Combining Architectural Patterns and Software Technologies in one Design Language

Uwe van Heesch
University of Groningen,
uwe@vanheesch.net

Sara Mahdavi Hezavehi
University of Groningen
saramhezavehi@gmail.com

Paris Avgeriou
University of Groningen
Groningen, The Netherlands
paris@cs.rug.nl

February 12, 2012

Abstract

Software technologies like frameworks, APIs, or libraries are pieces of software that can be used to solve recurring problems in particular domains. Just like patterns, they can be documented in a way that explicitly describes the context, in which they can be used; the problems they solve, and the solutions they imply. Based on this assumption, software technologies and patterns can be treated equally as architectural solutions capturing reusable architectural knowledge. In this paper, we describe the idea of relating patterns and technologies from a specific domain in one design language that provides useful guidance for software engineers who need to become familiar with the domain.

1 Introduction

Pattern languages preserve reusable architectural knowledge. They describe a web of related patterns, which can be used, for instance, to create software for specific application domains [1]. The patterns in the language offer generic solutions for isolated design problems in that domain [2]. By explicitly regarding the patterns' relationships and dependencies, and by indicating how complementing or conflicting pattern combinations may resolve a given problem, a pattern language not only assists designers in making the right choices; as a side-effect, it also preserves more design knowledge than just the sum of its parts [1, 3].

Although software patterns and pattern languages can assist software architects during the design phase [4], the development of a large software system often includes the choice of software technologies[1]; which means that both, patters and technologies, are frequently subject of design decisions made to solve a specific design problem.

As opposed to software patterns, which mainly contain design or implementation advice, software technologies can be considered as off-the-shelf solutions for particular problems in specific domains. Thus, software patterns generally offer design reuse, while technologies offer (binary) code reuse. Software patterns and technologies are different in many respects, but they are both reusable architectural solutions which can be applied to a recurring problem in a particular context. In principle, this similarity allows to mix both concepts in one language for architectural

[1]In this paper, we use the terms *software technology* and *technology* synonymously for compiled pieces of software or interface specifications that can be integrated in other software systems. This definition covers among others libraries, frameworks, drivers, APIs, database management systems and application servers.

solutions, which covers a part of a domain's solution space much deeper than pattern languages alone could.

In this paper, we present a design language consisting of a number of architectural and enterprise patterns, and certain well-known software technologies from this domain. The purpose of this language is to support architects in making and documenting design decisions related to the included patterns and technologies.

The focus of our work is not on eliciting new patterns or promoting unknown technologies. Instead, we concentrate on eliciting combinations of patterns and technologies that we repeatedly perceived in different software projects. Consequently, the language presented here does not claim to be complete (i.e. cover the entire solution space of a domain); it is rather an application-generic preparation of repeatedly made architecture decisions for reuse in other projects.

To make sure that the technology pattern combinations in our language are proven concepts, instead of incidental occurrences, we impose the following 2-rule on the combinations in our design language: A combination of patterns and/or technologies can be part of the design language, if it has been observed in at least two different software projects from the domain covered by the language. As one of the authors was involved in multiple software projects from the Java enterprise application domain, we decided to use this domain as a starting point.

The rest of this paper is organized as follows. Section 2 presents background information about patterns, architecture decisions and technologies. The next section contains the actual technology pattern language. In Section 4, we introduce tools we developed to support our approach. Finally, Section 5 concludes and points out directions for further research.

2 Background

2.1 Architecture Decisions

According to the new standard for architecture description ISO/IEC/IEEE 42010, software architecture can be defined as the "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" [5]. In the last decade, the software architecture community experienced a paradigm-shift concerning the adequate documentation of software architecture. While some authors mentioned the need to document the rationale behind architectural design long before (e.g. [6, 7]), Jansen and Bosch emphasized the importance of documenting architecture as a set of architectural decisions [8]. The perspective of looking at software architecture in terms of a set of architecture decisions is widely recognized today. Each architectural design decision should be documented by describing the problem it solves, the alternative design options considered, potential consequences of the decision, and the affected system concerns (including end-user requirements) [5].

2.2 Pattern Languages

Pattern languages offer support for developing software for specific technical or application domains [9]. They list and combine patterns from the respective domain in a way that reveals a generative, domain-specific sequence of design decisions. Meszaros suggests that "a pattern language should guide the uninitiated in building a system" (as cited in [9]). While the term *uninitiated* can be subject to discussion[2], they certainly contain candidate solutions for recurring problems occurring in a particular domain. According to Buschmann et al., pattern languages should provide developers with concrete guidance on how to develop a system by extracting the key problems to resolve, showing different alternatives to solve the problems, giving advice

[2]In our understanding, pattern languages provide guidance for software engineers at all levels of experience. However, they do not replace general software engineering knowledge and skills

for tackling dependencies between solutions, and finally by imposing a software development process on the consumer of the pattern language [9].

2.3 Technology choices

As described in the previous subsections, architecture decisions are the fundamental design decisions made in every software project. In the context of our research in the field of software architectural knowledge, we repeatedly experienced that many architecture decisions actually concern the choice of a pattern, an architectural style, or a tactic. But a great part of the decisions, in the projects we observed, concerned the choice of technologies to use. As we will describe in Section 3, we elicited the technologies and patterns for the presented design language from three different projects in the Java enterprise application domain. The architecture decisions of those projects were thoroughly documented by the respective architects of the system using decision views from the architecture documentation framework presented in [10]. In total, the architects of the system documented 96 decisions. An analysis of the decisions showed that 80 decisions (83%) concerned the choice of off-the-shelf technologies and only 16 decisions (17%) were about patterns (in this case architectural and enterprise application architecture patterns). Note that these numbers are hardly generalizable, but they can still be seen as an indicator that a great part of the design decisions architects consciously make concerns the choice of technologies.

