diversion.

My parents Walter and Gerlinde and my two brothers Walter and Thomas for a few days of relaxation, although there was not much time together in the last months (and years).

Nature, for the pleasures and small miracles it presents me and all the other people every day, and for putting me on earth in this technologically fascinating era—although, of course, there also would have been other centuries worth living in ...

Thank you all.

Contents

1 Introduction **1**

1.1 Intention 1

1.2 State of the Art 2

1.2.1 Tailoring of Process Models 2

1.2.2 Program Slicing 3

1.3 Slicing and Process Models 3

1.4 Preview of Results 3

1.5 Outline 4

2 Background Information **5**

2.1 Software Engineering Process Models 5

2.1.1 Introduction 5

2.1.2 The Software Process Engineering Metamodel 7

2.1.3 The Rational Unified Process 11

2.1.4 The IBM Global Services Method 11

2.2 Program Slicing 12

2.2.1 Basics 13

2.2.2 Control Flow and Data Flow 13

2.2.3 Computation of Reaching Definitions 16

2.2.4 Program Slicing Types 19

2.2.5 Slicing Algorithms 20
2.3 Portal Servers 21
2.3.1 “Portal” 22
2.3.2 Java Standards for Portal Applications 22
2.3.3 Portal Server Basics 23

3 Solution 27
3.1 Detailed Intentions 27
3.1.1 Using Program Slicing 27
3.1.2 A Portal-Based Prototype 28
3.1.3 An Easily Alterable Prototype 29
3.2 Using Slicing Techniques with Process Models 29
3.2.1 Analogies between Programs and Process Models 29
3.2.2 An Adapted Algorithm 31
3.3 The Application 36
3.3.1 Architecture Overview 38
3.3.2 Functionalities 38

4 Implementation of the Prototype 40
4.1 Design 40
4.1.1 First Attempts 40
4.1.2 Background Information 41
4.1.3 Graphical User Interface 41
4.1.4 Database Design of the Tailoring Extension 43
4.1.5 Application Design Overview 45
4.1.6 Application Design Details 49
4.2 Application Test and Usage 57

5 Conclusion **58**

5.1 Intention and Solution 58

5.2 Advantages and Disadvantages 58

5.3 Application Area 59

5.4 Future Work 59

5.5 Availability of this Work 60

Chapter 1

Introduction

This chapter introduces the thesis and provides an overview of it. Section 1.1 explains the thesis' goal. The state of the art in the fields of *tailoring process models* and *program slicing* is briefly described in sec. 1.2 on the next page. Afterwards sec. 1.3 on page 3 introduces the idea of slicing software development processes. Section 1.4 on page 3 gives a short preview of the results that were achieved within this thesis. Finally sec. 1.5 on page 4 shows an outline of the whole work.

1.1 Intention

The objective of this thesis is to develop a *web-based prototype* of a tool that is able to tailor (sec. 1.2.1 on the next page describes this term) a *software development process* by using *program slicing* techniques. The web-based prototype runs on a portal server (see sec. 2.3 on page 21) and can be used to tailor, for example, IBM's Global Services Method (for information on what has been implemented in the prototype see sec. 1.4 on page 3). It should be possible—without too much effort—to adapt the prototype to fit any other development process, provided that the process implements the OMG's Software Process Engineering Metamodel (see sec. 2.1.2 on page 7 for further details). The main aim of the prototype is to simplify the tailoring process by pointing out interrelationships of the software development process, which the user normally hardly can see. To compute these interrelationships, an attempt to apply program slicing techniques (see sec. 2.2 on page 12) to software development processes is made.

1.2 State of the Art

This section describes the current state of the art in the areas of tailoring process models and program slicing.

1.2.1 Tailoring of Process Models

When we use the term "tailoring" in the context of software development processes, we mean the process of adapting process models to the needs and requirements of specific projects. In [Kruchten, 2000], P. Kruchten uses the term "configuring a process" to distinguish between two types: *configuring an organizationwide process* and *configuring a project-specific process*. For the Rational Unified Process (RUP), configuring an organizationwide process means to take the RUP as it is delivered and adapt it according to the organization's practices and needs. This kind of configuration is typically done only once and for all kinds of projects of the whole organization. Configuring a project-specific process means the same as "tailoring" a process: process engineers take the organizationwide process and refine it for a given project; therefore, tailoring needs to be done for every single project. In the RUP, such a project-specific process is described as a *development case*. Another term mentioned in [Kruchten, 2000] is *implementing a process model*, which means the introduction of a process in a company; this must be done before the first process-based project starts and usually requires multiple steps, depending on the company's size.

IBM's Global Services Method (GSM) is an organizationwide software development process with various types of processes (called engagement models) that are designed to fit the requirements of different project types (e.g., an e-business project). Each of these engagement models was designed to be applicable to big projects with a high number of collaborators. In small projects the number of collaborators is much smaller (e.g., only five to eight), thus making the usage of a big process model an unproductive overhead. In order to avoid this, an engagement model is adapted to a given project; the main steps are:

- Choosing the roles which are needed in the project.
- Selecting the work products that will be created.
- Removing unnecessary tasks from the project.

Adapting an engagement model is already a software supported task. Anyway, there was a wish for more support: the tailoring tool should provide useful

context information. Let us take a look at an example: you decide to remove a task from a project. In that case it would be great if the tailoring tool informed you about work products that are included in the project but not produced anywhere any longer. Then you could decide to remove these work products, too, or add the task again. If the tool does not show such dependences, tailoring will be hard. The prototype developed with this thesis uses program slicing techniques to compute such dependences.

1.2.2 Program Slicing

Program slicing is—as its name implies—mainly used to analyze programs, i.e., source code, as desribed in sec. 2.2 on page 12. Anyway, there have also already been attempts to apply slicing techniques to other application areas. An example is the slicing of books and texts[1], where the text can be split into slices which contain only the information that the reader is interested in. Applying program slicing techniques to software development processes is a new idea that was originally born by C. Steindl, who worked a lot in the field of slicing object-oriented programming languages.

1.3 Slicing and Process Models

Program slicing is used to analyze source code; it can produce slices which contain only the information that the programmer is currently interested in. As mentioned in sec. 1.1 on page 1, the aim of this thesis is to apply such program slicing techniques to the area of software development processes.

The main idea is to find analogies between programs and process models. This should be possible, because both programs and process models describe a sequence of steps; in programs the steps are statements and in process models they are tasks. After similarities have been found and defined, program slicing can be used with process models (see sec. 3.2 on page 29) in a similar way as it is used with programs.

1.4 Preview of Results

We defined analogies between programs and software development processes and implemented a portal-based prototype, which uses an adapted program

[1] for more information see: `http://www.slicing-infotech.de/en/index.php`

slicing algorithm for computing slices (see sec. 3.2 on page 29). The slices are used to extract desired information about interrelationships of the process model.

The prototype uses IBM's Global Services Method as a process model and can partially tailor it to given projects. For complete tailoring some functionalities are missing (see sec. 5.4 on page 59). The following goals have been achieved with the prototype:

- A flexible design has been created, which allows the prototype to be adapted to all software development processes that implement the Software Process Engineering Metamodel.
- The prototype is portal-based, thus it possesses the typical advantages of web-based applications (see sec. 3.3 on page 36).
- It provides the process model's content in a structured manner and allows it to be browsed.
- Projects and collaborators can be managed, and tailoring of tasks can be done (this tailoring part uses slicing techniques).

1.5 Outline

Chapter 1 briefly introduces the thesis and provides an overview.

Chapter 2 gives background information about all thesis-relevant topics, as there are: *software process models*, *program slicing*, and *portal servers*.

Chapter 3 first exactly describes the problem, shows how program slicing is applied to software development processes, and finally introduces the prototype.

Chapter 4 deals with the detailed design of the prototype and gives information about how the prototype has been tested and what its weaknesses and strengths are.

Chapter 5 gives a short repetition of the thesis' intention and the developed solution. After this, advantages and disadvantages are discussed, the application area is defined, and ideas for future work are collected.

Chapter 2

Background Information

This chapter deals with topics that provide relatively important background knowledge for this thesis. It is intended to support the reader who might not be familiar with these themes. The first two topics—*Software Engineering Process Models* (sec. 2.1) and *Program Slicing* (sec. 2.2 on page 12)—are really essential because the thesis directly uses them. The third topic—*Portal Servers* (sec. 2.3 on page 21)—is not that essential for the thesis itself; however, it is important for a better understanding of the design and implementation parts.

2.1 Software Engineering Process Models

This section first introduces background knowledge about what software engineering process models are (sec. 2.1.1). Section 2.1.2 on page 7 then explains OMG's *Software Process Engineering Metamodel* (SPEM). Based on the SPEM, two concrete process models are briefly presented afterwards. The first one is the widespread *Rational Unified Process* (sec. 2.1.3 on page 11) and the second one is IBM's *Global Services Method* (sec. 2.1.4 on page 11), which the implementation of this thesis works with.

2.1.1 Introduction

"Different software development projects fail in different ways—and, unfortunately, too many of them fail . . ." [Kruchten, 2000]. One of the biggest problems is that many software development companies do not use any defined process model to develop their software; instead, decisions are often made in an undoc-

umented ad-hoc manner. In general, the major problems with software development are not technical problems, but management problems (cf. [SEI, 1995]).

According to [Kruchten, 2000] the main causes of development problems are:

- Ad hoc requirements management
- Ambiguous and imprecise communication
- Brittle architectures
- Overwhelming complexity
- Undetected inconsistencies in requirements, designs, and implementations
- Insufficient testing
- Subjective assessment of project status
- Failure to attack risk
- Uncontrolled change propagation
- Insufficient automation

The consequences are:

- Inaccurate understanding of end-user needs
- Inability to deal with changing requirements
- Poor software quality
- Team members in each other's way, making it impossible to reconstruct who changed what, when, where, and why
- And so forth ...

Software engineering process models address many of those problems and therefore support the development of software in project teams. Especially in big projects with many collaborators, a good process model is important in order to increase the chance of achieving the desired success. Additionally it is a good idea to use the best practices—these are practices which are commonly used by successful organizations—of software development (if the process model does not already specify them), such as:

1. Develop software iteratively
2. Manage requirements
3. Use component-based architectures
4. Visually model software
5. Continuously verify software quality
6. Control changes to software

A software development process has four roles, namely to [Kruchten, 2000]:

1. Provide guidance as to the order of a team's activities.
2. Specify which artifacts should be developed and when they should be developed.
3. Direct the tasks of individual developers and the team as a whole.
4. Offer criteria for monitoring and measuring the project's products and activities.

2.1.2 The Software Process Engineering Metamodel

The Object Management Group (OMG) released the first version of the Software Process Engineering Metamodel Specification (SPEM) in November 2002. We give an overview of the SPEM in this section, because, first of all, IBM's Global Services Method (GSM) is based on this metamodel and we used the GSM for the implementation of the slicing prototype, and secondly our prototype can be adapted to other process models which are based on the SPEM. If you are interested in more detailed information about the SPEM, we refer you to the specification [OMG, 2002].

The SPEM defines a template for software development processes. It is located at level M2 of OMG's four-layered architecture of modeling (see fig. 2.1 on the next page). Concrete process models—like IBM's GSM, the Rational Unified Process, DMR Macroscope, and Fujitsu SDEM—are on level M1, which means that they use the metamodel defined at level M2. If one of the process models of level M1 is used in a real-world project, this happens at level M0.

The SPEM is formally defined as an extension of a subset of the Unified Modeling Language (UML). The UML is defined by a metamodel, which is itself

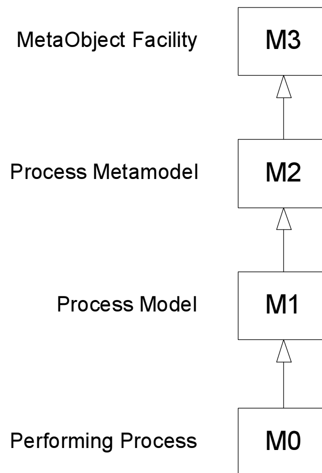

Figure 2.1: Four-layered architecture of modeling as defined by the OMG.

defined as an instance of the MetaObject Facility (MOF) metametamodel. This MOF metametamodel is located at level M3 in the architecture of modeling (see fig. 2.1).

The purpose of the SPEM is to support the definition of software development processes. It solely addresses the domain of process description and—as a metamodel—does not contain any content itself.

Conceptual Model

[OMG, 2002] describes the main idea of the Software Process Engineering Metamodel as follows:

> At the core of the Software Process Engineering Metamodel (SPEM) is the idea that a software development process is a collaboration between abstract active entities called *process roles* that perform operations called *activities* on concrete, tangible entities called *work products*.

Figure 2.2 on the next page shows the conceptual model and the coaction of the above-mentioned terms.

