Modularization and Demodularization: Levels of a Java Web Application for Open Health

Master Thesis

Torgeir Lorange Østby
University of Oslo
Department of Informatics
torgeilo@ifi.uio.no

February 1, 2008

Abstract

The outset of this thesis was to develop a solution for collecting separately deployable web modules into one seamless web application, a portal. The case study is the District Health Information Software 2, a modular web application with a Java back-end. The task included collecting the modules, giving the module web pages a common look, creating a menu system for accessing the modules, and creating common web widgets used by all the modules. As part of the development process, the thesis looks at the various levels of modularization in the Java web application. Modularization can be applied to all levels of an application, from the innermost levels of code organization in methods, through classes and combinations of classes and interfaces, to system modules and complete applications. The thesis focuses on the impact of code organization and the use of Java language constructs in order to promote module flexibility, extensibility, reusability, testability, and forward compatibility. It also looks at how tools and frameworks affect the modularization of applications, and how the case study and developed solutions compare to high level service architectures.

Acknowledgements

I would like to thank my teaching supervisor Knut Staring for his invaluable guidance and feedback during the writing.

A big thanks goes out to my loved ones for their support during the frustrating times, and their tolerance for my (still) never-ending presence by the computer.

I would also like to thank the Oslo team and the whole international DHIS 2 team for the great environment and all the interesting discussions, all from which I have experienced and learned a lot.

Contents

1 Introduction **12**

2 Literature **14**
2.1 Modularization 14
2.1.1 Object- and aspect-orientation 16
2.2 Modular service architectures 17
2.3 Information infrastructures in global and local contexts 19
2.3.1 Standardization and flexibility 20
2.3.2 Local and global contexts 20

3 Background **22**
3.1 Health Information Systems Programme 22
3.1.1 A quick look at history 22
3.1.2 So what is the Health Information Systems Programme? . . . 23
3.2 District Health Information Software 24
3.2.1 District Health Information Software version 1 25
3.2.2 District Health Information Software version 2 26
3.3 Where I come in 30

4 Research methods **32**

5 Concepts, tools and frameworks enabling modularization **35**
5.1 Maven 36
5.1.1 Source code organization 37
5.1.2 The correlation between Maven projects, Java archives and web archives 39
5.1.3 Modularization using Maven 42
5.2 Inversion of Control 44
5.2.1 Dependency injection 44
5.2.2 Aspect-Oriented Programming 46
5.3 The Spring Framework 46

5.4 WebWork . . . 47

6 The DHIS 2 web portal **50**
6.1 Assembling the portal . . . 51
6.2 Problems with the current solution . . . 57
6.3 A timeline perspective . . . 59
6.4 Creating a common look . . . 59
6.5 Common web page elements . . . 65
6.5.1 Module entry point and menu system . . . 65
6.5.2 Organization unit tree widget . . . 69

7 Discussion **75**
7.1 Java language constructs . . . 75
7.1.1 Parameterized inheritance versus Spring beans . . . 76
7.1.2 Cross-cutting concerns . . . 79
7.2 Modularization of frameworks and libraries . . . 80
7.3 The influence of tools . . . 82
7.4 A new web archive . . . 86
7.5 Web portal architecture . . . 90

8 Conclusion **94**

9 Further Research **97**

Glossary **99**

Bibliography **101**

Appendices **107**
Appendix A . . . 108
Appendix B . . . 110
Appendix C . . . 111
Appendix D . . . 114
Appendix E . . . 115

List of Figures

2.1 The hierarchy of standards. Each level is free to define their own extensions to the standards inherited from above. Copied with permission from [29]. 21

3.1 The layered architecture of DHIS 2; the classical three-layer architecture. 28

5.1 The project object model, or `pom.xml` file, in the top directory of a Maven 2 project. The `src` (source) directory is the base directory for all source code in the project. 36

5.2 A Maven 2 project with the conventional main directories and some exemplary content. Files are in italic. 38

5.3 A Maven 2 project with all conventional source directories as seen from the Eclipse SDK's [6] Package Explorer. 39

5.4 How a Maven project is packaged into a Java archive. Left out from the figure is a Manifest.mf file, which is automatically created and placed in the META-INF directory. Also, Maven puts the pom.xml file in a subdirectory of META-INF. 40

5.5 How a Maven project is packaged into a web archive. The lib directory contains libraries which the application directly or indirectly depends on. Left out from the figure is a Manifest.mf file, which is automatically created and placed in a META-INF directory in the *root* of the WAR file. Also, Maven puts the pom.xml file in a subdirectory of that same META-INF directory. 41

5.6 Basic structure of a parent project containing two sub-projects with source code defining one system module each. 42

5.7 a) shows a simple project hierarchy with one parent project and two project modules. Module-a is defined to inherit any configurations from the parent, but module-b is not. b) shows the system module hierarchy of the same project. Module-b is dependent on module-a at both compile-time and runtime. c) shows the combined dependency hierarchy defining the order of compilation of the projects into modules (bottom-up). 43

6.1 Conceptual image of web interface modules and a common web module, which, by applying the web portal, appear as one application to the users. The web portal as a web interface module in itself is not shown in the figure. 51

6.2 Conceptual image of how the web modules, packaged as JAR files, were merged into a web portal WAR file. The web resources are typically not included in JAR files. 52

6.3 Conceptual image of how deployable web modules were merged into a web portal WAR file using the first custom plugin for Maven. Configuration files with equal paths in the web modules had to be treated specially so that they would not overwrite each other during the merge. 54

6.4 Conceptual image of how deployable web modules were merged into a web portal WAR file using the second custom plugin for Maven. The classes and application resources were packaged in a JAR file and placed in the libraries directory of the web portal, keeping all configuration files separate without any special treatment. 56

6.5 Conceptual image of how deployable web modules are merged into a web portal WAR file using the latest Maven WAR plugin. The classes and application resources are packaged in a JAR file when each web module is packaged, so that no special treatment is needed when merging the modules into the web portal. 57

6.6 Timeline of Maven 2 releases and DHIS 2 events. Note that the distances between the dates do not correspond to the time differences. The two vertical lines symbolize greater jumps in time where uninteresting events have been left out. 60

6.7 How to define a Velocity result in an action mapping. 61

6.8 Action mapping with multiple templates in the result, which has to be supported by a custom Velocity resource loader. 61

6.9 How to specify a main template with dynamic includes of specified menu and page templates in DHIS 2. For different web pages with same or different menus, only the two static parameters need to be changed. The main template assures a common look to all the web pages. 62
6.10 The conceptual idea of how a main template written in Velocity can dynamically include a menu and page template based on outside configuration. Line 10 and 14 contain the include statements. 63
6.11 The module menu with roughly one "home module" (Welcome), eight management modules (Setting, Users, Organisation Units, Data Elements and Indicators, Datasets, Data Mart, Import-Export, and Data Quality), and three entry, reporing, and analysis modules (Data Entry, Report Tool, and Dashboard). 67
6.12 The organization unit tree widget with exemplary content. 70

7.1 Possible modifier combinations regarding the class interfaces for use and extension. 79
7.2 How a Maven project could be packaged into a web archive. Please compare with figures 5.4 (page 37) and 5.5 (page 38), on how Maven projects are actually packaged into Java archives and web archives. Left out from this figure is a Manifest.mf file, which should be automatically created and placed in the META-INF directory. 89

Chapter 1

Introduction

This master thesis started out as a development project on an open source health information system which was, at that time, not in use, but was soon to be implemented in selected districts and states in both India and Vietnam. I was introduced to the health information system, the District Health Information Software version 2 (DHIS 2), and the surrounding research network, the Health Information Systems Programme (HISP), through the INF5750 "Open Source Software development" course at the Department of Informatics, University of Oslo. The project caught my interest as I loved working on the task which I was assigned in the course—creating a customized data entry module—and because it was a real project with real requirements. I have always been fond of developing and solving problems, and so I stayed with the project, working with the core developers and taking part in the succeeding executions of the INF5750 course. Due to my participation I gradually received the "title" core developer.

Apart from helping out on different parts of the health information software, my main concern during the following semesters would become the challenge of merging the user interface modules into a single application with one overall look and feel, giving the users the impression of working with one single system. Several modules where developed during the first INF5750 course, but all of them had different looks, and there was no way of putting them together into one seamless application. The DHIS 2 is a web application with the back-end developed in Java with all sorts of supporting tools and frameworks taking part in the modularization and internal coupling of the components. These tools and frameworks laid the ground for the development of the web portal, as the target solution was suitably named, and so my two research objectives of this thesis became:

- **Primary research objective:** Find a working solution for combining web interface modules into a complete web application with focus on reuse and simplicity for developers, and transparency for users.

- **Secondary research objective:** Investigate how Java language constructs, tools, and frameworks affect and contribute to the modularization of a system.

As evident by the research objectives this thesis is a rather technical one, but we will touch upon less technical subjects along the way.

The rest of this document is organized as follows: Chapter 2 reviews the literature on modularization and standardization. Chapter 3 gives an introduction to the context in which the thesis was written with regard to the Health Information Systems Programme, the District Health Information Software, and my participation. Chapter 4 goes through the research methods used. Chapter 5 and 6 are both empirical chapters, where the first one dives into the technical details of how the different tools and frameworks used in DHIS 2 contribute to the modularization of the system. The second one describes the development of the DHIS 2 web portal. Chapter 7 discusses the empirical material, trying to reveal questions and answers related to the research objectives. Chapter 8 concludes the discussion. Chapter 9 outlines possible paths for further research. At the absolute end of the document are a glossary, the bibliography, and appendices.

Chapter 2

Literature

This chapter reviews the litterature on modularization and decomposition of code at multiple levels. Highly related to modularization is standardization and flexibility, and how modularization play a part in the struggle between universal standards and local adaptation and flexibility.

2.1 Modularization

Modularization is not a new concept within the field of computer science. Coding techniques and assemblers allowed quite early for independent development of program modules and reassembly and replacement of modules without needing to reassemble the whole system. Initially, modularization was a strategy for conserving computer memory, but as memory got bigger, the focus of modularization changed to making the computer systems easier to work with [67]. In 1972 Parnas [55] dicussed different criteria for system decomposition compared to the traditional strategies. A modularization is guided by design decisions which must be made before the development of each module can begin—in particular design decisions which affect more than one module. As a tool for assessing a given modularization, Parnas presents three expected benefits of modular programming:

- Managerial: Sparate groups of developers can work on separate modules simultaneously with little need for communication, thus decreasing total development time.
- Flexibility: It should be possible to make drastic changes to a module without the need to change others.
- Comprehensibility: It should be possible to study a system one module at a time, thus making it easier to get an understand of how the modules and the

system work. Better understanding of a system should result in a better design of it.

A computer system modularized by traditional aspects, focusing on conserving computer memory, would generally be decomposed by the sequence of different processing steps. When one module finished its processing, it would pass on or leave behind the processed data for the next module in the sequence to take over, and so on. While several of the expected benefits can be met, Parnas (ibid.) points to a typical problem with this type of modularization: Important design decisions, for example of how the data is represented or stored, are placed in the interfaces between the modules. If a design decision, manifested in one or more module interfaces, is changed, all modules using and implementing those interfaces must be changed too, clearly breaking the expected benefit of flexibility. Rather, a system should be decomposed using an *information hiding* strategy; to hide important design decisions behind module interfaces, making the interface definitions reveal as little as possible about the modules' inner workings. Changing a design decision hidden in the inner workings of a module does not affect other modules. As an example of such a decomposition, Parnas advices that

> "A *data structure*, its internal linkings, *accessing procedures and modifying procedures* are part of a single module" [55, p. 1056] (Parnas' emphasis)

In addition, algorithms, for example sorting mechanisms, and shared definitions should also be subjected to modularization.

Furthermore, Parnas investigates how his decomposition rules apply in different contexts, and finds that

> "...a careful job of decomposition can result in considerable carryover of work from one project to another." [55, p. 1057]

While Parnas' statement is related to different flavors of the same system, we will see in chapter 6 and 7 that external frameworks which are not properly modularized might lead to extra work for the developers when the framework has to be adapted to the case system.

Woodfield et al. [67] do a quantitative study of how different approaches to program modularization affect the program comprehensibility—the third expected benefit. As in Parnas' case the language used is Fortran. Two types of modules are defined to describe the approaches: A physical module is "a section of code whose boundaries are defined by the syntax of the current language". A logical module is "a section of code which implements only one function". Four different types of modularization were tested: *Monolithic* (non-modularization), where the whole program

is one physical module, *functional modularization*, where each natural logical module is in a separate physical module, *super modularization*, where the program is broken down into very small physical modules, and *abstract data type modularization*, where logical modules based on functional modularization are group into physical modules by the data type they manipulate. The study concludes that the abstract data type modularization was the easiest comprehensible modularization, and that the grouping of logical modules into physical modules should aim for minimized interface complexity between the modules. In other words to minimize the coupling and maximize the cohesion of the modules.

2.1.1 Object- and aspect-orientation

Both Parnas [55] and Woodfield et al. [67] use Fortran with one level of physical modularization. Submodule decomposition can be achieved by using the principle of information hiding recursively, but VanHilst and Notkin [63] present a different approach using object-oriented constructions where they relax the information hiding principle. The goal is to decouple change from design, to minimize the coding needs in case of change, and no submodule is treated any differently from any other submodule. Submodules implement one to few design *choices* each, where one is typically preferable. Submodules are classes which have parameterized inheritance, meaning that they can be combined into any hierarchy of super and subclasses as long as each submodule's dependencies, in regard to method calls, are provided for by any of its superclasses. Assembling a module defined by its design decision thus involves choosing the appropriate submodules and organizing them in an inheritance hierarchy, ready for compiling. Submodule linking is thus compiled as regular method calls between subclasses, minimizing submodule link overhead. Not all object-oriented languages support parameterized inheritance, but a similar result can possibly be achieved using pre-processing of the classes. In the discussion in chapter 7 I will compare this approach with the interface driven, information hiding design used in the case system of this thesis.

Johnson and Hueller [47] stress the importance of object-orientation. Their programming language of choice is Java (which is also the main programming language in my case study) and the focus is on enterprise applications. They draw upon their experiences with application development. Johnson and Hueller advice following good object-oriented design practices and programming to interfaces to enable architectual flexibility. According to Johnson and Hueller, the benefits of good object-oriented design are that implementation details can be hidden, commonalities can be implemented using inheritance and generic programming, code can be reused, and classes can be made extensible so that subclasses do not need to modify any existing

code. Furthermore, by using Inversion of Control, modules are highly decoupled and can be easily swapped by—and do not need to know about any—application wiring functionality, preferably an application framework. Finally, aspect-oriented programming provides for modularization of concerns which tend to cut across objects, which object-orientation itself cannot handle. Central to all this is the concept of plain old Java objects (POJOs), objects which are not aware of or tied to any frameworks, but which are pure “application-oriented” objects [47] [48] [49]. Frameworks supporting POJOs do not constrain application development with regard to design principles [49], like the different aspects of object orientation.

Garcia et al. [37] have done a technical study of how aspect-oriented programming improves modularization of object-oriented design patterns. They base the assessments on various countable metrics such as the number of concern switches for each concern in a pattern, and the number of coupling points from classes and aspects to other classes and aspects. While most object oriented patterns do target separation of concerns, the use of aspects further improved this in several cases. In particular, aspect-oriented approaches proved most effective in modularizing cross-cutting structures, and on the whole reduced the program sizes regarding the number of lines necessary to implement the pattern functionalities. Although separation of concerns was generally improved, the article also concluded that other factors such as coupling, cohesion, and size must be taken into account in the assessment of an aspect-oriented approach. My case study uses aspect-oriented programming in several areas, but only regarding cross-cutting concerns. The mentioned areas are security, internationalization, transaction management, and common information providers for the web interface.

While improved system comprehension by modularization should provide for improved quality of the system, there is evidence suggesting the opposite. According to a study of error rates in three independent software systems, smaller components contain proportionately more bugs than larger components [43]. Evidently a few large modules should result in a more reliable system than many small modules. The polemic paper points to a few possible explanations, like interface inconsistencies, to “save” the traditional belief, but the cases studied do not support any conclusive explanations.

2.2 Modular service architectures

My case study has a set of web modules with specific characteristics. They present functionality to the users over the web, and can be freely combined into a final system. They can be regarded as web service modules with the same base system, and which can communicate with each other. From this perspective, there are architectural styles

and inter-module communication models which are relevant in order to establish a better understanding of the case study. The concepts presented are Service-Oriented Architecture, Enterprise Service Bus, and Representational State Transfer.

Service-Oriented Architecture (SOA) is an architectual style for implementing and using network services based on business activities [46]. Functionality is distributed over a network, allowing clients and other services to make use of it in business processes. Important aspects are resusability, loose coupling between the services, and composability. Services can thus be collected to form a larger application, and services can be *orchestrated* to communicate with each other in order to meet business system requirements. If requirements change, orchestration is the only required task if the necessary services already exist. The services hide implementation details like which operating system or language they are based on, and they can be implemented using any communication protocol. Regarding modularization, SOA focuses on the services as reusable modules, thus operating on a high level of modularization. The web modules of my case study are comparable to such service modules, which will be further discussed in chapter 7.

SOA services are orchestrated to communicate with each other in order to combine businesses or business aspects into complete applications. Communication can be organizised as point-to-point connections between the services, but communication can also be managed by an Enterprise Service Bus (ESB) [32]. An ESB is an infrastructure based on standards, which works like a message broker between services or applications, reducing the number of communication connection points needed. The messages are also based on standards, for example XML schemas, possibly forcing the communication end points to adapt or convert the original message to the standards required by the bus—point-to-point communication lines between services do not necessarily have to follow the same standard. An Enterprise Service Bus can also be used internally in composite applications, on single computers, where orchestration can be used to combine application logic into new functionality.

Representational State Transfer (REST) is an architectual style which models a distributed application architecture with focus on user perceived performance and extensibility. REST was defined by Fielding [35] while working on the HTTP and uniform resource identifier (URI) protocols as a tool for communicating the different concepts of the World Wide Web. According to Fielding, REST models the web as it should be, but it is not bound to any specific protocol like HTTP. The heart of REST is a set of architectual constraints which, when implemented, give a system certain properties like cacheable web pages. The overall picture is to have a client request a resource from the distributed application, where the response is a representation of the resource according to its current state. The response can contain identifiers to other resources, allowing the client to progress through the application by following

the provided identifiers (for example links to other web pages). Both the resource identifiers and representations are required to be uniform, and the response is self-descriptive by its metadata specifying the representation format (for example that the response body is XML). All these constraints contribute to a uniform interface between clients and servers which simplifies the overall architecture and makes services independent of their implementations. The stateless constraint states that the client requests must contain all necessary information for the requests to be fully understood by the server, so that the servers do not need to keep any client session state between the requests. Request independence simplifies processing in multiple ways, including caching and recovering from failures, and allows clients to, for example, use the back button in a web browser without breaking communication flow. Caching is yet another constraint which allows clients to cache representations marked as cachable by the server in order to further enhance performance and scalability. REST does not impose which standards to use for the specific parts of the architecture, still, most people think of the web, and HTTP and URIs, when they think of REST.

Both REST and SOA focus on modularization of services and how they communicate. Vinoski [65] claims REST has specific advantages over SOA in that it focuses on the performance and scalability of the distributed system rather than on application design. Implementing SOA with an eye on REST can result in more scalable and better performing systems, according to Vinoski. The specific points are the uniform interfaces, data variability, and resource naming. Similar opinions are also expressed by others [38] [46].

2.3 Information infrastructures in global and local contexts

With the perspective of web modules offering services to users, as in the previous section, we also approach the subject of Information Infrastructure (II). Hanseth and Lyytinen [41] define II as:

> "*a shared, evolving, heterogeneous installed base of IT capabilities among a set of user communities based on open and/or standarized interfaces.* Such an information infrastructure, when appropriated by a community of users offers a *shared resource* for delivering and using information services in a (set of) community." [41, p. 213] (their emphasis)

As we saw in the previous section, SOA and REST are both architectures for modular infrastructures, and as we will see in this section, modularization is required for infrastructures and systems to be flexible and adaptable to various contexts.

2.3.1 Standardization and flexibility

Hanseth et al. [42] discuss the tensions between standardization and flexibility in information infrastructure and use the Internet as an example. They claim modularization is the only feasible way to cope with large and complex networks, and that "flexibility presupposes modularization" [42, p. 415]. For something to be flexible, it must be orthogonalized to everything else so that when the flexible unit is changed, the rest does not need to be changed—i.e. the modularization enables the flexibility. A black-boxed module with a standardized interface has full flexibility inside, because the interface separates *what* from *how*. Hanseth and Lyytinen [41] suggest using the principles of Parnas [55] for modularization and encapsulation of functionality. They also stress the importance of simplicity, because simple solutions are easier to change than complex solution [41]. There are two types of flexibility: Flexibility for change and flexibility of use. Use and change flexibility are related in the sense that increased use flexibility will decrease the need for change flexibility, and vice versa (ibid.).

Hanset et al. [42] distinguish between two forms of modularization: (1) To make a system layered or hierarchical, or (2) to make a system extensible by for example being prepared for additional functionality. An example of the latter is the HTTP protocol which can be extended with new, possibly application specific, request headers because clients and servers are allowed to ignore headers which they do not support, which again is an example of modularization of standards, and flexibility and scalability of standards by modularization (ibid., [57]). Furthermore, Braa et al. state that "standards should be modularized horizontally and vertically" [57, p. 26]. Vertical modularization is layering of software where each layer has a clear standardized interface to the layer above, while horizontal modularization is the notion of splitting a domain into multiple standards with interfaces between them. Loose coupling between a set of simpler standards is also suggested by Hanseth et al. [40] as a means to reduce socio-technical complexity in information systems. The elements which will not change are those which should be standardized.

2.3.2 Local and global contexts

Rolland and Monteiro [58] explore the balancing of the local and global in infrastructural information systems. Standardization and uniform solutions are motivated by the notions of rationalization, maintaining control, reducing risk, and curbing complexity. Developing and using a system based on standards enable coordination, which is important for management and control over distance. Standardization is necessary for a global system to work, but the literature shows that there is also a need to be sensitive to local contexts for a system to be successful (ibid.). Local contexts vary culturally and politically, and requirements and interests might differ. The balancing

of global standards and local variations should not be seen as a power struggle or a compensation for inaccurate design, but more as a necessary requirement for making it all come together. The tension will always exist, and because localization leads to multiple univerisalities, reaching for true universality is a futile project [29]. And so, Rolland and Monteiro [58] argue, the goal should not be to identify all the different ways global projects fail to meet local situations, but rather to analyze how global solutions are molded, negotiated, and transformed into workable solutions over time.

As a means for dealing with local universalities in my case study, Braa and Hedberg [29] have developed the principle of *hierarchy of standards*. The hierarchy of standards is based on a requirement for local contexts to follow global standards in collecting information while being able to extend the standards as to define their own information needs and their own local universality. The basic argument was that since it is not possible to agree on everything, we should agree on a basic minimum. The result was a hierarchy of (local) universalities, or a hierarchy of standards. Figure 2.1 shows a hierarchy of organizational levels where standards can be freely defined at all levels as long as they adhere to the standards set and inherited by the level above. Braa and Hedberg (ibid.) note that the hierarchy of standards has been important in lowering the tension between the levels in the standardization process.

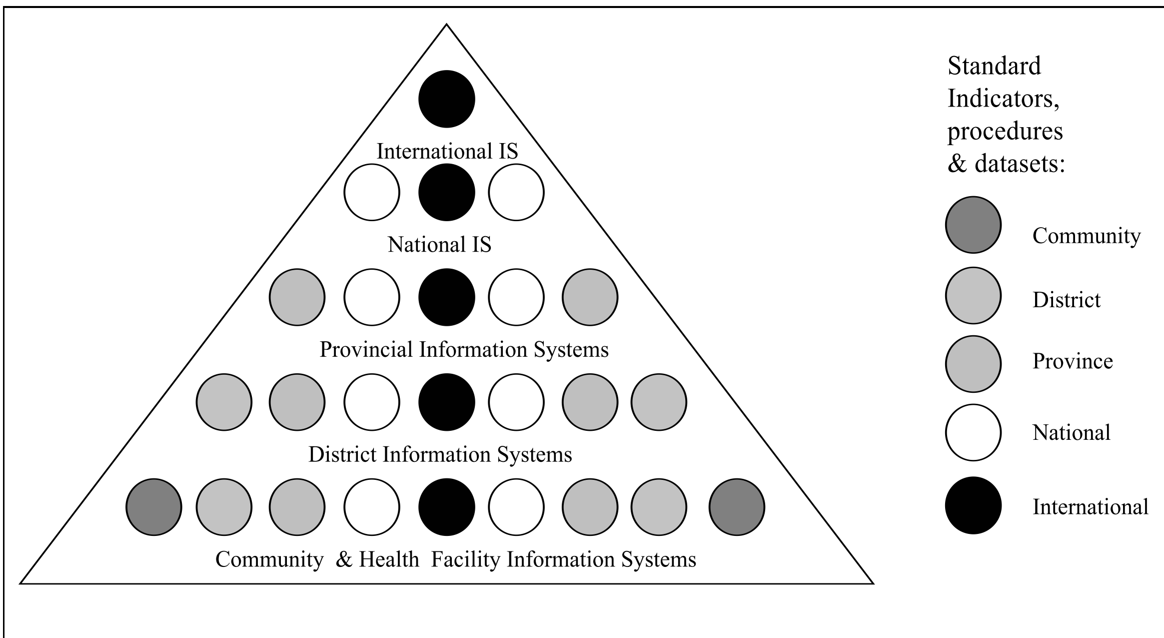

Figure 2.1: The hierarchy of standards. Each level is free to define their own extensions to the standards inherited from above. Copied with permission from [29].

The hierarchy of standards will not be directly used in the discussion in chapter 7, but it stands as an example of the importance of modularization.

Chapter 3

Background

This master thesis is based on my work in a global research collaborative called the Health Information Systems Programme, where most of my time has been spent taking part in the development of a computerized health information system called the District Health Information Software. My experiences from being one of the core developers of this system ground the empirical part of the thesis, where I seek practical, sustainable, and reusable solutions to the challenges of pulling together a medium sized enterprise application.

The following sections present the Health Information Systems Programme, the District Health Information Software, and where I come in with regard to these.

3.1 Health Information Systems Programme

We start with a quick look at the history of the Health Information Systems Programme in order to get a feel of how it all started, and then proceed to a short description of what it is today.