2.4 The synergy of patterns and technologies

At a first glance, software patterns and technologies have little in common. Patterns are long-lived, abstract knowledge containers describing solutions to recurring problems in a specific context. The abstract nature of patterns is one of their key advantages: Theoretically, a pattern can be applied in many comparable, but different situations without ever being applied the same way twice [11]. This even includes cases, which the author of the pattern did not have in mind when writing the pattern. As opposed to software technologies, which are concrete, compiled, often executable pieces of software. The purpose of patterns is to communicate a vision of a solution, whereas a technology is a solution itself. Another difference is the lifetime: Patterns, especially architectural patterns, can remain relevant for decades, while technologies are sometimes outdated after a few months. Nevertheless, patterns and technologies have something in common: Both are reusable architectural solutions, which means they can be the subject of a design decision made by an architect.

In our previous work [3], we described that patterns contain information about the problem space, design rationale, alternatives, consequences and trade-offs that are performed when applying them. This information has a strong overlap with the information that should be captured for the architecture decisions made in a system. We argued that the effort for documenting architecture decisions related to pattern choices could be drastically reduced by pointing to the pattern description when documenting decisions [3].

Unfortunately, the previously mentioned information, which is readily available for patterns, is usually not available for technologies. It is often not explicit which problems can be solved by using a technology, or which consequences arise for the rest of the architecture. This is unfortunate, because technologies like middleware systems, application programming interfaces (APIs), and frameworks are also solutions to a problem in a specific context. Consequently, they could be described using a pattern-like template. For instance, if the problem is that data needs to be stored in a central place in order to allow concurrent, transaction-safe access for multiple clients, then a database management system can be used to solve this problem.

We argue that, analogously to pattern decisions, the effort for documenting architecture decisions related to technology choices can be drastically reduced in the long term by describing

technologies in a pattern like (i.e. application-generic) way once, and referencing this generic description in multiple architecture decisions afterwards.

The main advantage however, is that using a pattern format for technology description allows to mix both concepts in one language for architectural solutions. This brings us closer to the vision of many pattern language authors stating that pattern languages describe a process for systematically resolving related or dependent software development problems (e.g. [12, 13, 14]). A process for designing new systems can only be *complete*, if it also takes off-the-shelf technologies into account. Based on the assumption that pattern choices are a significant part of all architecture decisions; and besides, that most of the other decisions concern the choice of technologies, we argue that the solution-space of a domain can be covered much deeper by combining both concepts, patterns and technologies, in one design language.

3 A design language for enterprise Java applications

In this section, we describe recurring patterns and technologies from the enterprise Java domain. As described in Section 1, we do not elicit unknown patterns or promote unknown technologies, but we focus on the proven combination of both in multiple software projects. The combinations of patterns and technologies presented in the language were collected from three different software projects from the enterprise Java domain:

Open Pattern Repository (OPR): This project is a Java based distributed system for managing and sharing patterns. The OPR provides web-services, JDBC database access, and a user friendly web application.

Open Decision Repository (ODR): The ODR is a publicly accessible, freely usable system that can be used to document architecture decisions made in a software project. It provides functionality to document architectural design decisions for software-intensive systems using different views on architecture decisions. The ODR provides RESTful web-services, a web application, and plugin support for the UML tool Visual Paradigm [15].

Raw Data Transfer System (RDTS): The RDTS is a measurement collector for network traffic analysis. It is part of an Internet early warning suite developed by a German organization operating mainly in the internet security domain. The part of the system we looked at is responsible for collecting data measured by probes that are located in different autonomous systems in the German Internet infrastructure. Details about the vendor, as well as the software itself, cannot be provided, as the organization asked us to treat this data confidentially.

The following process was used to elicit patterns, technologies, and their combinations from the three systems:

1. In the first step, the decision documentation of the three projects was browsed to identify decisions related to patterns and technologies. Note that all three projects had been thoroughly documented using the decision documentation framework presented in [10]. The decision documentation of the OPR and the ODR are both available online [16, 17]. As mentioned earlier, the raw data transfer system has to be treated confidentially. Nevertheless, the authors were provided with the complete decision documentation.

2. In the second step, the relationships between the decisions identified in the first step were analyzed. Especially decision relationship views from the previously mentioned documentation framework are useful to identify relationships between technologies and patterns. Figure 1 shows an example of a relationship view documented for the OPR. It shows that

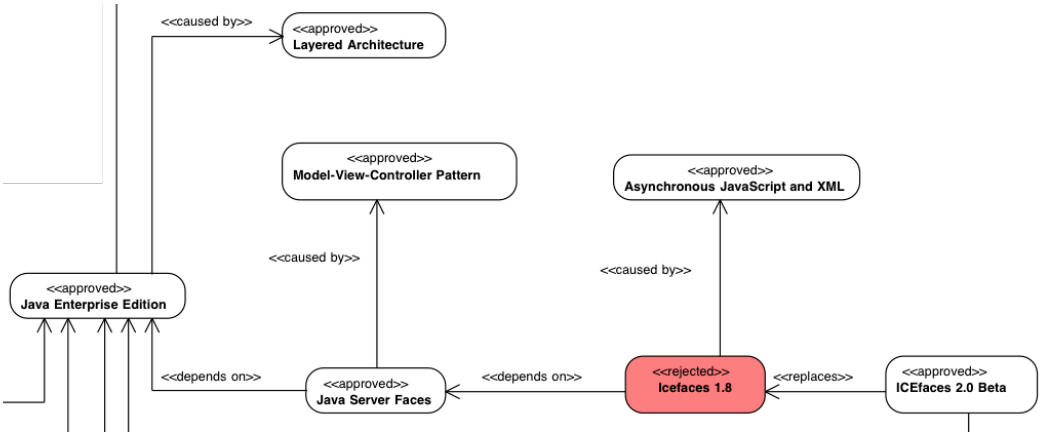

Figure 1: Excerpt from a decision relationship view documenting the Open Pattern Repository decisions