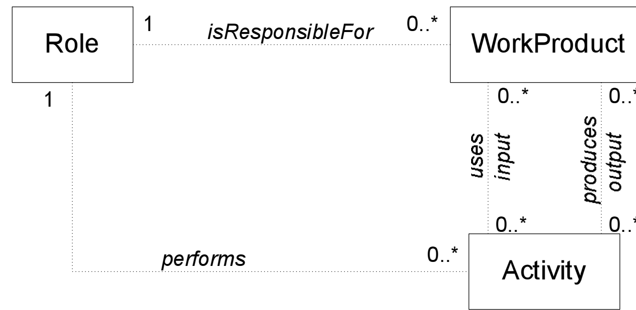

Figure 2.2: SPEM conceptual model: Roles, Work Products, and Activities.

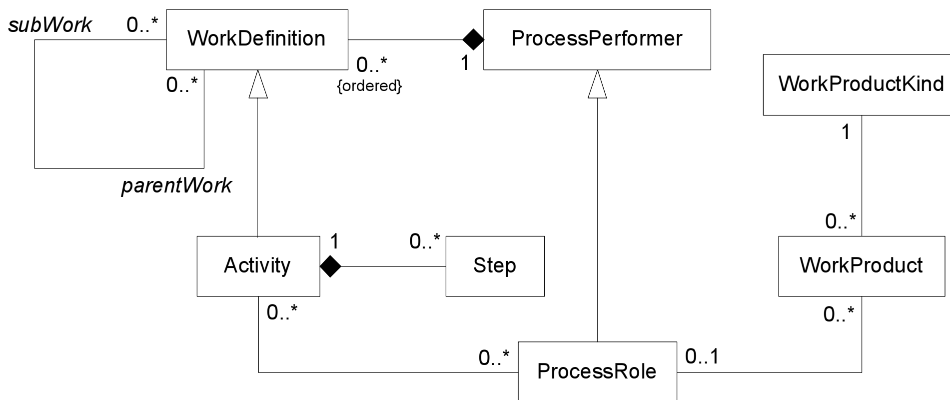

Figure 2.3: Marginally simplified version of the SPEM process structure shown in [OMG, 2002, sec. 7].

Process Structure

A simplified figure of the process structure defined in [OMG, 2002] is shown in fig. 2.3. Let us take a look at the figure's terms in order to understand this process structure:

WorkProduct and WorkProductKind *Work products* are anything that is produced, consumed, or modified by a process. This can be different kinds of documents, source code, models, and so on. Work products are assigned to *work product kinds* which describe various types of work products, such as a Text Document or an UML Model.

WorkDefinition *Work definitions* are pieces of work which the progression of the process can be described with. Therefore work definitions form a hierarchical structure by using some kind of recursive composition, which means that work definitions consist of other work definitions and so on (see the *subWork* - *parentWork* relation in fig. 2.3 on the page before). The development process can define multiple levels of work definitions. On the lowest level are activities, which represent the smallest pieces of work. On higher levels there are work definitions like phases, etc.—the names of these work definition types are not defined within the SPEM, resulting in differing names in SPEM implementations; also the number of levels is not defined and can vary. Furthermore, work definitions are related to their used work products through activity parameters[1]. These activity parameters define whether the work products are used as input or output.

Activity and Step *Activity* is the main subclass of *WorkDefinition*. Activities can be found on the lowest level of the hierarchical process structure. They describe pieces of work which can be performed by one *ProcessRole*. Although activities are located on the lowest level of the hierarchical process structure, they may themselves consist of even smaller, atomic elements called *steps*.

ProcessPerformer and ProcessRole The *ProcessPerformer* class defines a performer for work definitions that do not have a more specific owner. *ProcessRole* is a subclass of ProcessPerformer and defines specialized roles (e.g., Analyst, Technical Writer, etc.), that are responsible for specific work products, and that perform and assist in specific activities.

Terminology

SPEM implementations (IBM's GSM, and so on) do not necessarily have to use the same terminology as the SPEM does, but if the terminology differs, a correspondence list must be provided. In the case of IBM's Global Service Method, the terminology also partially differs a bit. Table 2.1 on the next page shows a correspondence list for IBM's GSM and the Rational Unified Process.

[1]this attribution is not shown in fig. 2.3 on the preceding page, because the corresponding figure in [OMG, 2002] also does not show this connection.

SPEM	RUP	GSM
ProcessRole	Role	Role
Activity, Step	Activity, Step	Task
WorkProduct, Information-Element	Artifact	Work Product Description
Discipline	Discipline	Domain
Lifecycle	Process	Engagement Model
Phase	Phase	Phase
Iteration	Iteration	Iteration
Guidance	Guidelines, ToolMentors, Templates	Technique

Table 2.1: Translation table for terms of the Software Process Engineering Metamodel (SPEM), IBM's Global Services Method (GSM), and Rational Unified Process (RUP).

2.1.3 The Rational Unified Process

We want to mention the Rational Unified Process (RUP) here, because it is a famous implementation of the SPEM. In [Kruchten, 2000] it says:

> The Rational Unified Process is a software engineering process. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end users within a predictable schedule and budget.

The RUP is available in a web-based form, which can easily be used in intranets. It is tailorable and fits the requirements of various kinds of projects. For further information on the Rational Unified Process we refer to [Kruchten, 2000].

2.1.4 The IBM Global Services Method

We used IBM's Global Services Method (GSM) for the implementation of this thesis work. As the Rational Unified Process, the GSM is also an implementation of the SPEM. We give an introduction to the basic structure and terms of the GSM in this section.

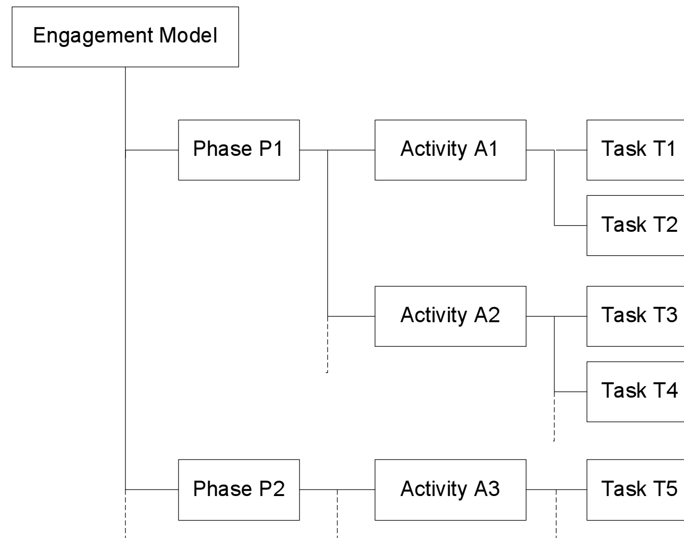

Figure 2.4: Structure of an engagement model of IBM's Global Services Method.

Unlike the RUP, the GSM provides a broad range of different processes, called *engagement models*. Each of these engagement models focuses on different kinds of projects (e.g., e-business projects). Figure 2.4 shows how an engagement model is structured into *phases*, *activities*, and *tasks*. Mind that the GSM's tasks are confusingly called activities in the SPEM.

Tasks have *perform roles* and *assist roles* assigned, take work products as input, and produce work products as output. Collaborators of a project are assigned to one or more roles and the roles specify which tasks each project member has to fulfill.

Implementation relevant details on the GSM are introduced in chapter 4 on page 40.

2.2 Program Slicing

In this section we give a short overview of what *Program Slicing* is (sec. 2.2.1 on the following page) and how slices can be computed (from sec. 2.2.2 on the next page to sec. 2.2.5 on page 20). Many of the explanations follow what is written in [Steindl, 2000], especially because [Steindl, 2000] includes a broad introduction to program slicing based on various papers and articles.

2.2.1 Basics

In [Steindl, 2000, sec. 1.1] a short but meaningful introduction to program slicing is given:

> Program slicing is a program analysis and reverse engineering technique that reduces a program to those statements that are relevant for a particular computation. Informally, a slice provides the answer to the question "What program statements potentially affect the value of variable `v` at statement `s`?"

Mark Weiser introduced program slicing, because he found out that programmers always had some abstractions in mind when they were debugging (cf. to [Weiser, 1984]). Programmers usually search for parts of the program resulting in an erroneous statement `s`, by following dependences from `s` backwards. The statements found in this way influence `s` either because they decide if `s` is executed at all (*control dependence*) or because they define a variable which is used by `s` (*data dependence*). Especially in modern programming languages, which support advanced concepts as object oriented programming, extracting these parts from a program just by taking a look at the source code can be very hard.

This leads us to why one of the most popular application areas of program slicers is supporting programmers during the debugging process: program slicing perfectly supports humans in debugging programs, because it has the ability to reduce the complexity of the involved source code. If the programmer knows that there is something wrong with variable `v` at a specific location in the source code, he or she can use a program slicer to compute a slice with variable `v` as starting point. This results in a shortened source code part that is still complete in the context of variable `v`, which makes it much easier for the programmer to find potential bugs.

Other application areas, where program slicing can be used to assist the programmer, are program integration, software maintenance, testing, and software quality assurance.

2.2.2 Control Flow and Data Flow

A slice of a program can be influenced by two kinds of dependences: *control dependence* and *data dependence*. Most program slicers need to analyze both

```
Node current = head;
int counter = 0;

while(current != null) {
  if(current.isActive())
    counter++;
  current = current.getNext();
} // while
```

Figure 2.5: A simple piece of source code.

control and data dependences to compute useful slices. We describe control flow and data flow in a compact form; if you need further information, see [Steindl, 2000] and the sources that are mentioned there.

Control Flow

High-level programming languages use control structures (such as `if`, `while`, and `return`) to define the flow of control within a program. For analyzing control dependences, program slicers use *control flow graphs*, which have to be computed first. Control flow graphs consist of *nodes* which are connected by *directed edges*. Nodes are basic blocks, which means that there is no choice of different flow possibilities. Basic blocks contain sequences of statements that are executed completely or not at all. The directed edges define the flow of the program: e.g., there is an edge from node `A` to node `B` if control can flow from block `A` to block `B`. Additionally, edges can be labeled T or F depending on whether their origin node (e.g., an `if` statement) evaluates to true or false. A control flow graph always starts at a node labeled *START* and ends with a block labeled *STOP*. Each block in between has to be reachable from *START* and must contain a path to *STOP*. Figure 2.6 on the next page shows the control flow graph of the short and simple piece of code of fig. 2.5.

Data Flow

Most variables of a program change their values several times during runtime. Data flow describes where the value of a variable may “flow” to, i.e., which parts of the program can be affected by the variable’s definition. *Data dependence* is

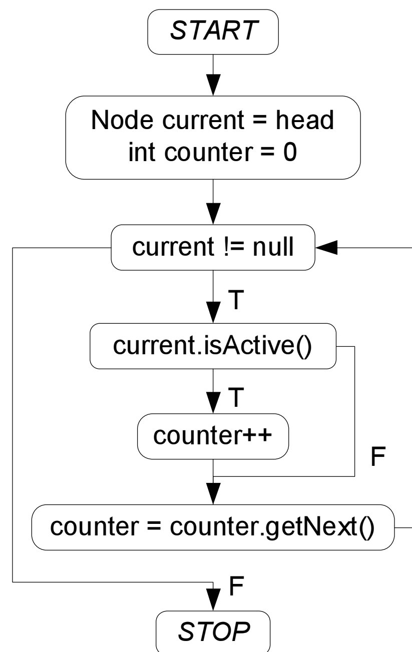

Figure 2.6: Control flow graph of the source code of fig. 2.5 on page 14.

defined as follows: node `U` is data dependent on node `D` if the following conditions are true (cf. [Steindl, 2000, sec. 2.3]):

1. Node `D` defines variable `x`.
2. Node `U` uses `x`.
3. There is no possibility of an intervening definition of `x` on all paths between node `D` and node `U`.

Node `D` is a *reaching definition* for node `U` if node `U` is data dependent on node `D`. Let us take a look at an example to illustrate data flow and data dependences; fig. 2.7 on the following page shows a short function which calculates the greatest common divisor of two numbers. The data dependence graph of this function can be viewed in fig. 2.8 on page 17 (this function was taken from [Steindl, 2000][2]).

[2] [Steindl, 2000] uses a different notation of the function, but it is based on the same algorithm.

```
int gcd(int u, int v) {
  int t;

  do {
    if(u < v) {
      t = u;
      u = v;
      v = t;
    } // if
    u = u % v;
  } while(u != 0);

  return v;
} // gcd
```

Figure 2.7: A function which computes the greatest common divisor of two numbers.