3.1.1 A quick look at history

After the fall of apartheid in 1994, South Africa was left with one of the least equitable health care systems in the world [29]. To improve on the situation, the new government launched a Reconstruction and Development Program, where one of the goals was to restructure the health care sector into decentralized health districts. Two years later, as part of this process, a pilot project for developing health information systems was launched in three districts in and around Cape Town in the Western Cape Province. The newly created Health Information Systems Programme (HISP), with people from the health care sector, the University of Western Cape, and the University of Oslo, took part in the pilot, seeking to identify the information needs

and to develop a computerized health care system supporting the new structures. The approach was

> "a slow incremental bottom-up process of aligning actors by enabling translation of their interests and gradually transforming social structures and information infrastructures where the resources already available form[ed] the base." [29, p. 116]

The team effort resulted in the development of an Essential Data Set, i.e. a minimal set of data to be captured in the districts, the hierarchy of standards, and the development of a District Health Information Software (DHIS) application supporting the implementation and use of such data sets. Because of the achievements in the pilot project, the Department of Health in South Africa adopted the HISP approach and the DHIS as the national standard in 1999, and it was implemented in all health districts in South Africa (ibid.). After the success in South Africa, HISP and the DHIS has spread to many other countries in Africa and Asia, including Mozambique, Malawi, Ethiopia, Tanzania, Nigeria, Botswana, Vietnam and India.

3.1.2 So what is the Health Information Systems Programme?

HISP is a research and education network. The University of Oslo and the University of Western Cape were part of the initial HISP team back in 1996, and since then academic institutions from several countries have joined. The field of research in HISP is information systems in developing countries, and both local and international master and PhD programmes are offered.

HISP is a health software development network [30]. HISP seeks to improve health information systems in developing countries, and utilizing computers is a natural step where applicable. After the success of the DHIS in South Africa, HISP has continued to develop the software, which currently comes in two flavors: Version 1.4, developed in South Africa using Microsoft technologies, and version 2, developed mainly in Norway, India, Vietnam, and Ethiopia using Java. Version 2 is mainly developed by master students doing their thesis in HISP. In addition, several other software projects have emerged as a result of local needs and lack of existing solutions. The DHIS has been free/libre and open source [59] from the beginning [29].

HISP is an implementation and support network. Evolving the health sector of developing countries requires not only usable information systems, but also people to implement the systems, train, and support the local health workers and administrators in the use of the systems. Sustainable health information systems is one of the key goals of HISP, and many of the master and PhD students continue their work with HISP after completing their degrees.

3.2 District Health Information Software

District Health Information Software (DHIS) is a computerized information system for handling aggregated health data. While a paper system provides fixed forms where output is the same as input—i.e. filled out forms—DHIS seeks to be flexible. When setting up the system for the first time, you define a hierarchy of administrative health units, denoted as *organization units* in the system, which represent the health structure in the country or region where the system is to be used. Each of these organization units can define *data elements*, which represent *what* data should be captured, for example the number of live births, or the number of vaccinated children. The corresponding *data values* are also tied to *periods* (*when*), for example January 2007, or Q1 2007, resulting in a unique combination of where, what, and when for each value. Furthermore, data elements can be freely grouped into *data sets*, which constitute the equivalence of paper forms containing data elements. This is a direct result of the development of the Essential Data Set during the pilot phase in South Africa. An Essential Data Set might change in time and geographical space, and thus needs to be flexible. Typically, one would create separate data sets for different health programmes, like HIV/AIDS, Malaria, RCH[1], etc. In addition, other types of logical groupings can be made as desired using the concepts of *data element group* and *organization unit group*.

Due to the hierarchy of organization units, the data values can be aggregated up in order to get the bigger picture for multiple facilities and geographical areas. One important point here is that anyone can do this aggregation and can, given that they have sufficient data, compare themselves to the broader situation and other organization units, typically on the same hierarchical level. The key notions are flexibility and local empowerment, which are some of the main goals in HISP [29].

In addition to data elements, a second, important metadata dimension exists. The *indicators* combine data elements in formulas, expressing interesting relations between different phenomena. A typical indicator could be the number of vaccinated childen divided by the number of live births. If multiplied by 100 this would give you the vaccination coverage in percent for a specific organization unit and period in time. Also, basing the indicator calculation on further aggregated data, as explained, will again give you the bigger picture.

Currently, the DHIS comes in two versions, version 1 and version 2. The following sections describe them both.

[1] Reproduction and Child Health

3.2.1 District Health Information Software version 1

DHIS 1 is based on Micrsoft Access and Visual Basic, and was started in South Africa in 1996. Since then, the system has grown into a quite comprehensive and complete DHIS, with a large userbase and solid foothold in about ten African countries. The latest major release was of version 1.4 in May 2006. DHIS 1 has been developed in South Africa based on Scandinavian traditions in information system development [29] of user participation, with one leading developer from Norway, and with the help of a few other developers over the years. One reason why the development team has not be larger, might be because of how the source code is stored as "modules" in the binary Access Database files (the data and the source code are both stored in the same binary file with an mdb extension) instead of being stored in plain text files. This makes it very hard for the developers to utilize any source code management system, which, in fact, has not been done, and leaves it up to the developers to manually cut and paste code between instances of the software (inside the mdb files after being opened in Microsoft Access). Still, the source code is fully open source and free of charge, but you do need a Microsoft platform and Microsoft software in order to read and use it.

DHIS 1 has received criticism for the choice of technology, for one thing in a report to the Mozambican Ministry of Heath. The dependency to Microsoft means that developing countries have to spend lots of money on proprietary software in order to use DHIS 1. In the Indian state of Kerala, the state officials had decided to use only free and open source software within the governmental sector, so DHIS 1 was not an option here since it requires Microsoft solutions. Furthermore, the awkward storing of the source code in binary files makes it hard to build a community of developers working on the same source code. It makes it harder to decompose the software into modules of different concerns—everything is basically in one binary lump (mdb)—which again makes it harder to customize and adapt the software to different contexts and requirements in different countries. People have also expressed a desire to use other database management systems (DBMS) than the Microsoft Jet Database Engine, like MySQL or Oracle. As more developing countries get faster and more reliable internet connections, centralizing the database and management of the systems starts to be a natural next step. DHIS 1 falls short of all this, and HISP saw the need for a new version of the DHIS being platform independent, DBMS independent, web enabled, modularized, and customizable with a central source code management system for keeping track of all the development. Meet DHIS version 2.

3.2.2 District Health Information Software version 2

DHIS 2 is based on Java and is developed using a lot of open source Java tools and frameworks. The project started in 2004 with master students as developers, and a new course at the University of Oslo was created in the Spring of 2005 to support the development and recruit new master students for the project. For the most part, DHIS 2 seeks to replicate the DHIS 1 functionality, but DHIS 2 is not a port of DHIS 1 into Java. Everything has been created from the ground up, and deviations from the DHIS 1 solutions do occur when the functionalities are discussed, rationalized, and translated into DHIS 2 functionalities. Using different base technologies allows for different possibilities, and different best practices. The development of DHIS 2 is coordinated from Norway and in most parts developed in Norway. However, there is an increase in development efforts from the developing countries where DHIS 2 is being used. The countries in question are India, Vietnam, and Ethiopia. The developers are typically master students, writing their thesis related to the HISP project.

We will noe look more into the general technical properties of the DHIS 2.

Web interface

DHIS 2 has a web interface, which means that it can be installed on a central server and used by multiple clients with a connection to the server. It is also possible, however, to install the system on a single computer, running both the server software and the client software (the browser) on the same machine. The last approach is the most common one so far because of limited bandwidth and the lack of stable connections, but there are tendencies towards wanting centralized solutions. One of the benefits of a central solution is that the management of the system will be centralized too. Also, a clinic might not have to report to its parent organization unit if they both use the same server. The web layer is based on an open source web framework called WebWork, which uses the Java Servlet API, defined by Sun in the Java Servlet Specification [61], to communicate with the web server waiting on requests. WebWork adapts XWork, a generic command pattern framework, to the web tier. The idea is to translate textual commands, or HTTP request, into Java action executions and return a result based on the outcome of the execution. In most cases the result of an action execution is a dynamically created web page. DHIS 2 uses Velocity, a template engine, for this. Using these frameworks in DHIS 2 has not been straight forward, as DHIS 2 is modular, and XWork has had problems with being configured from muliple sources. The desire to make all web pages of DHIS 2 have the same look and feel without duplicating work has also been a challenge. The web layer of DHIS 2 will be covered in much greater detail later.

Modularization

One of the most important goals of DHIS 2 was to modularize it. As recognized by Rolland and Monteiro [58] and Braa and Hedberg [29] one solution cannot possibly work in all situations. Thus DHIS 2 has been organized to consist of a larger set of modules where one can choose which modules should be part of the final system. Being able to choose different modules in different countries or contexts is almost necessary for the system to be accepted in the first place. The countries implementing the DHIS are independent and have different requirements, but at the same time might have the same integration needs with other systems, both internally and across countries and organizations. A few modules define the core functionality, and multiple user interface modules build on top of these—possibly combined with other service modules. Maintaining a common core yields standarization of concepts and data representations, which can be extended with custom service modules as necessary by local requirements, and used in various ways by user interface modules. Good modularization reduces the overhead and duplication of work regarding multiple flavors of the same system.

Because the development of the system is distributed, modularization improves the ability to coordinate and manage the development [58]. Until now, core modules have mainly been developed in Oslo, while developers in Ethiopia, India, and Vietnam have worked on context specific modules and other customizations of the system. So far, most of the context specific modules have been included as standard modules too, as they have been of interest to other implementation nodes in the HISP network as well. Several tools and frameworks play together regarding the modularization in DHIS 2, making the chosen modules interact seamlessly. Later chapters are dedicated to this subject, describing the matter in detail.

The modules are organized, both horizontally and vertically, in the three general application layers of web applications [49]: Persistence, service/business, and presentation (see figure 3.1). The presentation layer is the web interface layer just mentioned. It should be made as thin as possible, containing only functionality related to the presentation, so that, if it was to be replaced, all business functionality would still be in the application. The business functionality should be located in the service/business layer. This functionality is typically guided by use cases [48], and a strategy for designing methods is that one method implements one use case. In DHIS 2, no clear strategy has been chosen, but most use cases are handled by one method. The persistence layer does exactly what the name implies, it persists objects without any further manipulation. The layer functionality contracts of the service and persistence layers are defined in Java interfaces [49], visible as thick lines in figure 3.1. The object model is used by all layers and thus does not belong to any specific layer. In DHIS 2 there are also support modules for all layers providing common functionality,

implementing cross-cutting concerns like transaction management, security, and the like.

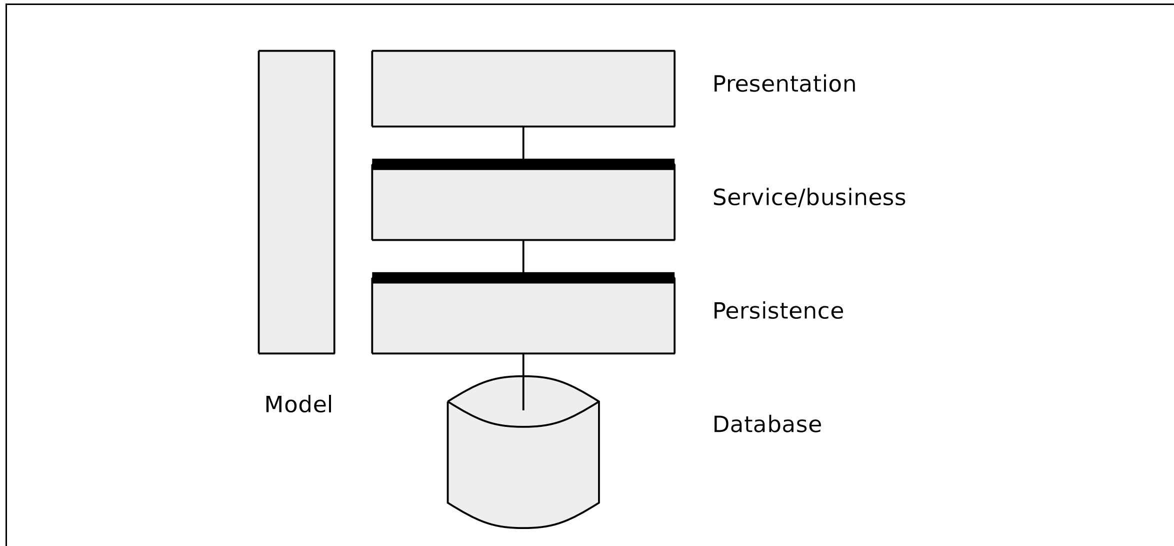

Figure 3.1: The layered architecture of DHIS 2; the classical three-layer architecture.

Database independence

Apart from being platform independent, one goal of DHIS 2 is to be database management system independent. Countries taking DHIS 2 into use might already have production databases which they are comfortable with and wish to continue using. Having DHIS 2 independent makes it possible for the system to run on any preferred existing solution. Although this is a goal, DHIS 2 has not been tested on other database systems than MySQL [14] and PostgreSQL [17], which are the two most commonly used open source database management systems. Currently, DHIS 2 is running with PostgreSQL in Vietnam, and MySQL in India. From a technical perspective, the DBMS independence is provided by an object-relational mapping framework named Hibernate [8]. Hibernate provides a generic layer on top of all its supported database systems, where one can configure which driver and connection URL Hibernate should use, and which built-in SQL dialect Hibernate should use to communicate with the database. In most cases this generic layer is good enough, but in some cases more advanced queries are required, and one needs to be careful not to break the support for the desired database systems implementing the advanced queries. In DHIS 2, we leave the configuration of the database to the implementors of the system, so that DHIS 2 can be easily adjusted to the local situations. Object-relational mapping comprises the task of translating between an object model in an object-oriented environment to relations in a relational database.

Source code management

The DHIS 2 approach to source code management is shared with most other open source projects. A centralized management system is used with which the developers synchronize their copies of the source code. In DHIS 2 we use the Subversion [22] version control system. A version control system is a tool which keeps track of all versions of files and directories, called revisions, so that it is possible to, for example, look at the history of a document, or go back in time and recover "lost data" if needed. Subversion holds the revisions in a central repository created with the Subversion administration utility, where the size of the repository is held down by Subversion storing only the differences between each revision. So when a file is accessed, it is first assembled before it is returned to the user. Access is done through checkouts from and commits to the central repository, typically over a network connection. A checkout results in a local copy of the latest revision at the time of the checkout, which the developer can work with. When the developer feels that enough work has been done to constitute a new revision, she commits her changes back to the repository. Others can then update their working copies with the latest changes in the repository. Conflicts can occur when people change the same lines of the same files, but I will leave that subject to the Subversion documentation. A more well-known version control system is CVS (Concurrent Versioning System) released first time in 1986 [39], which Subversion seeks to surpass [33].

Although a repository can be used by one person for, for example, personal documents, I would like to stress the important aspect of collaboration using such a tool. The possibility to look at the document history is more like a bonus compared to the offered collaboration support. Without a version control system, if more than one person is working on a set of text files, they have to manually copy each other's changes into their personal copies of the documents. Mess is lurking in the bushes, and snippets are easily lost if one is not careful. A tool like Subversion collects all the individual changes, automatically merging them into complete documents, and making sure nothing is lost. It is an essential tool when developing computerized systems like DHIS 2.

A source code management tool is also important for maintaining context specific modifications to a system. While modularization takes care of course grained customization, there is often a need to make smaller configuration changes, like for example setting the default system language. By making a copy of a module which needs smaller internal modifications in the repository, one can use the source code management tool to maintain the local changes while still keeping the module up to date with the latest changes from the main development branch. Merging changes, or copying changes, from one branch to another is a central aspect of a source code management tool. So far, this strategy for maintaining context specific modifications

has not been used in conjunction with DHIS 2. I have mentioned it on the developer mailing list (see appendix B) without much response, except for a private positive one from another core developer, but nothing is stopping the local teams from using this strategy if they would like to.

3.3 Where I come in

I entered the stage first time in January 2005 when I took the INF5750 "Open Source Software Development" course at the University of Oslo. I was not aware of where that would lead me later—I chose the course because it looked interesting. At that time DHIS 2 was nearly started, and only basic functionality was present in the system. There was no user interface, only the decision that it should be web based and which framework the web layer should use for producing dynamic web pages. During the course, I and the other group member created a data entry module with cusomtomizable entry forms. I found the module was fun to create, as we managed to create something working and at the same time useful and wanted. In addition, I got to know a bit more about system development and the different computer languages, tools, and framework in use. The course triggered my interest in the project, and when I was asked to be a lab assistant for the same course the following semester, I accepted gladly. Although the module was a success at the time, there was no strategy for merging the module with modules developed by the other students, and because there were no guidelines on how the web pages should look, all modules had their own style. So in the end, none of the modules developed during the INF5750 course spring 2005 ended up as part of DHIS 2.

Only after the course I really got to know the core modules of the system and how to utilize the different tools and frameworks. I gradually took more part in the development of the core modules and eventually got the title "core developer". Because of my experiences with the web layer, I was asked to create a web portal, collecting all the web modules and make it look like one larger application. It is the work on the web portal that forms the basis for this thesis. By late 2005, we had a working portal, and in early 2006 the first milestone of DHIS 2 was released. Kerala was in great need of a platform independent DHIS, and was pressuring HISP for a release of the first version. Around this time I found myself in Saigon, Vietnam, to help the Vietnamese HISP team in learning and taking part in the development of DHIS 2. The release of milestone one took place in the middle of the Vietnamese two week celebration of new year, Tết, which basically ment that most of the holiday was spent coding on the portal and on the first web modules to be released.

I have participated in the development of DHIS 2 quite heavily. I have touched most aspects of the system, including the main model and basic functionality for

operating on the model, database configuration and transaction management, user administration and security, internationalization, the web portal and common functionality and looks for web modules, and technical administration of the server hosting the DHIS 2 development tools. I have taken initiatives to larger refactorings to clean up the code and to avoid design "mistakes" which otherwise would have made further development more complex and cluttered. Because of my engagement and experiences with DHIS 2, I have also worked as a consultant for other developers working on the same or different modules, offering my opinion about good or bad solutions, or to simply help out in the case of a less obvious problem or bug. HISP has invited me to join them on various trips abroad; one to Addis Abeba, Ethiopia, to give the Ethiopian team a quick start in using DHIS 2 technologies, one to a workshop in Cape Town, South Africa, in relation to the first international DHIS conference, one to Cape Town in relation to an OpenMRS[2] conference, and one to Geneva, Switzerland, to meet UNAIDS [25] and discuss collaboration between UNAIDS and HISP, and communication between DHIS and the Country Response Information System (CRIS) developed by UNAIDS. I am naturally grateful for everything I have been invited to take part in.

[2] OpenMRS is an open source patient based information system [15].

Chapter 4

Research methods

The HISP activities of implementing health information systems, and in particular, computerized health information systems, require software to implement. The circumstances which lead to the DHIS being developed and widely adopted have lead to a demand for dedicated and skilled developers who can keep up with the requirements, produce new functionality, and in general provide the software system which HISP thinks is the most appropriate according to the action research conducted in parallel. The development model in DHIS 2, using primarily master students, is one of the research areas in HISP, but the software development in itself is also research [52].

This thesis is a case study [68] of DHIS 2, rooted in software development. My main types of questions are, as set out by the research objectives, “how” and “why”, and because I take part in my case study as a developer, it is of the exploratory kind (ibid.).

Marcos [52] compares software development with software engineering research in order to establish a parallelism between the two. The parallelism is based on a set of general research method steps which are applicable to any scientific process. For each of the steps, Marcos compares the research methods of software enginering science and software development. An overview of the steps and the related software development methods follows:

- Documentation: The body of knowledge which defines the situation in which action is to take place. For software development this includes the field of application, known techniques for solving problems, and so on.
- Problem determination: A software developer must analyse the field and identify necessities, leading to hypothesis creation.
- Hypothesis creation: The hypothesis describes the object to be constructed, which in software development corresponds to setting up a specification of requirements.

- Definition of the method: A software developer must choose a set of tools and principles with which the problem is to be solved such as object-orientation, extreme programming, and methods for adapting to the problem.

- Resolution, validation, and verification: Through case studies and creativity, prototypes can be developed and tested, which again might lead to iterations with the previous step. While action research is a common research method within software engineering, Marcos points to its similarities with extreme programming, which might have a better fit with software development depending on the context of the developing.

- Analysis: Checking the hypothesis corresponds to comparing the developed solution with the requirements.

- Final report: The developed solution must be appropriately documented, for example as manuals, guides, and similar.

As the empirical part of this thesis, by and large, is grounded in software development as set out by the first research objective, my research method is based on the presented list of research activities. All of the requirements for the solutions are grounded in discussions with other core and main developers and development coordinators of DHIS 2. The discussions have taken place on the developer mailing list, through instant messaging chats and group discussions, and in face-to-face meetings. Sadly, not all of it has been logged in the process, but the outcomes of the conversations have been incorporated into the solutions. The technical solutions have been grounded in object-oriented programming, aspect-oriented programming, information hiding, and the languages already in use such as Java and XML. This document does not contain the full technical documentation of my developments, though, as it would take up a lot of pages and contain technical details which are very specific to the DHIS 2 case, and not immediately relevant to the overall solutions and discussions.

My approach to software development has been based on extreme programming (XP), which is a set of principles and practices for guiding development [31]. XP is based on four principles: Simplicity, communication, feedback, and courage. Simple designs take less time to develop and are easier to change than complex designs, and change is likely to happen. Constantly refactoring the solutions to keep them simple minimizes the cost of future changes (ibid.). Communication between developers and customers is important, but communication is also about keeping code simple, as simple code is easier to communicate than complex code. Unit tests [45] can contribute to the communication as they show how the functionality they test works, in addition to that they provide immediate feedback on the correctness of the code [31]. From a requirements perspective, XP encourages short release cycles for rapid

feedback, in order to minimize wasted development time. Courage is about choosing simple solutions instead of designing for future changes and additions. It is part of the goal of short roundtrips from development to feedback (ibid.). In a few cases I might not have been this courageous, as I have incorporated flexibility for possible additions through means of configuration.

I also draw upon my experiences from working with the international DHIS 2 teams. I spent the first three months of 2006 working with the Vietnamese team in Saigon as part of a capacity building project which Nordal [53] and Øverland [64] have covered in detail. During this stay, I got to learn how the distributed development model of DHIS 2 plays out in practice in relation to the importance of modularity on multiple levels and the challenges of being more or less experienced with the modular way of thinking. I would not use the term action research on the work I did in Vietnam, as I did not directly play the reflective card of action research on the Vietnamese team, but rather worked with them and helped them out in their search for understanding, learning, and contribution to the system development. At the same time, I developed some of the solutions presented in this document. Similarly I have had the pleasure of working with the Ethiopian and Indian teams, though both in Oslo and using electronic tools, communicating and developing requirements, suggestions, and solutions. Furthermore, participating in the execution of the INF5750 “Open Source Software development” course at the University of Oslo has greatly contributed to the overall experience, seeing the solutions being used in practice, and the confirmation of the importance of modularity, understanding, and in the end good, sustainable solutions.

Chapter 5

Concepts, tools and frameworks enabling modularization

In order to be modular, DHIS 2 uses a set of tools and frameworks which indirectly play together to create an application which can be assembled as desired by developers around the world. Each module defines a set of dependencies to other modules which are automatically included if the depending module is included to be part of the final system. This means that some requirements are built into the modules to make them work properly, but by using object oriented methods and a framework for wiring the system together during startup, even required module dependencies can be replaced with other dependencies implementing similar functionality. This provides full flexibility with regard to assembling the system before deployment, and allows for locally customized modules to replace general modules.

The following sections describe the different concepts, tools, and frameworks that are part of the modularization, and how they play their parts. They are the building blocks in DHIS 2, setting the stage for solving problems, and it is thus important to understand how they work and how they contribute to and hinder modular solutions. They are, in the order presented, Maven, Inversion of Control, the Spring Framework, and WebWork.

The choice of tools and frameworks was taken before I became a DHIS 2 developer in the spring of 2005. Thus the tools and frameworks presented in this chapter have counter parts which might as well be better choices now than what they were then. I will not discuss the alternatives as such, but rather point to them where they could have been used instead, or are otherwise interesting with regard to the context.

5.1 Maven

> "Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information." [12]

Maven is a command line tool for managing Java projects, not unlike Ant [2] combined with Ivy [10]. It is platform independent—it is written in Java[1]—and it is hosted by The Apache Sofware Foundation. Maven seeks to help developers "comprehend the complete state of a development effort in the shortest period of time" [12]. Maven does this by providing conventions for organizing project source code, resources, and documentation, and by providing extensible project management through functionality plugins. Basically, all of Maven's functionalities are implemented as plugins, Maven just orchestrates the execution of them. An example of such a plugin is the core Maven Compiler Plugin which takes care of the key task of compiling the Java source code of a project. We will bump into a few more plugins later.

The central piece of information mentioned in the quote, the project object model, is an XML file residing in the top directory of the project as it is placed in the file system. I often refer to this file as the project descriptor file. A Maven 2 project is simply defined by a directory containing such a project descriptor file, with a sensible content. Figure 5.1 shows the basic structure of a Maven 2 project. In addition to the mandatory components, this project has the base directory for source code. We will see later that not all projects have source code.

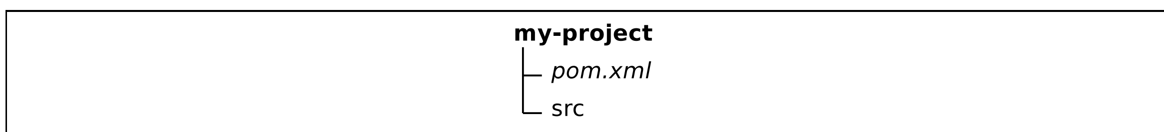

Figure 5.1: The project object model, or `pom.xml` file, in the top directory of a Maven 2 project. The `src` (source) directory is the base directory for all source code in the project.

Two versions of Maven exist. Maven 1 was used for managing DHIS 2 during the first year of development, but at that time only the most basic Maven features where used due to the lack of complexity in DHIS 2. Maven 2 is a great improvement over Maven 1 and is currently the version of choice, so no time will be used on discussing Maven 1 features and problems. For ease of read, Maven 2 will from now on be addressed simply as Maven. I am not going to go into detail on all of Maven's features,

[1] It is perfectly possible to create systems in Java which are not platform independent. Just hard-code a reference to e.g. `C:\` and the system is immediately locked to the Microsoft platform.

only those which are relevant to the context. The relevant features are basically the mechanisms for source code organization, packaging, system modularization, and module dependencies, all described in the following sections.

5.1.1 Source code organization

Maven provides conventions for organizing source code. The rationale is quite simple; if everyone uses the same basic structure for their projects, less time is wasted getting to know other projects. The conventions are defined by the Maven Super POM [28] which all project descriptors inherit automatically. It is possible to override these to fit custom structures, but in DHIS 2 we use the defaults. The conventional source code directories are as follows:

- `src/main/java`: The *Java source code* is located in the main java directory and consists of Java files and their package directories. All the Java source code in a Maven project is located in this directory, and nothing else. Maven does not force you to follow this rule, but there is no reason not to if you want a nice and tidy project where files can be looked up quickly.