the Java Enterprise edition (JEE) was chosen to implement a layered architecture (Layers pattern). It also contains a relationship between JEE and Java Server Faces, which in turn was used as an implementation of the Model View Controller Pattern. Finally, IceFaces was used as component library to complement the Java Server Faces technology. The result of this step is a set of candidate patterns, technologies and their interrelationships.

3. After steps one and two were completed, we compared the candidate patterns and technologies from the three projects to each other to find correspondences. All similar decisions were taken over into the design language presented in this paper.

The result of this process is a generic design language for the domain of the analyzed projects. We assume, that the language grows with additional projects analyzed until a certain stable body of design knowledge is documented. In this particular case, we took over all patterns and technologies that had occurrences in at least two projects. This is a rather low limit, chosen here, because we only analyzed three projects to exemplify the approach. If more projects are taken into account, the limit can be increased to the magic 3-known usage rule from the pattern community.

3.1 Overview over the design language

Figure 2 provides an overview over the design language. The ovals in the figure represent patterns and technologies. Patterns have a green background and are tagged with *<<Pattern >>*. Technologies are tagged with *<<Technology >>*. The arrows between the ovals represent relationships. These are either verbally described, e.g. *provider for*, or *strategy of*, or the word *impl* shows that a technology can be used to implement, or implements the pattern the arrow points to.

The overall architectures of the systems we looked at are dominated by the *Layers*[3] [1] pattern. Like in many other web-based applications, the developers of the systems used the *Model View Controller* pattern [1] to separate the view code from controllers and data models. Finally, a *Shared Repository* [18] was used to realize a central datastore.

Different database management systems can be used to implement the *Shared Repository*. In the three analyzed projects, *MySQL DBMS*, *Derby DBMS* and *Oracle DBMS* were considered by the developers. The *Java Enterprise Edition* (JEE) was used as a middleware to implement the *Layers* pattern with *JBoss* and *Glassfish* as concrete application servers. In all projects,