2.2.3 Computation of Reaching Definitions

This section takes a look at how reaching definitions can be computed. We only show the very basics here, complete program slicing has to face advanced problems such as arrays, record fields, etc. (for further details see [Steindl, 2000]).

Before starting to compute reaching definitions we need to know *used variables* and *defined variables* of each statement in the program. A variable is used in a statement if its value is used. For example the statement `a = b + c` uses the variables `b` and `c` and defines variable `a`. There are two possibilities for a variable to be defined: the first one unambiguously assigns a new value to the variable, which is also called *killing definition* because the variable is assigned a new value for sure. The second one is an ambiguous assignment, which means that it is not sure if the variable gets a new value or not. This kind of definition is called *non-killing definition*. An example would be a procedure call with a call-by-reference parameter, where it is not sure if the procedure assigns a new value to the variable or not.

At this point we have to introduce some new terms again. First of all, each statement of a program is assigned a label to clearly identify it. Then we can define the following terms:

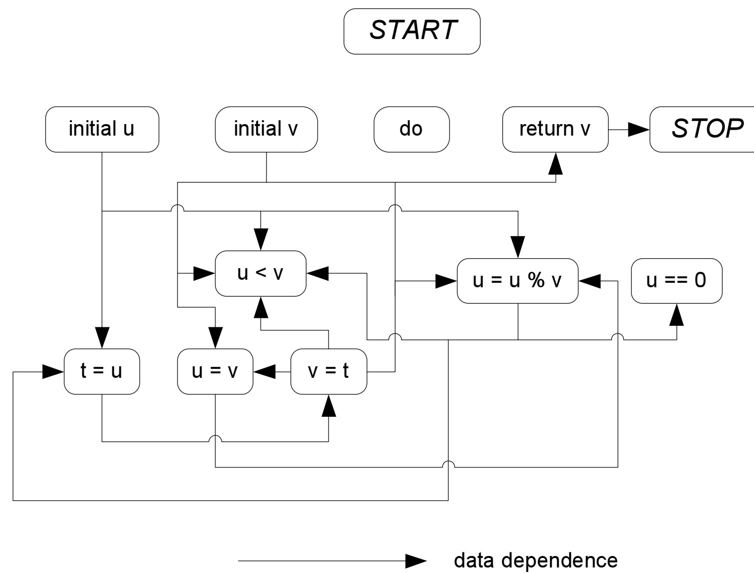

Figure 2.8: Data dependence graph of the source code of fig. 2.7 on the page before.

Definition set Each variable x has a definition set which contains the labels of all statements that define x (both killing and non-killing definitions).

Gen set and kill set Each statement S has a gen set and a kill set. The gen set contains the labels of all definitions that are generated by S. The kill set contains the labels of all definitions that are killed by S.

In set and out set Each statement S has an in set and an out set. The in set contains the labels of all definitions that reach S. The out set contains the labels of all definitions that leave S.

The algorithm for the computation of reaching definitions consists of two iterations:

1. In the first iteration the definition set of each variable and the gen and kill sets of each statement are computed.
2. The second iteration computes the reaching definitions in a syntax directed manner and inserts links from the usage nodes of variables to all its reaching definitions.

Computing the gen and kill sets and the reaching definitions requires us to solve the data flow equations for all statements of the program (we only take a look

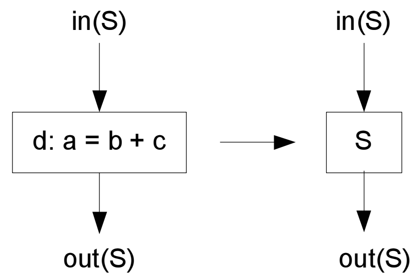

Figure 2.9: Scheme of a killing assignment.

at the data flow equations that are relevant to this thesis; for more information we refer to [Steindl, 2000]).

Data Flow Equations for Assignments

An assignment to a variable generates a definition. If the assignment is ambiguous, it is a non-killing definition (empty kill set). Otherwise it is a killing definition (non-empty kill set). Figure 2.9 illustrates a killing assignment which results in the following data flow equations:

- gen(`S`) = {`d`}
- kill(`S`) = DefinitionSet(`a`) - {`d`}
- out(`S`) = gen(`S`) ∪ (in(`S`) - kill(`S`))

Each assignment is given a label; in case of fig. 2.9 the assignment got the label `d`. The gen set contains only this label `d` because the statement generates the definition for variable `a` only. All previous definitions of `a` are killed, thus the kill set of the statement consists of `a`'s definition set minus the current definition `d` (which is—of course—not killed). Finally the out set contains all definitions that are generated by `S` (i.e., gen(`S`)) plus all definitions that reach `S` (i.e., in(`S`)) and are not killed by `S` (not in kill(`S`)).

Data Flow Equations for Statement Sequences

If two or more statements are executed in a sequence, their effects can be combined. Figure 2.10 on the following page shows the scheme of a sequence of two statements. The corresponding data flow equations are:

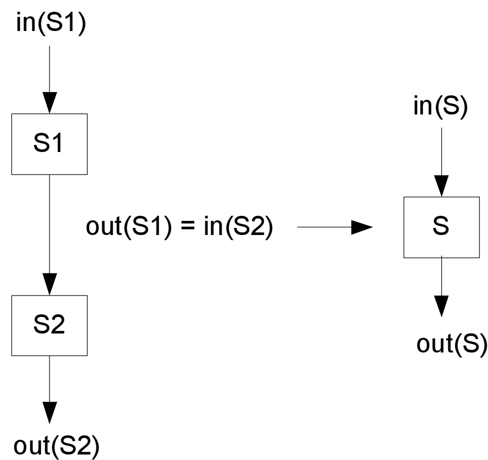

Figure 2.10: Scheme of a sequence of two statements.

- gen(`S`) = gen(`S2`) ∪ (gen(`S1`) - kill(`S2`))
- kill(`S`) = kill(`S2`) ∪ (kill(`S1`) - gen(`S2`))
- in(`S1`) = in(`S`)
- in(`S2`) = out(`S1`)
- out(`S`) = out(`S2`)

Statement `S` generates everything that `S2` generates (i.e., gen(`S2`))—because there is no chance that those definitions are killed—plus everything of what `S1` generates and which is not killed by `S2` (i.e., gen(`S1`) - kill(`S2`)). The statement's kill set is computed in a similar way: it contains everything that `S2` kills (i.e., kill(`S2`)) plus the killing definitions of `S1` which are not defined by `S2` again (i.e., kill(`S1`) - gen(`S2`)).

2.2.4 Program Slicing Types

Program slicing can be divided into different types: *static slicing* versus *dynamic slicing*, *backward slicing* versus *forward slicing*, and *intraprocedural slicing* versus *interprocedural slicing*. This section briefly describes the differences between these types.

Static Slicing and Dynamic Slicing

Static slicing analyzes the program's source code without knowing anything about a particular runtime state. So, variables do not have concrete values. Thus, since static slices have to take all possibilities of program flow into consideration, the generated slices can be quite large. In contrast to static slicing, dynamic slicing knows the values of variables in the context of a particular execution of the program. That is why dynamic slicing produces smaller slices and is more precise than static slicing.

Backward Slicing and Forward Slicing

These two types have pretty self-explanatory names: backward slices contain all parts of the program that might have influenced the variable at the selected statement. Forward slices contain all parts of the program that might be influenced by the variable.

Intraprocedural Slicing and Interprocedural Slicing

Intraprocedural slicing concentrates on one single procedure and does not take calls of other procedures into account. Interprocedural slicing analyzes the interactions between procedures (a procedure can be called multiple times from different places in the program, etc.).

2.2.5 Slicing Algorithms

Slicing algorithms evolved over time; early slicing algorithms saw slicing as a data flow problem, later slicing was seen as a graph-reachability problem. We give a short introduction to a data-flow-based algorithm here (take a look at [Steindl, 2000] if you need more detailed information). Chapter 3 on page 27 shows how such an algorithm can be used with process models.

Slicing as a Data Flow Problem

This is the initial method which M. Weiser used in the early days of program slicing. He took a control flow graph as intermediate representation for his slicing algorithm. At each node the data flow information of all *relevant variables* had to be computed, which made it possible to extract a correct slice. The

algorithm for computing the sets of relevant variables for the slice of node `N` and variables `V` worked as follows (cf. to [Steindl, 2000, sec. 3.1]):

1. Initialize the relevant sets of all nodes to the empty set.
2. Insert all variables of `V` into *relevant(N)*.
3. For `N`'s immediate predecessor `M`, compute *relevant(M)* as:
   a) relevant(`M`) = relevant(`N`) - def(`M`)
   b) if relevant(`N`) $\cap$ def(`M`) $\neq \{\}$ then
   c) relevant(`M`) = relevant(`M`) $\cup$ ref(`M`)
   d) include `M` into the slice
   e) end
4. Work backwards in the control flow graph, repeating step 3 for `M`'s immediate predecessors until the entry node is reached or the relevant set is empty.

Explanation of step 3:

a) Exclude all variables that are defined at `M`.

b) If `M` defines a variable that is relevant at `N`.

c) Include the variables that are referenced at `M`.

See table 2.2 on page 25 for an example of how this algorithm works. All statements that are part of the computed slice have bold and italic node numbers.

For structured programs—with conditions, loops, etc.—the algorithm must be extended. We do not describe these extensions here because they are not used in the current implementation (see chap. 3 on page 27).

2.3 Portal Servers

Using a *portal server* as a platform for the application has been a given fact since the very beginning of our work. The tailoring tool was supposed to be a web application based on the *IBM WebSphere Portal Server*. In this section we will take a look at the term "portal" (see sec. 2.3.1 on the following page), possible standards and current implementations (see sec. 2.3.2 on the next page), and the very basics behind implementing applications for a portal server (see sec. 2.3.3 on page 23).

2.3.1 "Portal"

The word "portal" is one of those "overloaded" words that can mean anything and nothing at the same time. In reality there are a lot of different portals with basic similarities. The term *portal* has evolved over time resulting in more or less slightly different meanings. The first portals were simple search engines or collections of links. Both types offered their visitors one single place for entering the Internet and finding information they were looking for.

Today there are different types of portals with different functions; examples are *Personal Portals*, *B2B Portals*, *B2C Portals*, *Enterprise Information Portals*, and so forth. Like the early portals, they also try to form some kind of central point at which users find as much desired information as possible. Users can customize the contents of such portals according to their own priorities and needs. And the portals customize available content depending on previous user actions and behavior; this is called "personalization."

Developing extensive portals means a lot of work and headaches. To prevent this and to make common tasks easier, so-called *portal servers* were invented. *IBM WebSphere Portal Server*, for example, is one of the Java-based portal servers; the following sections (see sec. 2.3.2 and sec. 2.3.3 on the following page) deal with such servers. Of course, there are also other—non-Java-based—portal server systems available, like the *Microsoft SharePoint Portal Server*; however, they have no relevancy to this thesis.

2.3.2 Java Standards for Portal Applications

Currently there are quite a few Java-based portal servers on the market (from IBM, BEA Systems, Oracle, the Apache Software Foundation, and others). Most of these servers provide comparable functionality although there is no standard for portals. Java-based portals are based on a J2EE compliant application server. They are constructed with *portlets*, which are much like servlets (see sec. 2.3.3 on the next page for further details).

Portlets are not part of the standard Java platform; however, if things work out they will become an official extension to the J2EE 1.4 platform soon. Let us take a look at the history of this standardization process. The origin of the Portlet API is located at the Apache Software Foundation (cf. [JCP, 2003a, sec. 2.5]). They invented the first versions and built the public available Jakarta JetSpeed portal server [Apache, 2003]. Later IBM submitted a Java Specification Request [JCP, 2003a] for standardizing the "Portlet API" and a little bit later

Sun Microsystems also submitted a similar request for a specification called "Java Portlet Specification" [JCP, 2003b]. Both requests must have been issued in late 2001[3]. In January 2002 they were withdrawn and replaced by a new Java Specification Request [JCP, 2003c], "Portlet Specification,". IBM and Sun Microsystems together took on the leadership role of this request, which is supported by a broad expert group of 19 well-known companies such as BEA Systems, Borland Software Corporation, SAP AG, Oracle, and others. Since the community review process was finished recently, the first version (1.0) of the Portlet Specification has been released in late August 2003. It was not possible to take this specification into consideration within this thesis, because of its recent release.

It is hoped that all of the portal server providers will implement the Portlet Specification in the next versions of their portal servers in order to become specification compatible. One and the same portlet will then work on many servers without needing to be modified.

2.3.3 Portal Server Basics

In this section we want to show what *portlets* are and how they are used to build portal applications for the IBM WebSphere Portal Server. Although there is currently no official specification for portlets, as shown in sec. 2.3.2 on the preceding page, most of the concepts which we are going to show here should be valid for all portal servers. For further information on these concepts, see IBM's *Portlet Development Guide* [Buckner et al., 2003].