- `src/main/resources`: The *application resources* are located in the main resources directory and consist typically of configuration files, language files, and other necessary information for the system to run properly. As an example, the application context specification (the `beans.xml` configuration file in DHIS 2) is located within this directory. All framework configuration files are typically also located here. The resources directory can be omitted if there are no resources, but this is rarely the case.

- `src/main/webapp`: The *web application resources* are located in the main webapp directory and may consist of HTML templates, style sheets, JavaScript files, images, and everything else that is related to and needed by the web interface of the application. This includes the special WEB-INF directory and web.xml file as defined by the Java Servlet Specification [61, Chap. 9]. The webapp directory can, and should, be omitted if the Maven project does not have a web interface, if it is, for example, a library, a service module, has a Swing user interface, or other. Basically, only projects packaged as web archives should have this directory. Web archives will be discussed later.

- `src/test/java`: Java classes for unit tests go into the test java directory. The unit tests are automatically executed by the Maven Surefire Plugin when building the system, but the files are never included in the final build as they do not contribute to the functionality of the system—they just test the functionality.

- `src/test/resources:` If the unit tests require special configuration or the like, then these files go into the test resources directory. These files take precedence over files with equal names located in the main resources directory. As with the actual unit tests, these files are not included in the final build of the project.

The use of any of these directories is optional, but normal projects usually have the two first, at least. Maven does not complain if there is no Java source code in the project, in that case you just do not get any compiled Java classes in the final build. Figure 5.2 shows the three main directories with some exemplary content.

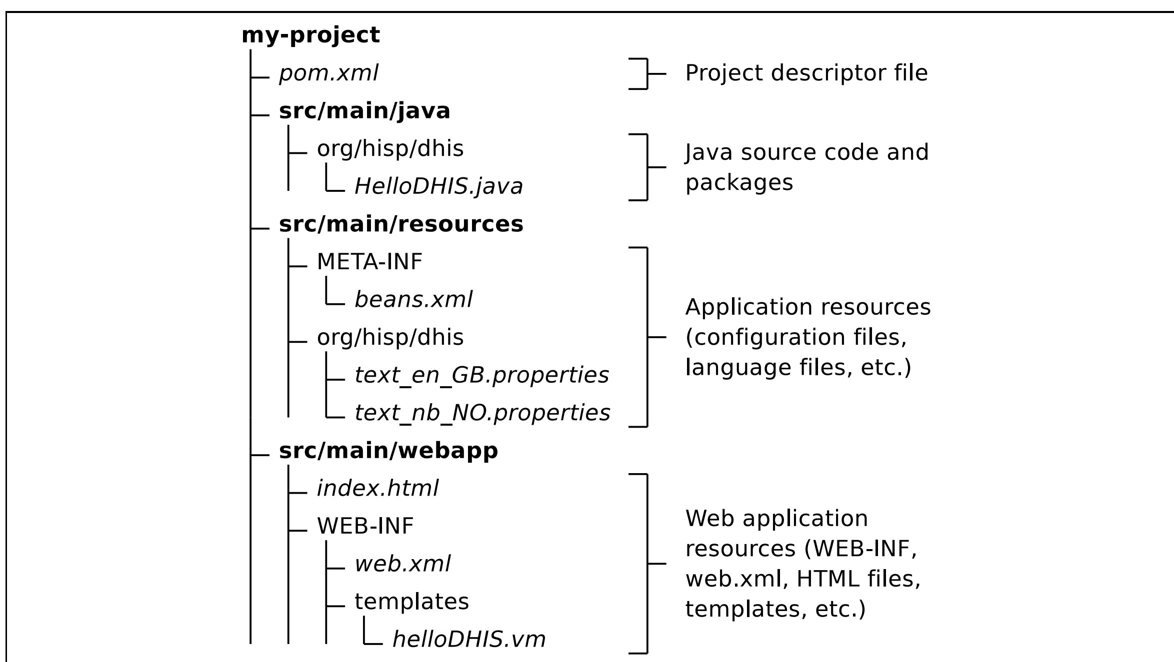

Figure 5.2: A Maven 2 project with the conventional main directories and some exemplary content. Files are in italic.

Figure 5.3 shows how all the conventional source code directories look in the Eclipse SDK. The directories are set up as build directories in Eclipse so that when files are edited they are automatically compiled, if needed, and copied to the target directory. The target directory is the output directory by the Maven conventions, and contains all the products of the source so that the products, for example the compiled Java classes, do not interfere with and mess up the source code. It is possible to generate Eclipse project files from the Maven project descriptor file by using the Maven Eclipse Plugin so that you do not have to set up the Maven conventions in Eclipse manually. The separate `src` directory visible in figure 5.3 gives access to directories not set up as build directories.

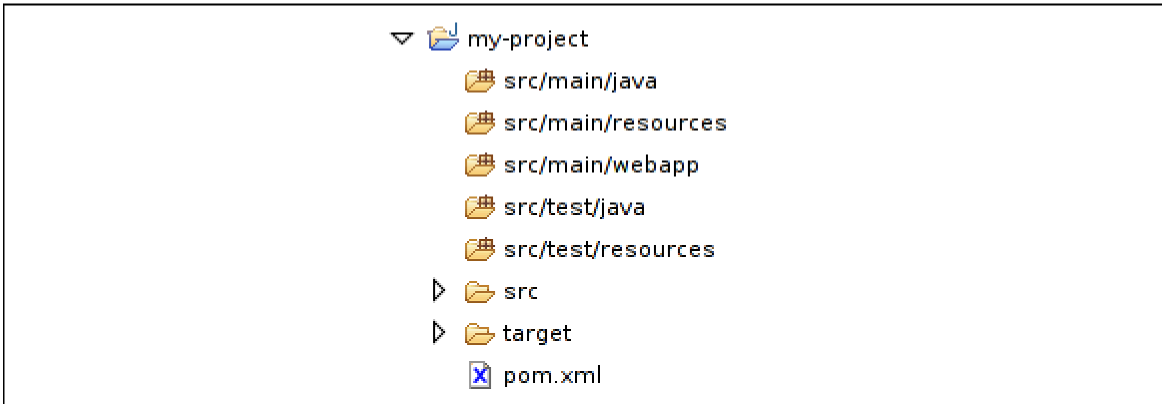

Figure 5.3: A Maven 2 project with all conventional source directories as seen from the Eclipse SDK's [6] Package Explorer.

5.1.2 The correlation between Maven projects, Java archives and web archives

This section describes how Maven projects in general, when built and packaged, are turned into Java archives (JAR files) and web archives (WAR files). The understanding of this is important for understanding the comming sections on building and putting together the DHIS 2 web portal. The Java archives are described first, then the web archives.

The main purpose of Java archives is to ease distribution of classes and resources by collecting them in a compressed file. A Java archive is a zip file, but with the .jar extension instead of .zip. The content structure of the Java archive is defined by Sun [62]. The transformation from a Maven project to a JAR file is simple, and is taken cared of by the Maven Jar Plugin. The archive is not made for web interfaces, so any `src/main/webapp` directory in the Maven project is just ignored. The other two, the Java source code and the resources, are compiled, merged, and zipped together without any additional structuring (see figure 5.4). The Maven specific directories, `src/main/java` and `src/main/resources`, are not included—it is the *contents* of these directories, which are included. Thus, if you look inside a Java archive, you will find the compiled classes from the `src/main/java` directory and the resources from the `src/main/resources` directory directly in the root of the archive.

A web archive is also a zip file, but with the extension .war instead of .zip. A web archive can contain a complete web application which can be deployed directly to web servers which know how to handle these archives. The content structure of the web archive is also defined by Sun [61]. The main difference from a JAR file is that the WAR file can also contain the web application resources and other JAR files, thus the structure is different (see figure 5.5). The web application resources from `src/main/webapp` are located in the root of the web archive. These files can

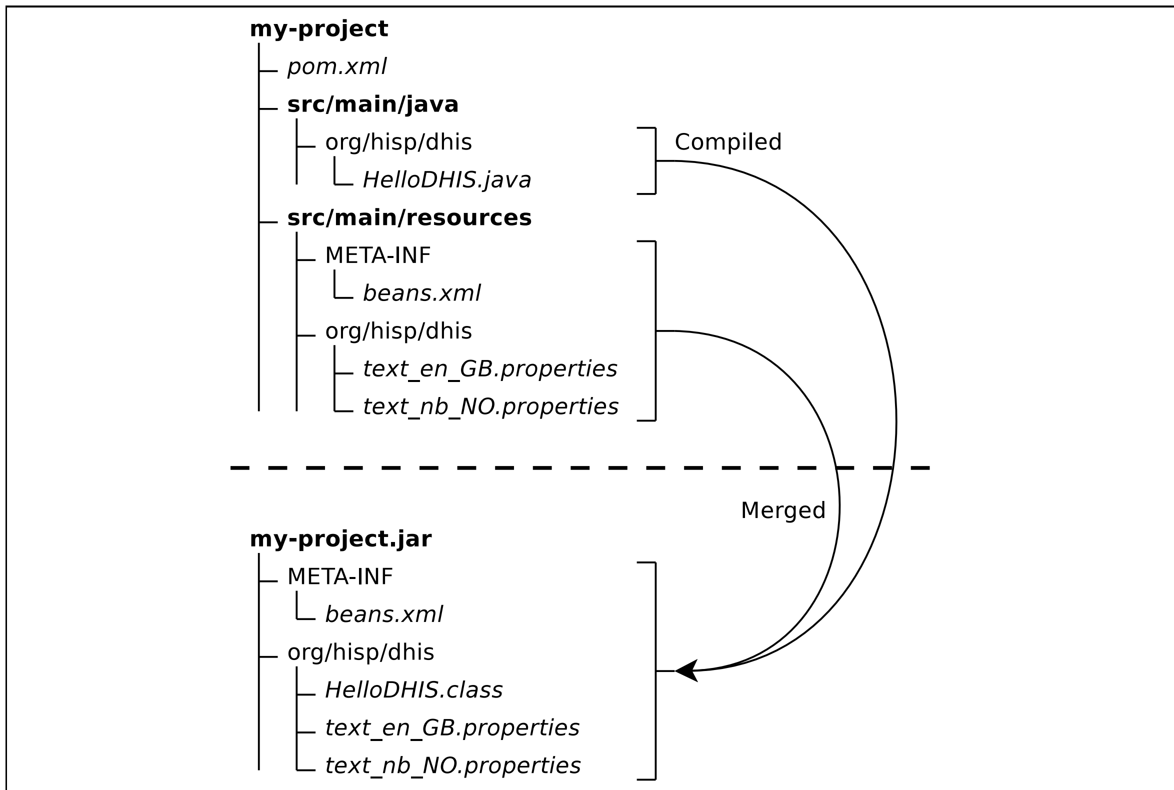

Figure 5.4: How a Maven project is packaged into a Java archive. Left out from the figure is a Manifest.mf file, which is automatically created and placed in the META-INF directory. Also, Maven puts the pom.xml file in a subdirectory of META-INF.

be directly returned to a client by a web server supporting WAR files. Everything else, which should not be returned by a web server is located inside a special `WEB-INF` directory in the root of the archive. The compiled classes and resources are put in the `/WEB-INF/classes` directory. Notice that the contents of `/WEB-INF/classes` are exactly the same as the contents of the JAR file in figure 5.4. Additional JAR files, on which the application depends, can be put in the `/WEB-INF/lib` directory, and a web application deployment descriptor file is required to exist and be named `/WEB-INF/web.xml`. The corresponding location for the `/WEB-INF/web.xml` file in a Maven project is `src/main/webapp/WEB-INF/web.xml`. It is not required that the `/WEB-INF/classes` directory contains any classes. The packaging of a Maven project into a WAR file is performed by the Maven WAR plugin.

An example should make it clearer how Java archives and web archives are used together: Say that you have two Maven projects, one service module which is packaged as a JAR file, and one web interface module which is packaged as a WAR file. The web interface module has a dependency to the service module, and so the service module is packaged first, then the web interface module, where the service module

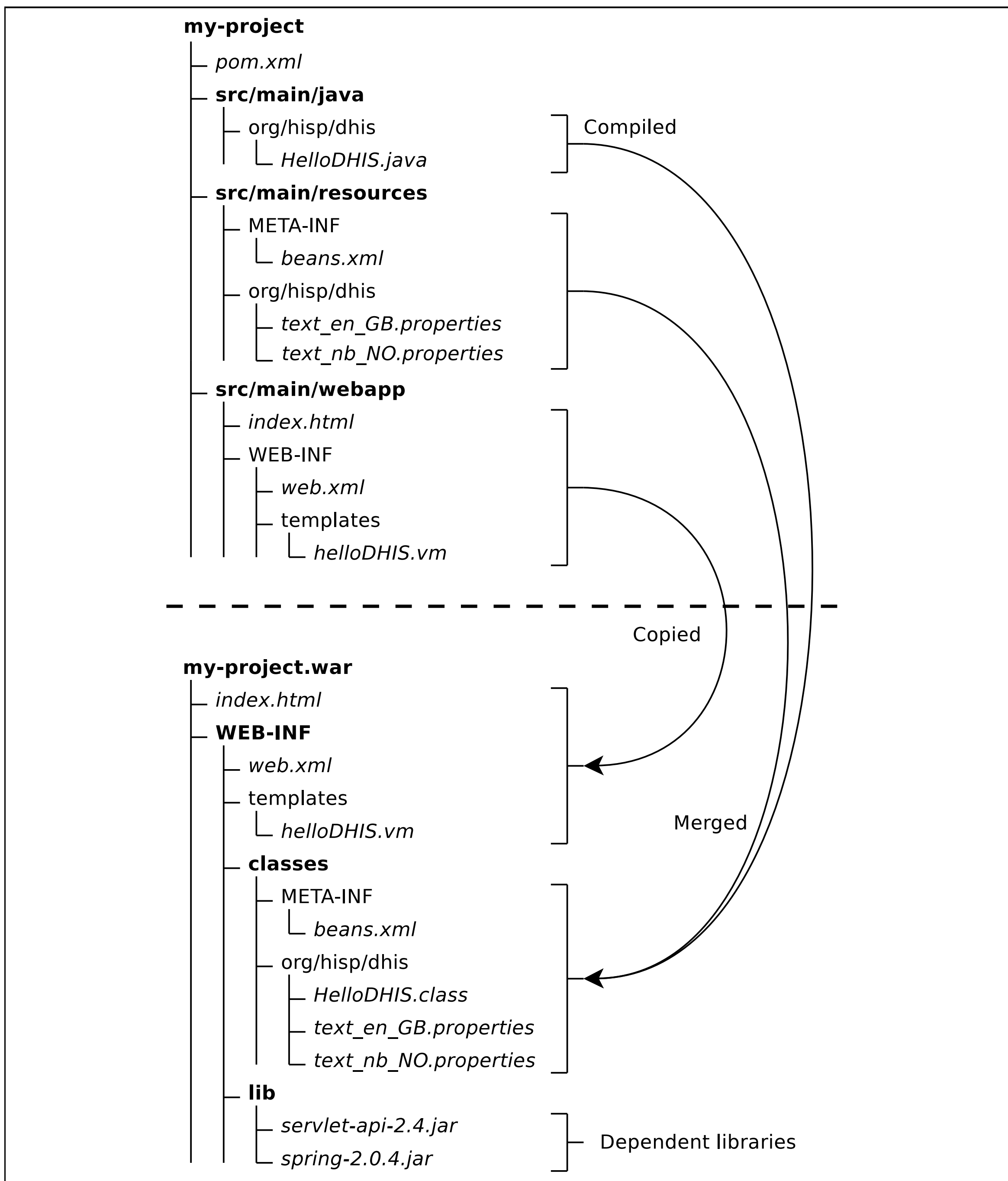

Figure 5.5: How a Maven project is packaged into a web archive. The lib directory contains libraries which the application directly or indirectly depends on. Left out from the figure is a Manifest.mf file, which is automatically created and placed in a META-INF directory in the *root* of the WAR file. Also, Maven puts the pom.xml file in a subdirectory of that same META-INF directory.

ends up in the `/WEB-INF/lib` directory of the web archive. This makes the WAR file self-contained, and can be deployed in a suitable web server.

An important issue about JAR files and WAR files, which is not related to Maven, is that only JAR files can be used as Java functionality libraries. It is not possible to make use of the Java code in a WAR file from other Java and web archives. Only a servlet container[2] knows where the compiled Java code of a WAR file is located. So if one wants to make a support project containing both common functionality and common web application resources, one actually needs to create two support projects: One for the functionality, which must be packaged as a JAR file for the functionality is available to other modules, and one for the web application resources, which must be packaged as WAR file. How to include web resources from one WAR file into another is discussed in a later section.

5.1.3 Modularization using Maven

As already explained, a Maven project is basically a directory with a project descriptor file and source code organized in special subdirectories. A project can also contain sub-projects, breaking a system up into a hierarchy of projects and modules (see figure 5.6). I tend to use the terms project and module interchangeably, but strictly speaking, a system module is *defined* by a Maven project, and because projects containing sub-projects cannot contain source code, not all projects constitute modules. In the end, only modules are in the classpath[3] of a running system.

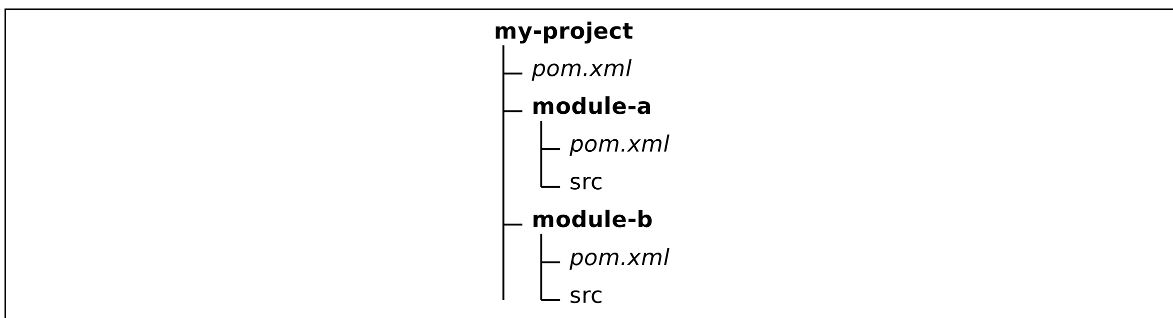

Figure 5.6: Basic structure of a parent project containing two sub-projects with source code defining one system module each.

Notice the important difference between the hierarchy of projects and the hierarchy of system modules. The projects are in an administrative, preferably logically and semantically organized hierarchy, where the sub-projects can inherit the configuration in the project descriptor files of their parent projects (figure 5.7a). The hierarchy of

[2]See the glossary for a brief explanation of the servlet container.
[3]See the glossary for a brief explanation of the classpath.

the system modules is, however, defined by the compile-time and runtime dependencies between them (figure 5.7b). This distiction is not made easier by that Maven denotes projects in the hierarchy of projects as modules—project modules, that is, not necessarily system modules.

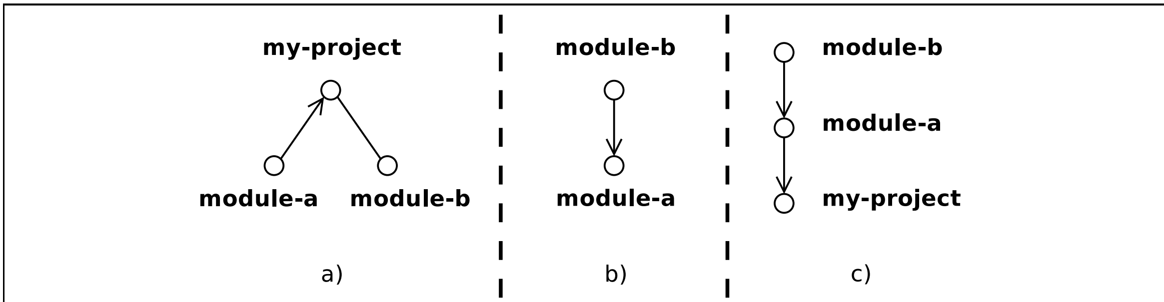

Figure 5.7: a) shows a simple project hierarchy with one parent project and two project modules. Module-a is defined to inherit any configurations from the parent, but module-b is not. b) shows the system module hierarchy of the same project. Module-b is dependent on module-a at both compile-time and runtime. c) shows the combined dependency hierarchy defining the order of compilation of the projects into modules (bottom-up).

It is possible to specify the inter-module dependencies, across the project hierarchy, in the project descriptor files located in the root of the project directories. Maven will handle the dependencies, automatically building and packaging the projects in a depth first traversal of the system module hierarchy, extended with the dependencies from child to parent in the project hierarchy (figure 5.7c). Circular dependencies will not work, making Maven complain and stop.

In addition to handling inter-module dependencies, Maven also handles external dependencies where libraries, like the Spring Framework, can be automatically downloaded from a central repository available on the Internet [36] and installed locally for use with the projects. Transitive dependencies are also dealt with, spoiling the developers who hardly ever need to do anything except listing the libraries the different projects directly depend on in the project descriptors. Actually, Maven does not differentiate between dependencies to system modules and dependencies to external libraries in the dependency configuration. When issuing the build command on a project, all sub-projects are build too, and missing dependencies are attempted downloaded.

By using Maven's support for modularization, one can quite quickly create a multi module project with both inter-module and external dependencies handled automatically by Maven. Projects and sub-projects are just regular directories and subdirectories, and all the configuration is done in the project descriptor files. There

are no restrictions on where a project might be located in a computer, all it takes is a directory.

5.2 Inversion of Control

Inversion of Control (IoC) is the concept of extracting control, or knowlegde, out of classes implementing system functionality to make the classes know only what they need to know in order to fulfill their intended tasks. A class should only do one thing, and hopefully do it well. When implemented correctly, classes will have low coupling between them, meaning that they can easily be taken out of context to, for example, be replaced, tested in isolation, or reused elsewhere without affecting other parts of the system. The key is *separation of concerns*, and used together with Maven, IoC provides a complete solution for modularization of system functionality.

There are two complementing strategies for implementing IoC; dependency injection and Aspect-Oriented Programming (AOP). None of these replace Object-Oriented Programming (OOP), but rather build on and complement OOP. They will be described in turn.

5.2.1 Dependency injection

Dependency injection is the concept of extracting instantiation of classes and hard-coded references and values out of the classes, and injecting these from the outside. For example, a business layer class might need to persist model objects using a persistence layer class. The business layer class does not need to know *how* the persistence layer class does this, it only needs to know *what* the persistence layer class is supposed to do by knowing which methods are provided. If the persistence layer functionality is given by an interface, and there are different implementations of the interface, the business layer class does not need to know which implementation it actually is invoking methods on because the interface defines the contract. By letting some outside functionality inject a reference to the desired implementation of the persistence layer interface into the service layer class, the latter will be happily unknowing of which implementation it receives, given that it only uses the interface internally. Regarding modularization, this allows for multiple implementations of the same interfaces to be placed in separate modules, and depending on which modules are included in the final system, different implementations might be instantiated and injected into the depending classes. On the technical side, there are three approaches to implementing dependency injection, which can be mixed as desired:

- *Setter injection* is the most commonly used type of dependency injection, and works by having one set method, often called a setter, for each dependency to

outside functionality. It is then simple for any outside wiring functionality to set the references between the different implementations by using the set methods.

- *Constructor injection* works by having the constructor receive all the dependencies needed. The benefit of this compared to setter injection is that the class will be ready to use after instantiation, while with setter injection, the class will not work properly before its set methods have been called. The downside of using constructor injection is that you cannot have cyclic dependencies—one class has to be instantiated before another.

- *Interface injection* works by letting the interfaces decide which dependencies a class needs. It is then up to the wiring functionality to detect which interfaces different classes use and give them the right object references.

In all cases mentioned, there is a need for some central functionality, instantiating all the implementations and wiring them together as needed. One solution is to do this programmatically, for example in the main class when starting the system. But this might make it hard to modularize the system as a replaced module typically means replaced implementations, which the wiring functionality needs to adjust itself to. A better solution is to define the dependencies in configuration files, using IDs to match implementations and dependencies. For example, one configuration file might define that the service layer class mentioned earlier has a dependency to an implementation with ID "persister". Another configuration file might then map "persister" to a specific implementation of the interface the dependency requires. The wiring functionality can then easily match these. Also, the module containing the interface implementation can easily be replaced by another module with a different implementation, and with a different configuration file mapping its local implementation to "persister". Naturally, the wiring functionality must either require all ID mappings to be unique, or have some defined way of handling multiple implementations mapped to the same ID.

A slightly different approach to dependency injection is *dependency lookup*, where the classes are themselves responsible to ask a central piece of functionality maintaining all the implementation instances in order to satisfy their own dependencies. The approach is often refered to as the service locator pattern. A downside of this pattern is that all classes are tied to the service locator, and thus cannot as easily be taken out of its context to be tested in isolation or reused elsewhere. In most cases, dependency injection is prefered to dependency lookup.

5.2.2 Aspect-Oriented Programming

Aspect-Oriented Programming is the concept of extracting cross-cuting concerns, logic that applies to multiple objects and methods, into separate modules/classes of code. Take the case of transaction management. In a persistence implementation there is a need for starting a transaction before, and ending the transaction after persisting data to a database. The procedure is the same for every method persisting data. By defining the transaction management as an aspect, mapped to the relevant methods, one can simply provide the same functionality before and after each method call while having the all the code in one place instead of all over the place. The aspect can be implemented as a method interceptor, applied using AOP proxies, which are available in Java 1.5 and later, and third party libraries like the Code Generation Library (cglib). The interceptor will then contain all the transaction management logic necessary, and the persistence implementation will never know about it or any transaction management. The persistence implementation is then able to focus on its intended job only, namely persisting data. By using similar configuration methods as with dependency injection, one can declaratively apply interceptors between dependers and dependencies, across modules, striving for the ultimate separation of concerns.

DHIS 2 uses Inversion of Control heavily, in all parts of the system. As one might expect, there are multiple frameworks available providing IoC capabilities, for example the Spring Framework [19] and PicoContainer [16]. AspectJ [4] should be mentioned as one of the most comprehensive AOP frameworks for Java. Otherwise, the Spring Framework is possibly the most popular framework implementing all the IoC concepts, and which supports AspectJ as well. DHIS 2 uses the Spring Framework both for dependency injection and AOP.

5.3 The Spring Framework

The Spring Framework is a lightweight application framework, which means that you use it for implementing the structure of an application. Although Spring provides support and best practices approaches to lots of other tools and frameworks, Spring's most important feature is its container for wiring the application together using IoC. Classes are mapped up as *beans* in XML configuration files, where one bean can have dependencies to other beans, using all types of dependency injection. The word bean originates from the JavaBeans™API specification by Sun.