[3]Elements of the presented design language are printed in italics

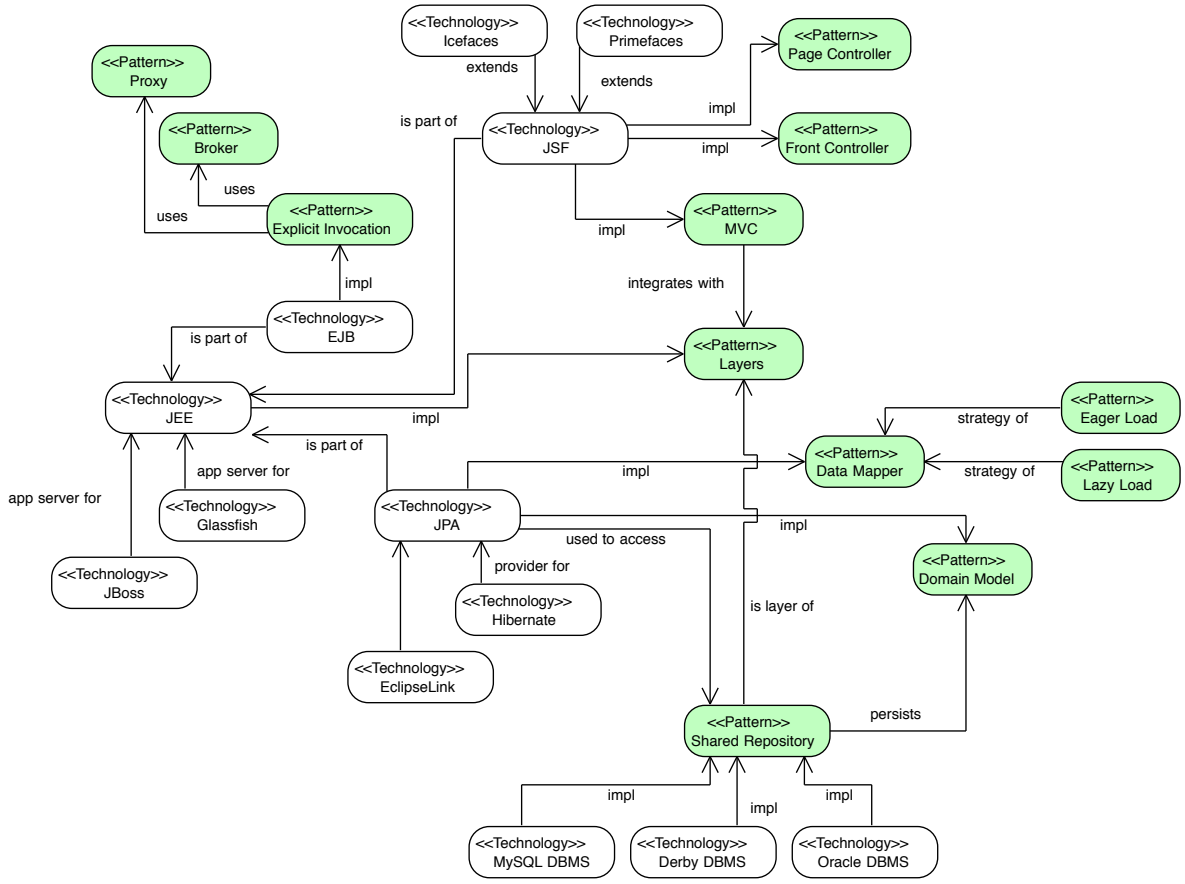

Figure 2: Overview over the technology-pattern language

the *Java Persistence API* (JPA) was used as implementation for the *Data Mapper* pattern [19], which describes a solution for object-relational mapping between the object-oriented *Domain Model* [19] used in the JEE application, and the underlying relational database. *Eclipselink* and *Hibernate* were considered as concrete persistence providers. *Enterprise Java Beans* (EJBs) encapsulate the business logic in the business layer of the layered architecture. *Java Server Faces* (JSF), which is the standard JEE web technology, was used to implement the *Model View Controller* (MVC) pattern. Likewise, *JSF* uses *Front Controller* and *Page Controller* (both [19]) to organize workflows within the web layer. *Icefaces* and *Primefaces* are JSF based frameworks providing additional view components.

The design language might be useful for software architects and software engineers designing applications using enterprise Java technologies. We present "thumbnails" for the patterns and technologies used in the language in the appendix.

4 Tool support

The approach exemplified in this paper benefits from holistic tool support. On the one hand, decision need to be analyzed in order to develop the design language (see process described in Section 3). On the other hand, the design language itself should be documented in a way that allows to easily find patterns and technologies for a particular domain, to browse relationships between patterns and technologies, and finally to reference elements of the language for the documentation of design decisions made based on the language. In this section, we describe two of our research projects started to support these use cases. The projects were already mentioned

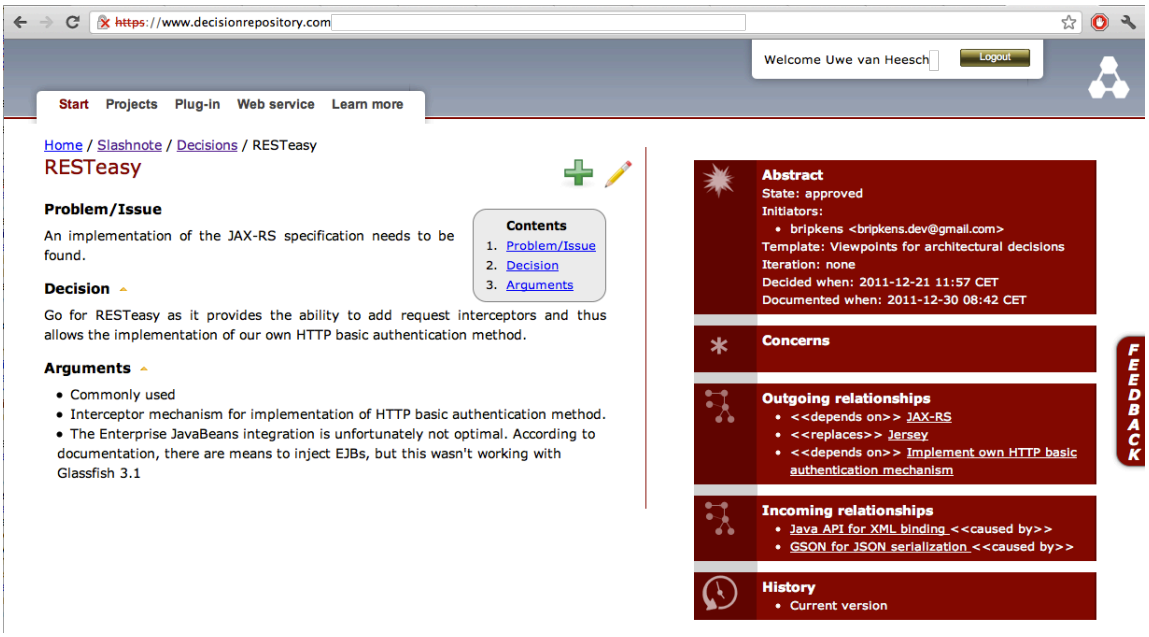

Figure 3: Use cases related to creating and using the design language and their mapping to the OPR and ODR

in Section 3, because they were used to elicit parts of the design language itself:

Open Pattern Repository: The Open Pattern Repository (OPR) is a publicly available and freely usable online repository for patterns with a focus on architectural- and design software patterns. An emphasis is put on capturing patterns including their known variants, their consequences and especially their various interrelationships. Additionally, the pattern repository captures technologies, frameworks, and programming languages in a pattern-like format consisting of context, problem and solution statements. The OPR provides functionality to add, change and remove patterns; to browse patterns according to categories; to find patterns that match certain criteria; and finally to create a documentation of a set of chosen patterns. A part of the functionality is also available via restful web-services. That way, other repositories and knowledge management platforms can easily integrate the OPR. The open decision repository makes use of these web-services to reference patterns and technologies in architecture decision documentation. The architecture, design and source-code of the OPR is available in its Google Code repository [16]. Companies or universities can download the OPR to deploy it in their own organizational context. A public version of the OPR can be found at `http://www.patternrepository.com`.