According to the latest specification request [JCP, 2003c, sec. 2.1], "Portlets are web components—like Servlets—specifically designed to be aggregated in the context of a composite page. Usually, many Portlets are invoked in the single request of a Portal page." Apart from minor differences in the environment, one main difference between portlets and servlets is that portlets generate only a fragment of a page and not whole pages as servlets do. That means that the programmer has to be careful not to use page-level tags like `<html>` in a portlet's output. Such portlet output would result in wrecked pages on the client's screen.

The first two basic features of portlets that we want to introduce here are *portlet states* and *portlet modes*. Within a webpage, portlets are often rendered in the way typical windows are in Microsoft Windows operating systems. Like a window, they can be in three different states: normal, maximized, and minimized.

[3] Unfortunately it was not possible to find out the exact date when the requests were made.

For example, if a user maximizes a portlet, the page will be reloaded and there will be only the maximized portlet filling up the whole page.

A portlet can operate in three possible modes. The normal mode is the *view mode*, where the portlet shows the information that it is supposed to show. In *edit mode* the user can set user specific, persistent settings for the portlet, e.g., how much information he or she wants the portlet to show. Finally, the user can switch the portlet to *help mode* in order to get some hints on how to use the portlet. The only mode that a portlet has to support is the *view mode*. All other modes can be implemented optionally.

A portal's content is hierarchically structured by the use of "page-groups." Page-groups consist of page-groups or pages. Pages finally contain the portlets that are used for creating the pages' content. Portlets that are located on the same page can communicate with each other by sending simple messages. One message can be received by one or more other portlets (see fig. 2.11 on the next page).

Figure 2.12 on page 26 shows different variations of a portlet during its life-cycle. We decided to show a more general life-cycle than the one described in [Buckner et al., 2003][4]. At first a *portlet class file* is generated by a Java compiler. At runtime the class file is deployed to a portal server. If necessary, the portal server creates a *portlet instance*. When the portlet is placed on a page a *portlet window* is created. The portlet window can be adjusted to its page by page-specific settings. Portlets can store user-specific data, so that the user is able to customize it to his or her personal needs. To accomplish this, the portlet window is parameterized by a persistent object for each user. Additionally, for each login of a user, a portlet session object is created. This session object stores transient data for the portlet window.

[4] [Buckner et al., 2003] describes a more complicated, proprietary life-cycle model that will presumably not be part of the portlet specification.

n	statement	ref(n)	def(n)	relevant(n)
1	`b = 1`		b	
2	`c = 2`		c	b
3	`d = 3`		d	b, c
4	`a = d`	d	a	b, c
5	`d = b + d`	b, d	d	b, c
6	`b = b + 1`	b	b	b, c
7	`a = b + c`	b, c	a	b, c
8	`print(a)`	a		a

Table 2.2: Computation of a slice for statement 8 (node `N` = 8) and variable `a` (variables `V` = {`a`}) plus computation of the relevant sets.

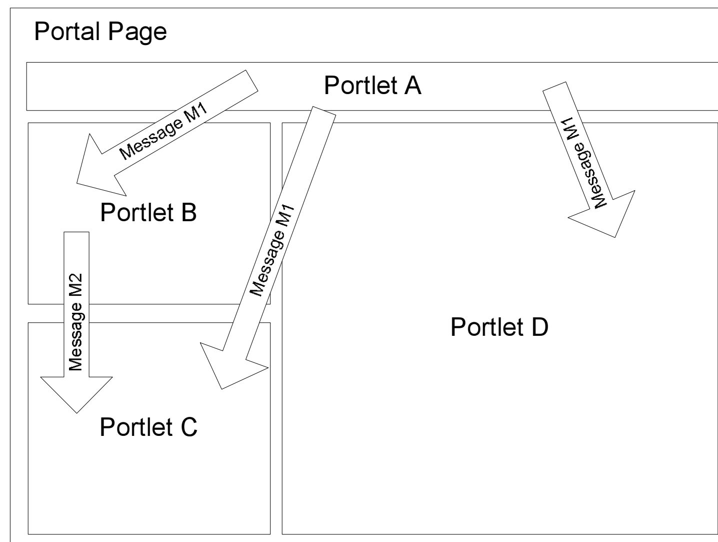

Figure 2.11: A portal page with portlets. The portlets communicate with each other by sending messages. In the example, portlet A sends out a message M1 which is received by the portlets B, C, and D. Another message M2 is sent from portlet B and received by portlet C only.

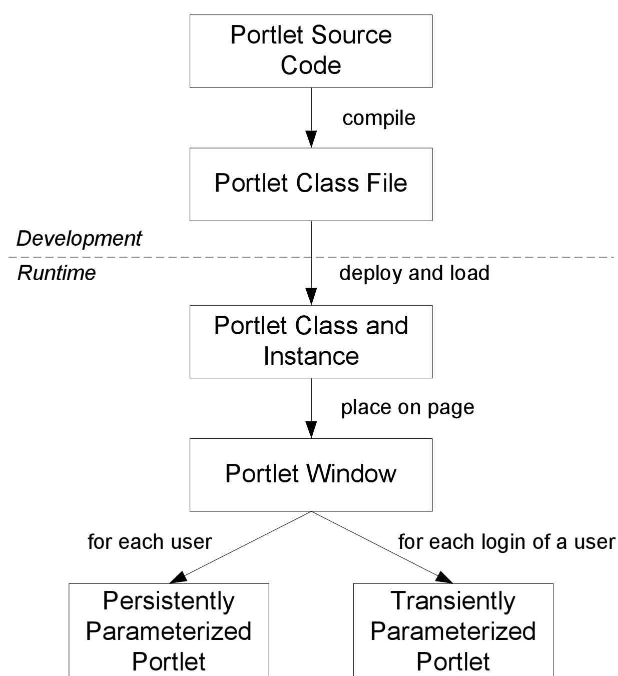

Figure 2.12: The life-cycle of a portlet.

Chapter 3

Solution

In this chapter we discuss our solution. First the intentions of our work are detailed (sec. 3.1), then we describe how program slicing has been applied to a Software Process Engineering Metamodel based software development process (sec. 3.2 on page 29), and finally we introduce the application that resulted from our work (sec. 3.3 on page 36). Chapter 4 on page 40 takes a look at implementation details of the application.

3.1 Detailed Intentions

According to sec. 1.1 on page 1 the main goal is to create a web-based prototype for tailoring software development processes. The most important requirements on the prototype are:

- The prototype should use program slicing techniques for computing slices.
- The application should be portal-based (which implies that it has to be web-based).
- The application should be easily alterable, so that it can be used with various process models.

In the following we go through these requirements and explain them.

3.1.1 Using Program Slicing

The prototype is supposed to provide usefull information about interrelationships within a software development process, supporting the user in his or her

decision-making. The interrelationships of the kind “task `A` depends on task `B` because `B` produces a work product which `A` requires as input” have to be computed on the fly, because they change depending on the current tailoring state of the process model (e.g., task `B` could be removed).

For computing these interrelationships and dependences, the prototype uses program slicing techniques. Program slicing was originally used for analyzing pieces of source code (sec. 2.2 on page 12 gives more details on program slicing). C. Steindl had the idea of applying program slicing techniques to software development processes, because he saw that there were some analogies between process models and programs. In sec. 3.2 on the next page we describe which analogies we have found and how program slicing has been applied to process models.

3.1.2 A Portal-Based Prototype

My employer wanted the prototype to be portal-based; IBM’s WebSphere Portal Server is used as a platform for it. In fact, applying some program slicing techniques to a piece of a process model would not have been such a great challenge. The aspect of implementing the whole prototype as a web-based application strongly influenced the prototype’s design and layout (the design is discussed in detail in sec. 4.1 on page 40).

The reasons for choosing a portal server as a platform are the usual advantages of web applications plus the advantages a portal server brings about. Typical advantages of web applications are:

- usage of thin clients (does not need to install anything on the clients since a web browser can be assumed to be standard)
- can be easily used in intranets or the Internet
- is easily accessible from anywhere
- application logic is completely located on a central server (in our case: WebSphere)

In addition to that, the usage of a portal server brings further advantages:

- built in user management
- persistent storage of preferences for each user

- consistent GUI (which additionally can be changed very easy without needing to modify any code)
- usage of portlets results in a modular structure of the GUI and the logic that lies behind

Of course web applications do not only provide advantages. Disadvantages and how they affected the prototype's design are also discussed in sec. 4.1 on page 40.

3.1.3 An Easily Alterable Prototype

Another challenge was to create an alterable application design, so that the prototype can be adapted to other software development processes without too much effort. Developing an appropriate application design was one of the most demanding tasks of the whole work. In sec. 4.1 on page 40 we take a look on the final design and we also describe how the design evolved over time.

The problem with different software development processes is that, although they are based on the same metamodel, they are available in different forms. The protoype has to be alterable in such a way that these different forms can be supported, thus making it process independent. Information on how this problem has been handled can als be found in sec. 4.1.

3.2 Using Slicing Techniques with Process Models

In this section we want to show how program slicing techniques can be applied to software development processes. First we define analogies in sec. 3.2.1 and then we describe how a program slicing algorithm can be used with a process model in sec. 3.2.2 on page 31.

3.2.1 Analogies between Programs and Process Models

Software development processes and computer programs obviously share some common purposes. Both describe a strict sequence of actions that have to be performed in a defined order.

Computer programs basically consist of control-related and data-related instructions. Control-Related instructions are everything that defines the flow

of a program; these can be loops, conditional statements, calls of self-defined functions, and so on. Data-related is everything that deals with the storage of values; these are, e.g., declarations and definitions of variables, assignments to variables, and so forth.

Software development processes have a similar structure, although they are much simpler. They describe the actions that have to be performed to achieve a defined goal, which in most cases would be the completion of a successful project. The actions are hierarchically structured (see sec. 2.1 on page 5 for more information on process models). Concrete tasks—these are the smallest units of actions—produce outcomes in the form of work products; other tasks require these work products as input.

In contrast to programs, software development processes based on the Software Process Engineering Metamodel (SPEM, see sec. 2.1.2 on page 7) lack a lot of control-related parts. There are no conditional points in a process model, where the process could proceed in different directions depending on some criteria. And there are no loops which would repeat one and the same sequence of actions multiple times. In fact, the SPEM uses and defines *iterations*, but these iterations are nothing that could be used in terms of a program's loops. The SPEM mentions an iteration as "composite WorkDefinition with a minor milestone" [OMG, 2002]. This means that a process model may contain similar sequences of actions with different goals. The SPEM specification provides an example for such iterations from the DMR Macroscope process model, where the phase "Preliminary Analysis" contains the iterations "First Joint Requirements Planning (JRP) Workshop" and "Second Joint Requirements (JRP) Workshop". Both iterations contain many common activities and steps, but anyway, they differ slightly and are specified separately. This way the typical advantages of loops—as we know them from programs—get lost.

The only control-related part that process models contain, is their definition of the process flow. They define phases, activities, and so on. The smallest pieces of process flow are activities (SPEM notation) or tasks (GSM notation, which we use throughout this section). Tasks are very similar to a program's functions, because

- they do a small and manageable piece of work,
- they often need some input so that they can do their work,
- and in many cases they produce some output.

Data-related parts of a program are everything that has something to do with variables. Variables store different kinds of information, provide the ability to modify their content, and transport information throughout the whole program. Software development processes possess something with similar characteristics, namely work products. As described in sec. 2.1 on page 5, work products can be documents, source code, diagrams, and so on. These work products also store some kind of information, transport it through the whole process, and provide the ability of their content to be changed.

This led us to the following analogies between programs and software development processes:

- Both describe a strict sequence of actions
- Tasks are similar to functions
- Work products are similar to variables

With this analogies defined and the restrictions of process models in mind, we could say that a software development process is a very simple straightforward program without branching, loops, recursions, or any analogical constructs.

In a software development process, work products appear in two different kinds, as there are input work products and output work products. Input work products are the input of a task. This means that the task needs the work products in order to be performed, resulting in input work products being like input parameters of a program's functions. Output work products contain the result of what the task does. Therefore, output work products are like a function's output parameters or return value. Sometimes a work product appears as input and output of one task. This means that the task presumably modifies the content of the work product and can be compared to a function with pass-by-reference parameters.