One important aspect of the bean configuration is that is can be spread among multiple configuration files, which is more or less required in a modularized computer system. Typically, you have one configuration file in each module, mapping up the

functionality that the module provides. Spring will automatically find and load all configuration files based on a set of resource paths given at startup. Combine this with Maven, and you get dependencies on two levels. First, the Spring configuration contains bean to bean dependencies, but the dependencies can typically be fulfilled by any implementation of the right interfaces—so the mapping is indirect. Second, the project dependencies in the Maven project descriptors tell the system which modules are present at compile time and runtime, and thus affect which beans are matched by the indirect mappings of the bean dependencies. This weighs for putting the interfaces and implementations in separate modules, so that a set of implementations can be easily swapped with another set of implementations by simply changing a project dependency, while having the interfaces fixed and independent of the swap. During the development of DHIS 2, we have had a few discussions on where to put the interfaces. The solution so far is one DHIS 2 API module which contains all the layer boundary and commonly used interfaces together with the object model, and then have interfaces which are local to modules in the modules themselves.

5.4 WebWork

To combine Java and web, Sun Microsystems defined the Java servlet. Being about 10 years old, the definition of a Java servlet has been refined, and is currently

> "a Java™technology-based Web component, managed by a container, that generates dynamic content." [61, p. 19]

The mentioned container is part of the web server which receives the HTTP requests, and transforms the requests into an object format as defined by the Java Servlet Specification. The requests are then redirected to servlets, which process the intentions of the requests, and return appropriate responses, typically dynamically generated web pages.

Many implementations of web servers with servlet containers exist. The most known and popular one in the open source world is probably Apache Tomcat [3], which was originally developed by Sun as the reference implementation of the servlet and servlet related specifications. In the DHIS 2 development, Mortbay's lightweight web server called Jetty [13] is the most commonly used servlet container. The reason for this is a Jetty plugin for Maven, which allows for easy deployment of a Maven web project. With a simple Maven command, the module is built and the Jetty plugin starts Jetty and deploys the WAR file automatically. In production, however, Tomcat seems to be preferred by the DHIS 2 implementors[4].

[4]A Maven 2 plugin for Tomcat is currently under development [24], which deploys web archives through the Tomcat manager web interface

Servlets are simple and to the point, which means that there is no built-in support for input validation, input type conversion, internationalization, etc. Sun provides the Java Server Pages (JSP) specification for creating dynamic web pages based on servlets, but developers must discipline themselves to make such pages usable and maintainable. A servlet is by definition not thread safe—all requests to the same servlet share the same servlet instance, and the complete responsibility for structuring the application wisely is on the developer, JSP templates offer no guidelines or constraints. So rather than implementing servlets directly, larger projects introduce a web framework which builds on top of the Java Servlet Specification, and helps to structure the application and support common tasks. Well known web frameworks are, among others, Struts [21], WebWork [51], Struts 2 [20], which is based on the two just mentioned frameworks, Spring MVC [49], Tapestry [23], Wicket [27], and Java Server Faces [11]. DHIS 2 uses WebWork.

WebWork, developed by OpenSymphony, is based on a generic and extensible command pattern framework called XWork. What XWork basically does is to transform string commands into action executions. This is quite analogous to executing Java code based on URLs. WebWork provides web specific extensions to XWork which transforms it into a web framework. So while the framework is called WebWork, the action configuration file is called xwork.xml.

WebWork receives the HTTP requests from the servlet container through a servlet filter. Typically, all URLs ending with .action are sent to WebWork. WebWork will then look up the right action class to execute based on the action configuration from xwork.xml. A new instance of the action class is created for each request, making it thread safe, and all the URL parameters are automatically matched to developer defined setter methods in the action class, type converted accordingly, and set. The action class is then executed by a defined method which processes the values and prepares any output values in object members backed by corresponding getter methods. Finally, the action class returns a status code, which WebWork compares to result mappings in the configuration. When a matching result mapping is found, WebWork executes the result, which typically consitutes executing a template engine like Velocity or FreeMarker on a given template, and then returns the product to the client.

WebWork has built in support for the Spring Framework, which means that action classes can be defined as beans in Spring, and WebWork will ask Spring for the specific action instances to execute. The great benefit of this is that it makes the wiring of the web interface layer with the rest of the application no different from any other part of the system, and the action classes can receive the same benefits as any other Spring managed bean. This also makes it up to the developers to decide whether an action class is a singleton or not. While Spring provides AOP support, WebWork has its own

built in support for interceptors. The interceptors are collected in interceptor stacks which are executed around action executions. A lot of the XWork and WebWork functionality itself is implemented as interceptors, for example the functionality for converting and setting the HTTP request parameters in the action classes before they are executed. The XWork configuration allows for full flexibility in which and where, and in which order, the interceptors are applied. It is also simple to implement custom interceptors and add these to the interceptor stacks.

The XWork configuration supports defining multiple packages with action mappings. A package can inherit configurations from another package, which allows for centralized base configurations. Packages can also be namespaced, which makes a modular web interface easier to maintain. Although the configuration files have working support for modularization, the XWork configuration loading mechanisms have had issues. During the development of DHIS 2, different versions of XWork have been used with variying support for multiple configuration files. General customization of the XWork configuration loading has been required in order be able to detect and load each web module's configuration file. Extendability has been poor too, resulting in, at the time of writing[5], a complete copy and internal fix of one of the XWork provided classes.

That concludes the presentation of the various concepts, tools, and frameworks which form the base for developing modular solutions in DHIS 2. The next chapter describes the requirements, development, and issues of the DHIS 2 web portal.

[5]The currently used version of XWork is 1.2.1, used by WebWork 2.2.4. Latest versions are XWork 1.2.3 used by WebWork 2.2.6.

Chapter 6

The DHIS 2 web portal

The DHIS 2 web portal is a web interface module, but the name has broader meaning. According to Webster's Online Dictionary [26], a portal is "a grand and imposing entrance". In the field of computing, web portals are entry points to web sites (ibid.), but a portal can also be a web page containing *portlets*:

> "A Portlet is a multistep, user-facing application delivered through a Web application. [. . .] Hence, a portal page can contain a number of Portlets that users can arrange into columns and rows, and minimize, maximize, or customize to suit their individual needs." [34, p. 628]

The DHIS 2 web portal is not based on portlets. The DHIS 2 web portal is about collecting all the different web interface modules into one complete web application and providing a common front page to the users. This includes packaging all the Maven modules into one big WAR file, and creating a menu which gives access to the functionality in the different modules. The web portal is also about applying a common look to all the pages, like a common header, the module menu, module specific menus, style sheets, etc. The DHIS 2 web portal has evolved quite a bit, from a single module solution, via custom Maven plugins, to the current three module solution. The Maven project called dhis-web-portal is the final module which, when packaged, contains the whole DHIS 2 system in a web archive.

I use the terms web modules and web interface modules. I define a web module as a module related to the web layer of the system. Thus, a module packaged as a JAR file which contains common web functionality for use by other web modules is a web module itself. A web interface module is always packaged as a WAR file, and can be deployed to a servlet container for separate execution. It is possible to have WAR files which are not meant for direct use in a servlet container, for example a module containing web application resources like templates which are common to other web modules, but which does not contain anything sensible to deploy in itself.

In DHIS 2, there are all these types of modules. There is a JAR file which contains common functionality, there is a WAR file which contains common web resources, and there are lots of web interface modules which can be run separately and make sense in a servlet container. The web portal module is also a web interface module, but one which includes everything from the other web interface modules. Figure 6.1 shows web interface modules and a common web module, which, by applying the web portal, appear as one application to the users.

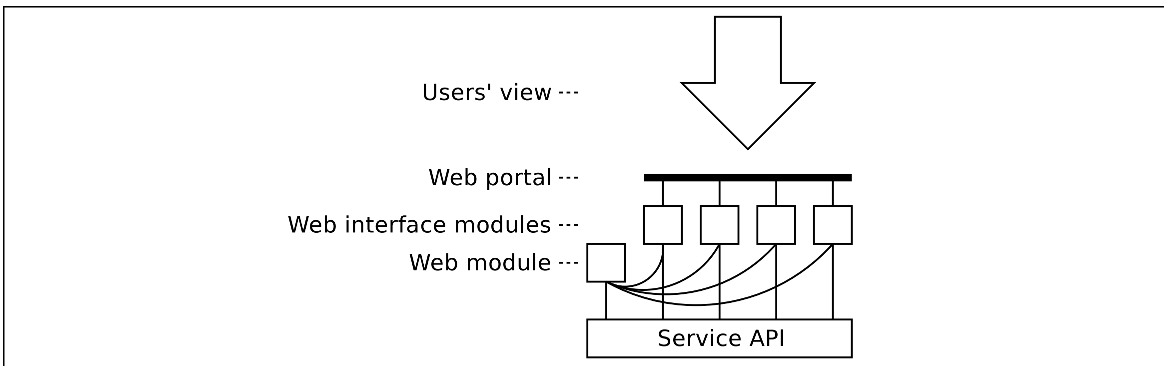

Figure 6.1: Conceptual image of web interface modules and a common web module, which, by applying the web portal, appear as one application to the users. The web portal as a web interface module in itself is not shown in the figure.

As mentioned, DHIS 2 consists of several separate web interface modules. Splitting up the user interface into separate modules adds flexibility for composing the final system. It was also set as a requirement, by one of the core developers, that each web interface module should be standalone so that it could be executed and tested in a web browser separately from all other web modules. Not having to build the entire application makes development of modules a more independent process, thus less influenced by bugs in other modules, plus it shortens the round trip from developing to testing.

In all, three major versions of the DHIS 2 web portal have existed. All three have been developed by me, and are described next in two sections; Assembling the portal and Creating a common look.

6.1 Assembling the portal

The first attempt at creating a web portal was made in late October 2005. The requirements were misunderstood, and the portal was never actually used in DHIS 2. At this point in time, Maven did not support dependencies from web archives to web archives, only from web archives to Java archives. So the challenge of merging separate web modules into one big web application was solved by letting all the web

modules be packaged as JAR files, and then have one Maven project packaged as a WAR file, called the web portal, including all these JAR projects in its `/WEB-INF/lib` directory when built (see figure 6.2). This worked, except that web modules packaged as Java archives cannot be executed separately in a servlet container. And because the web application resources of web modules are usually not packaged with a JAR file, the project descriptors of the web modules were altered so that the web application resources would be treated the same way as the regular application resources; be placed in the root of the Java archive. This again lead to the problem of having web application resources inside the JAR files. By the WAR file specification [61], web application resources are to be stored directly in the web archives, not inside JAR files in `/WEB-INF/lib`, so the servlet containers could never find the HTML files, style sheets, JavaScript files, and images from the various web modules. Thus, a custom resource response servlet was made to handle the clients' requests to these resources.

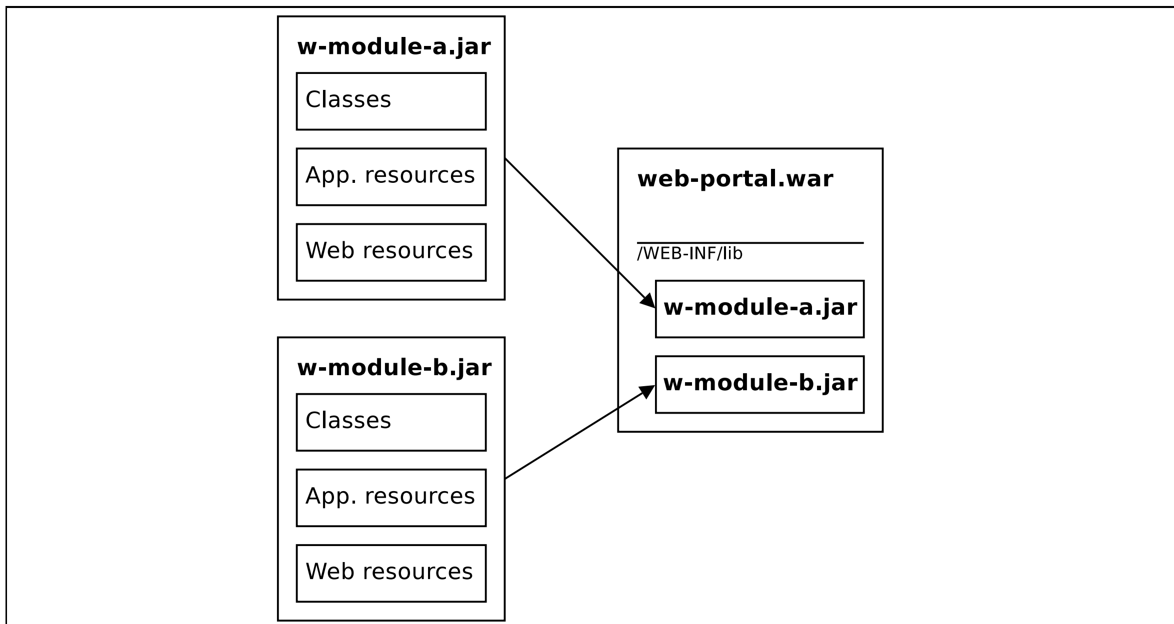

Figure 6.2: Conceptual image of how the web modules, packaged as JAR files, were merged into a web portal WAR file. The web resources are typically not included in JAR files.

One positive effect of packaging the web modules as JAR files was that the WebWork configuration files—the xwork.xml files—would not conflict with each other when the web portal was build. Even if the xwork.xml files were located in the same directory in every web module, the files would not overwrite each other when the modules were merged because the configuration files were packaged in JAR files first, with the different names of the modules, and then placed in the libraries directory of the portal WAR file. The positive effect had a backside, and that was that WebWork

did not know how to find these configuration files when located inside the JAR files. WebWork was hard-coded to look for only *one* xwork.xml file, located directly in the `/WEB-INF/classes` directory of the portal WAR file. A custom configuration provider factory was made, locating all the xwork.xml files and feeding them to WebWork's configuration manager[1]. This specific solution did not work when the portal was ran *exploded*, i.e. when the WAR file is unpacked by the servlet container when deployed. Not because it was not possible, but because the problem was not solved back then.

The Spring Framework bean configuration files—the beans.xml files in DHIS 2—are also located in the same directory in all modules. As with the xwork.xml files, they too ended up in their respective JAR files, but unlike WebWork, Spring knows how to locate them all, regardless of how the WAR file is deployed.

Again, this web portal was never used, because it did not allow for web modules to be deployed separately. Also, the need to have a custom servlet for loading web content instead of letting the web server do its inteded job, was not particularly compelling. A new solution was needed. So the prerequisite for the next attempt at creating a web portal was that all web modules must be packaged as web archives, including the portal itself. A solution was created in November 2005. Maven did still not support dependencies from web archives to web archives—the Maven WAR plugin was in a beta version, and the support for dependencies to web archives came with the 2.0 release of the plugin in May 2006.

Because WAR files cannot contain other WAR files, the only way to create a portal from multiple web archives is to unpack all the archives, merge the files, and pack everything together again into one big WAR file (see figure 6.3). A custom Maven plugin was made, which replaced[2] the Maven WAR plugin in the package phase of the Maven build cycle of the web portal project. The plugin worked by specifying the dependencies to the web modules in the plugin configuration section of the portal pom.xml file, and when executed, located all the web module builds and unzipped them into the build directory of the portal, before zipping it all up into one big WAR file.

A couple of special cases had to be taken care of. As mentioned earlier, both the xwork.xml file used by WebWork and the beans.xml file used by Spring were in the same directory in every web module, so if they did not get special treatment, they would overwrite each other during the great merge of the modules. This was solved by prepending the module name to the corresponding xwork.xml and beans.xml files,

[1]Details: A custom ConfigurationProvideFactory located the xwork.xml files, created instances of WebWork's XmlConfigurationProvider class and gave these to WebWork's ConfigurationManager.

[2]The custom plugin did not completely replace the Maven WAR plugin. The Maven WAR plugin executed first, and then the custom plugin was executed, overwriting the assembled WAR file made by the Maven WAR plugin. A way to replace the Maven WAR plugin completely in the build phase was not found.

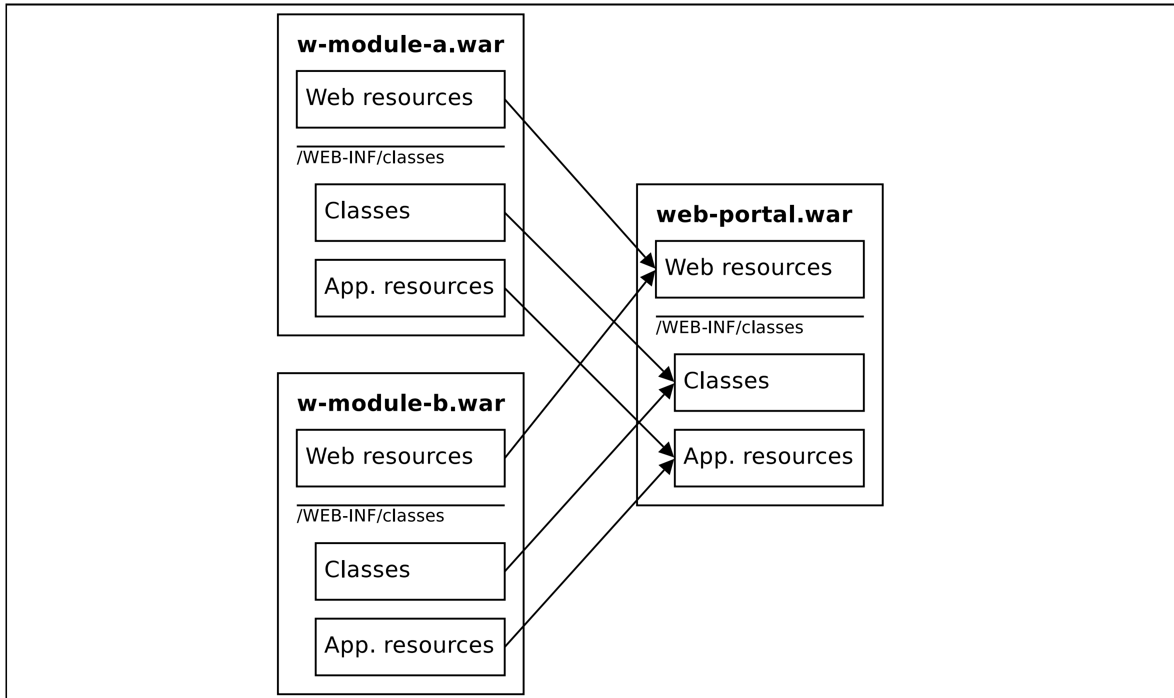

Figure 6.3: Conceptual image of how deployable web modules were merged into a web portal WAR file using the first custom plugin for Maven. Configuration files with equal paths in the web modules had to be treated specially so that they would not overwrite each other during the merge.

and then in the end create one xwork.xml and one beans.xml which included all the others using already supported include elements in the XML formats. For Spring, this did not change much—Spring would find the beans.xml file as always—but for WebWork this meant that the custom configuration locator and provider from the first web portal solution was not needed. WebWork was able to find this single xwork.xml file by itself, and then find all the namespaced xwork.xml files by parsing the first one.

The downside of this solution was that, due to technical details, only the web portal could use the plugin. The plugin could only be used once in a build sequence because of how the namespacing of the beans.xml and xwork.xml files was executed. Also, the plugin configuration was relatively ugly, considering that Maven should have dealt with the dependencies. So only the web portal project had this configuration, and thus could use the plugin upon building. The result of this was that all the web interface modules needed to contain all the web application resources they needed when deployed separately. All common templates, images, style sheets, etc. had to be placed in every single web module—the most obvious violation of the DRY[3] [45]

[3]Don't Repeat Yourself

principle. Any update to a common file had to be manually copied to all the other modules, which is simply wasting time and space in addition to being more error prone. Only later, with the third web portal solution, could we collect common resources in a common web project.

In May 2006, between milestone 3 and 4 of DHIS 2, a new plugin was created with a completely different approach than the previous one. Instead of trying to replace the Maven WAR plugin in the packaging phase, it was made to operate in the *process resources* phase as a supplement to the build process. This time around, more of Maven's provided functionality was used, making the plugin significantly smaller and simpler. Regular Maven dependencies were used in the project descriptor file. The WAR dependencies were ignored by Maven, as the Maven WAR plugin still did not support WAR file dependencies, but the custom plugin managed to get hold of them through Maven's services. While the previous plugin had to take special measures to handle the application configuration files, the new plugin solved this by splitting all the WAR dependencies into two: The web resources and the rest. The web resources were merged with the portal as before, only now in a much earlier stage and using the conventional directories by Maven, which the Maven WAR plugin also use. The rest, which is what is packaged in a JAR file if the Maven project is packaged as a JAR file, was, in fact, packaged as a JAR file by the custom plugin and placed in the `/WEB-INF/lib` directory of the portal (see figure 6.4). The result of this was that the beans.xml and xwork.xml files ended up in JAR files with the names of the modules instead of conflicting upon a direct merge of the files, the same as with the first portal solution. The plugin configuration was also reduced to a minimum, and placing it in a parent project descriptor file for all the web modules meant that all the web modules could have dependencies to other WAR projects without any other configuration than the inherited plugin configuration. With the possibility of having dependencies to WAR projects from all web modules, came a central WAR project containing all the templates, images, style sheets, etc. common to the web modules. A great step forward in the portal solution.

With the xwork.xml files being back in the JARs as with the first portal solution, a special WebWork configuration file locator was again needed. In the coming versions of XWork, this turned out to not be a completely smooth ride. XWork's later configuration file reader has not supported the necessary path type required to address the configuration files. And because of poor extendability of the reader class, a copy of the source code had to be taken and modified directly.

A couple of days after the new plugin was taken into use, the first stable (by definition) version of the Maven WAR plugin was released, version 2.0. This version had support for merging WAR files, but because it could not handle the collision issue of the application configuration files it could not be used. So while waiting for better

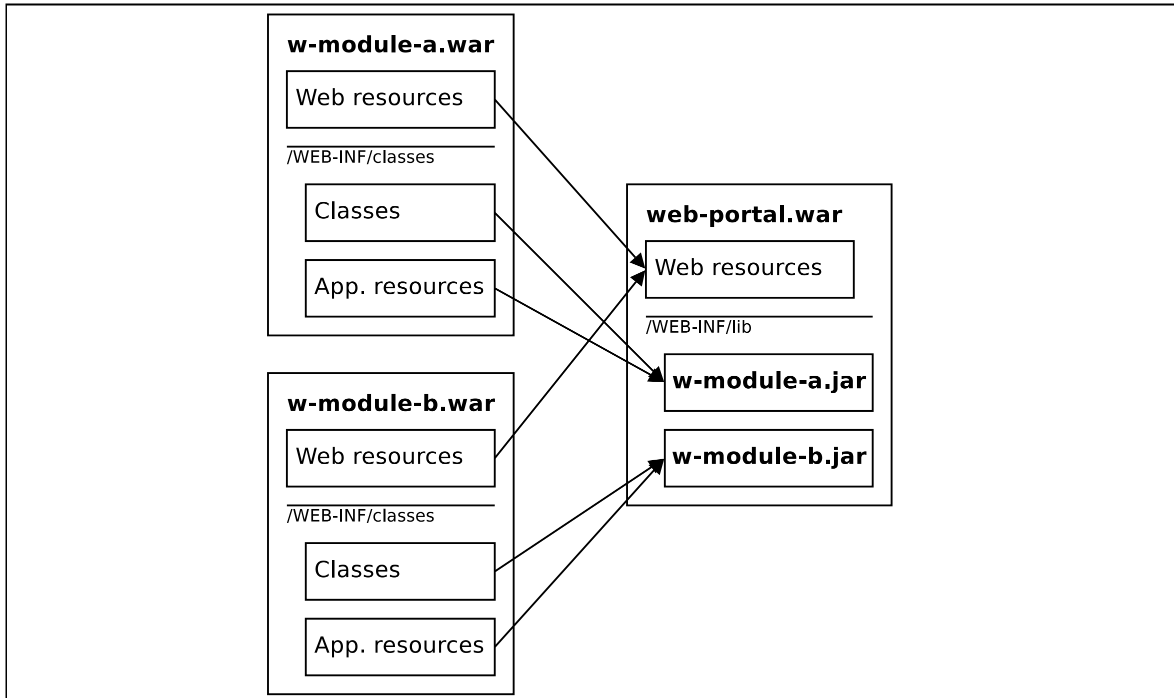

Figure 6.4: Conceptual image of how deployable web modules were merged into a web portal WAR file using the second custom plugin for Maven. The classes and application resources were packaged in a JAR file and placed in the libraries directory of the web portal, keeping all configuration files separate without any special treatment.

alternatives, the custom plugin was kept, and the beta version of the Maven WAR plugin was forcibly used in the Maven configurations, because it simply ignored the WAR file dependencies.

Then, in early June 2006, I came across a feature request in Maven's bug tracker regarding how WAR files are packaged (see Appendix A for a screenshot of this issue). The suggested feature was similar to what the latest custom DHIS 2 plugin did, to move the Java classes and resources into a JAR file in `/WEB-INF/lib`, only this time as part of the web project packaging stage and not as part of the web portal merging (see figure 6.5). I commented the issue, directly based on the experiences from DHIS 2, and voted for it in hope of increasing its weight and focus, as the feature would mean that no custom plugin would be needed in DHIS 2:

> "This feature makes it easier to overlay war files if you have files with the same name and path in the war projects you want to merge (e.g. in src/main/resources). With this, these files will end up in their respective jars instead of replacing each other during the overlay process (take the application context descriptor as an example)" [comment 3 in the Appendix A screenshot]

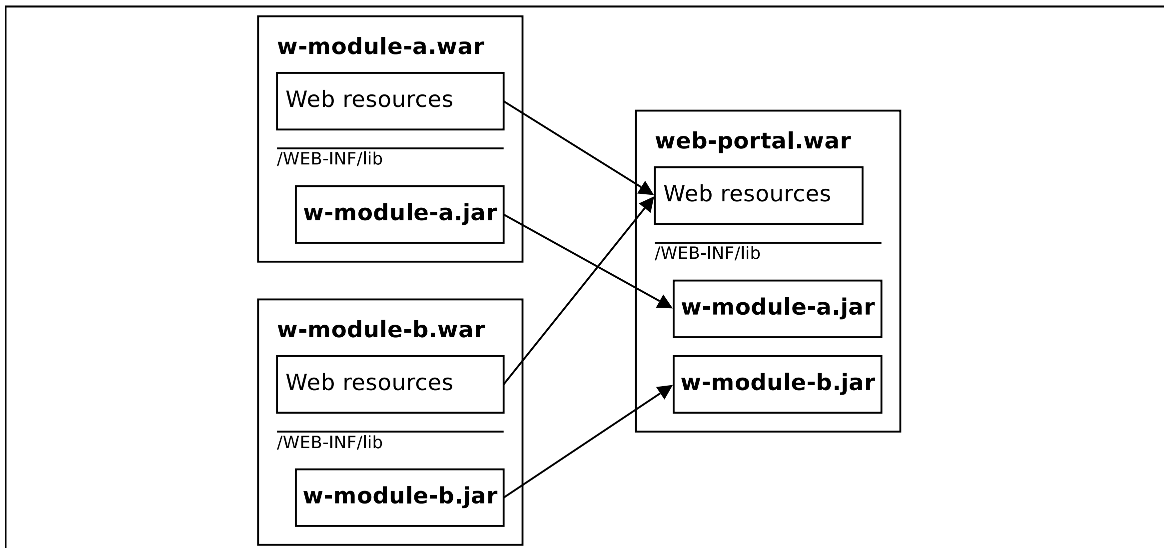

Figure 6.5: Conceptual image of how deployable web modules are merged into a web portal WAR file using the latest Maven WAR plugin. The classes and application resources are packaged in a JAR file when each web module is packaged, so that no special treatment is needed when merging the modules into the web portal.

