Metadata Modularization Using Domain Annotations

José Roberto C. Perillo
Aeronautical Institute of Technology
jrcperillo@yahoo.com.br

Eduardo M. Guerra
Aeronautical Institute of Technology
guerraem@gmail.com

Jefferson O. Silva
Aeronautical Institute of Technology
jefferson.o.silva@uol.com.br

Fábio F. Silveira
Federal University of São Paulo
fsilveira@unifesp.br

Clovis T. Fernandes
Aeronautical Institute of Technology
clovistf@uol.com.br

Abstract

Many recent frameworks use the metadata configuration by applying code annotations in the application classes to customize the behavior at runtime. These annotations usually add elements related to the infrastructure in classes that are related to the domain. This practice makes the domain classes coupled to the framework metadata and mix infrastructure and domain information in the same class. This paper presents the use of domain annotations to modularize the metadata definition, develops a case study and uses assessment techniques to evaluate it as a design rule.

General Terms: Measurement, Design.

Keywords *Metadata; Code Annotations; Domain-Driven Design; Frameworks;*

1. Introduction

Attribute-oriented programming is a program level marking technique that allows developers to mark programming elements, such as classes and methods, to indicate application-specific or domain-specific semantics [1]. In Java platform, this programming style has become popular with the native support to code annotations [2].

Metadata-based frameworks [3] are frameworks that that process their logic based on the metadata of the classes whose instances they are working with. In these frameworks the metadata can be defined using attribute-oriented programming, but also using external sources like XML files or even by code conventions [4].

Many mature frameworks and APIs used in the industry nowadays use code annotations, such as Hibernate [6], EJB 3 [7] and Spring Framework [8]. In these, the framework annotations are placed in the application domain classes and consumed by the framework at runtime. Aspect-oriented frameworks also benefits from the use of metadata, especially in cases which contains many variabilities in the same crosscutting concern [5].

Those annotations add the framework semantic, usually related to infrastructure, to the domain classes. According to the domain-driven design [9], this is a practice to be avoided. Using framework specific annotations the application also gets coupled with it. if is necessary to change to another framework or even to another version of the same framework usually is necessary to use some annotations refactoring [10].

This paper proposes the use of domain annotations [11] supported by the Daileon framework [12] as an alternative to modularize the metadata definition, resulting in domain classes independent of frameworks specific annotations. It also presents a case study where this design rule is applied and the modularity achieved evaluated.

This paper is outlined as follows. Section 2 the main concepts about the use of domain annotations. Section 3 presents Daileon [12], a tool used to enable the support the use of domain annotations in frameworks that do not support it. Section 4 presents a detailed case study, suggesting development steps for the use of domain annotations. Section 5 uses DSMs to evaluate the modularity achieved with the technique application. Section 6 concludes the paper presenting the main contributions and the limitations of this work.

2. Domain Annotations

Nowadays, it is widely accepted by the software development community that, when solving a particular problem by the construction of new software, the real complexity, in most cases, is predicated on understanding the business domain [12]. The approach known as domain-driven design [9] provides a set of practices that helps creating, maintaining and evolving software for most part of domains. One of these practices is the technique called domain modeling, which helps dealing with the complexity of a particular domain. Its goal is to create an abstraction of it, contemplating the aspects that are relevant to the development of the new software. Domain model is not a diagram, and thus it can be represented by practically anything. In terms of application code, the domain layer is the manifestation of the domain model. It should be isolated from other layers and should concentrate all business logic being implemented by the software.

There are inevitable situations in which the purity of the object model has to be compromised. For instance, frameworks often require annotations to be added to code related to the domain. The concept of domain annotations [11][12] proposes representing domain-specific metadata via annotations on domain objects. These annotations can be mapped to other annotations, and thus one domain annotation can represent one or several annotations.

There are several benefits from the domain annotations concept usage: i) the code becomes cleaner with fewer annotations; ii) it helps keeping the domain layer isolated from other layers; iii) annotations on domain elements clearly have a relationship with the domain; and iv) the dependency on external annotations that has to exist is encapsulated. However, the greatest values this concept provides are modularity and reusability. Mapping a domain annotation to external annotations allows changing external frameworks without effectively changing code related to the domain. In order to achieve it, the domain annotations must be translated to their corresponding annotations, which can be done by using the Daileon framework [12], presented in the next section.

3. Daileon Framework

The Daileon framework allows Java-based applications to be developed using the concept of domain annotations. Essentially, it provides two main functionalities: i) it allows frameworks to be created with the capability of interpreting domain annotations; and ii) it also allows domain annotations to represent annotations of frameworks that do not support such a concept. Therefore, Daileon allows frameworks to support the concept of domain annotations and it also allows domain annotations to be created even when using frameworks that are not able to recognize annotations other than their own.

3.1 Domain Annotations for New Frameworks

A framework that is capable of interpreting a domain annotation is a framework that is able to identify which known annotations a domain annotation corresponds to. That means that, even if an element is annotated with an annotation that is not known by the framework, it is still able to recognize it and verify which known annotations it represents, as long as this annotation is annotated with annotations known by the framework. Figure 1 depicts the technique to be applied with Daileon for such cases.

```
/* The definition of the domain annotation */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@DomainAnnotation
@FrameworkAnnotation1
@FrameworkAnnotation2
public @interface Adminisrative {
    // This annotation does not actually have
    // attributes. The code that consumes this
    // annotation only uses the annotations
    // defined in it.
}

// A method of a domain class,
// annotated with the domain annotation
@Administrative
public void placeCurrentOffer() {
    // A business method of a domain class.
}
```

Figure 1. The definition of the domain annotation.