3.2.2 An Adapted Algorithm

Modern program slicing deals with challenges of today's programming languages. As described in sec. 3.2.1 on page 29, software development processes are not that complex. Therefore, we decided to take the data-flow-based algorithm described in sec. 2.2.5 on page 20 as starting basis. This algorithm uses a control flow graph of the program for computing slices. In our case the process model can be seen as simple program, thus we need not compute a control flow

Task T1	↓WP A, ↓WP B, ↑WP A, ↑WP C
Task T2	↓WP A, ↓WP C, ↑WP C
Task T3	↓WP C, ↑WP X, ↑WP F
Task T4	↓WP F, ↑WP A
Task T5	↓WP C
Task T6	↓WP C, ↑WP Y
Task T7	↓WP F, ↑WP F, ↑WP X
Task T8	↓WP Y, ↑WP D, ↑WP E
Task T9	↓WP E, ↑WP G
Task T10	↓WP C, ↑WP K

Table 3.1: This is a simplified part of a process model. The tasks T1–T10 appear in the given order. Work products (WP) are marked as input (↓) or output (↑).

graph because we already know the control flow of the process. It starts with the first task and works its way right through the process until the last task is reached.

Anyway, the computation of one slice can result in different slices because the process model may be in various tailoring states. The removal and adding of tasks and work products influence the computed slices.

With the algorithm described in sec. 2.2.5 on page 20 we can compute backward slices for a given task and work product in a very simple way. The backward slice contains all tasks that produce the work products which can influence the selected task. Additionaly we are interested in a forward slice, which shows us all the tasks that can be influenced by the selected task. This goal can be achieved with a similar computation. And finally we do not only want to receive a simple slice from our algorithm, we also want to know detailed dependences between work products. Let us take a part of a simplified software development process as example (see table 3.1) and develop the algorithm step by step.

First we use the base algorithm to compute a backward slice for task T5 and work product C. The algorithm does not need to be changed for this task. We simply changed the "program terms" to "process terms". The algorithm has the following four steps (assumed that we want to compute a slice for task T and work product P):

1. Initialize the relevant sets of all tasks to the empty set.
2. Insert work product P into the relevant set of task T.

3. For T's immediate predecessor S, compute the relevant set as:
   a) relevant(S) = relevant(T) - def(S)
   b) if relevant(T) $\cap$ def(S) $\neq$ {} then
   c)   relevant(S) = relevant(S) $\cup$ ref(S)
   d)   include S into slice
   e) end
4. Work backwards in the process, repeating step 3 for S's immediate predecessors until the first task is reached or the relevant set is empty.

Table 3.2 on the following page shows the result of computing the backward slice for task T5 and work product C. First of all—in terms of program slicing—the *defined* and *referenced sets* have to be computed. In fact, we do not need to compute anything here, because we know this information directly from the process model. The output work products are the work products a task defines and the input work products are the work products a task references. The next step is to insert work product C into the relevant set of task T5. Then we compute the relevant set for T5's predecessor T4. The relevant work products of T4 are the relevant work products of T5 (=C) without the work products T4 defines. T4 defines work product A, so the relevant set of T4 is C. Because the intersection of T5's relevant set and T4's defined set is the empty set, T4 does not become part of the slice. This means that T4 does not define anything which would be relevant for T5, so T4 is not important. In the next iteration, the relevant set of T3 is set to C, too. Then T2's relevant set is set to the empty set, because T2 defines C and C is the only content of T3's relevant set. The intersection of T3's relevant set and T2's defined set is not the empty set (it contains C), so the referenced set of T2 (=A, C) is added to T2's relevant set. This means that T2 defines C and requires A and C to do that, thus A and C become relevant for the slice. T2 is added to the slice because it is important. In the last iteration T1 is handled in the same way and therefore also added to the slice.

This way we can compute a backward slice which contains all tasks that are important for task T5. Additionaly we are also interested in how the relevant work products depend on each other, because this can be valueable information for the user of our tailoring tool. To take work product dependences into account, we need to modify the algorithm so that it can build up a dependence tree. In step 1 the selected work product C has to be added to the dependence tree and step 3 has to be extended by an additional operation inside its `if` statement:

			def.	ref.	rel.
⇒	Task T1	↓WP A, ↓WP B, ↑WP A, ↑WP C	A, C	A, B	A, B
⇒	Task T2	↓WP A, ↓WP C, ↑WP C	C	A, C	A, C
	Task T3	↓WP C, ↑WP X, ↑WP F	X, F	C	C
	Task T4	↓WP F, ↑WP A	A	F	C
⇒	Task T5	↓WP C		C	C

Table 3.2: This is the result of computing a backward slice for task T5 and work product C. All tasks that are part of the slice have an arrow (⇒) in front of them. The three columns show the defined, referenced, and relevant work products of each task.

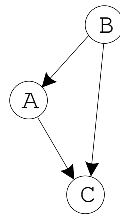

Figure 3.1: Work product dependence tree for work product C from task T5 backward.

3.e) add the work products of S's defined set to the dependence tree, so that the work products of T's relevant set depend on them. Dependences which would lead to self-dependence relations are not inserted.

With this extension a work product dependence tree for our backward slice can be computed. Figure 3.1 shows the structure of this dependence tree. Node C is the starting point of the tree. Arrows point from the "master work product" to the one which depends on it. In task T2 A and C are referenced and C is defined. So C becomes dependent on A and C. Since it does not make any sense that C would depend on its own, this dependence can be ignored and only the dependence on A is added to the tree. In task T1 A and C are defined and A and B are referenced. So, for both A and C the two referenced work products A and B have to be inserted as "master work products". A and C become dependent on B. All other dependences are ignored because they are either self-dependences or already part of the tree.

At this point we have computed a backward slice containing all tasks that influence T5 and a dependence tree with information on how the work products,

which are important in the context of the slice, depend on each other. Now we are interested in a forward slice that shows us which of the tasks `T6`–`T10` can be influenced by `T5`. This slice can be computed with a similar technique as the backward slice was. Additionally, we can build up a dependence tree for work product dependences again. The algorithm for computing the forward slice and the work product dependence tree works as follows.

1. Initialize the relevant sets of all tasks to the empty set. Insert `P` into the dependence tree.
2. Insert work product `P` into the relevant set of task `T`.
3. For `T`'s immediate successor `U`, compute the relevant set as:
   a) relevant(`U`) = relevant(`T`)
   b) if relevant(`U`) $\cap$ ref(`U`) $\neq$ {} then
   c) relevant(`U`) = relevant(`U`) $\cup$ def(`U`)
   d) include `U` into slice
   e) add the work products of `U`'s defined set to the dependence tree, so that the work products of `U`'s referenced set depend on them. Dependences which would lead to self-dependence relations are not inserted.
   f) end
4. Work forwards in the process, repeating step 3 for `U`'s immediate successors until the last task is reached.

The explanation of this algorithm is easy to understand. At each task, we verify if the task references a work product which appears in the relevant set. If this happens, the task is added to the slice and the work products of the task's defined set are added to the relevant set. In this way all work products that can be influenced by work product `P` are added to the relevant set. Depending on the process structure a slice can become pretty large, but anyway, the result is always precise because all dependences are taken into account.

Table 3.3 on the following page shows how this algorithm is used to compute a forward slice for task `T5` and work product `C` of our example process. Figure 3.2 on the next page illustrates the resulting work product dependence tree.

It is now possible to compute backward and forward slices of a process model. The two work product dependence trees computed by the forward and backward

			def.	ref.	rel.
⇒	Task T5	↓WP C		C	C
⇒	Task T6	↓WP C, ↑WP Y	Y	C	C, Y
	Task T7	↓WP F, ↑WP F, ↑WP X	F, X	F	C, Y
⇒	Task T8	↓WP Y, ↑WP D, ↑WP E	D, E	Y	C, Y, D, E
⇒	Task T9	↓WP E, ↑WP G	G	E	C, Y, D, E, G
⇒	Task T10	↓WP C, ↑WP K	K	C	C, Y, D, E, G, K

Table 3.3: This table shows the result of computing a forward slice for task T5 and work product C. All tasks that are part of the slice have an arrow (⇒) in front of them. The three columns show the defined, referenced, and relevant work products of each task.

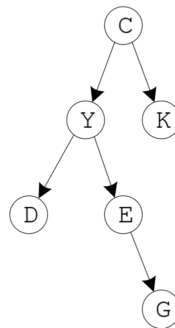

Figure 3.2: Work product dependence tree for work product C from task T5 forward.

slicing algorithms could also be combined; this would result in a full dependence tree of one work product. The example showed a simple process model part which did not contain complex data dependences. For example, there could also occur circular dependences of work products. Such dependences can also be computed with the methods we have shown in this section.

3.3 The Application

This section makes up a short introduction to the prototype application. It shows what the prototype does and in which context it works, thus with which entities it interacts.

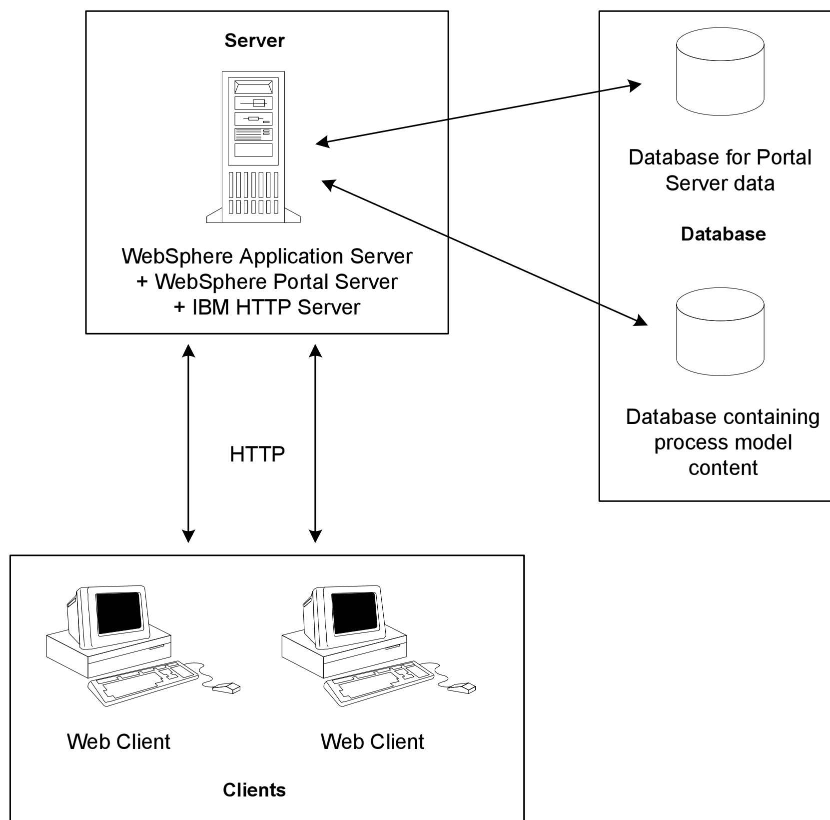

Figure 3.3: Architecture overview of the prototype.

3.3.1 Architecture Overview

The prototype is a portal-based application. It solely works on a portal server which itself is usually based on an application server. In our case we use IBM's WebSphere Portal Server which is an extension to the WebSphere application server. The content of the Global Services Method is available in a database. In general, the prototype could also be adapted to software development processes which are not available in databases. A second database is used by the portal server for storing persistent data, configuration settings, and so on. Clients can access the prototype application through web browsers. On client side there is absolutely no domain logic, so all the "work" is done on the central server. Figure 3.3 on the preceding page provides an architecture overview of the prototype.

3.3.2 Functionalities

The functionalities of the prototype are:

- Browsing the process model's content
- Managing projects
  - Creating projects
  - Deleting projects
- Managing collaborators
  - Creating collaborators
  - Deleting collaborators
  - Assigning collaborators to projects
  - Assigning collaborators to roles within projects
- Tailoring tasks
  - Including tasks into projects
  - Excluding tasks from projects
  - Compensating tasks
- Tailoring work products
  - Including work products into projects
  - Excluding work products from projects

  - Compensating work products
- Exporting tailoring results

Tasks and work products can be “compensated”. The idea behind this is to reduce the complexity of big process models in small projects by reducing the number of elements that have to be handled. A task can compensate one or more other tasks which means that it additionally does what the other tasks would do. The result is that the input work products of all compensated tasks become input work products of the compensating task, the output work products of all compensated tasks become output work products of the compensating task, and all roles appearing in the compensated tasks become roles of the compensating task. In terms of work products compensation means that one work product can be replaced by others, or that one work product includes the content of other work products. This way the number of tasks and work products can be decreased without loosing content. In small projects with few collaborators this can reduce the overhead of managing a lot of tasks and work products without completely renouncing them.

Chapter 4

Implementation of the Prototype

In this chapter we want to take a closer look at the prototype. Our goal was to develop a prototype which fulfills the requirements defined in sec. 3.1 on page 27. Section 4.1 describes the prototype's design in a detailed way. Section 4.2 on page 57 then shows how the prototype has been tested and points out its strengths.