Three short weeks later, the Maven WAR plugin version 2.0.1 was released with the new feature, and a couple of days later all custom DHIS 2 plugins were removed from the web module configurations, contributing to the infrastructure. So now all web interface modules are packaged as WAR files with their own Java classes and resources inside JAR files in `/WEB-INF/lib`, and no special care has to be taken when web archives are merged.

6.2 Problems with the current solution

As the size of DHIS 2 has grown—by more web modules and more external libraries—the build time has increased. Not just because of the relatively large number of archives, but because the unpacking, merging and re-packing require huge amounts of processing and hard drive operations. The common web resources project, dhis-web-commons-resources, has no direct dependencies to other modules or libraries itself, but inherits a few from its parents, giving it, at the time of writing, 35 JAR files in `/WEB-INF/lib`—a total size of 9MB. Each web interface module depends on the common web resource module, so each time any of these projects are build, the common web project is unzipped, merged, and zipped with the dependent project, 9MB each time. The resulting packaged web modules vary in size from 9 to 24 MB. At the final step of building, the same process is repeated with the web portal, except that the web portal depends on all the already packaged web interface modules, in addition

to the common web resources module. This amounts to approximately 160MB of zip files being unpackaged, merged, and packaged with the web portal module, with the current default set of web interface modules (M8-SNAPSHOT). On a computer with limited resources[4], building the web layer of DHIS 2 takes time. With the web portal being approximately 29MB after build, it is obvious that there are a lot of common libraries in the web interface modules. This is the effect of wanting every web interface module to be deployable separately—they have to contain most of the same libraries as each module is based on the same service layer and support structure. If the web archives did not contain their dependencies, this would not be an issue.

Related to the fact that WAR files contain their dependencies is the fact that if you change a service module which several web modules depend on, either directly or indirectly, you need to rebuild all the WAR modules in order to know which version of the service module ends up in the final portal. This is because, when the web portal is built, all these dependencies are merged together in the `/WEB-INF/lib` directory, and there is no control in the Maven WAR plugin on which build of the same module ends up in the libraries directory during the merge. It takes time to rebuild all the web modules just to test a small change in a browser. This has been frustrating to developers, but not all these developers have been aware of the fact that each web module can be deployed separately—you do not need to build and run the web portal to test a change which only concerns one of the web modules.

The Jetty Maven plugin supports running the servlet container directly on the source code, which means that in a normal project you can start the servlet container, change a template or a JavaScript file or similar, and then simply refresh the browser to see the changes immediately. In DHIS 2 this does not work because, in this mode, Jetty only looks at the source code of the web module in question, and not the source code from other dependent WAR files. All the web modules in DHIS 2 depend on the dhis-web-commons-resources module which contains the common main template, the default style sheet and so on. So when running Jetty in "directly-from-source" mode, the common web resources are not included, breaking all the web pages. Thus, a web module must always be packaged before deploying in a servlet container so that all dependent WAR files are merged with the module before launch.

To summarize; the current DHIS 2 web module solution has long round-trips from development to testing in a browser because all web interface modules need to be rebuilt if a common dependency is changed, and because all web interface modules depend on at least one other WAR file, it is impossible to run Jetty on the source

[4]This includes sub-optimal file systems. On my computer I have been using NTFS in Windows XP and ReiserFS in Gentoo Linux. ReiserFS is known to be one of the fastest file systems to date, and the differences in speed from NTFS have been visible, especially when executing Subversion commands on the root of the DHIS 2 working copy.

code of a web module. Large WAR files yield huge amounts of time consuming hard drive operations in addition to the compression and decompression algorithms.

6.3 A timeline perspective

Due to the early adoption of Maven 2, the web portal development has stretched over quite some time. The following timeline, shown in figure 6.6, puts the development in perspective with regard to the development of Maven 2.

Maven 2 was entering the DHIS 2 stage already as an alpha 2 version. At some point during the next four months, all modules were converted into using Maven 2, and the Maven 1 project files were eventually deleted. The first web portals were developed after the final version of Maven 2 was released, but the Maven WAR plugin was just in a beta stage at this time, and would be for another seven months. In the mean time, three milestones of DHIS 2 were released with the current custom Maven 2 plugin for assembling the web portal. The first stable release of the Maven WAR plugin in May 2006 did unfortunately conflict with the current solution as it no longer ignored WAR file dependencies, but instead handled them in a way that was not useful nor helpful to the web portal solution, and so had to be ignored for the time being. Another two releases of DHIS 2 went by before the 2.0.1 version of the Maven WAR plugin was released with the MWAR-45 issue fixed, which enabled us to remove all custom plugins, and only use the Maven WAR plugin. The next milestone of DHIS 2, with the latest and current portal solution, was, for different reasons, released in December the same year, and the next version of Maven 2, version 2.0.5, was released in February 2007.

I cannot completely explain the time difference between the first custom Maven 2 plugin and the second and improved custom plugin. I was in Vietnam, working with the Vietnamese team during the first three months of 2006, where the web portal solution itself was not directly in focus. Still, the idea for the improved portal plugin did not emerge until May that year. I remember being slightly puzzled about not coming up with the idea at an earlier stage.

6.4 Creating a common look

For the collected modules to appear as one application, it is crucial that they have the same looks. This includes a common header, footer, menus, and so on. A very common solution to this is problem is to create a header template and a footer template, and have the page content templates include the header and footer templates at the top and bottom of the page, respectively. In WebWork, the end status of an

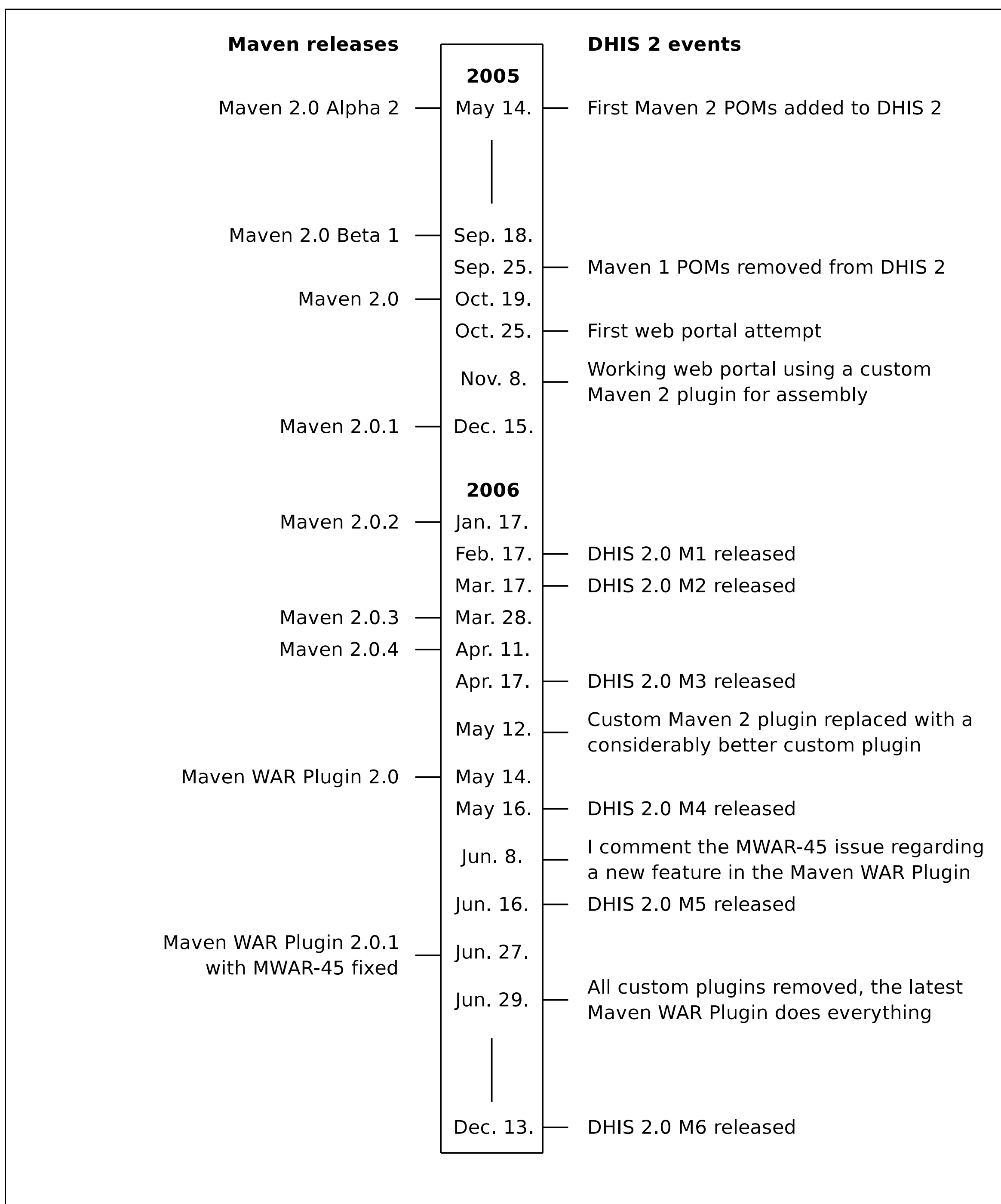

Figure 6.6: Timeline of Maven 2 releases and DHIS 2 events. Note that the distances between the dates do not correspond to the time differences. The two vertical lines symbolize greater jumps in time where uninteresting events have been left out.

action class execution is mapped to a result, for example a Velocity result with a path to the template which should be processed, similar to figure 6.7.

```
1 <action name="relativeURL" action="actionClass">
2   <result name="success" type="velocity">template.vm</result>
3 </action>
```

Figure 6.7: How to define a Velocity result in an action mapping.

So by specifying the page template in the action mapping, and making the page template include a header and footer template, the problem should be solved. But having all these includes can be seen as every template knowing and controlling its own environment, which might not be a good thing if the environment would change in some way or another. With the first web portal assembly attempt, I created a similar solution in which, instead of specifying the header and footer templates in the page content template, all template paths were to be specified in the action mappings as a pipe-separated string of paths, similar to figure 6.8.

```
1 <action name="relativeURL" action="actionClass">
2   <result name="success" type="velocity">
3     header.vm|template.vm|footer.vm
4   </result>
5 </action>
```

Figure 6.8: Action mapping with multiple templates in the result, which has to be supported by a custom Velocity resource loader.

Upon an action result, a custom Velocity resource loader would receive the template paths, load them in the order specified, and return a concatenated stream back to Velocity. Velocity would thus regard the loaded templates as one large template. A positive consequence of this was that any set of templates could be combined as specified in the action mappings, and Velocity would also automatically cache the composite template, because caching takes place after loading. While this worked just fine, it would be very unfortunate if the header and footer templates, for some reason, were to change location or name. This holds for all variants of solutions having separate header and footer templates. The custom Velocity resource loader also had a technical issue which should have been addressed if it were to be used in a production system. There were no restrictions on which files could be concatenated, so any path to any file on the classpath would be accepted by the resource loader.

Shortly after, a safer solution was found. This was a resource loader based on key information at the top of each template that were asked for, which put the template

in a category for adding specific header and footer templates around it. The header and footer templates were predefined in a properties file, thus collecting the path specifications in one place. If the key information did not match any of the predefined templates, it was returned as is and an error message was sent to the system log. A small inefficiency with the solution was that the requested template had to be parsed in order for the key information to be extracted, but the great benefit of this way of doing things—and also of the previous solution—was that it did not require much code, and no web framework would really need to know about the solution. The Velocity resource loader just had to be registered as the one to be used by Velocity, everything else was transparent. Here too, Velocity would automatically cache the composite templates as single complete templates.

One of the core developers was sceptical to having custom resource loaders, and the resource loaders would have become more complex when the web portal would have been deployed unwrapped, as then the templates are not on the classpath (I will get back to the location of the templates in the next chapter). The core developer had a wish for something different, and thus a month later a new solution for putting together web pages was created. The opposite of having header and footer templates is to have one "background template", or main template, containing the header and footer, which includes the content and other elements where they belong in the document. In DHIS 2, we needed a page content template and a page menu template, but because a result in the action mappings can only contain one template reference when using the Velocity and WebWork provided resource loaders, the main template had to be specified here as it had to be the first template to be loaded. So, by defining that the same main template was to be displayed for all actions, and then give separate parameters deciding which templates would be included by the main template, the desired result was met. Figure 6.9 shows how this was achieved in the action mapping. Figure 6.10 shows the basic concept of the main template in Velocity including the menu and page templates.

```
1 <action name="relativeURL" class="actionClass">
2   <result name="success" type="velocity">main.vm</result>
3   <param name="menu">some/menu.vm</param>
4   <param name="page">some/page.vm</param>
5 </action>
```

Figure 6.9: How to specify a main template with dynamic includes of specified menu and page templates in DHIS 2. For different web pages with same or different menus, only the two static parameters need to be changed. The main template assures a common look to all the web pages.

```
<html>
  <head><title>DHIS 2</title></head>
  <body>

    <div id="header">
      ...
    </div>

    <div id="menu">
      #parse( $menu )  ## Referencing the menu parameter
    </div>

    <div id="content">
      #parse( $page )  ## Referencing the page parameter
    </div>

    <div id="footer">
      ...
    </div>

  </body>
</html>
```

Figure 6.10: The conceptual idea of how a main template written in Velocity can dynamically include a menu and page template based on outside configuration. Line 10 and 14 contain the include statements.

By itself, this is not enough. WebWork does not automatically make static parameters from the action mappings available to Velocity, WebWork only makes the parameters available to the corresponding Java action classes. So it is by default the action classes' responsibility to have two setter methods and two getter methods matching the parameter names, in order to transfer the incomming parameter values from the action mappings to the Velocity template engine. This is a concern cutting across all action classes, and thus is best not implemented by every single action class. A small change in the names or the addition of new parameters would mean that all action classes in the system would need to be updated. There are two ways of solving this: Create a superclass which all action classes must extend, or create a WebWork interceptor which makes the parameters bypass the action classes. The latter solution was chosen, making the action classes completely unaware of what is going on and not forcing them into unnecessary class hierarchies. The interceptor was added to the default interceptor stack, which is used upon every action execution. In addition, the interceptor was made general in the way that the parameters that it should recognize and pass on were configurable in the bean mapping of the interceptor. This has

turned out useful, as more properties for including page specific JavaScript files and style sheets have been added without touching any Java code. This template system is still being used in the spring of 2008.

There are a few benefits of having a main template. For instance, the template is viewable in a web browser during design, without needing to include the contents, so it is a more attractive solution to web graphic designers. All included templates are standalone—they do not contain any knowledge of their surroundings, and can thus be reused much more easily. And control is centralized in the main template. On the negative side, the current technical implementation requires the main template to be specified in every action mapping that ends with a Velocity result. If the main template was to be moved or renamed, all action mappings need to be updated. On the other hand, the current solution allows for specifying different main templates if they exist. Another issue is that static parameters in an action mapping are local to the action mapping and not the result mapping. Thus, if two different Velocity results were mapped up for the same action, they would necessarily get the same page and menu. A workaround is to let one of the result mappings chain to another action mapping, which then returns the correct web page, but this is a hack.

A slightly different implementation of the same concept is to create different Velocity result types, one for each "background" template. This would move the main template paths from the action mappings to the result types, centralizing the paths, and then the page template paths could be set in the result mappings instead of static parameters. But it would tie the solution to the template engine being used, in our case Velocity, while the current solution is template engine independent. It is a trade-off, and I would go for the solution which is being used today, because it only requires one custom class, a simple, generic interceptor, which is completely transparent and initially just a helper class for forwarding static parameters to the result.

Yet another alternative is to use a decoration framework such as OpenSymphony SiteMesh [18]. My personal opinion is that SiteMesh is overkill when templates including other templates is an option. SiteMesh works by parsing the web pages returned from the application to remove the meta information and outer HTML elements from the document, so that it can be surrounded by new, common HTML. The meta information extracted is also partially merged into the header section of the new document. For complex enterprise web sites where the web pages are produced by multiple independent subsystems, a SiteMesh type of framework is a good alternative for creating a common look to the subsystems. For applications and sites with a single web page source, I would try hard to incorporate the common look functionality into the application itself, in order to avoid the processing overhead of parsing and generating new web pages which already have been generated by the application.

6.5 Common web page elements

Up to this point, the responsibilities of the developer of a web module have been (except for the basic setup of a Maven web project):

- Define module dependencies to the two common web modules so that common functionality, templates, and other web resources can be utilized.
- Make the module xwork.xml configuration file include and extend a centrally located XWork configuration file with default interceptor stacks, error handling mappings, etc. The interceptor stacks are crucial, as they provide the support for the common web page looks, in addition to transaction management and security features.
- In addition, each module should use a namespace for its templates and action configurations, so that everything is uniquely addressable, and will not collide when they are merged with the other web modules.

The next step is common web page elements. Most DHIS 2 pages have a common menu system. An organization unit tree widget is also shared between modules. Naturally, these elements have been placed in the common web modules as far as possible during the development of the portal. The following sections go into the details.

6.5.1 Module entry point and menu system

With each module specifying a set of web pages for interacting with a specific part of the system, there is a need for one or more entry points through which a user can get access to the module pages—a menu. A discussion on the developer mailing list outlined a few alternatives. The initial question was: What is the menu going to contain except for the web modules? A single entry point to each module is a good start, but with each module offering a set of related functionality to the user, it is likely that each module would need its own menu. This was pointed out by one of the core developers. I continued to suggest making one, large menu:

> "A possible implementation of this [large menu where all modules contribute] is that the necessary information is provided in an xml file in each module. The xml files will be parsed only once, at startup, and no changes has to be done to the object model of the menu.
>
> Other suggestions? A more dynamic approach?" [Me on the developer mailing list. October 29, 2005]

This is actually a semi-dynamic approach, as the menu is dynamically loaded from XML files, but the contents of the XML files are rather static. It is a good approach with regard to caching of the menu, as the menu will be the same on every page, but the menu will be quite large, and not all parts of it are equally interesting when working with the different modules. The reply from one of the main developers is more in the direction of how the menu system was finally solved:

> "We're talking local navigation, which should be relatively stable. The VM-files [Velocity templates] could provide any context stuff or dynamic navigation, if necessary." [DHIS 2 developer on the developer mailing list. October 31, 2005]

The key here is local navigation. And so the menu system was solved by having one main menu on the top of every page with one entry point to each module, and then have a module menu included by the main template. The main menu is assembled by searching through the XWork configuration and listing all action classes mapped up with the name "index". The name of the menu item is based on the unique package name in which the index action is defined. The package name should be, by DHIS 2 specification, the Maven artifact ID of the module. The package name is then used as a resource key for internationalized strings, making the module menu internationalized (but if no translations are available, the package name is used). Each module must make sure they have an index action in order to be accessible from the module menu, and that this index action points to a natural starting point for navigating the module pages. The order of the menu items can be specified by configuring one of the menu management beans, which will do its best to adapt the configuration to the discovered set of modules included in the final system.

The main template is created to include a menu template in a sidebar of the web pages. This menu area is designed for page specific menus, and is configured for each web page separately in each module. In most cases, the same menu template is used for all pages of a module, as multiple menus per module might be messy and confusing. Initially, anything can be put in this menu area, as any regular template can be specified, but the intention is to let the developers provide the users with options for navigation within the module.

The DHIS 2 solution is simple, at least for the developers—they only need to define an action named index, and possibly define a nice name in the common part of the internationalization resources. The system itself takes care of the rest. From a user's perspective, this menu might not be the best construction. Considering that, at the moment, most of the DHIS 2 web modules are management modules for setting up the system, and not data entry and reporting modules, which are the ultimate

business providing modules, the system seems to be all about management (see figure 6.11).

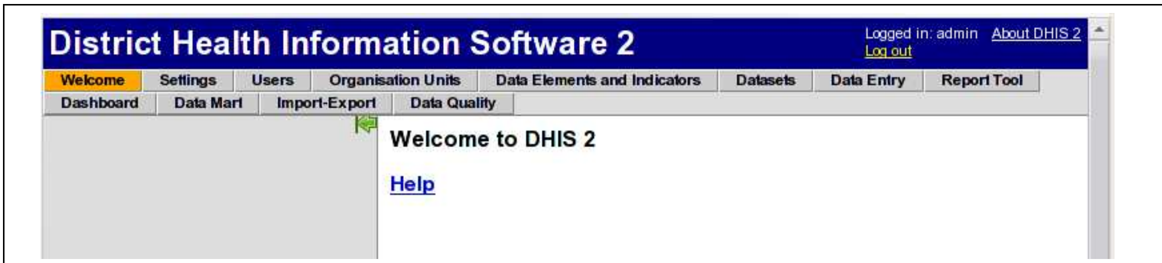

Figure 6.11: The module menu with roughly one "home module" (Welcome), eight management modules (Setting, Users, Organisation Units, Data Elements and Indicators, Datasets, Data Mart, Import-Export, and Data Quality), and three entry, reporing, and analysis modules (Data Entry, Report Tool, and Dashboard).

An early solution to the problem of having too many management modules on the main menu was suggested, which merged all the menu buttons of the maintenance modules into one, and with a common page menu for all the modules. The only problem with this proposal was internationalization, as each module contains its own translations. And because only the translations of the currently active module is loaded at any time, most of the common page menu would be left untranslated, as their translated texts would not be loaded. The solution was implemented around the first release of DHIS 2, but it was later removed when the system was properly internationalized. A side effect of this, which is still visible today, is that all the maintenance modules are placed with a common parent Maven project in the project hierarchy.

While it is now possible to hide the modules a user should not have access to, by either excluding the modules at build time or configuring the user privileges in the user management module, it is not given that all available web interface modules are equally "important" and should be aligned on the same menu. Why should the management modules not be hidden behind a separate management menu button? And why should each module only have one entry point? There is no requirement in DHIS 2 saying that one interface module can only contain one naturally cohesive set of web pages. In fact, the current data dictionary web module contains management pages for multiple model domains (the "Data Elements and Indicators" menu item in figure 6.11).

In order to have more flexibility with regard to the mentioned issues, I will propose a new menu solution, where each module can define which module entry points it offers, and where these should be placed in a system wide menu hierarchy.

Technical proposal for changing the menu solution in DHIS 2: *The proposed solution has a main menu like today, but each menu item is a "bucket" containing entry points. Each web interface module will contain a specific configuration file (not unlike my initial suggestion for creating the menu system), which specifies all of the module's entry points, and which buckets the entry points should be available from. On startup, the system will load all the configuration files and construct and cache the complete system wide menu. As several modules might claim the same position and name in the menu, the system must either have a mechanism for conflict resolution, or simply complain and ask that the problem is rectified. If a bucket only contains one entry point, the entry point can be promoted to the main menu. For example, if the system only contains one data entry module, and this module has only one entry point which is defined to be placed in a "Data Entry" bucket, then it would be natural to promote this entry point to the main menu. A click on the "Data Entry" menu item will take you to the data entry module. At first, it might seem that the entry point promotion makes it hard to expect what is behind a main menu item, but one must not forget that the system, and hence the menu, does not change after being installed on a computer. A question that needs to be answered is whether the promoted entry point should keep its own name or take the name of the main menu item. In the data entry example, it might be desirable for it to take the name of the main menu item, so that the main menu does not end up with an item named, for example, "Standardized Data Entry". But if an entry point to a user management module is promoted to the "Management" main menu item, it might be desirable for the entry point to keep its own name, for example "User Management". A good solution would resolve this through configuration, so that the decision is externalized. Naturally, if the module menu configuration files are not coordinated, one might end up with an even worse main menu than the current DHIS 2 menu. How each bucket of entry points should be presented is still open, and can be changed regardless of the menu item structure. One solution would be to list the entry points on a separate web page when clicking the corresponding main menu item. Another solution would be to present the entry points in a drop down list, close to the corresponding main menu item. For simplicity, alphabetical ordering of the entry points is currenly assumed, but support for any ordering or further grouping of entry points can be added.*

The suggested solution is rather similar to my initial suggestion for creating a menu system, yet slightly more sophisticated. Note that I did not suggest to remove the page specific menus—I believe that there will still be a need for local navigation—but it would be preferable to remove most of the page menus, as they tend to occupy valuable screen space, which has been complained about (many of the monitors used

are 14 inches). As a response to the complaints, we did make it possible to collapse the page menu when it was not needed, but a redesign of the meny system, allowing for improved navigation from the main menu would solve some of these problems.

6.5.2 Organization unit tree widget

In DHIS, the organization unit tree is a hierarchy of administrative health units. The top unit of the hierarchy is typically the national ministry of health, and underneath follows provinces, districts, health facilities, etc (for an example, see figure 6.12). In many countries, there are no computers at the lowermost levels where most of the data is captured, so paper reports are still being used to propagate data up the hierarchy. At some point, computers take over, meaning that all the paper reports from the lower organization units must be entered into the computers. To make the process of entering the data more efficient, one needs to be able to specify which organization unit one is working with, even though a user is typically associated with only one organization unit by default. This was solved by having a tree widget representing the organization unit hierarchy in the page menu, making it possible to quickly select the organization unit one is entering data for. Similar situations apply for making reports. The following quote is taken from the developer mailing list:

> "Yes, a user should be associated with an OrganizationUnit, but it should be possible to change by choosing in a hierarchy. This hierarchy could maybe be the default view of the organizationunit admin part (the welcome screen of that module..." [DHIS 2 core developer on the developer mailing list. October 28, 2005]

A placement of the hierarchy widget is suggested in the quote, but this was later changed in the way that each module can include the widget where it wants to, but preferably in the page menu, or a form. Figure 6.12 shows an example of the tree widget.

There are actually two organization unit tree widgets in DHIS 2, with slightly different behaviour. The first one is a tree widget where the current "active" organization unit is, at any time, selected. Only one organization unit can be active at a time. This is the tree which is used when entering data, and the state is kept across modules and pages. The other one is a tree widget for just selecting a set of organization units for use in a specific context—a form input field. Of the two widgets, the form input widget is less frequently displayed, as it is only used in a few forms. We will now look into how these widgets are constructed.