Open Decision Repository: The Open Decision Repository (ODR) is a publicly accessible, freely usable web application that can be used to document architecture decisions made in a software project. It provides functionality to document architectural design decisions for software-intensive systems using different views on architecture decisions. Among others, it visualizes decisions in a graphical manner suitable for quickly grasping the main decisions and the relationships among them. As many decisions concern the choice of patterns or technologies, the ODR uses the web-services of the OPR to link the application-generic information stored in the patterns or technologies to specific architecture decisions. That way, decisions can be documented with minimal required effort. The architecture, design and source-code of the ODR is available in its Google Code repository [17]. Likewise the OPR, companies can run the ODR in their own networks to document projects. A public

version of the OPR is available for evaluation under `http://www.decisionrepository.com`. Figure 3 shows a screenshot of a decision shown in the ODR.

Figure 4 lists use cases for creating design languages, as promoted in this paper, and their mapping to the Open Pattern Repository and Open Decision Repository respectively.

Use Case	Open Pattern Repository	Open Decision Repository
Browse decision documentation to find pattern and technology candidates for the design language		X
Search multiple software projects for decisions related to the pattern and technology candidates		X
Analyze relationships between patterns and technologies		X
Understand rationale behind pattern and technology choices		X
Document patterns and technologies	X	
Establish generic relationships between patterns and technologies	X	
Search for patterns and technologies in a design language	X	
Browse relationships between patterns and technologies in the design language	X	
Document the decisions made by using the design language	X	X

Figure 4: Use cases related to creating and using the design language and their mapping to the OPR and ODR

5 Conclusions and Future Work

In this paper, we introduced the idea of mixing architectural patterns and technology descriptions in one design language. This design language covers parts of a domain's solution space much deeper than pattern languages typically do, because they involve off-the-shelf solutions as well. Likewise pattern languages, this type of design language can assist designers in making design decisions in a particular domain.

The proposed approach is exemplified by a language for enterprise Java applications. The patterns and technologies forming the language, as well as their relationships, where mined from three software projects from the enterprise Java domain.

Design languages comprising patterns and technologies enable designers who are inexperienced in a specific domain to explore the space of possible solutions. Furthermore, they allow to efficiently document design decisions related to the choice of patterns or technologies.

We presented tools that can be used to elicit and document design languages and to make use of them in architecture documentations of software projects.

In the future, we plan to extend the technology pattern design language presented in this paper and to elicit more languages from other application domains.

We would like to invite the pattern community to contribute to the building of such languages. We believe that domain specific software pattern languages are a good basis, which has to be complemented with the respective technologies available in the domain.

Acknowledgements

The authors would like to thank Michel de Jong, Martin Verspai, Christian Manteuffel, Ben Ripkens and Stefan Arians for their contributions to the Open Pattern Repository and the

Open Decision Repository.

Also, we would like to thank Ernst Oberortner for providing valuable feedback during the shepherding for EuroPLoP 2011.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc. New York, NY, USA, 1996.

[2] James Coplien. *Software Patterns*. SIGS, June 1996.

[3] U. van Heesch and P. Avgeriou. A pattern driven approach against architectural knowledge vaporization. In *Proceedings of the 14th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee*, 2009.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable object-oriented software. *Addison-Wesley Professional Computing Series*, page 395, 1995.

[5] ISO. Systems and software engineering — architecture description. *ISO/IEC CD1 42010*, pages 1–51, Jan 2010.

[6] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[7] P. Kruchten. The 4+ 1 View Model of Architecture. *IEEE Software*, 12(6):50, 1995.

[8] A. Jansen and J. Bosch. Software Architecture as a Set of Architectural Design Decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 109–120. IEEE Computer Society, 2005.

[9] F. Buschmann, K. Henney, and D.C. Schmidt. *Pattern-oriented software architecture: On patterns and pattern languages*. John Wiley & Sons Inc, 2007.

[10] U. van Heesch, P. Avgeriou, and R. Hilliard. A documentation framework for architecture decisions. *Journal of Systems and Software*, 2011.

[11] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A pattern language*. Oxford Univ. Pr., 1977.

[12] J.O. Coplien. A generative development-process pattern language. In *The patterns handbooks*, pages 243–300. Cambridge University Press, 1998.

[13] R.S. Hanmer and K.F. Kocan. Documenting architectures with patterns. *Bell Labs Technical Journal*, 9(1):143–163, 2004.

[14] D.C. Schmidt, M. Stahl, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture, Volume 2: Patterns for concurrent and networked objects*. John Wiley & Sons Inc, 2000.

[15] Visual Paradigm Int. Visual Paradigm for UML. `http://www.visual-paradigm.com/product/vpuml`, February 2012.

[16] University of Groningen, Software Engineering and Architecture Group. The Open Pattern Repository. `http://code.google.com/p/openpatternrepository/`, February 2011.

[17] University of Groningen, Software Engineering and Architecture Group. The Open Decision Repository. http://code.google.com/p/opendecisionrepository/, February 2011.

[18] P. Avgeriou and U. Zdun. Architectural patterns revisited–a pattern language. In *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPlop), Irsee*, 2005.

[19] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[20] Oracle. Java EE at a Glance. http://www.oracle.com/technetwork/java/javaee, February 2011.

[21] Java.Net. GlassFish - Open Source Application Server. http://glassfish.java.net/, February 2011.

[22] JBoss.org. Community driven open source middleware. http://www.jboss.org/, February 2011.

[23] blogs.sun.com. Glassfish v2, the Fastest Application Server on the Planet. http://blogs.sun.com/nazrul/entry/glassfish_v2_the_fastest_application, February 2011.

[24] Oracle. Java Persistence API. http://jcp.org/en/jsr/detail?id=317, February 2011.