4.1 Design

Section 4.1.1 shortly describes the first attempts and the problems that occurred. In sec. 4.1.2 on the next page we then give some background information about things that influenced the design. Afterwards sec. 4.1.3 on the following page shows an example of how the GUI was designed. Then we take a look at the design of the tailoring extension of the database in sec. 4.1.4 on page 43. An overview of the application design is given in sec. 4.1.5 on page 45, followed by detailed application design topics in sec. 4.1.6 on page 49.

4.1.1 First Attempts

Creating a good design for the prototype took some time. The main problem was that when I[1] started to work on the prototype, I did not know much about the Global Services Method. And I have to confess that I underestimated the

[1] I use the first person here, because this represents my own opinion and experience.

complexity of such a big development process a bit. The result was that I did not spend enough time on thinking about a good design of the application at the beginning of my work. Later I had to completely reengineer the design in order to fulfill the requirements.

The first version of the design was somewhat conservative, resulting in a few classes with many static methods and bad object-orientation. For the beginning this was satisfactory, but later things got too complex and the whole application was really inflexible. After I knew the disadvantages of this first version it was at least easier to develop a better design for further development.

4.1.2 Background Information

The design of the prototype is especially influenced by the database design of the Global Services Method and the fact that the application has to be web-based.

Tailoring the Global Services Method (GSM) is the main function of the prototype. In order to do this, the application has to use a lot of the GSM's content, which is available in a database. The whole content and coherences are defined in this database, thus making the application very database-intensive. The GSM's database contains about 40–50 tables; approximately 15 of them are of interest for our work. We cannot explain details about the database here, because it is classified confidential by IBM. Regardless of this, knowing details about the database should not be essential to understand the design, although, of course, it would be interesting.

The web-based approach leads to a request- and response-based communication. The server cannot keep tailoring information about different projects in memory, because this would lead to additional problems. For example, depending on the number of projects, this would result in too high memory load at the server. So the current tailoring state is always stored in the database and the server loads the required data on each request. That sounds time extensive, but tests showed that—provided that a good database is used—the prototype has really good response times, even if a lot of database accesses are required for one request.

4.1.3 Graphical User Interface

As the prototype is portal-based, we use portlets to build up the webpages (see sec. 2.3.3 on page 23 for background information on portlets). Creating good-

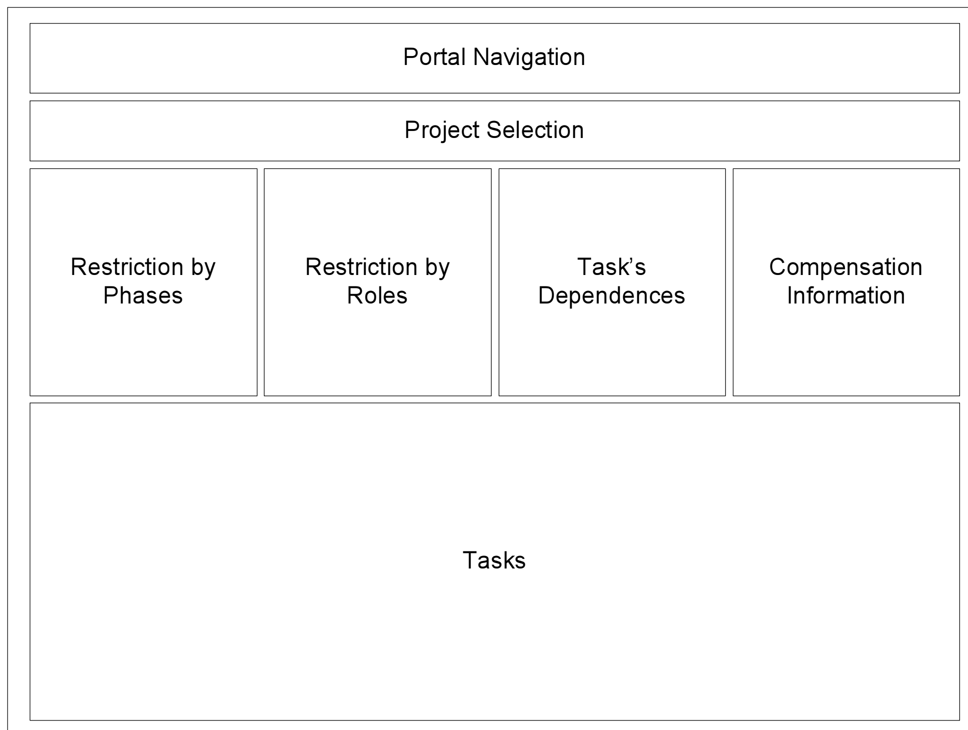

Figure 4.1: Typical design of a portal page. The outer box marks the browser window and the rectangles inside symbolize the portal page's portlets.

looking webpages with a portal server is not that difficult, because the server defines layout rules (font sizes, link colors, text colors, navigation layout, etc.); this guarantees that the whole webpage looks fine and the programmer does not need to care about such details. Later the user can select his or her favorite out of different layout schemes.

Building up the webpages with portlets results in well-structured and clear webpages. Figure 4.1 shows how one of the webpages is made up of various portlets. The uppermost box labeled "Portal Navigation" is provided by the portal server. The other boxes represent five portlets that have been placed on this page. The subject of this page is to manage the tasks of a project. Its layout is typical for the prototype, as some of the other pages are composed in a similar way.

Let us take a look at how this page operates. The portlet named "Project Selection" allows the user to select the project he or she wants to work on. Each project is bound to an engagement model (see sec. 2.1.4 on page 11 for an explanation of the GSM and its terms), thus, selecting a project also defines

the process that lies behind it. When the user selects a project, all the other portlets show data that belongs to the selected project. The most important portlet on this page is the "Tasks" portlet, which has several functionalities; it can be used to include tasks in the project, exclude tasks from the project, and compensate (see sec. 4.1.4) tasks by other tasks. Additionally it uses different colors to indicate extra information about the tasks. The two portlets "Phases" and "Roles" can be used to restrict the shown tasks; this is useful if the current engagement model is very large, and showing all tasks at once would be too complex. Restriction by phase means that only the tasks which appear in the selected phase are shown. Restriction by role means that the user can select a role (e.g., System Analyst) and the portlet "Tasks" then only shows tasks which are done by this selected role. Of course, these restrictions can also be combined. If the user selects a task in the "Tasks" portlet, the "Task's Dependences" portlet will show all dependences of this task by computing slices for the task and all of its work products. As the name implies, the "Compensation Information" portlet additionally shows compensation information for the selected task.

4.1.4 Database Design of the Tailoring Extension

We had access to a database containing the contents of the Global Services Method. From our point of view, these tables are static and cannot be changed by our prototype, however, in reality the tables can change; e.g., if a new version of the GSM is released, the database will have to be updated. The prototype provides the functionality to create projects (based on engagement models of the GSM) and tailor the process to the user's needs.

The data concerning the current tailoring state of a project has to be stored in a database by the prototype. Figure 4.2 on the following page shows a diagram of this database extension. The shown database tables have the following functionalities.

Project Every project that is created is stored in this table. It has a name by which the user can identify it, a reference to the engagement model of the Global Services Method, and the ID of the portal user who created the project. The ID of the user who created the project can be used to show only a user's own projects in the "Project Selection" portlet.

Collaborator Collaborators can be stored in this table; they are independent of a specific project and are managed separately. Collaborators can be assigned to projects and roles.

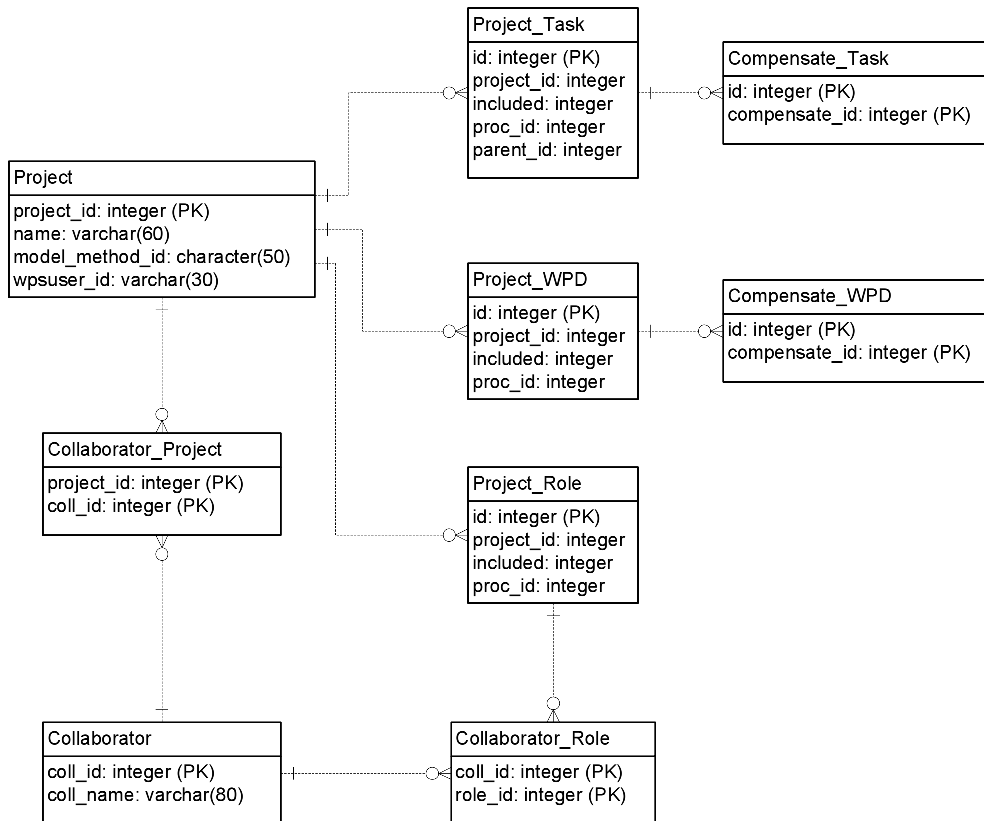

Figure 4.2: Database extension for storing tailoring information.

Collaborator_Project This is an associative table which connects collaborators and projects. One collaborator can take part in many projects and one project can have many collaborators.

Project_Role This table creates a connection between a project and a role of the Global Services Method, thus it represents a role in a specific project. The Project_Role table stores information about whether the role is included in the project or not, because the user of the tailoring tool can decide to remove roles from a project.

Project_WPD Like the Project_Role table, the Project_WPD table also creates a connection to the Global Services method; it connects projects and work products. As the user can include and exclude work products, this table also has to store this information.

Compensate_WPD Work products can be compensated by other work products. The idea behind this compensation is that one work product can be replaced by others, or that one work product includes what originally would be part of a separate work product (see sec. 3.3.2 on page 38). This can be helpful in small projects to keep the number of work products manageable.

Project_Task This table connects projects with tasks of the Global Services Method. Again, tasks can be included and excluded by the user. This table needs additional information about the parent of each task in order to clearly identify each task; the parents of tasks are activities in the Global Services Method. The information about a task's parent is necessary, because a task—identified by one unique ID—can occur in multiple activities and we want to be able to exclude and include each single task separately.

Compensate_Task Like work products, tasks can also be compensated. In terms of tasks this means that one task can take on the work which would originally be done by another task (see sec. 3.3.2 on page 38).

4.1.5 Application Design Overview

In this section we want to discuss the high-level design of the prototype. Figure 4.3 on page 47 shows the three main layers *presentation*, *domain logic*, and *data access* with a few sample classes. In sec. 4.1.6 on page 49 we take a closer look at each of the three layers and describe their interior structure.

The *presentation* layer contains classes for all portlets plus JSPs to build the HTML output of the portlets. `MethodSlicerPortlet` is the common base class of all portlet classes; it contains some common functionalities like sending messages to other portlets and showing help texts for portlets. Sending messages is supported by the portal server, but we simplified it a bit by providing one single method in the `MethodSlicerPortlet` super class. The super class additionally supports sending messages with more than one value as content.

Portlet classes are similar to servlet classes; they provide methods for generating content, handling parameters that come with requests, and so on. The `actionPerformed` method of a portlet class is usually the place where a computation at server side starts. For example, the user has selected an engagement model for browsing it. The portlet class then stores information about the selected engagement model in the user's session. Then the portal server builds up the next HTML page for the client, thus the `doView` methods of all portlets, which are on the page, are called and the content is put together. Each portlet then accesses the *domain logic* and *data access* layers to generate the content that it has to show. The portlet class itself only initiates these computations, but the most work is done in the *domain logic* layer. Portlet classes prepare the data, which has to be shown, and use JSPs to generate the HTML output. The JSPs themselves use objects of the *domain logic* layer, but they never access the *data access* layer.