From the users' perspective, the widgets look like regular computer representations of trees, with plus and minus signs for expanding and collapsing branches of the tree.

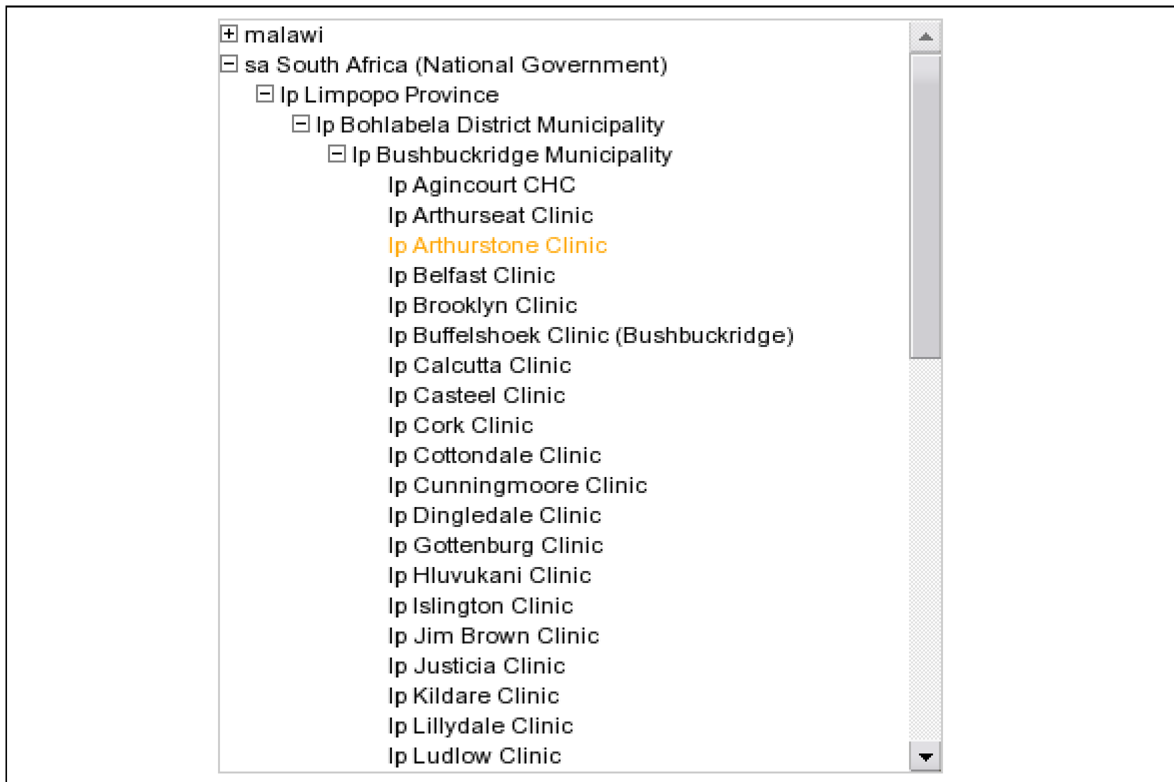

Figure 6.12: The organization unit tree widget with exemplary content.

When an organization unit is selected, it is given a different style—currently a different color. The current organization unit tree keeps its state across pages, so the state must be stored on the server. For each click in the tree, an AJAX [50] request is sent to the server with a note of the change. The state is user specific, so the HTTP session is used as the state holder. In case of a branch expansion, the server returns the newly visible organization units, which are then rendered using JavaScript and DOM manipulation. When a page containing the tree is loaded, the expanded branches of the tree are constructed in HTML before the page is returned. Further manipulation of the tree is done using JavaScript. The form input widget is not partially created on the server. Upon page load, the tree widget is empty, but it immediately sends an AJAX request to the server for getting all visible and selected organization units, and does the complete rendering using JavaScript. The reason why AJAX was chosen in both trees, is that otherwise each click on the widgets would refresh the whole page. Refreshing a page typically means loosing all the data on that page which has not already been sent to or provided by the server. Such refreshes are utterly frustrating, and more or less require the user to do the tree selections first, before continuing the work on the web page. AJAX solves this by being able to send asynchronous requests to the server, and handling the responses in JavaScript.

The benefit of saving the tree state on the server is that they can operate more or less independently from whatever else is being worked on in the web pages. WebWork actions can manipulate the tree states before the web pages are constructed, adapting the trees to the current situation and context. A problem is that this makes it hard to open multiple pages of the application at the same time, for example to compare values from different organization units—the server state is always the same for all open pages from the same user. This was pointed out by me in the initial discussion about the widgets:

> "...the user's username and the corresponding organization unit must be kept in the session variables. A potential problem is that a user might open two browser windows and try to choose two different organization units, which of course doesn't work, but I guess this is a stupid scenario."
> [Me on the developer mailing list. October 29, 2005]

Although it was suggested to use browser cookies instead, HTTP session was eventually used. I do not recall having received any complaints from people having multiple windows open at the same time.

In the case of the form input widget, there is an awareness problem too, because the use of the widget is form specific, but the state is user specific, and a previous state is generally not wanted in a different form. The result is that all actions which lead to a page containing the form input widget must be aware and always set the state of the input tree according to the context. The tree does not automatically fall back to a default (safe) state between use. Thus, the form input widget should probably be reconstructed to keep its state on the current web page only, to be submitted as regular form data. In such a case, the initial tree state must also be set on the web page at generation time. This would allow for multiple instances of the form input widget on one web page.

Technical proposal for changing the organization unit tree widget used in input forms in DHIS 2, to not use HTTP session: *The goal is to change the form input widget from saving its state in the HTTP session, to saving its state in the current web page, and ensure that the state is submittable as a typical form element. For the state to be submittable, it must use a hidden input element with a defined state representation format. The simplest solution is probably a comma separated list of organization unit IDs, where the IDs represent the selected organization units. If any organization units are pre-selected, the hidden input element must be initialized using the same format. Because multiple action classes and pages will be using the widget, it is best to hide these details and let widget related functionality take care of them.*

Ideally, the actions will communicate with the widget using Java sets[5] *of organization units, which means that the widget must be able to convert IDs to organization unit objects. The only way to do that is by, at some point, having a bean dependency to the organization unit service. A WebWork type converter could be made to automatically convert the data between the web pages and the action classes—it would hide all the conversion details from the action classes—but it might not be desirable or possible to have a bean dependency from such a converter. Another option is to create the converter as a prototype bean, which the action classes must depend on, and which will receive the raw data string from the forms using getters and setters. If the name of the widget's hidden input field is set to match a setter of the converter bean, and the action class makes the converter bean accessible to WebWork, the action class does not need to handle the raw data in any way. It can communicate with the converter bean directly, using sets of organization units. The hidden field name could have a default value, but it should be possible to override it if multiple instances of the same widget is used on the web page.*

A common web page template should be made to include the hidden input element and the appearance of the widget, so that web pages only need to include the template when using the widget. JavaScript must be created to keep the hidden input element up to date and correct, according to the user clicks, so that the form can be submitted at any time. Also, JavaScript must, upon page load, send an AJAX request to the server, asking for all organization units which need to be displayed, in order for any pre-selected organization units to be visible in the form. Naturally, further expansion of tree branches will also require AJAX requests for the organization units details to be displayed. The goal should be to create an as small as possible interface between the widget and the using classes and web pages.

An important issue is how the root element of the tree is defined. Both of the current tree widgets support setting the root of the displayed tree to any organization unit in the hierarchy. If the root element is form specific, then the ID of the root could be hidden in the web page together with any pre-selected organization units. But because there is no guarantee that values sent from the client are the same as were returned from the server, this is not immediately secure. While one might argue that the user is already authenticated at this point, and thus should not pose a security risk, proper validation of the values from the client is required. The action classes responding to the AJAX requests cannot do this validation, as they do not know the context in which the tree widget is being used. In addition, manipulation of the form data can happen independently of the AJAX requests, meaning that validation done by the AJAX actions do not guarantee that only legal values are submitted with the form. Thus, the action classes receiving the input form values must do the validation,

[5] java.util.Set

for example by letting the converter bean know the effective root, and let it do the whole job of checking the incomming IDs. By avoiding using the HTTP session, there will be no conflicts between any simultaneous form widgets.

As already mentioned, the tree widget for setting the currently active organization unit generates the visible parts of the hierarchy in HTML, so that no AJAX requests are required when the web page is displayed. This is beneficial, as the widget is often constantly visible in the page menu, and because the generated HTML can be cached between pages. A technical problem I had while developing the tree was that action classes must be allowed to modify the tree state before the corresponding web page is generated. Thus the tree preparation functionality had to be placed after the action execution, but before the web page was generated. It would be natural to put this in a WebWork interceptor, but the interceptors are only executed before the action is executed, and after the web page has been prepared for assembly. So any functionality placed after the action execution in an interceptor will not be able to affect the action result. The solution was to implement the provided pre-result listener interface, and register the listener in an interceptor before action execution. The problems back then were that the pre-result listener was executed even if the action class failed, in which case the tree widget functionality is likely to fail too, and is rather unimportant after the first failure. Furthermore, the pre-result listener interface does not allow the implementors to throw exceptions, and the service layer in DHIS 2 was full of checked exceptions. Of course, a checked exception can be wrapped in an unchecked exception[6], but the solution felt ad-hoc because no explicit handling of exceptions was provided, compared to action classes and WebWork interceptors. Following the execution thread, any exceptions thrown by a pre-result listener would propagate up the same path as any other exception from an action or interceptor. So why not provide the pre-result listeners with the same specified robustness as the other elements of an action execution? Common functionality after action execution and before result assembly seems to be given less priority than common functionality after result assembly, which strikes me as a bit odd. Regretably, I have to admit, I never sought an answer to that question, but instead created an abstract implementation for pre-result listeners, wrapping the execution in a try/catch block, just logging any exceptions thrown, and not executing the pre-result listener if the action class failed first. Later, all checked exceptions in the service layer have been replaced with unchecked exceptions, as they represented states which the system is not expected to handle directly, so now no checked exception is ever thrown, and no exception is ever expected to be thrown from the organization unit tree widget. After all, a solution was found that worked, and the abstract wrapper, hiding the exception handling,

[6]See the glossary for a quick introduction to checked and unchecked exceptions.

made it less ugly. Furthermore, this small piece of functionality is the only pre-result listener in DHIS 2.

Appendix E contains a couple of e-mails to the developer mailing list which show some of the frustration around these and a few related issues. The responses to the e-mails were rather fruitless, as I already had the most experience with these matters in the team. The best response, though, was:

> "This seriously hurts my brain..." [DHIS 2 developer on the developer mailing list. January 21, 2006]

Chapter 7

Discussion

Modularization applies to different areas and parts of an application. An application can be modularized at a high level with regard to business processes and user interfaces. High level modules make it possible to assemble the final system by selecting the desired and necessary application modules according to requirements, similar to orchestration in SOA, or choosing the right Maven 2 dependencies in the DHIS 2 web portal. Each application module can be decomposed internally following principles of, for example, information hiding in order to promote flexibility, reusability, testability, and maintainability. Using the Java language, this involves designing interfaces and classes which interact in some way. Together, the interfaces and classes present a defined outward interface for interaction with other application modules. At the innermost level, code is organized in methods.

7.1 Java language constructs

The Java language has a range of constructs which are related to modularization. Classes are themselves modules, collecting related state and behaviour, and are typically created around single tasks, responsibilities, or concepts. Functionality can be standardized through interfaces, hiding the implementation details, and providing a black box for flexibility [42]. Using the Spring Framework, such an interface/implementation pair is called a *bean* [47], which can be seen as a module. A bean is often dependent on other beans in order to fulfil its defined contract. Examples are service layer interfaces with implementations which use persistence layer interfaces with implementations for persistence [48]. Inheritance provides generalization and specification, and hierarchies of standards and implementations. Methods can be made public, private, overridable, and not overridable by using specific modifiers, giving a class, in practice, two class interfaces; an *interface for use* and an *interface for extension*. By class interface, I mean the outward access points of a compiled class

which are programmatically approachable without changing the bytecode or manipulating modifiers using reflection. The interface for use can be represented by Java interfaces, and the interface for extension is used by subclasses. The next sections will talk more about these interfaces. Furthermore, (Java) interfaces and classes are organized in packages (directories) and different types of archives (ZIP files). Packages and archives can be mixed as desired, resulting in cross-archive organizations of classes.

Generics, which were introduced in Java 1.5, cannot be used for modularization. I choose to comment on this here because it might not be obvious to the less experienced Java developer. Generics were added to move type checks from runtime to compile-time, which means that the compiler will complain on a type mismatch in the source code instead of getting a cast exception when the program is running. Generics are similar to using all-eating Object references in that one can create classes which work with multiple types without knowing which types. The difference is that with generics, the type is specified in the code instantiating the generic class, and it is forced by the compiler to be the same for all references to that particular instance. So generics cannot be used for collecting common functionality for multiple types any differently than what is possible by using Object references.

7.1.1 Parameterized inheritance versus Spring beans

In DHIS 2, archives are maintained through Maven 2, and the Spring Framework is used for mapping together interfaces and classes from different archives to working beans, and wiring the beans together. By placing the interfaces in one archive and the implementations in another, the implementation archive can easily be replaced by a different archive of classes implementing the same interfaces. Thoughtful grouping of classes and related configuration into archives means that archives can be treated as modules, and beans can be seen as submodules.

VanHilst and Notkin [63] use parameterized inheritance in their module decomposition. They form complete modules by creating classes with appropriate methods and combining them using inheritance. While parameterized inheritance is not directly supported in Java, it can possibly be achieved using pre-processing of the classes. The pre-processing must be based on some sort of *prior explicit configuration* which defines the class hierarchy, as well as which methods will call which other methods using the implicit interfaces between the classes. The question becomes: How is this different from programming to interfaces and combining the beans using external configuration and wiring? First of all, beans can be created from class hierarchies, so one could use parameterized inheritance in order to create a single bean. But a bean has a defined interface which already should be a set of related methods addressing

a coherent target for minimized complexity, as adviced by Woodfield [67]. So further decomposition of the implementation using parameterized inheritance should not be immediately necessary, and should maybe be taken as a sign of poor interface design. Inheritance is still useful in beans, though, for example as a means for supporting later adjustments. But VanHilst and Notkin are not targeting this level of decomposition, they are targeting the decomposition of the module. The implicit interfaces between the classes in the parameterized hierarchy must be compared to the explicit interfaces of the beans. The interfaces in the parameterized hierarchy are class interfaces for extension. Calling the interfaces implicit is not completely accurate, as they are explicitly defined by the classes, but they only exist if the implementations exist, and thus must be formally defined elsewhere—for example on paper. The beans have explicit Java interfaces which are themselves the formal definitions and contracts.

The parameterized inheritance and the beans both need configuration. The configuration of beans define which implementations are used behind which interfaces, and wire them together into a complete module. The configuration of parameterized inheritance is the choice of parameters which define the complete class hierarchy of the module. The specific difference is that the bean configuration can be changed without the need to recompile the classes, resulting in a loose coupling of both source and compiled code, while the hierarchy configuration requires compiling, resulting in a loose coupling of source code, but a compiled coupling of compiled code. The latter might reduce link overhead, but the former requires less effort to re-configure. For automatic unit-testing, both types of submodules can be taken out of context, but I would argue for the benefit of having explicit Java interfaces which define the intended behaviour of both the implementation and its dependencies, and the greater flexibility of working with decoupled compiled code. For example, dynamic configuration, or externalization of control flow [66], require that configuration can take place after compiling.

In general, parameterized inheritance and beans are quite similar—they consist of classes combined by configuration—but the parameterized hierarchy has one interesting property which has been missed in the development of DHIS 2; the option of hiding submodule interfaces from the public regardless of package. For example, in a layered system one might want to hide the persistence layer from the user interface layer, forcing the user interface layer to use the service layer only. If a parameterized hierarchy includes both the persistence and service layer related to a specific part of a system, the class interfaces define what is public and not by using suitable modifiers. The interface for extension can provide methods which are public to subclasses but not public to outside functionality. Appendix D contains an e-mail from one of the core developers which outlines the wish to hide an interface, although with a slight error regarding the scope of no modifier. The Java language supports creating non-

public interfaces, but such interfaces are only available to the package in which the interface is defined. While the same package can be used across archives and modules, this constraint might be too strict, as it requires all persistence layer interfaces and implementations, and all submodules which use the persistence layer, including the service layer implementations, to be in the same package. Though this is technically possible, it might conflict conventional structuring, and would require a fair amount of refactoring in the case of DHIS 2. On the other hand, a more flexible solution to hiding interfaces would require specification, which introduces more configuration and complexity. A possibility would be to define an interface as public to all classes unless otherwise specified, and that such a specification includes a listing of packages the interface is public in. It could also be possible to specify a complete subtree of packages using wildcards.

According to the previous comparison, we need to further specify the definition of the class interface for extension. Does the interface for extension include the interface for use? All combinations of public and private, and overridable and not overridable methods can be made[1]. Figure 7.1 shows the possibilities. If a method is public, it can be used by a subclass even if it cannot be redefined. The method is part of the interface for use, it cannot be overridden, but it can be used in other overriding methods. It is impossible to make a method public to other classes and not public to subclasses. In contrast, it is perfectly possible to define methods which are only public to subclasses, and also only by use. If the interface for use is defined as the collection of all methods and fields which are publicly available to non-subclasses, then methods which are only public for use by subclasses must be in the interface for extension. Similary, extension is required for such methods to be used by other classes than the defining class, and intentionally so, thus being part of the interface for extension. Unconditionally public methods are, however, available to both outside classes and subclasses. If the developer's intentions are included in the argument, then a method might not be intended to be used directly by subclasses although it is public, and thus not be part of the interface for extension. This constraint is not supported by the Java compiler, but it can be provided in a comment to the method. Compared to Java interfaces, comments are part of the contracts the interfaces define, and if the same is said to be true for classes, then the distiction exists, and the interface for extension does not include the interface for use. But they may overlap.

[1] I choose to ignore the fact that protected methods and methods without modifiers are public to the package, as we are interested in unconditionally public methods here.

Modifiers	Class interface for Use[1]	Extension Use	Extension Override
public	×	×	×
public final	×	×	
public final[2]	×		
protected		×	×
protected final		×	
private			

[1] Usable from all packages

[2] Not to be used in subclasses, as intended by the developer, and not supported by the Java compiler

Figure 7.1: Possible modifier combinations regarding the class interfaces for use and extension.

7.1.2 Cross-cutting concerns

With Java 1.5 came native support for dynamic proxies, which enables the creation of AOP interceptors for cross-cutting concerns. But third-party libraries for applying AOP functionality have existed for quite some time already, with more possibilities than what dynamic proxies offer. DHIS 2 is using the Code Generation Library [5] through the Spring Framework, because DHIS 2 was originally built with Java 1.4, but the same results could have been achieved with Java 1.5 dynamic proxies today. Hibernate [8], the object-relational mapping framework mentioned in the DHIS 2 introduction, uses proxies for loading data lazily from the database. That is, if an object just fetched from the database has a reference to another object (typically a relational foreign key), a proxy will take the place of the referenced object and fetch the actual object from the database when someone tries to follow the reference. The result is that only accessed objects are fetched from the database. DHIS 2 uses proxies for interceptors for transparent transaction management and other cross-cutting concerns. As showed by Garcia et al. [37], AOP can greatly improve the modularization and separation of concerns in applications. Combined with dependency injection and cross-module configuration, as can be done with the Spring Framework, an interceptor can be placed in a semantically coherent module, and be simply referenced and applied to the rest of a system, without touching any Java code [48]. While transparent interceptors can be created using proxies, a framework can also implement support for interceptors on specific targets through provided interfaces and configuration, as in the case of XWork.

The Java language provides multiple means for creating programs which are modular on multiple levels. It is up to the developers to know the possibilities, and to know

how to use them, in order to achieve the best modularization of their applications.

7.2 Modularization of frameworks and libraries

The modularization of external frameworks and libraries also affect the modularization of the application. This is evident in the following two examples of subclassing and programming to interfaces.

A class structure is made up of methods, fields, and their modifiers. Methods and classes are often based on the DRY principle [45] and "do one thing and do it well". If properly structured, classes can present an interface for extension where subclasses do not need to repeat code they do not want to override. In DHIS 2, there have been a couple of cases where framework provided classes did not completely fit with the system and needed to be adapted. In the case of the XML configuration loader of XWork, the loader class was so poorly structured that the only way to make it work was to copy the whole source code of the class, and then make the necessary internal changes. The basic problem was that it did not support addressing and loading configuration files with equal names, located in separate archives. If the relevant parts had been separated out in overridable methods, a simple subclass would have sufficed. Parnas [55] experienced how proper modularization reduced the overall work of fitting a system to different contexts. A system related to technical contexts is similar to a library or framework related to systems. If a framework is not properly modularized, its usefulness in different situations might be reduced, even though it is intended for supporting the general problem.

Furthermore, a poor structure might also reduce forward compatibility, because you might need to repeat code in your subclasses which is related to the inner workings of the library, and which might be changed in future versions. Differently put, improperly structured classes can leave "holes" in the interfaces, from a subclass' perspective, which reduces potential flexibility. Evidently, there are interesting principal similarities between the interface for use and the interface for extension. Each method signature defines what the methods should do, which again defines what a default content can be replaced with in a subclass. Creating a subclass is similar to implementing an interface—the method signatures are defined, except that the methods have default behaviour if not being abstract. So creating a class involves standardization of two class interfaces for securing internal flexibility and reusability. Or in other terms, minimize the interface complexity in order to minimize the coupling between the classes [67]. Both class interfaces are very important in this regard.

In DHIS 2, most classes are programmed to Java interfaces. This follows the rules for application development recommended by Johnson and Hueller [47]. Dependency

injection is also used, so that implementations can be easily replaced by configuration. The Acegi Security framework for Spring [1] uses the same principles. When applying the Acegi Security framework to a system, the initial job is to choose the implementations you need and want to use, and wire them together using dependency injection. The benefit is simple adaptation of the framework, as any of the provided implementations can be replaced with custom implementations of the same interfaces. Acegi makes use of interceptors for applying the security checks since security is a cross-cutting concern. Using the Spring Framework as the application framework for wiring everything together, programming to interfaces is more or less required by the application. If Java 1.5 proxies are being used for creating method interceptors, interfaces are required[2].

The two examples above show how the modularization of external libraries and frameworks affect the application. A framework is created to guide solutions to common problems by setting a frame of (believed) best practices, where the developer only needs to "plug" the application specific "holes" in the solutions. A framework provides conventions (a focal point for distributed development), promoting consistency, seeking to minimize the development efforts [60], including taking care of boring, repetitive, and tricky tasks. An interesting metric in this regard in framework development is *intrusiveness*, which measures the degree of framework awareness in the main parts of an application. For example, an application using the Spring Framework can be developed to not have any programmatic references to the framework except in the main method starting the framework [48]. Such a solution makes it possible to replace Spring with any other suitable application framework without modifying any core classes—the framework is used in a non-intrusive manner. On the other hand, it is possible to make core classes implement various Spring specific interfaces, tying the application to the framework. There seems to be a current trend in making frameworks non-intrusive, supporting plain old Java objects (POJOs)[3], which again can partly be seen as a reaction to Sun's Enterprise Java Beans [47]. Non-intrusiveness makes it possible to develop application classes with "clean" code related to nothing but the actual functionality of the system, and which can easier be taken out of context to be tested in isolation or reused elsewhere. The Acegi Security framework for Spring is non-intrusive by its use of interceptors to provide basic se-

[2]One of the arguments to the java.lang.reflect.Proxy.newProxyInstance() method is an array of interfaces, which are the eventual interfaces implemented by the proxy. The AspectJ [4] library for applying AOP uses compile-time weaving, which provides more options.

[3]In [49, p. 18], the Spring Framework and Spring MVC are mentioned to contribute to the "The Return of the POJO". Starting from XWork version 1.1 (WebWork version 2.2), action classes were allowed to not implement the Action interface [54], thus allowing action classes to be POJOs. This change was anticipated in [51]. Starting from Tapestry version 5, component classes (comparable to actions in XWork) were changed to be POJOs [23].

curity checks, although intercepting by proxying requires programming to interfaces. And if you have chosen to use Spring and Acegi Security for your application, you have probably already chosen to program to interfaces.

Although the main goal of a framework is to set the scene for solving one or more specific problems, the scene must be customizable by the application. WebWork provides simple means for creating custom result types, interceptors, and interceptor stacks. But for placing functionality between action execution and result assembly, as in the case of the organization unit tree widget in DHIS 2, WebWork "only" provides an event hook. Predicting where applications might want to plug in functionality to a framework in order to solve certain problems is a challenge. WebWork could for example have provided one double or two interceptor types—around action and around result. On the other hand, if the pre-result listener is always used in conjunction with one specific result type, a new result type could have been created instead, which could either include the pre-result functionality or simply be wrapped in an AOP method interceptor. Which solution is the best would depend on the number of pre-result listeners and the number of different result types these listeners are used with, but I would, in general, prefer a more explicit guideline or documented possibility for extension from the framework. The goal here is not to bash WebWork or the team behind it—the popularity speaks for them—it is merely to point out how framework developers influence the development of the applications using the framework in ways that they might not have intended, and that developers need to be aware of this fact. An unclear stance from a framework's perspective, including its documentation, might result in more ad-hoc and unmaintainable solutions than in a case of a clearly documented unsupported feature—keeping in mind the important aspect of forward compatibility.

7.3 The influence of tools

Maven has been used for managing the modules of DHIS 2 since the beginning of the project, partly because one of the initial developers was engaged in the development of Maven itself. Choosing a project management system like Maven also implies choosing a set of source code organization conventions and patterns for working with them. While it is possible, in the case of Maven, to override the conventions in configuration files, it is preferable to keep a certain standardized set of rules which modules need to follow, so as to minimize the maintenance overhead and managerial differences between modules. Included in such considerations is the aspect of debugging and helping other developers from remote locations—conventions and standardization limit the number of permitted configurations, thus narrowing the field in which to look for errors.

Behind the Maven conventions are concepts, like Java code and different types of resources, so if conventions are changed within the scope of the concepts, the changes merely correspond to redirected folder locations. Hopefully, the Maven plugins are properly designed to handle the concepts and adjust to the configured conventions. Otherwise, the Maven plugins would have to be modified with any changed convention. Using the provided Maven conventions in DHIS 2—because we do not see a need to change them—actually helps avoiding potential problems with the plugins. In a perfect world this would not be an issue, and it should not be used as an argument for not setting up the system conventions to fit with the required structure. Because Maven is open, it would be better to rather engage in the development of the tool, by at least submitting bug reports, so that any shortcomings can be worked out.