[25] Eclipse.org. Hibernate. http://www.hibernate.org/, February 2011.

[26] JBoss.org. EclipseLink. http://www.eclipse.org/eclipselink/, February 2011.

[27] Santiago Rodriguez. JPA implementations comparison: Hibernate, Toplink Essentials, OpenJpa, Eclipselink. http://terrazadearavaca.blogspot.com/2008/12/jpa-implementations-comparison.html, April 2011.

[28] P. Lalanda. Shared repository pattern. In *Proceedings of the Conference on Pattern Languages of Programs (PLOP)*, 1998.

[29] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[30] msdn.microsoft.com. Front Controller pattern. http://msdn.microsoft.com/en-us/library/ms978723.aspx, February 2011.

[31] builderau.com. Two patterns that promote code reuse in ASP NET. http://www.builderau.com.au/program/html/soa/Two-patterns-that-promote-code-reuse-in-ASP-NET/0,339028420,339130833,00.htm, February 2011.

[32] java.sun.com. Java sun - Patterns Front Controller. http://java.sun.com/blueprints/corej2eepatterns/Patterns/FrontController.html, February 2011.

[33] enode.com. Mark up tutorial. http://www.enode.com/x/markup/tutorial/mvc.html, February 2011.

[34] Anthony Lauder. Event ports. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 80–86, London, UK, 1999. Springer-Verlag.

Appendix - Pattern and technology thumbnails

We present thumbnails for all patterns and technologies in the language. A *thumbnail*, in this context, is a short description that includes the context, problem, main forces, and the solution proposed by the pattern or technology. The patterns and technologies are not described in detail. The thumbnails are useful for readers who are already familiar with the patterns and technologies; our descriptions may help to understand when and how to use the patterns and technologies. Note that we use the same format to describe both patterns and technologies. We ask the reader to refer to the references for more detailed descriptions.

Technology: **JEE**
Context: An enterprise application is developed, which requires a web-based user interface and exposes parts of its functionality using web services. **Problem**: The application should be build on modular, reusable components in a distributed environment. It has to operate in a heterogeneous application environment, in which a variety of standard and proprietary communication protocols is used. The requirements indicate transaction aware services that scale well in the number of concurrent users. The application developers are experienced Java programmers. **Solution**: The Java Platform Enterprise Edition (JEE) [20] is a widely used platform for server programming in the Java programming language. JEE comprises a set of specifications to create fault-tolerant, distributed, layered applications based on modular components, which are managed by an application server. This allows programmers to focus on implementing business logic, instead of caring for subsidiary services. **Related Patterns and Technologies**: JEE can be used to implement the LAYERS [1] pattern. Different application server vendors implement the JEE specification set including, among others, GLASSFISH [21] and JBOSS [22].

Technology: **EJB**
Context: A JEE based application requires reusable, distributed components to encapsulate the business logic. **Problem**: A JEE based multi-tier application uses different clients (e.g. different web frameworks and web service clients) to access the business logic. Code duplication must be avoided. Components must be decoupled from class names and invoked transparently for clients. **Solution**: Enterprise Java Beans (EJB) are the JEE technology of choice for implementing reusable, distributed, transaction-aware services. EJBs are plain old Java objects that publish parts of its functionality using different kinds of business interfaces. They provide out of the box support for web services and transactions using simple annotations. **Related Patterns and Technologies**: The EJB specification is part of the JEE. EJB containers are often implemented using the EXPLICIT INVOCATION pattern.

Technology: **JBoss**
Context: An application server needs to be chosen that implements the JEE specification. **Problem**: The JEE application server should be a certified platform for developing and deploying enterprise Java applications and portlets. It should be an open source solution that is well documented and widely used on the market. Additionally to the JEE features, support for clustering and caching is needed. **Solution**: The JBoss Application Server [22] is the #1 most widely used Java application server on the market. It is an open source project, driven by professional open source developers who have contributed to the JBoss Application Server over years. JBoss is a fully certified Java EE platform guaranteed to fulfill the complete specification set. It is free of use. JBoss integrates Apache Tomcat as its Web container and includes capabilities for data caching, clustering, messaging, transactions, and an integrated web services stack. **Related Patterns and Technologies**: The JBOSS application server implements the JEE specification set.

Technology: **Glassfish**
Context: An application server needs to be chosen that implements the JEE specification. **Problem**: The used application server should be an open source product to allow developers to analyze and change the code if necessary. Support for clustering is required and the application demands high performance. Commercial support for the server platform should be available 24x7 using phone or email. Tool support is necessary to monitor applications at runtime. The underlying operating system provides a desktop system, which allows using graphical user interfaces for installation and configuration. The hardware platform provides at least 4GB of physical memory. **Solution** The Glassfish application server [21] is JEE 5 certified, aiming at a JEE 6 certification. It provides high performance when executed on SUN Sparc servers. It generally has the reputation to be faster than other professional JEE application servers [23]. The Glassfish server provides a task-oriented administration console and configuration wizards that simplify routine administrative tasks. Additionally, it provides out of the box support for clustering. **Related Patterns and Technologies**: The GLASSFISH application server implements the JEE specification set.