The *domain logic* layer contains several classes that represent the data of the Global Services Method and some classes for tailoring purposes. All computations are done in this layer; it comprises the whole logic of the prototype. Classes of the *domain logic* layer never access anything of the *presentation* layer. They only use the *data access* layer to get data out of the database or to store information. Our *domain logic* layer is based on the ideas of the *Domain Model* pattern described in [Fowler, 2003]. It is a rich domain model which differs from the database design and therefore does not use any Enterprise Java Beans. The *domain logic* layer exclusively uses POJOs[2] to model the *Domain Model*. The main reason for this is that an EJB-based approach would not have provided us with any advantages. However, the POJO-based domain model has advantages; Fowler describes some of them in [Fowler, 2003]:

> A POJO domain model is easy to put together, is quick to build, can run and test outside an EJB container, and is independent of

[2] Martin Fowler found out that "normal" Java objects did not have a defined name while he was preparing for a talk in 2000. Therefore, Fowler, R. Parsons, and J. Mackenzie decided to give them one: POJOs (plain old Java objects) [Fowler, 2003].

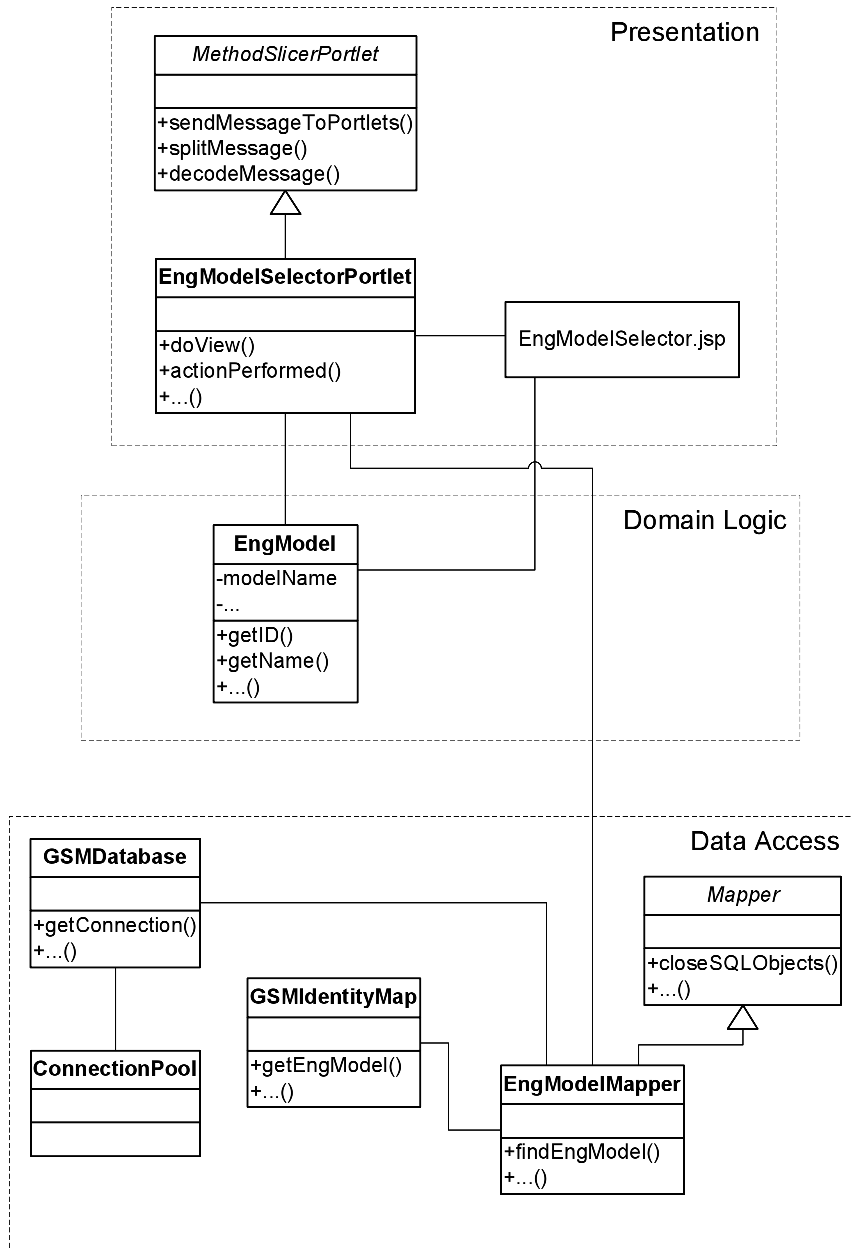

Figure 4.3: This is a coarse design overview of the prototype.

> EJB (maybe that's why EJB vendors don't encourage you to use them).

And some paragraphs later he hits the mark:

> The biggest frustration for me with the use of EJB is that I find a rich *Domain Model* complicated enough to deal with, and I want to keep as independent as possible from the details of the implementation environment. EJB forces itself into your thinking about the *Domain Model*, which means that I have to worry about both the domain model and the EJB environment.

Apart from the fact that the Global Services Method's database is too complex for modeling the *Domain Model* with entity beans, this was another reason why we did not spend our time on EJBs.

The *data access* layer is responsible for providing access to the data of the database. In the first version of the prototype's design, the *data access* layer was pretty chaotic, because it just grew as needed and was not consistent. Even classes of the *domain logic* layer sometimes accessed the database directly.

The current *data access* layer design is smarter, easier to maintain and extend, and faster than the old one. It is mainly based on the *Data Mapper*, *Identity Map* and *Lazy Load* design patterns explained in [Fowler, 2003]. In [Fowler, 2003] Martin Fowler explains:

> The *Data Mapper* is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. With *Data Mapper* the in-memory objects needn't know even that there's a database present; they need no SQL interface code, and certainly no knowledge of the database schema.

This results in the big and important advantage that the *domain logic* layer does not need to know anything about how and where the data of the process is stored. For our prototype it means that switching to another process model (e.g., the Rational Unified Process) mainly requires adapting the mapper classes which map domain objects to the database. In fact, it is not even necessary to use a database, because the mappers could also read and write data from and to file streams or any other storage.

Identity Maps additionally support the *data access* layer. Fowler gives an amusing introduction to *Identity Maps* in [Fowler, 2003]:

> An old proverb says that a man with two watches never knows what time it is. If two watches are confusing, you can get in an even bigger mess with loading objects from a database. If you aren't careful you can load the data from the same database record into two different objects. Then, when you update them both you'll have an interesting time writing the changes out to the database correctly.
>
> Related to this is an obvious performance problem. If you load the same data more than once you're incurring an expensive cost in remote calls. Thus, not loading the same data twice doesn't just help correctness, but can also speed up your application.

For the prototype, the usage of *Identity Maps* brings amazing perfomance increases. Especially the Global Services Method content, which never changes, is loaded from the database step by step—depending on what the actual client requests demand—until nearly everything is kept in memory as objects. *Identity Maps* store references to all loaded objects in `HashMap`s, resulting in very fast access times. For example, let us take a look at how objects from the *presentation* and *domain logic* layers get a reference to an `EngModel` object. They use the `EngModelMapper`'s `findEngModel` method to retrieve the desired reference and need not care about where the mapper gets the object from. The `EngModelMapper` itself first requests a reference to the object from the `GSMIdentityMap`. If there is no success, the `EngModelMapper` loads the required data from the database, creates a new `EngModel` object, stores the reference in the identity map for future requests, and returns a reference to the caller.

In many classes of the prototype we used a *Lazy Load* design pattern to further improve performance. *Lazy Load* means that data is only loaded when it is needed. For example, if you request an `EngModel` object from the accordant mapper class, only the information about the engagement model is loaded (its name, etc.). Without *Lazy Load* the content of the whole engagement model would be loaded, thus, hundreds of other objects would be created, even if they are never used.

4.1.6 Application Design Details

In this section we take a closer look at the three layers we introduced in sec. 4.1.5 on page 45.

Presentation Layer

The *presentation* layer is straightforward. Each portlet is represented by a portlet class and uses its own JSP for generating output. As shown in fig. 4.3 on page 47, all of the portlet classes extend the `MethodSlicerPortlet` base class, which provides some common functionality. We did not include a figure of the complete *presentation* layer, because all the classes look alike.

Domain Logic Layer

The *domain logic* layer can be divided into two parts: the first one deals with the content of the Global Services Method and the second one deals with the process model's tailoring.

Figure 4.4 on page 52 illustrates the first part of the *domain logic* layer. Let us take a look at each of these classes separately:

`ProcessElement` This abstract class plays a central role, because most of the other classes extend it. The structure is based on a *Composite* design pattern [Gamma et al., 1995]. `EngModel`, `Phase`, `Activity`, `Task`, `Role`, and `WorkProduct` are all "process elements". The `ProcessElement` class has a `ProcessElement` array as instance field, thus a hierarchical structure is defined, like the process model describes it. One of the advantages of this approach is that the structure becomes flexibel.

`ProcessElementComparator` Objects of this class can be used as comparators when `ProcessElement` objects have to be sorted. There are two ways of creating `ProcessElementComparator` objects: the first is to use the default constructor which creates a comparator for alphabetical ordering, and the second one requires a `ProcessElement` object as parameter and can be used to sort the elements in the order they appear inside the specified `ProcessElement`.

For example, if you create a `ProcessElementComparator` with phase `P` as parameter, you can use it to sort elements—in this case it would only make sense for `Activity` objects—so that they appear in the right order.

`EngModel` This is the top of the hierarchical structure.

`Phase` Represents a phase of the process model.

`Activity` Represents an activity of the process model.

`Task` The class `Task` plays a special role; objects of this class are the smallest pieces of work that occur in the process model. Therefore, `Task` objects differ from the higher-level elements (`EngModel`, `Phase`, and `Activity`). `Task` objects have two kinds of children: `WorkProduct` objects, which represent the work products that occur in the task, and `Role` objects, which represent the roles that perform the task.

`WorkProduct` Objects of this class represent the work products of the process model. They are process elements, because they are part of the hierarchical structure. Additionally, they store extra information about whether they are input, output, or input and output in the tasks they occur in. Different from most other process elements (except `Roles`) `WorkProduct` objects are divided into various domains.

`Role` `Role` objects represent the roles of the process model. Like `WorkProduct` objects, they belong to domains, too.

`Domain` Objects of this class represent the domains of a process model. In the case of the Global Services Method there are six big domains (e.g., "Application"). The domains are themselves subdivided into sub-domains, but we did not take these sub-domains into account.

As decribed above, `ProcessElement` plays an important role in the process structure. In fact, most operations are directly handled in this class. Classes of the types `EngModel`, `Phase`, and `Activity` would nearly be unnecessary; the only functionality they have is that the design becomes more understandable, e.g., when an `EngModel` object is requested, the `EngModelMapper` class is used to create or find it. Anyway, in general it should be no problem to drop the three classes, and handle the process structure in `ProcessElement` only.

An example of the advantages gained from the `ProcessElement`-based structure is the following. The `ProcessElement` class has a method to get its sub-elements (`getSubElements()`). This method works straightforward, it simply returns the direct sub-elements of the `ProcessElement` object it is called for. Additionally, `ProcessElement` has a method `getSubElements(int type)`, which returns all sub-elements of a particular type (e.g., `EngModel`, `Phase`, etc.).