In the development of DHIS 2, there has been little reluctance to create new system modules. If a new piece of functionality did not fit with any of the existing modules, a new module would be created for it with a suitable name. This can be partially explained by the fact that Maven makes it easy to handle many modules. The requirements for setting up a basic Maven project are few and simple, especially because of the possibility of configuration inheritance. Defining a few Maven module dependencies is, and should be, easier than manual alternatives. But it has lead to quite a proliferation of DHIS 2 modules, which developers have complained about. The more modules, the more time it takes to build, because of the overhead related to constructing the dependency tree, resolving the dependencies, running the Java compilation processes, and so on for each module. There has been some effort in trying to reduce the number of modules in DHIS 2, but the process is rather staccato in the sense that modules are merged when developers see fit, typically after a quick discussion on the developer mailing list. The general guideline for grouping persistence and service beans into modules is based on model objects, but later guidelines are more focused on core functionality versus (theoretically) optional functionality. The various support modules are grouped by theme, such as Hibernate, web, and unit tests. The web interface modules are grouped based on related features and use cases, including the web interface modules for maintaining the database, which are also centered around related model objects. While each software project must choose its own rules for modularization, DHIS 2 has been developed under the philosophy that modules are important and useful organizers, and that they ease distributed development, much in line with Parnas' expected benefits [55] of modularization. And that when using a good module management system like Maven, there is no reason for not creating a lot of modules. I am quite sure DHIS 2 would have been differently modularized if a different project management tool had been used. That said, most of the discussions we have had around how to organize the code have been grounded in logic and principles. Appendix C shows an e-mail from one of the main developers

on the subject of reducing the number of modules in DHIS 2. The decisions in this e-mail have, for reasons unknown to me, only been partially implemented. The corresponding issue is still marked as open and of minor importance in the bug tracking system.

While Maven adds overhead to the build process, the largest consumer of build time in DHIS 2 is, by far, the unit tests. Unit tests are important for verifying that the code is correct, and they are automatically executed by Maven in conjunction with building a module. Unit tests [45] are to be executed in isolation, so that environmental randomness does not affect the outcome of the tests. For example, if unit tests which test against a database always get an initially empty database, there is no need to worry about existing data in columns with uniqueness constraints or similar, and it is simple to create checks regarding the amount of data in the database. In DHIS 2, this has proven to be a relatively time consuming setup, as Spring, Hibernate, and the database, a Hypersonic database [9] running in memory only, are all currently started and stopped for every single unit test. While each initialization is only a matter of seconds, the repeated process adds minutes to the total system build time, especially on less powerful computers. But the current solution is simple, because it leaves the whole job of setting up the test environment to Spring and Hibernate. A less time consuming solution would be to keep Spring and Hibernate started, and only clear out the database between the tests. In such a scenario, Spring and Hibernate would only be started once for each Maven project. The only problem with this approach would be to find a simple solution for clearing out a complete database without hard-coding any references to tables, as the set of tables created by Hibernate varies according to which modules are present in the current classpath[4]. Since this problem is not in the immediate scope of my thesis, I have only had a quick look at it and possible solutions, which are not worth mentioning here. A result of the unit tests taking considerable time compared to not running them, is that some developers tend to skip the unit tests every time they build the system, thus failing to verify that everything still works.

In all the previous paragraphs there is an interesting human aspect related to how the development of DHIS 2 is managed. As can be seen, most of the technical decisions are made by the developers. How the source code should be structured in directories and Maven projects has largely been the concern of the developers. Also, any quality control of source code committed to the Subversion repository must be carried out by the developers. The development of DHIS 2 has so far been "lucky" to have had a few committed core developers, master students, able to work as technical

[4]All modules in DHIS 2 can contain Hibernate mappings. The dependencies from each module define which modules are present in the classpath when each module's unit tests are executed, and the included Hibernate mapping files are automatically discovered and loaded on startup.

officers and advocates of principles, and staying with the project over long periods. The project coordinators have given these core developers the freedom to do what they think is best for the DHIS, and because participation in the development is voluntary, the core developers have had the power to do what they think is best, or what suits them the most. Related to the tools used, the core developers have chosen how they want to utilize them. Discussions have taken place on how to organize the system modules, and the tools being used have enabled the wishes of the developers, without ignoring the fact that the discussions are initially based on the tools at hand. But if, for example, Maven had been less plugin-oriented, and it was hard to extend Maven's functionality to create a web portal similar to the current solution, without ending up with an "ugly" result, a completely different approach would have to be taken to reach the desired goal, probably without greater influence from the project coordinators. Personally, I have appreciated this freedom, although a clearer stance from above has been missed a few times. So while a tool like Maven influences how DHIS 2 evolves in terms of system modules, and how the modules are combined to form a complete application, there is room for adoptation of alternatives if the current was not to suffice.

At the same time, it is always a good idea to try to keep the number of tools to a minimum. Even though Maven did not support dependencies between web archives, and DHIS 2 needed this, there was never really a question of trying a different tool. Maven 2 was already being used throughout the system for "regular" project management, and because Maven took care of packaging the web modules into web archives, it felt natural to have Maven take care of the final step of packaging the portal, too. The web portal project is but a web module itself, only with different dependencies. Pushing it a bit, what saved Maven was its extendibility, which allowed for custom, working solutions. The Maven documentation was rather poor in the beginning, so a few hacks in the first plugin were replaced by proper use of Maven provided functionality when the possibilities were discovered. Maven source code was read. But overall, the plugin system greatly contributed to making Maven worth using, as it provided options and customizability, while still taking care of all the other common tasks related to project management. One of the core developers of the DHIS 2 actually made the first Maven plugin which deployed a web archive directly in Jetty. Mortbay [13], the creators of Jetty, developed their own plugin much later. New releases of the Maven WAR plugin triggered changes in the web portal solution, as the goal was to not have any custom solutions. All the common web resources were collected in one or two common web modules. The day I could scrap the custom DHIS 2 plugins was a happy day, in spite of all the time and effort which had been put into them. The target solution for the web portal was more or less set with the first working web portal, even if it could not be immediately accomplished

in terms of good internal solutions. The web portal has basically been the same, in theory, during the development and time presented in section 6.3. Only the technical carry-out of the solution has changed, in line with the Maven releases and progressing developer ideas.

The parts of the portal which have not been directly affected by Maven, the composite web page system, the menu system, and the widgets for organization unit hierarchies, have had rather stable solutions. Although there were initial challenges to create them, and although there have been technical improvements here too, they have not gone through the same type of evolution as the portal merging mechanism. The web solutions are more pure Java and web solutions, using Java language constructs and the available frameworks for solving their intended jobs. And because the relevant parts of the frameworks have been quite stable, the solutions have been more immediately dependent on their developer or developers for improvements and change.

7.4 A new web archive

The initial building block for creating web applications with Java is the Java web archive. As defined by the Java Servlet Specification [61], a servlet container recognizes web archives as web applications which can receive HTTP requests through Java servlets. We have seen what WAR files contain, and how Maven projects can be converted to them, yet the WAR file is only the result of a specification building on the Java languange and web protocols. If there was to be a better way of organizing the resources of a web application and the communcation links between a web server and the application, a new specification could be designed. While a new standard might not be widely adopted because of the existence and widespread use of the Java Servlet Specification, the possibilities are there. The Java Servlet Specification is nothing more than a framework for the basic functions of a Java web application.

Keeping that in mind, one of the benefits *and* problems in DHIS 2 is that web archives contain their dependencies. The benefit of this is that WAR files can hold complete applications, making it simple to distribute and run the applications. The problem is, as explained earlier, that merging web modules require all the dependencies to be of the same build in order to have some control over which version ends up in the final application. At the same time, decompressing and compressing all the archives take a lot of time because of the compression algorithms and related disk operations. The feature of all web interface modules being standalone applications, as well as being able to be merged into one large application, is a great convenience for developing and testing web interface modules in isolation. But it necessarily comes with the issues just mentioned. For the most efficient building of the web portal, each

web interface module would not contain its dependencies. The transitive dependencies, from the portal's perspective, can be resolved by looking at the project descriptor files anyway. However, such web interface modules are not independent, deployable applications, which is important for isolated execution and short round trips when testing the modules. This duality of wanting to be complete in one situation and only contribute itself in another is less than obvious to solve. It is possible to create two different builds of each web interface module, with the default being the web portal friendly variant without dependencies. The other variant, a fully deployable application, can be specifically built when desired. That solution requires that a new plugin is created which can take the Maven WAR plugin's place in the build cycle, since the Maven WAR plugin does not have the required features. Also, the Maven Jetty plugin, which automatically executes all the phases in the build cycle up to the package phase, needs to tell the package plugin to include all the dependencies in the build, otherwise the default variant is executed. It should be technically possible, but I have unsolved problems with replacing a default plugin with a custom one, putting the project in a stale state. A benefit of the current solution is that it is less complex. Packaging a web interface module results in the same type of build every time—a fully functional web application.

Another problem with web archives is that the Java compiler does not automatically handle WAR files on the classpath—you cannot compile against classes which only exist in a dependent WAR file. The WAR file dependency is currently a Maven construct, handled by the Maven WAR plugin by overlaying the archives. The module for common web resources in DHIS 2 had to be split in two, because compile-time functionality is only available with JAR files, and web application files such as web pages and images can only be held by WAR files. A combination would have been appreciated, as the functionality is highly related to the web application files, and thus belong together in one module. A slightly different internal structure in web archives could have solved this without larger changes in the Java compiler, by making a WAR file an extended JAR instead. The current solution is simple, though: Every web interface module has two dependencies instead of one, one to the functionality and one to web application files.

An alternative to merging the WAR files to form a complete application would be to collect all the archives in an enterprise archive, an EAR file [44]. Enterprise archives are simply archives that can contain other types of archives, such as Java archives and web archives, in addition to one or more deployment descriptor files. A downside to EARs is that they require more complex application servers to run instead of just a servlet container, thus ruling out our beloved Tomcat and Jetty. Furthermore, if all the web interface modules in DHIS 2 are still packaged with their dependencies, the EARs will end up being as big as all the archives together,

with around fifteen equal copies of each common dependency. Thus EARs have not been seen as a good alternative for bringing together the portal, but I like the idea of a pool of dependencies and web interface modules. If a web archive did not contain its dependencies, and the deployable file corresponded to an application archive containing such “thin” WAR files and dependencies as separate JARs, then maybe it would have been easier to assemble different types of applications as needed in DHIS 2. What I mean is to turn web archives into simple holders of web content and related functionality, the direct contents of a Maven web project, which are not necessarily deployable by themselves, depending on the presence of dependencies. In such a case, the package phase of a Maven project would consist of building a Java archive or a web archive depending on the type of module, and then another plugin would take care of assembling the project build and all its dependencies into a deployable application archive. The Jetty Maven plugin would depend on the application assembly plugin being executed instead of the WAR plugin. If the web archives could be treated as regular Java archives as well, then functionality could be reused across web modules. A web archive would be nothing more than a JAR with extra resources, organized in special directories recognizable by a supporting web server.

The differences from an EAR might feel subtle at first, but the main difference is how WAR files are restructured into JAR files with web resources instead of being all that different as they currently are. A second difference is that application archives do not need more complex treatment than what a regular servlet container could handle—the main difference being unpacking the archive and deploying multiple web archives as one application, with all the dependencies in the same application archive. Figure 7.2 shows how a Maven project is suggested to be packaged into a web archive. The Java classes and application resources are compiled and merged exactly like a JAR file (compare to figure 5.4 on page 40), thus voiding the `/WEB-INF/classes` directory. The module dependencies are left out, voiding the `/WEB-INF/lib` directory. While the contents of the `/WEB-INF` directory in a regular web archive is not supposed to be returned to the clients by the web server, this solution needs to change that. I have kept the WEB-INF name for simplicity, but moved the public directory from the root of the archive to the `/WEB-INF/public` directory. So now everything is hidden from the public, except the contents of the just mentioned directory (compare to figure 5.5 on page 41).

The advantages are as follows: The files which are the same as in a Java archive, are placed in the exact same location in the web archive as in a Java archive. Thus both archive types can be used on the classpath for compiled references between classes in different modules. The web archive does not contain its dependencies, thus being as small as possible in size, and convenient for either merging or collecting

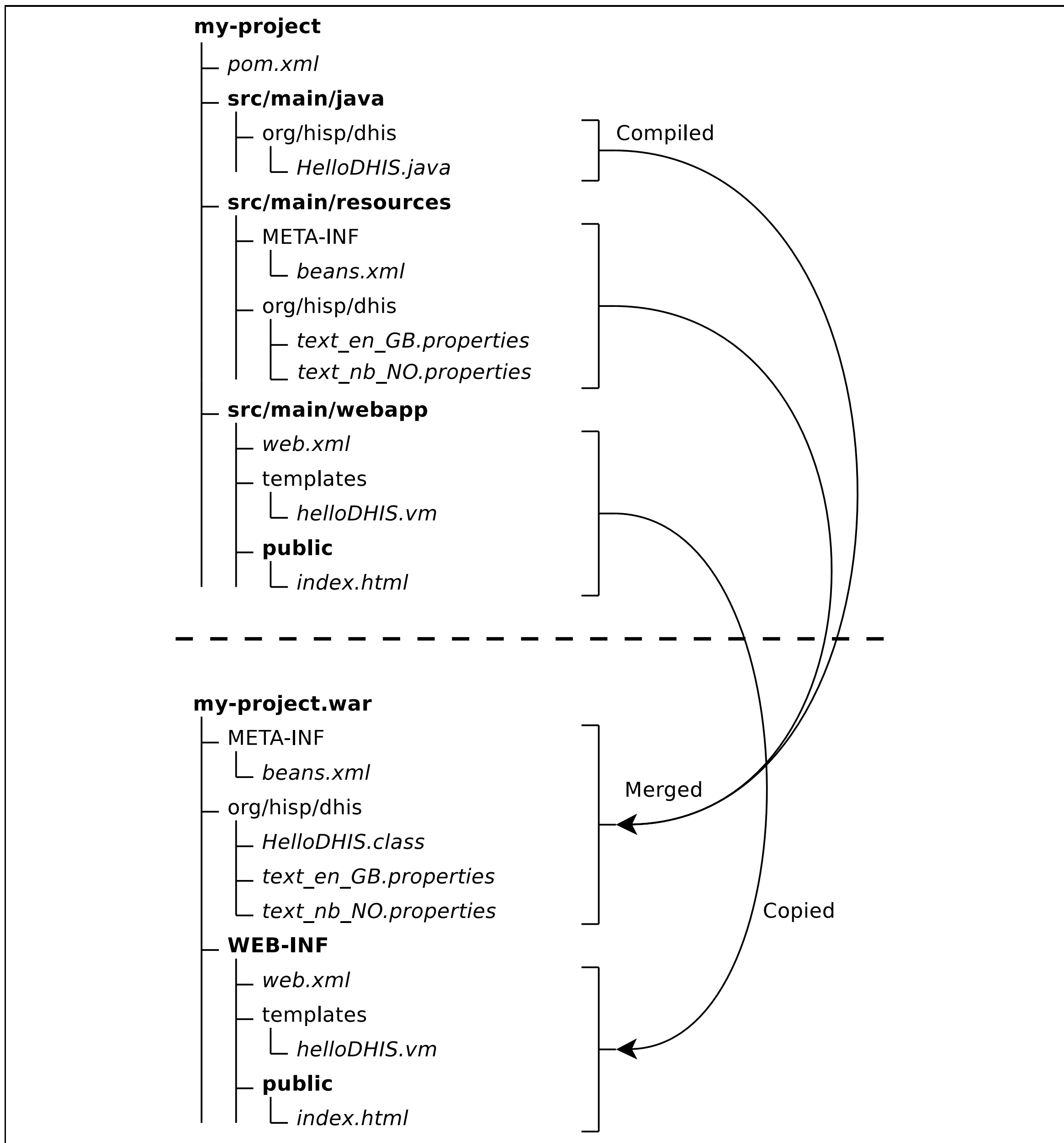

Figure 7.2: How a Maven project could be packaged into a web archive. Please compare with figures 5.4 (page 40) and 5.5 (page 41), on how Maven projects are actually packaged into Java archives and web archives. Left out from this figure is a Manifest.mf file, which should be automatically created and placed in the META-INF directory.

in a larger archive, just like Java archives. If web servers were built to support multiple web archives in a single application, there would be no need for merging them. Another advantage, which has not been touched upon before, is how web application resources are found in a servlet container context. If a web application is executed while being packaged as a normal web archive, the servlet container must put the root of the web archive on the classpath, so that the web resources can be fetched using classpath resource loaders. If, however, the web application is executed in an exploded state, that is, being extracted from its ZIP file before executed, the web application resources will not be on the classpath. It is then the responsibility of the servlet container to set a *realPath* property of the *servlet context* as defined in the Java Servlet Specification. The web application must then get hold of this value from the servlet context, and use regular file system mechanisms when accessing the web resources. During the development of DHIS 2, we have had repeated issues with this "real path" when trying to configure the Velocity template engine, as Velocity has constantly complained about not being able to find the templates (this is partially due to a poorly implemented Velocity configuration class in WebWork which is hard to improve by inheritance and overrides). The suggested new structure of a web archive, however, has the property of always having the web application resources on the classpath, as the root of the application must be on the classpath. One needs only to prepend the resource path with `WEB-INF/public` for the classpath resource loader to find the files. There would no longer be a need for the "real path".

I do think the alternative web archive would make a difference, but I am far from convinced that it would be adopted in the real world. The required efforts to build support for an alternative format are huge because of the momentum of the installed base [41]; the widespread use and implementation of the Java Servlet Specification [61].

7.5 Web portal architecture

If the DHIS 2 web portal architecture was to be characterized by a single expression, I would choose "configuration merging". The point of the portal is to make the web modules appear as one application, with one main channel for human interaction—through HTTP. In some way or another, client HTTP requests need to be redirected to the right web module for processing. All the web modules in DHIS 2 have their own namespace in the form of a virtual directory in the URLs, so identifying the target of a request is inherently trivial. The question is: Who or what is doing the redirection? It should be noted that the web interface modules in DHIS 2 do not communicate between themselves, they are, as repeatedly stressed, standalone modules, and none of them have any periphery detection systems for discovering

other modules. So communication is only initiated by incoming HTTP requests, and the request redirector in DHIS 2 is WebWork, or actually XWork. Upon system startup, WebWork is fed with all the modules' XWork configuration files (because it is unable to grab them itself), containing namespaced packages with action mappings. The configuration files are merged inside WebWork when loaded and transformed into an internal object model. Similarly, the Spring Framework creates a network of bean factories, instantiating action classes from all modules by requests from WebWork. Thus none of the frameworks are actually aware of the fact that files and resources are provided by a collection of initially separate web modules. The Java classpath hides the boundaries between the archives, as it presents a single interface to locating files, similar to a regular file system. The "real path", if present, is a single directory where all the web resources of the modules have already been merged by the Maven WAR plugin. And the XWork and Spring configuration files work as composite address books targeting the classpath and the "real path". Even if the files were spread out and not fetchable through one or two interfaces, the framework in which the configuration files are loaded would still need to support loading files from the various locations specified in the configurations, regardless of which modules use which types of locations. The configuration files simply specify paths and appear as one configuration when loaded by a single framework. The web interface modules are effectively demodularized into one seamless application when packaged into the web portal project. Hence the term "configuration merging".

In a Service Oriented Architecture (SOA), the services are distributed, and communicate with each other using remote interfaces [65]. And the concept of *orchestration* requires the communication channels to be dynamically configurable. DHIS 2 web interface modules do not currently need dynamic channels, and the closest DHIS 2 comes to orchestration is selecting the desired Maven dependencies in the web portal module before merging and packaging the portal, or on a submodule level, setting up the bean dependencies in the Spring Framework. But then we should consider the differences between a SOA service and a DHIS 2 web interface module. SOA services are typically geared towards offering access to and linking businesses, while DHIS 2 web interface modules are more in the direction of implementing different use cases for the same business. Both are high-level, but a SOA service can be seen as a business application, with or without a graphical user interface. An outward interface of such a service could possibly be decomposed into separate interface modules, similar to the DHIS 2 web portal. Strictly speaking, the DHIS 2 web portal does not have a graphical user interface, but one is constructed by a web browser based on the information which is returned by the portal. So it is not that far fetched to portray the DHIS web portal as one SOA service, although it is not intended as or built to be one. The DHIS 2 application, which is a result of combining the web interface

modules, targets one business, the business of routine health management. A SOA service targeting the same business could, for example, communicate with a service for human resources or supplies, or simply another routine health system in the organizational hierarchy. Linking DHIS 2 instances over a network has actually not been discussed, although the discussions have not been influenced by a SOA perspective. Another aspect here is that web interface modules from a different business could as well be created and added to the portal. "Bloating" the DHIS 2 is perfectly possible, but communication wise these modules would typically use the internal service layer for cross-business communication, and not a configurable message bus.

Even though the web interface modules in DHIS 2 do not communicate between themselves, an Enterprise Service Bus could have been utilized for routing requests from server to module. Such an approach would have made the modules less dependent on using the same technology and frameworks for producing responses—an ESB can be made independent of end point implementation details such as language and platform [32]. A decoration framework such as SiteMesh [18] can be used for aligning the looks of web pages from the various modules, that is, operating on the server side of the ESB. In a global network of development teams like HISP, this could really lower the barrier for participation in the development of the system, because everyone would be able to use their favourite languages and frameworks. I would worry a little bit about performance, though, as the ESB, the site decorator, and every added technology used in various services add computational overhead to the application, and the DHIS is typically not installed on high-end servers. System installation might also become more complex, and maintenance wise you would become more dependent on the various developers. In these regards, the limited number of technologies and the merged nature of the DHIS 2 web portal might be preferable, as there are only a couple of frameworks taking care of all the requests, and all internal communication is based on Java method calls. Of course, it is perfectly possible to have an Enterprise Service Bus between modules using the exact same technology (which OpenHealth [7][5] does).

A cache on the server side of the ESB would help on the perceived performance, but this requires that application state is more in the hands of the client. If the provided URL contains the page state, a cached page with that exact URL should be close to correct with respect to the time difference between the requests. If the client is dependent on application state being stored on the server, the server is bound to process the whole request every time, in order to return an accurate response. In DHIS 2, the state of the organization unit tree widget is always kept on the server, thus

[5] The reference is to HealthMapper, which is the next version of OpenHealth. HealthMapper is based on the same technologies as DHIS 2, in addition to that it uses an internal ESB, and is RESTful [35].

making it harder to cache pages containing it. For example, if one specific organization unit is selected, and the user navigates to another page where another organization unit is selected, then the server and client would become out of synchronization if the user was to go back to the first page, without that page being regenerated by the server. And a form submit could result in data being stored with the wrong organization unit. DHIS 2 is not RESTful [35]. It is partly the organization unit tree widget's fault—the way it is constructed. If the tree state was represented in the URL, then all URLs on a page would have to be dynamically replaced after each change in the tree widget, if the widget was still using AJAX for loading its content. Otherwise, all the URLs would have to be generated on the server according to the state. In any case, it would require a considerable refactoring of DHIS 2 if the state of the tree was to be incorporated into the URLs. Most of the pages which do not contain the organization unit tree widget are RESTful, though, because the pages are simple and their state requires mostly zero or one argument, typically an ID to an object to display. Without going into details, there are other elements too which are not directly addressable, most of them using AJAX, and most of them using the HTTP session for storing the current state. It might be easier to create RESTful web applications if JavaScript was used more extensively. A link can be made to call a JavaScript method, which can assemble the URL from state variables, and then send the appropriate request to the server. It seems to me that RESTful web pages should be either completely generated on the server, or close to completely generated on the client with regard to generating URLs, but this requires further research (see chapter 9). Static pages with a touch of dynamics, like many of the DHIS 2 web pages, require intervening scripts from dynamic sections to static sections, adding a layer of complexity. There is no current need for making DHIS 2 completely RESTful, but I would seriously consider its compelling aspects in other projects.

There are plans to integrate DHIS 2 with other health information systems like OpenMRS [15] and OpenHealth [7] (see footnote 5) on the application level. SOA and REST can support this infrastructure, and OpenMRS has already a REST module which can be used for inter-system communication.

Chapter 8

Conclusion

The starting point of this thesis was to create a framework for combining a set of web modules into a seamless application, as defined by the primary research objective.

- **Primary research objective:** Find a working solution for combining web interface modules into a complete web application with focus on reuse and simplicity for developers, and transparency for users.

The primary research objective has been met by the proof of concept development of the DHIS 2 web portal. The different parts of the web portal are the process of merging the web interface modules into one deployable application archive, the template solution for creating web pages with a common look across modules, the menu system providing access to the module web pages, and the organization unit tree widgets, which are common to all the web modules. The problems are general and relevant for many information systems other than DHIS 2. And the solutions have been developed as general as possible, so that thay can be applied and adapted to other systems with little effort.

Maven 2, the project management tool used in DHIS 2, has provided the base for merging the web interface modules. Because of the early adoption of Maven 2 and its development alongside DHIS 2, the web portal has gone through multiple phases of technical solutions, including custom Maven 2 plugins, web application resources spread out over the web modules, and support functionality for loading framework configuration files. While the technical solution has changed with time, the web portal idea and the ultimate result of it have remained the same. This part of the web portal solution, as it stands today, is a simple configuration of the Maven WAR plugin, common web application resources and functionality collected in two common web modules, namespaces for collision avoidance, and helper functionality for loading XWork configuration files. Each web interface module is separately deployable in a servlet container, in the same way as the web archive produced by the web portal

solution. The solution is generic, and can be used with all frameworks that support loading and merging of configurations from multiple sources.

The common look across the module web pages is provided by a simple template system built on top of WebWork. The concept behind this is to have background templates which include page content templates and web resource references at desired locations in the web documents. The template system uses static parameters in the action mappings, and a WebWork interceptor which forwards selected static parameters from the action mappings to the action result assembly. The result is a completely transparent solution, where the action classes have no awareness of the template system, and any template engine supported by WebWork can be used.

For the users to get access to the module web pages, a menu system has been created, which lists the index actions from the module action mappings. This solution is simple, and can be adapted to any web framework which provides access to its configuration. But the simplicity does not necessarily result in the most suitable menu from a user's perspective. Therefore, I propose a new menu system which is based on a configuration file in each module. The configuration files are loaded by adaptable functionality, which should provide more flexibility for the developers than what the current solution does.

The last part of the web portal solution is the organization unit tree widgets. The tree widgets are common to all modules, and are thus best placed in common web modules. Both trees use AJAX for communication with the server, so that no page state is lost when the users interact with the widgets, and the tree states are saved in the HTTP session. There has been a few technical issues with the development of the tree widget which prepares the tree state on the server before the web pages are returned, but nothing that could not be solved or worked around. A new technical implementation of the form input widget has been proposed, so that it will save its state in the web form in which it is included, instead of in the HTTP session. The transformation will make each instance of the form input widget completely separate, which means that there is no need to reset the widget between each web page which uses it, and multiple instances can be used simultaneously by the same user.