Technology: **JPA**

Context: Business objects in a Java based application have to be persisted using a relational database management system.
Problem: The DOMAIN MODEL pattern was applied in an application. A SHARED REPOSITORY is used as a central data store for the entities, using the DATA MAPPER pattern. The business objects were implemented using plain old java objects (POJO) and a transaction aware service is required to synchronize them with the relational data store. In addition, a powerful query language is needed to efficiently search for persistent objects. Finally, the application must remain portable and support a wide range of different persistence providers.
Solution: To satisfy these requirements, the Java Persistence API (JPA) [24] can be used. Being a part of the Java Enterprise Edition (JEE), the JPA is a specification for persistence providers that allows to persist the state of lightweight Java objects to tables in a relational database. Instances of such an entity correspond to individual rows in the table. Relationships between entities are expressed through object/relational metadata. Object/relational metadata can be specified directly in the entity class file by using annotations, or in a separate XML descriptor file distributed with the application.
Related Patterns and Technologies: Well-known persistence providers implementing the JPA specification are HIBERNATE and ECLIPSELINK.

Technology: **EclipseLink**

Context: A Java based application uses the Java Persistence API as object relational mapper.
Problem: The state of persistent objects should be stored in a relational database using the JPA. A powerful query language is needed to search for objects. On the one hand, the domain objects must not be polluted with metadata about the underlying database. On the other hand, no changes to the database schema must be necessary. The database schema and the object model evolve independently. Finally, tool support is needed that integrates nicely in standard Java development IDEs like Netbeans or Eclipse.
Solution: EclipseLink [25] offers the flexibility of expressing persistence metadata using standard JPA annotations, deployment XML, or both. Optionally, EclipseLink JPA annotations and persistence.xml property extensions can be used to keep the database independent from the object model.
Related Patterns and Technologies: EclipseLink can be used as persistence provider underneath the JPA, or as a standalone persistence service.

Technology: **Hibernate**
Context: A Java based application uses the Java Persistence API as object relational mapper. **Problem**: The state of persistent objects should be stored in a relational database using the JPA. Tool support is needed that integrates nicely in standard Java development IDEs like Netbeans or Eclipse. CPU and memory consumption are not critical. The application requires to store huge amounts of new objects quickly. **Solution**: HIBERNATE [26] is an object relational mapping tool for JAVA and .NET applications. It is fully JEE compatible, but it can also be used as standalone persistence service. Unlike other persistence providers, HIBERNATE allows to make full use of native SQL calls to retrieve persistent objects. A performance test concluded that Hibernate is faster than ECLIPSELINK, when it comes to persisting objects [27]. **Related Patterns and Technologies**: Hibernate can be used as persistence provider underneath the JPA, or as a standalone persistence service.

Technology: **MySQL DBMS**
Context: A database management system (DBMS) is needed to realize a central data store. **Problem**: The DBMS should be open source and provide high performance. The business model requires commercial support for end customers. The DBMS must be used on different hard and software platforms, be easy to backup, and support online transaction processing (OLTP). The database vendors must provide a proprietary Java database connectivity (JDBC) driver in order to connect the database to JEE based application servers. **Solution** The MySQL Standard Edition allows developing high performance, scalable database applications that require online transaction processing. It is highly reliable and supports replication using easy to use built-in mechanisms. **Related Patterns and Technologies**: MYSQL can be used to implement the SHARED REPOSITORY pattern.

Pattern: **Layers**
Context: A large system needs to be decomposed into independent organizational units responsible for a coherent set of tasks. The system comprises low level components on a low level of abstraction, as well as high level components. **Problem**: The system needs to be flexible in a way that components are exchangeable without affecting great parts of the whole system. Some components comprise functionality provided by lower level components. **Solution**: Structure the system using layers at different levels of abstraction. Each layer comprises components at a particular level of abstraction. Components in layer x make use of services provided by components in layer (x-1), and provide services for components in layer (x+1) [1]. The LAYERS pattern helps to keep dependencies localized and improves exchangeability and reusability [1]. **Related Patters and Technologies**: The SHARED REPOSITORY pattern can be applied in one of the bottom layers of a layered architecture. The MVC pattern also integrates nicely into layers, as it may be used for communication between layers; in other words, Model, View and any number of controllers can be located in any number of Layers.

Pattern: **Shared Repository**

Context: Different components in a system need to access the same information.
Problem: The components in a system need to communicate and exchange large and evolving data; sometimes they are not aware of each other's existence. The system has to provide a certain level of performance. Using sequential architectures to share data between components may be inefficient, especially for large data sets [18]. Since relationships between components may change or evolve, the communication mechanisms of the system's components should be flexible. Also, the data exchanging should be reliable, meaning that data should be thoroughly and correctly transmitted between source and destination components [28].
Solution: Organize the system in a way that all data is stored in a SHARED REPOSITORY. All the components of the system can access the central repository, i.e. store data in and retrieve data from the repository. By using a central data repository, components do not need to be aware of each other's existence and can communicate indirectly. The shared repository should be scalable [18] and data kept in the repository should represent the system current state [28].
Related Patterns and Technologies: Oracle DBMS, Derby DBMS, or MySQL DBMS are examples for database management systems, which can be used to implement a shared repository.

Pattern: **Data Mapper**

Context: The system makes use of object-oriented data containers, while the information is stored in a relational database management system.
Problem: Object models encapsulate business logic and provide an efficient mechanisms to categorize data and its behavior. If information from the object model has to be persisted using relational database management systems, then data has to be transferred from object oriented structures to relational structures.
Solution: Provide a layer of software, which transfers data between in-memory objects and relational tables. The layer abstracts from the database in a way that in-memory objects do not need to know the details of the underlying database schema [19]. As a consequence, database schema and object model can evolve independently.
Related Patterns and Technologies: LAZY LOAD or EAGER LOAD can be used as a strategy to load objects from the database [19].