Figure 4.5 on page 53 shows the `getSubElements` method. The public method can be called in order to retrieve a `Vector` containing all the desired elements. Internally a `TreeSet` is used to store the process elements; this has the advantage that it is fast, the elements are sorted by name (an object of type `ProcessElementComparator` is used for that), and multiple occurrences of one

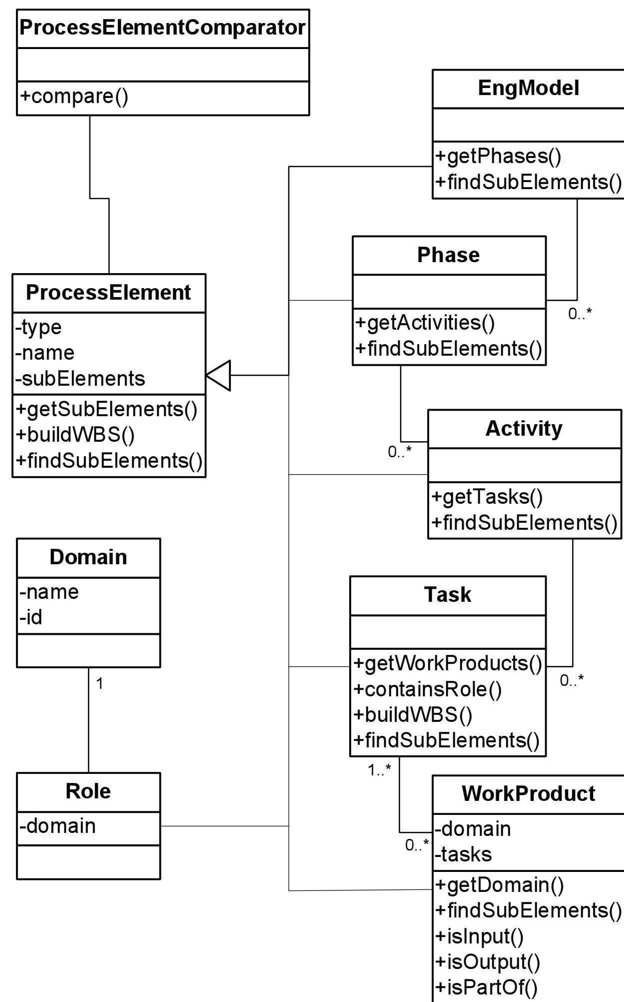

Figure 4.4: The part of the *domain logic* layer which models the content of the process model. The figure does not contain all fields and methods of the classes.

```
public Vector getSubElements(int type) {
  TreeSet tmpSet =
    new TreeSet(new ProcessElementComparator());

  getSubElements(type, tmpSet);

  if(tmpSet.size() > 0)
    return new Vector(tmpSet);
  else
    return null;
} // getSubElements

private void getSubElements(int type, TreeSet set) {
  if(this.type == type)
    set.add(this);
  else {
    ProcessElement[] subElems = getSubElements();

    if(subElems != null)
      for(int i = 0; i < subElems.length; i++)
        subElems[i].getSubElements(type, set);
  } // else
} // getSubElements
```

Figure 4.5: Method `getSubElements` of the class `ProcessElement`. It recursively finds all sub elements of the defined type.

element are ignored. The private `getSubElements` method then recursively searches for the specified element type and adds all occurrences to the `TreeSet`.

As these two methods take advantage of the `ProcessElement`-based data structure, a lot of different questions can be easily answered with their help. For example, you can call `getSubElements(TASK)` on an `EngModel` object to retrieve all tasks that occur in this engagement model. Another example would be to call `getSubElements(ROLE)` on a `Phase` object; this would result in a list containing all roles that occur in the given phase.

The second part of the *domain logic* layer contains the functionality to tailor process models; fig. 4.6 on the next page illustrates this part of the layer. Objects of these classes represent the current state of the tailoring process. As the content always changes, these objects are never kept in memory; instead, they are created per request. Because of the easy database structure (cf. fig. 4.2 on page 44), the creation of these objects works fast enough to ensure good response times of the system. The following describes the classes in detail:

`Project` An instance of this class represents a project which a user can create. The object has references to `Collaborator` objects of collaborators that take part in the project, and references to all `ProjectRole`, `ProjectTask`, and `ProjectWorkProduct` objects of the roles, tasks, and work products that occur in the engagement model the project is based on. A `Project` object also has a reference to the `EngModel` object of its engagement model, thus the whole content of the model is accessible by the `Project` object.

Usually a `Project` object is first created, then something is changed, and finally its state is written to the database again (by the accordant `ProjectMapper`). Therefore, the `Project` object stores information about all `ProjectElements` that changed (`modifiedElements` field).

`ProjectElement` This is the base class of all project-related elements. It contains common functionality, e.g., information about whether the element is part of the project or if the element compensates other elements.

`Collaborator` Represents a collaborator. Collaborators can be defined by the user; they do not depend on specific projects. This means that one collaborator may take part in more than one project. Collaborators can be assigned to roles; therefore, `Collaborator` objects have references to `ProjectRole` objects.

`ProjectRole` Represents one specific role of the engagement model in a project.

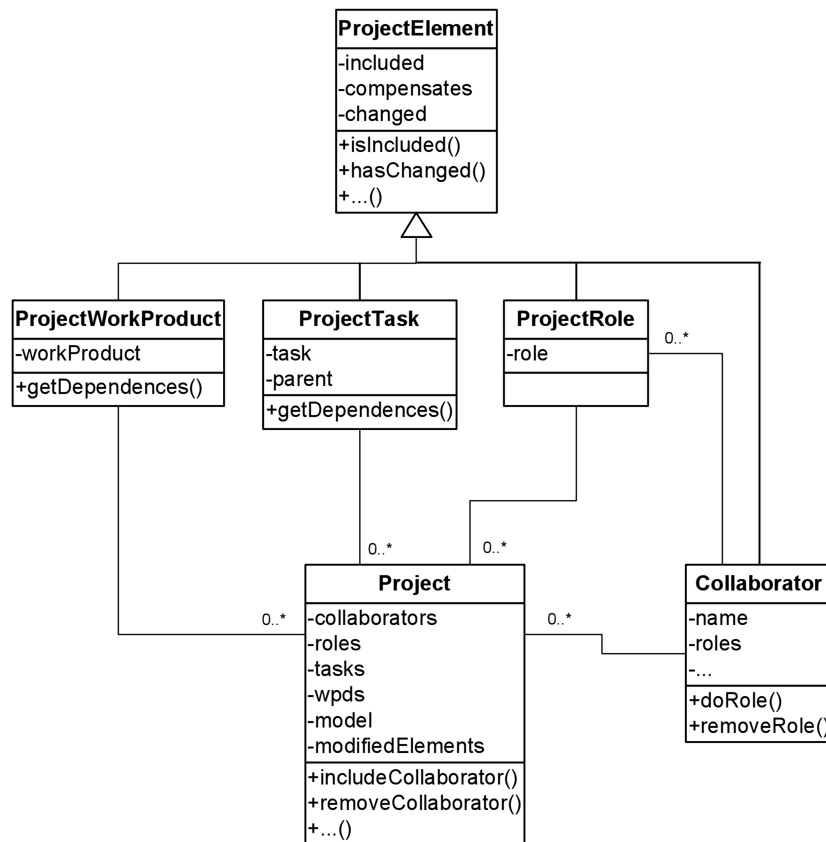

Figure 4.6: This is the tailoring-relevant part of the *domain logic* layer. The figure does not contain all fields and methods of the classes.

Objects of this class have a reference to the `Role` object of the process model.

`ProjectTask` An object of this class represents one specific task of the engagement model. It has references to the `Task` object it represents and its parent `Activity` object. This is important, because in the hierarchical process structure one and the same task can occur in more than one activity. Our prototype supports inclusion and exclusion of same tasks in different activities separately.

The `getDependences` method uses adapted slicing methods to compute a slice containing all dependences of the task. Section 3.2.2 on page 31 explains how slicing methods have been adapted and how the dependences are computed.

`ProjectWorkProduct` Represents one specific work product of the engagement model in a project. `ProjectWorkProduct` objects have references to the `WorkProduct` objects they represent. Like `ProjectTask` objects, they also use adapted slicing methods in their `getDependences` method in order to compute dependences of the work product.

Data Access Layer

The *data access* layer contains mapper classes—based on the Data Mapper design pattern described in [Fowler, 2003]— for all classes of the *domain logic* layer. The mapper classes all look similar, so we do not show an extra figure here. Section 4.1.5 on page 45 shows and discusses the structure of the *data access* layer.

For example, let us take a look at the `ProjectMapper` class. It provides four methods:

`findProject(int id)` This method can be used to get a reference to a project, provided that the project's ID is known. In most cases this method is used, because normally we know which project we are interested in.

`findAllProjects()` This method returns a collection containing references to all projects that exist in the database. This is useful for initial tasks, like selecting a project.

`storeProject(Project project)` Stores the given project to the database. The `Project` object's `modifiedElements` field is used to update only what has been changed.

`deleteProject(Project project)` This method deletes a project. All database entries of elements which belonged to the project are removed, too.

4.2 Application Test and Usage

Application Test

As the application is a prototype and time was rare, it was not tested systematically. We just informally defined some test cases with defined results and manually proved whether the prototype worked correctly. Of course, there would have been better ways of testing. Anyway, the prototype appeared to be stable and delivered correct results in all tests.

Hardware Requirements

First of all, the prototype needs a functional installation of IBM's *WebSphere Portal Server*. This requires a computer with about one gigabyte of RAM and a CPU faster than one GHz to function perfectly.

The prototype itself does not need many resources. The *domain logic* layer requires approximately 25–30 megabyte of RAM, when it is in an advanced state and has loaded nearly all data from the database. In relation to the one gigabyte of RAM, which the server needs anyway, this should be no problem at all. We developed and tested the prototype on a computer with an AMD Athlon XP 2000+ (1666MHz) processor, where it performed quite fast and had very good response times; thus, it should also satisfactorily run on computers with slightly weaker CPUs.

Strengths

Our prototype has the following strengths:

- It has a flexible design, allowing it to be adapted to other software development processes.
- It is based on a portal server, which improves the appearance of the application and provides some useful functionalities.
- It is a good basis for further development.

Chapter 5

Conclusion

This last chapter reviews this thesis, shows application areas of our prototype, and gives some ideas for possible future work.

5.1 Intention and Solution

The intention of this thesis was to create a tool for tailoring software development processes with program slicing techniques. Therefore, analogies between process models and software programs had to be found, a program slicing algorithm had to be adapted to a software development process, and a prototype of a tool for tailoring a process model had to be developed.

The result was that we found useful analogies between software development processes and software programs, and defined them (see sec. 3.2.1 on page 29). Afterwards, we adapted and extended a program slicing algorithm to fit our needs (see sec. 3.2.2 on page 31). Finally, we designed and implemented a prototype of a tailoring tool (see chap. 4 on page 40).

5.2 Advantages and Disadvantages

This section reviews some advantages and disadvantes of our work.

Advantages

- Our solution is based on a simple program slicing algorithm, which perfectly suits our requirements.

- The prototype has a flexible design which allows it to be adapted to other software development processes without too much effort.
- The prototype is portal-based; thus, it benefits from the portal's functionality.

Disadvantages

- A portal-based prototype also has disadvantages; for example, our prototype cannot operate without a portal server. If we wanted it to work on its own, the whole *presentation* layer would have to be rewritten.

5.3 Application Area

Our prototype can be used for browsing and tailoring software development processes based on the OMG's Software Process Engineering Metamodel. We used it on IBM's Global Services Method; however, it is designed to be adaptable to any other process model.

A fully functional application based on our prototype would be a great tool for supporting project managers in tailoring process models for their projects.

5.4 Future Work

Most parts of the prototype are finished. At its current state it can be used to browse the Global Services Method, create and manage projects and collaborators, and tailor tasks. What is missing is the part for handling work products and a function to export the tailoring results in a form that can be used for further processing.

The prototype could also be extended with more powerful program slicing algorithms in order to compute dependences of process models containing more control structures (like loops, etc.). Of course, this would only be theoretical work, because the Software Process Engineering Metamodel currently does not define such control structures; however, it might be interesting, too.

5.5 Availability of this Work

This thesis is available for download at `http://www.dietrichsteiner.com/gerhard/thesis`. On this webpage, you will also find additional information on the topic, interesting links, and maybe information about improvements and further development of the prototype.

Bibliography

[Apache, 2003] The Apache Software Foundation. Jakarta JetSpeed. `http://jakarta.apache.org/jetspeed/site/`, April 2003.

[Buckner et al., 2003] Ted Buckner, Stephan Hesmer, Peter Fischer und Ingo Schuster. Portlet Development Guide, March 2003. Second Edition.

[Fowler, 2003] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Pearson Education, 2003. ISBN 0-321-12742-0.

[Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Pearson Education, 1995. ISBN 0-201-63361-2.

[JCP, 2003a] The Java Community Process. JSR 162 Portlet API. `http://jcp.org/en/jsr/detail?id=162`, April 2003a.

[JCP, 2003b] The Java Community Process. JSR 167 Java Portlet Specification. `http://jcp.org/en/jsr/detail?id=167`, April 2003b.

[JCP, 2003c] The Java Community Process. JSR 168 Portlet Specification. `http://jcp.org/en/jsr/detail?id=168`, April 2003c.

[Kruchten, 2000] Philippe Kruchten. *The Rational Unified Process – An Introduction*. Addison-Wesley, second edition, 2000. ISBN 0-201-70710-1.

[OMG, 2002] Object Management Group. Software Process Engineering Metamodel Specification. `http://www.omg.org/technology/documents/formal/spem.htm`, November 2002. Version 1.0.

[SEI, 1995] Mark Paulk, Charles Weber, Bull Curtis und Mary Chrissis. *The Capability Maturity Model – Guidelines for Improving the Software Process*. Addison-Wesley Longman, 1995. ISBN 0-201-54664-7.

[Steindl, 2000] Christoph Steindl. *Program Slicing for Object-Oriented Programming Languages*. Dissertation, Johannes-Kepler-Universität Linz, Universitätsverlag Rudolf Trauner, 2000. ISBN 3-85487-151-1.

[Weiser, 1984] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, July 1984.