The decomposition of the web layer comes with a cost. Each web archive must contain its dependencies for all the web interface modules to be separately deployable. And because of that, large amounts of computer resources are used when merging the web archives together, making the build processes more time consuming. In addition, measures have to be taken in order to know which build of a service module is included in the portal, basically forcing all web modules to be rebuilt. And web archives with common functionality must be split in two for the functionality to be available to other modules. All of this lead to the design of an alternative web archive which does not contain its dependencies. This solves the mentioned problems, but another

archive must be created for deployable applications, which can contain the new web archive and all its dependencies. However, because the current web archive is defined in the Java Servlet Specification [61], and because of all the effort it would require to build support for it, it is not likely that a new, independent web archive would easily gain a foothold.

The final result of the web portal solution is a demodularized web application, where the Maven modules are not visible to the users.

- **Secondary research objective:** Investigate how Java language constructs, tools, and frameworks affect and contribute to the modularization of a system.

We have seen that the Java language has multiple constructs which enable modularization. Methods and their modifiers define the two important class interfaces for use and extension, which, when properly designed, secure flexibility, extensibility, reusability, testability, and forward compatibility. Classes can be combined in hierarchies, and can, together with interfaces, form black boxes for internal flexibility. Classes and interfaces can be organized in modules, managed by for example Maven, and modules can be combined into applications. Java proxies provide native AOP capabilities for extracting the cross-cutting functionality out into separate classes, yielding the ultimate separation of concern.

We have also seen how the choice of a project management tool like Maven 2 implies a choice of conventions and work patterns, including lowering the threshold for creating new modules. The development of Maven 2 alongside DHIS 2 has affected the development of the web portal with regard to the complexity of the portal solution, and to where it is possible to place web application resources. And the modularity of Maven 2 itself has strongly influenced the solutions developed.

Finally, we have seen how the quality and cross-system influence of frameworks and libraries depend on the developers' intentions and predictions, and their use and awareness of the Java language constructs. Frameworks and libraries are likely to be adapted to application contexts, and the construction and extensibility of the frameworks and libraries affect how the applications are formed, and how application solutions might need to work around potential problems.

Modularity means that developers in different teams can work independently of each other on the same code base. Context specific classes and modules can be created for local customization of well modularized systems, while the core internal structures can be standardized across the installations. DHIS 2 requires that all modules are developed in Java, but with a different internal architecture other technologies could have been utilized, at the expense of a more complex system.

Chapter 9

Further Research

There are several open ends regarding the problems and solutions presented in this document. These are worth following up, and are described here in various detail:

- **Implement the suggested menu system:** In section 6.5.1, a new menu system is suggested, which would provide more flexibility to the developers in setting up a systemwide main menu.

- **Implement the suggested organization unit tree widget:** In section 6.5.2, the organization unit tree widget which is intended for use in input forms, is suggested to be changed to not use the HTTP session for storing its state. This would make the tree function more like a regular input form element with the described benefits.

- **Make the Maven WAR plugin behave differently, depending on whether the Jetty plugin is executed or not:** In section 7.4, I suggested that the web interface modules are packaged without their dependencies by default, and only packaged with their dependencies if the Maven WAR plugin is executed in a build process which ends with the execution of the Maven Jetty plugin (or any plugin which needs the web archives to contain a deployable application). This would minimize the size of the web archives when they are merged in the web portal, but the web interface modules would still be separately deployable.

- **Look at the cost of making DHIS 2 RESTful:** There are several elements in the DHIS 2 web modules which are not RESTful, but should be possible to convert. What would it entail to change DHIS 2 into being RESTful?

- **Look at how the unit tests can be made faster:** In section 7.3, the unit tests are described as a major source to the long build time, partly because the unit tests themselves are inefficient. If the unit tests could be changed to not

start and stop the Spring Framework, Hibernate, and the DBMS every single time, this would save a lot of time.

- **Look at alternative architectures**: Related to modular web interface systems is a relatively new architectual style called Service-Oriented Front-End Architecture (SOFEA) [56], which would be worth looking into. SOFEA seeks to decouple presentation tier processes into application download, presentation flow, and data interchange, and supports applications based on REST.

Glossary

- **Classpath**: The classpath is an argument to the Java Virtual Machine (JVM) which consists of a set of paths to user-defined classes and packages. Classes will be automatically searched for on the classpath and loaded as needed by the JVM, but a program can also load classes and resources from the classpath using dedicated class loaders. Java archives can be conveniently put directly on the classpath in packaged state and used as if they were unpackaged.

- **Exceptions, checked vs unchecked**: Checked exceptions are exceptions which must be explicitly declared or handled using the *try/catch* and *throws* clauses. Methods which throw checked exceptions must declare that they can throw such exceptions. Checked exception should be used when an error situation is recoverable and expected to be handled by the system. Unchecked exceptions do not need to be declared as throwable in a method signature, and they do not require any explicit handling by potential callers of a method. Unchecked exceptions should be used for error situations which a system is not expected to recover from—or unexpected problems—and to indicate programmatical errors in the system. Unchecked exceptions are subclasses of `java.lang.RuntimeException` or `java.lang.Error`, while checked exceptions are subclasses of `java.lang.Exception`.

- **Packaging vs Java packages**: Java packages are directories in which Java classes are organized. Packages typically work as namespaces, and classes, methods, and properties can be made accessible only to the package in which they are declared, if desired. Packaging is the process of creating an archive out of a set of packages, classes, and resources. Archives are compressed files, usually ZIP files.

- **Servlet container**: A servlet container is a part of a web server which conforms to the Java Servlet Specification. The servlet container can handle and forward web requests to Java web applications through servlets implemented by the applications. For a Java web application to be deployable in a servlet container it must also adhere to the Java Servlet Specification.

- **Web server**: A piece of software which responds to HyperText Transfer Protocol requests on a specified port on a computer, typically port 80. The responses must also be HTTP.

Bibliography

[1] Acegi Security (website). http://www.acegisecurity.org.

[2] Ant (website). http://ant.apache.org.

[3] Apache Tomcat (website). http://tomcat.apache.org.

[4] AspectJ (website). http://www.eclipse.org/aspectj.

[5] Code Generation Library (website). http://cglib.sourceforge.net.

[6] Eclipse (website). http://www.eclipse.org.

[7] HealthMapper (website). http://www.who.int/health_mapping/tools/healthmapper/en/index.html.

[8] Hibernate (website). http://hibernate.org.

[9] Hypersonic database (website). http://hsqldb.org.

[10] Ivy (website). http://ant.apache.org/ivy.

[11] Java Server Faces (website). http://java.sun.com/javaee/javaserverfaces/.

[12] Maven 2 (website). http://maven.apache.org/.

[13] Mortbay (website). http://www.mortbay.org.

[14] MySQL (website). http://www.mysql.com.

[15] OpenMRS (website). http://openmrs.org.

[16] PicoContainer (website). http://www.picocontainer.org.

[17] PostgreSQL (website). http://www.postgresql.org.

[18] SiteMesh (website). http://www.opensymphony.com/sitemesh.

[19] Spring Framework (website). http://springframework.org.

[20] Struts 2 (website). http://struts.apache.org/2.x/.

[21] Struts (website). http://struts.apache.org.

[22] Subversion (website). http://subversion.tigris.org.

[23] Tapestry 5 (website). http://tapestry.apache.org/tapestry5/.

[24] Tomcat Maven plugin (website). http://mojo.codehaus.org/tomcat-maven-plugin.

[25] UNAIDS (website). http://www.unaids.org.

[26] Webster's Online Dictionary (website). http://www.websters-online-dictionary.org.

[27] Wicket (website). http://wicket.apache.org.

[28] The Apache Software Foundation. *POM Reference: The Super POM*, April 2007. http://maven.apache.org/pom.html#the_super_pom.

[29] Jørn Braa and Calle Hedberg. The struggle for district-based health information systems in south africa. *Inf. Soc.*, 18(2):113–127, 2002.

[30] Jørn Braa, Eric Monteiro, and Sundeep Sahay. Networks of action: Sustainable health information systems across developing countries. *MIS Quarterly*, 28(3):337–362, 2004.

[31] Eric M. Burke and Brian M. Coyner. *Java Extreme Programming Cookbook*. O'Reilly, mar 2003.

[32] David A. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[33] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, June 2004.

[34] Oscar Diaz and Juan J. Rodriguez. Portlet syndication: Raising variability concerns. *ACM Trans. Inter. Tech.*, 5(4):627–659, 2005.

[35] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2(2), May 2002.

[36] The Apache Software Foundation. Maven 2 central repository. Website. http://www.ibiblio.org/maven2/.

[37] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uriá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: A quantative study. 2005.

[38] Greg Goth. Critics say web services need a rest. *IEEE Distributed Systems Online*, 5(12), December 2004.

[39] Dick Grune. Concurrent versioning system. Website, June 2007. `http://www.sc.vu.nl/~dick/CVS.html`.

[40] Ole Hanseth, Eduardo Jacucci, Miria Grisot, and Margunn Aanestad. Reflexive standarization: Side effects and complexity in standard making. *MISQ*, 30, 2006.

[41] Ole Hanseth and Kalle Lyytinen. Theorizing about the design of information infrastructures: Design kernel theories and principles. 4:207–241, 2004.

[42] Ole Hanseth, Eric Monteiro, and Morten Hatling. Developing information infrastructure: The tension between standardisation and flexibility. *In Science, Technology and Human Values*, 11(4):407–426, 1996.

[43] Les Hatton. Is modularization always a good idea? *Information and Software Technology*, 38:719–721, 1996.

[44] Matthew Hosanee. Dissecting EARs - Enterprise ARchive (.ear) Files. March 2003. `http://access1.sun.com/techarticles/DissectingEARs/DissectingEARs.html`.

[45] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, October 1999.

[46] Mamdouh Ibrahim, Kerrie Holley, Nicolai M. Josuttis, Brenda Michelson, Dave Thomas, and John deVadoss. The future of soa: What worked, what didn't, and where is it going from here. *ACM*, October 2007.

[47] Rod Johnson and Juergen Hoeller. *Expert One-on-One: J2EE^{TM}Development without EJB^{TM}*. Wiley Publishing, Inc., 2004.

[48] Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, and Dmitriy Kopylenko. *Professional Java Development with the Spring Framework*. Wrox Press Ltd., Birmingham, UK, UK, 2005.

[49] Seth Ladd, Darren Davison, Steven Devijver, and Colin Yates. *Expert Spring MVC and Web Flow (Expert)*. Apress, Berkely, CA, USA, 2006.

[50] Reuven M. Lerner. At the forge: Ajax application design. *Linux J.*, 2006(152):9, 2006.

[51] Patrick Lightbody and Jason Carreira. *WebWork in Action (In Action)*. Manning Publications Co., Greenwich, CT, USA, 2005.

[52] Esperanza Marcos. Software engineering research versus software development. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[53] Kristian Nordal. The challenge of being open - building and open source development network. Master's thesis, University of Oslo, 2006.

[54] OpenSymphony. Issue xw-295: Support pojo actions. Website. `http://jira.opensymphony.com/browse/XW-295`.

[55] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[56] Ganesh Prasad, Rajat Taneja, and Vikrant Todankar. Life above the service tier: How to build application front-ends in a service-oriented world. October 2007.

[57] Jørn Braa, Ole Hanseth, Arthur Heywood, Woinshet Mohammed, and Vincent Shaw. Developing health information systems in developing countries: The flexible standards strategy. *MIS Quarterly*, 31, August 2007.

[58] Knut H. Rolland and Eric Monteiro. Balancing the local and the global in infrastructural information systems. *The Information Society*, 18(2):87–100, 2002.

[59] Walt Scacchi. Free/open source software development: recent research results and emerging opportunities. In *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 459–468, New York, NY, USA, 2007. ACM.

[60] Howard M. Lewis Ship. *Tapestry in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[61] Sun Microsystems Inc. *Java^{TM}Servlet Specification Version 2.4*. `http://java.sun.com/products/servlet/download.html`.

[62] Sun Microsystems Inc. *JAR File Specification*, February 2007. `http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html`.

[63] Michael VanHilst and David Notkin. Decoupling change from design. 1996.

[64] Lars Helge Øverland. Global software development and local capacity building: A means for improving sustainability in information systems implementations. Master's thesis, University of Oslo, 2006.

[65] Steve Vinosky. Rest eye for the soa guy. *IEEE*, 2007.

[66] Urjaswala Vora. Modularization with externalization of control flow. 2007.

[67] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. 1981.

[68] Robert K. Yin. *Case Study Research : Design and Methods (Applied Social Research Methods)*. SAGE Publications, December 2002.

Appendices

Appendix A

Screenshot of Maven WAR plugin issue MWAR-45 regarding where classes should be placed in a WAR file (`http://jira.codehaus.org/browse/MWAR-45`). Comment 3 and 4 (second part of the screenshot) are my contributions to the discussion of the issue.

[#MWAR-45] add config prop to specify webapp classes should be zipped and placed into WEB-INF/lib/xxx.jar instead of placed ...

File Edit View Bookmarks Widgets Tools Help

http://jira.codehaus.org/browse/MWAR-45

codehaus

History | Log In

HOME BROWSE PROJECT FIND ISSUES QUICK SEARCH:

Issue Details (XML | Word | Printable)

Key:	MWAR-45
Type:	New Feature
Status:	Closed
Resolution:	Fixed
Priority:	Major
Assignee:	Jason van Zyl
Reporter:	Ian Springer
Votes:	2
Watchers:	3

Operations

If you were logged in you would be able to see more operations.

Maven 2.x War Plugin

add config prop to specify webapp classes should be zipped and placed into WEB-INF/lib/xxx.jar instead of placed in WEB-INF/classes/

Created: 30/May/06 06:28 PM Updated: 11/Jun/06 01:34 PM

Component/s:	None
Affects Version/s:	2.0
Fix Version/s:	None

Original Estimate:	Unknown	**Remaining Estimate:**	Unknown	**Time Spent:**	Unknown
File Attachments:	1. mwar-45.patch (6 kb)				

Description « Hide

I think this is a common enough practice that there should be an option provided for it.

All Comments Work Log Change History Sort Order:

Prasad Kashyap - [31/May/06 11:29 PM] [Permlink | « Hide]

<prasad> bporter: if I had to do something like that in m2.. jar clases into web-inf/lib.. how would you recommend I do that ?
<bporter> it probably needs to be an option of the war plugin
<bporter> I'm yet to see why this is a good idea
<bporter> oh course, you can always put it in 2 projects now, too
<prasad> let me give you the req'ment from Geronimo..
<prasad> we recently ran into a problem where the path names were too long for windows to handle..
<prasad> so we jar'ed them into web-inf/lib
<bporter> heh
<bporter> ok, well you could code up a patch for the war plugin and we'll apply it
<bporter> the jira was filed just yesterday I tink
<prasad> i see it.. this.. http://jira.codehaus.org/browse/MWAR-45
----

patch introduces config parameter "classesArchiveName"

David Jencks - [07/Jun/06 10:39 AM] [Permlink | « Hide]

I think adding this capability is a very good idea. IIUC the only argument against it is that it is possible to get the same effect by putting the classes into a separate project. I think this ignores the different effects and affect of having 2 projects:

1. Having 2 projects forces you to maintain what you are likely to be thinking of as one project in two places: e.g. even without jsps, the web.xml is going to be far away in the file system and IDE from the classes it is the descriptor for. I think breaking the way people think about their project in this way is inadvisable.

2. If you have jsps, you cannot precompile them and include them in the project classes jar without moving them into the classes project. What should be a packaging option requires major svn changes to your project.

3. With 2 projects you are forced to publish the classes jar. You may not want to pollute your repo with this artifact that you may regard as unnecessary. Also, you are apt to want to name both projects with the same name.

4. If you want 2 profiles, one to pack classes into a jar and the other to leave them in WEB-INF/classes, you are forced to unpack the jar if the jar is from a separate project. This is going to be a bit slower than not creating the jar in the first place.

I'm sure there are more, this is just what came to mind during a moments consideration. war files are a pretty strange and perhaps inadvisable construction, but I think it is better for maven to try to adapt to what they are and how they are used rather than trying to force everyone to essentially use something else.

(Screenshot continued)

Torgeir Lorange Ostby - [08/Jun/06 05:04 AM] [Permlink | « Hide]

This feature makes it easier to overlay war files if you have files with the same name and path in the war projects you want to merge (e.g. in src/main/resources). With this, these files will end up in their respective jars instead of replacing each other during the overlay process (take the application context descriptor as an example).

Torgeir Lorange Ostby - [09/Jun/06 05:41 AM] [Permlink | « Hide]

Would it be better if classesArchiveName defaults to ${project.build.finalName} and then have an additional parameter/flag which triggers the feature? I guess

```
<configuration>
<classesArchiveName>${project.build.finalName}</classesArchiveName>
</configuration>
```

would give the same result, but it is not that intuitive that this is what actually makes the plugin behave differently.

Ian Springer - [09/Jun/06 07:55 AM] [Permlink | « Hide]

+1 to Torgeir's suggestion. I think having a boolean parameter for enabling the option is more intuitive.

Jason van Zyl - [11/Jun/06 01:26 PM] [Permlink | « Hide]

I agree that there should be a boolean parameter and the JAR name should come out with the standard Maven naming conventions.

Jason van Zyl - [11/Jun/06 01:34 PM] [Permlink | « Hide]

Patch applied and snapshot deploy, so grab the snapshot and make sure it works correctly for you. I tried it quickly with a webapp I'm working on and it seems to work.

There is now a boolean option to archive the classes and the JAR produced follows standard Maven naming conventions.

Powered by a free Atlassian JIRA open source license for Codehaus. Try JIRA - bug tracking software for *your* team.

Atlassian JIRA the Professional Issue Tracker. (Enterprise Edition, Version: 3.9-#233) - Bug/feature request - Contact Administrators

Appendix B

E-mail sent to the developer mailing list by me regarding the use of Subversion for maintaining context specific branches of DHIS 2:

```
On 11/28/07, ************* <address at hidden> wrote:
>
> Torgeir Lorange stby wrote:
> > On 11/28/07, ********** <address at hidden> wrote:
> > > On Nov 28, 2007 5:17 AM, ********* <address at hidden> wrote:
> > >
> > >
> > > > Yes, we almost based on MM module developed by you.
> > > > What we did are: fix some bugs, treat ICD as another
> > > > dimension a long with dataelement, period, orgunit. We
> > > > did not restrict to sex, age etc but rather allow
> > > > users to define these flexibly as a new dimension.
> > >
> > > Excellent *****, please commit what you have done to the repo.
> >
> > It might not be that simple to just commit to the repo. It depends on
> > how ***** has organized his code. I’m wondering, should we keep
> > branches for countries or "installations"? I mean, if the Vietnamese
> > team had their own branch they could work on it, commit to it, make
> > the local and custom changes they need, and copy/merge from trunk when
> > they wanted new features or bug fixes done by other people. Also, it
> > would be much easier for us to copy/merge from them when they make
> > something we want in trunk.
>
> Do we not have local/vn, incubator and sandbox for this? Do you mean an
> additional place in the repo?

I guess I mean to replace trunk/local/vn with branches/vietnam. The
trunk/local/ directory only contains single custom modules. But in
many cases the local teams need to customize other modules too, like
e.g. changing the default user interface locale in the i18n module.
Such local customizations need to be maintained somehow, and
Subversion supports this by the ability to merge code. It would also
give the local teams greater flexibility in choosing when and what to
grab from trunk. We’ll still have a joint effort in developing DHIS 2,
only now the local teams use Subversion to a greater extent for
managing their customizations, and we can all easily grab from each
other (much like how distributed version control works, but that’s
another thing). Of course, the local teams must be willing for this to
work.

Torgeir
```

Appendix C

E-mail sent to the developer mailing list by one of the main developers regarding reducing the number of modules in DHIS 2:

```
Thu May 24 15:38:22 CEST 2007

Hello all.

We've had several discussions over the last few days as to how to make
DHIS 2 smaller and simpler. The discussions around the dhis-core project
has been going on for a very long time. Torgeir, ******* and I tried to
conclude on it a couple of days ago.

Basically, we're suggesting to merge most of the stuff found in
dhis-services into the core, with the exception of a few projects. The
core will not be organized as dhis-services as now, with a series of sub
projects (dhis-service-organisationunit, dhis-service-dataset). Rather,
most of the stuff will be consolidated into three modules and one parent
project:

dhis-core/
  dhis-core-api
  dhis-core-persistence(-hibernate?)
  dhis-core-services

The contents of dhis-core-services module will be something like this

src/
  main/
    java/
      org/hisp/dhis/
        aggregation
          DefaultAggregationService.java
        organisationunit
          DefaultOrganisationUnitService.java
        dataelement
        period
        etc
    resources/
  test/
    java
    resources

Why? Because:
a) It will reduce the number of modules in the system, meaning less
modules to build, meaning less build and test environments to set up and
```

fewer cycles in Maven. Furthermore, it means that there will be fewer places for developers to search for functionality.
b) It means an extreme reduction in the number of directories in the system. Each module is itself a directory and contains src/{main,test}/{java,resources} directories, pluss individual org/hisp/dhis/xxx directories x 2. This should reduce the total size of the system, and also less .svn-folders everywhere, which should hopefully reduce the SVN download time.
c) It means less dependencies to put in your POM files. Instead of dependencies to dhis-api, dhis-service-core, dhis-service-dataset, dhis-service-organisationunit, etc, etc, you'll often have a maximum of three dependencies.
d) Makes it easier to separate the core project from other stuff, with the goal of releasing it as it's own general purpose project in the future.

Consequences:
- It may, but not significantly, improve the overall performance of the running system.
- The change will initially not affect the API, so the code in the web modules will not have to be rewritten. The only change will be changing (ie reducing) the dependencies in their POM files.
- The original three layers, api, persistance and service, will be retained.

So: This is a change for simplification, not for speed.

We suggest merging in the following modules:
dhis-service-aggregationengine-default - Essential for doing anything with indicators.
dhis-service-organisationunit-hibernate - Default, most commonly used implementation of Source, which is in the core.
dhis-service-expressionengine-default - (Will be) Used by indicators and validations alike.
dhis-service-user-hibernate - May/should be connected to the storedBy field in data values.
dhis-service-minmax-hibernate - Are/should be connected to sources, not org units.
dhis-service-validation - Are/should be connected to sources, not org units. dhis-service-dataset - We like it that way. Nuff said.

The new modules:
dhis-core-api will contain most of the current stuff in dhis-api, all the interfaces and model objects related to core functionality. DataElement, DataElementStore, DataElementService.

dhis-core-persistence will contain all the store implementations for the different model objects, HibernateDataValueStore, HibernateDataElementStore, etc. We're a bit uncertain how to deal with

different implementations, for example stuff using JDBC, related to the
ability of replacing the persistence layer.

dhis-core-services will contain default implementations of the core
services: DefaultDataElementService, DefaultOrganizationUnitService, etc.

Modules that will remain outside the core:
dhis-support - It's support, most modules will depend on these modules
anyway.
dhis-service-dataprovider-hibernate - Extra service for getting data.
dhis-service-importexport-default - Data interchange, not necessary for
the core. Depends on dataprovider, which is outside.
dhis-i18n - It's an aspect of the system, but not necessary.
dhis-useradminandsecurity > dhis-security - It's an aspect of the system,
but not necessary.
dhis-populator - Extra service for test data
dhis-service-datamart - Extra service for pivot tables.
dhis-options - Extra service for user and system settings.

These modules may be rearranged and renamed.

What will need to be done:
- i18n must be disconnected from the core even more, if possible. I leave
it up to ******* and Torgeir to battle this out in a caged pit, with
weapons of minor destruction strewn randomly around on the floor.
- Lots of boring work moving the files and directories around.
- Changing (reducing) the dependencies in most of the modules.
- Possibly: Connect users to the storedBy field, and add functionality for
disabling users. People may quit, but their data values must remain and
their users must not be allowed to enter more data. Will make a ticket for
this.

-- =

Appendix D

E-mail sent to the developer mailing list by one of the core developers regarding hiding the persistence layer from the user interface layer in DHIS 2. The e-mail contains an error regarding the scope of using no modifier. Sub-packages do not have access to package-private interfaces or classes.

```
Subject: Restricting access to DataStore

Hi,

Take note of this issue:
http://www.hisp.info/jira/browse/DC-122

Restrict access to org.hisp.dhis.service.datastore.DataStore

"We should restrict access to the DataStore interface, since _NO_
other classes should use this except classes in the same package or
sub-package (implementations of DataStore and implementations of the
interfaces above it; RoutineDataStore, ...., PeriodDataStore). We
will set it to package private (no specifier)."


--
```

Appendix E

Two e-mails sent to the developer mailing list by me regarding placement of functionality in and around WebWork actions, interceptors, and pre-result listeners in relation to the organization unit tree widget, transaction management, and lazy loading.

E-mail one:

```
Subject: XWork/WebWork architecture image

Have a look at this:
http://www.opensymphony.com/webwork/wikidocs/Architecture.html

According to this image, all the interceptors are executed completely
before and completely after the action and result is executed. This
should mean that the transaction, controlled by the
transactionInterceptor, is open until after the result is executed.
But this doesn't happen. The transaction is closed before the result
is executed (all of you who have gotten the
LazyInitializationException know this). So the transaction is closed
before the result is executed, but if I use the
organizationUnitTreeInterceptor to create the orgUnit tree before the
transaction is closed, the tree is not displayed on the resulting web
page. I.e. the result is created before the
organizationUnitTreeInterceptor is executed. So how can the
transactionInterceptor close the transaction before the result is
executed, and the result doesn't contain the orgUnit tree when the
tree is create before the transaction is closed?

Currently the tree is created in a PreResultListener, i.e. after the
action is executed and before the result. This works, except that the
PreResultListener cannot throw any exceptions (#@!$) and is executed
regardless of any exceptions thrown in the action class. Everything
works fine also, when the orgUnit tree is created before the action
invocation, but then the action cannot change the values of the
organizationUnitSelectionManager before the tree is created. If the
tree is executed after the result is executed, the tree will, of
course not make it to the web page, but then again, the transaction is
closed after the tree is created but before the result is executed.

What I get from all this is that the result is executed in two steps.
The result is prepared before the execution thread returns through all
the interceptors, but the template values are inserted after
everything, i.e. outside the interceptors, excluding all the values
created by the interceptors' after() methods.

Any thoughts? Am I missing something?
```

```
Torgeir
```

E-mail two:

```
Subject: RE XWork/WebWork architecture image

A small followup:

The image is correct, but for some reason, parts of the Velocity
templates are build within the interceptor stack, and the rest outside
the interceptor stack. Looks to me like most values are inserted into
the Velocity templates within the interceptor stack, but when there
are proxies between the objects, the values are fetched and inserted
outside the interceptor stack.

Torgeir
```