Pattern: **Lazy Load**

Context: Data from the database needs to be loaded into memory.
Problem: Data should be simply loaded from the database, and at the same time loading all the required objects from the database needs to be avoided. The decision might be to load a set of related objects, whenever you load an object of interest. But sometimes this strategy will result in loading of a huge object graph containing a number of useless objects which will eventually decrease performance [19].
Solutions: A Lazy Load strategy interrupts objects' loading processes, leaving a marker in the object structure in such a way that the object can only be loaded when it is required. That way, loading unnecessary objects is avoided. As a consequence, efficiency is increased [19].
Related Patterns and Technologies: Lazy Load can be used as a strategy for loading objects in the DATA MAPPER pattern.

Pattern: **Eager Load**
Context: Data should be loaded from database into the memory. **Problem**: Several objects need to be initialized; when requesting an object you can estimate you will probably need its children at the same time. Performance is not your main concern. **Solution**: The Eager Load strategy is the opposite pattern of Lazy Load. In this method, objects are initialized beforehand; i.e. you do not wait to fetch them until you actually need them. Since Eager Load can affect performance adversely, this strategy is valuable particularly when performance is not an issue, and you do not mind waiting a few seconds to fetch objects from the database. **Related Patterns and Technologies**: Eager Load can be used as a strategy for loading objects in the DATA MAPPER pattern.

Pattern: **Domain Model**
Context: The business logic has complex behavior. An object model of the domain including both behavior and data is required. **Problem**: The business logic is often subject to changes, there should be a solution to handle changing business rules and decrease the total cost of changes [19]. **Solution**: Domain Model creates a conceptual view of objects and their relationships. Each of the objects represents some meaningful individual. Using a Domain Model in an application includes inserting a layer of objects which models the business area, and because the behavior of the business often changes, it is important to be able to modify and test this layer easily. By minimizing the dependency of this layer to other parts of the system, modifiability of the Domain Model increases [19]. **Related Patterns and Technologies**: In Java, different persistence services assist the usage of the domain model pattern, the Java Persistence API (JPA) is a common specification for these kinds of persistence services.

Pattern: **Proxy**
Context: A client should be able to indirectly access services offered by another component. **Problem**: The client should not change its calling behavior which it uses to call local components. Accessing the component should be simple and transparent. The communication between client and the component should be safe, run time efficient, and cost effective. **Solution**: A proxy object acts as a representative of the original component by offering the interface of the original component. So the client should communicate with the placeholder, i.e. the proxy, instead of the original component. As a consequence, clients are decoupled from the location of servers; changes in the network infrastructure and server relocation does not affect the client [9].

Pattern: **Broker**
Context: There is a distributed system including independent cooperating components. **Problem**: Components should be able to access each others' services. All the limitations and dependencies emanating from communication between components should be eliminated. It should be possible to replace, add or eliminate a component at run time. Particular system details required to be hidden form components users. **Solution**: A broker component knows the location of servers and forwards the requests' of client to server, and sends back the response to the client. The broker provides location transparency and exchangeability of system's components [9].

Pattern: **Front Controller**
Context: A complex web application requires dynamic and configurable navigation based on a set of rules. **Problem**: When handling a request, several similar issues should be taken into account; including security issues, internationalization, and also providing specific views for certain users [29]. To avoid duplicated behavior you should not spread out the input controller behavior across multiple objects. It should be possible to change behavior at runtime [29]. **Solution**: Front Controller channels all the requests through a single controller (handler object). The new object can be modified at runtime [30]. Moreover, since there is just one centralized controller, it is possible to enforce all the application policies like security on the controller [30]. **Related Patterns and Technologies**: The Java Server Faces (JSF) framework internally used the Front Controller pattern.

Pattern: **Page Controller**
Context: You are developing a complex application which constructs web pages dynamically, but the navigation between the pages is mainly static. **Problem**: Code duplication should be minimized to increase reusability and flexibility. The amount of code in scripted server pages needs to be minimized to develop reusable logic. Also, it should be possible to implement common appearance and navigation across your web application and reuse presentation logic all over the application [30]. **Solution**: Provide a central class, which implements all common behaviors needed to deal with the HTTP request, update the model and then send the request to the suitable view [31]. This centralized decision point reduces the amount of code, and promotes code reuse by reducing the business logic [32]. **Related Patterns and Technologies**: The Java Server Faces framework provides means to implement a Page Controller.

Pattern: **Model-View-Controller**
Context: You have an interactive application. **Problem**: Multiple views of the same data need to be kept and displayed to the users. Users should be able to modify the data and the updates should be stored in the data store. You need to be able to easily change the views and these changes should not affect the core of the application. **Solution**: Clearly separate controller and model from the view: Model for keeping data and functionalities, Views for displaying data, and Controller for handling inputs that affect data or views [9], [33]. By using MVC, you can have different views of the same data, and since they are not dependent on the model, you may change the views not worrying about affecting the core of the system. Also, when the model is modified, all the dependent views and controllers are notified by the change-propagation mechanism [9]. MVC introduces a new level of indirection and therefore adds complexity to the system. **Related Patterns and Technologies**: MVC integrates with Layers pattern.

Pattern: **Explicit Invocation**
Context: There is a system consisting of collaborating components. **Problem**: They need to somehow communicate with each other, and certain components (clients) need to use services defined in other components (servers). Client is aware of the server, but the client could be anonymous to the server, you intend to have coupling among client and server. **Solution**: Explicit Invocation allows client to invoke services on server directly and forwards the results back to the client. In this mechanism, the client is tightly coupled to the server [18], [34]. **Related Patterns and Technologies**: Explicit Invocation is used for message passing purposes in Broker pattern.