Figure 1 shows how a domain annotation can be defined when frameworks use Daileon to support their creation. An annotation is annotated with two annotations of a framework that is external to the domain. Since this annotation is not known by the framework, some mechanism to recognize it is required. In this case, frameworks can use the first functionality provided by Daileon. This functionality provides several ways to recognize annotations that are not known by frameworks and identify which known annotations they correspond to. The `DomainAnnotationsHelper` class provides several static methods that frameworks can use to identify domain annotations and recognize which known annotations they represent.

3.2 Domain Annotations for Existing Frameworks

Currently, most part of frameworks still does not support the concept of domain annotations, which means that they are not able to interpret annotations that are not known by them. Consequently, replacing their annotations by domain annotations does not have any effect in their environment, even if the domain annotations represent annotations already known by them.

For these cases, the second main functionality provided by Daileon can be used. Essentially, this functionality allows translating domain annotations to their corresponding annotations in the bytecode level. Hence, each domain annotation must indicate a class or class element in which their corresponding annotations can be found. The case study in the next section presents an example of this mapping.

3.3 Daileon's Internal Behavior

When defining domain annotations to be used with existing frameworks, Daileon acts in the application classes after the compilation process. It changes the classes' bytecodes to incorporate the annotations of the original frameworks. The order of Daileon actions are shown in Figure 2. The configuration file indicates the classes that are annotated with domain annotations. In this case, each domain annotation indicates where the annotations of the original frameworks are located. The domain annotations are then evaluated properly (that is, translated to the corresponding annotations of the original frameworks) and are finally placed in the classes' bytecodes, adding the corresponding annotations. Classes are then saved in a different directory, so that the domain annotations are not replaced in the original class files, allowing them to be replaced by other annotations when necessary.

Daileon uses the ASM bytecode engineering tool [13] to manipulate the classes' bytecodes. Fundamentally, each domain annotation is evaluated to annotations of existing frameworks that do not support the creation of domain annotations at all. After the manipulation, the external annotations are added to their corresponding elements in the code. For frameworks that use Daileon to support the creation of domain annotations, it provides a class called `DomainAnnotationsHelper`, in order to recognize a domain annotation and inform which known annotations it corresponds to.

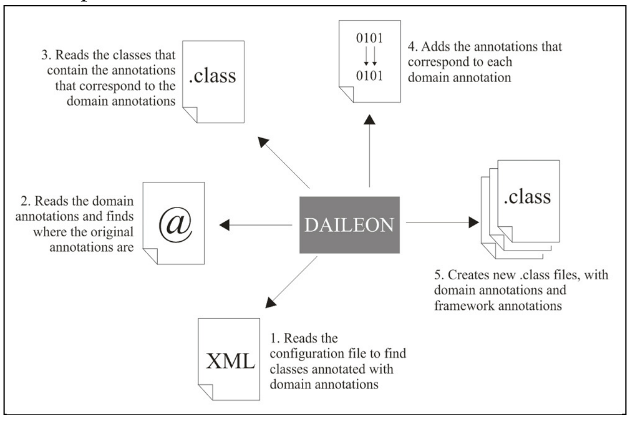

3. Reads the classes that contain the annotations that correspond to the domain annotations
.class
0101
0101
4. Adds the annotations that correspond to each domain annotation
2. Reads the domain annotations and finds where the original annotations are
@
DAILEON
.class
5. Creates new .class files, with domain annotations and framework annotations
XML
1. Reads the configuration file to find classes annotated with domain annotations

Figure 2. Daileon actions, when translating domain annotations in the bytecode.

4. Case Study

The objective of this section is to present a case study to illustrate the use of domain annotations and to be use as a reference for the evaluation of modularity. The case study is a service to manage and achieve information about papers. It has concerns about transaction management, access control and logging. EJB 3 standard annotations [7] are used for transaction and security and the Metadata-based Logger [5], an aspect-oriented framework, is used for logging.

The following subsections describe the case study in detail. Each subsection can be considered a step in the modeling of the domain annotations.

4.1 Service Definition

The first step for the case study is the definition of the service API. The Figure 3 represents the source code of a stateless session bean [7], with the method implementations omitted, that represents the service used as case study. The class Paper represents the paper information provided and method names are very descriptive about their responsibilities.

```
@Stateless
class PaperService implements PaperServiceRemote{

  public void addNewPaper(Paper p){ ...}
  public void updateExistentPaper(Paper p){ ...}
  public void deletePaper(Paper p){ ...}
  public Paper getPaperByName(String name){ ... }
  public List<Paper> listPapersByAuthor(
                  String authorName){ ... }
  public List<Paper> listPapersByKeyword(
                  String keyword){ ... }
  public List<Paper> listPapersByConference(
                  String conferenceName){ ... }

}
```

Figure 3. PaperService API

4.2 Domain Annotation Definition

Analyzing the service API the methods can divided in three different kinds: methods for data management, methods that retrieve free information and methods that retrieve paid information. To communicate this difference between the methods, tree domain annotations are created to annotate the methods: @DataManagement, @FreeQuery and @PaidQuery. The result is represented in Figure 4.

```
@Stateless
class PaperService implements PaperServiceRemote{

   @DataManagement
   public void addNewPaper(Paper p){ ...}
   @DataManagement
   public void updateExistentPaper(Paper p){ ...}
   @DataManagement
   public void deletePaper(Paper p){ ...}
   @FreeQuery
   public Paper getPaperByName(String name){... }
   @PaidQuery
   public List<Paper> listPapersByAuthor(
               String authorName){ ... }
   @PaidQuery
   public List<Paper> listPapersByKeyword(
               String keyword){ ... }
   @PaidQuery
   public List<Paper> listPapersByConference(
               String conferenceName){ ... }
}
```

Figure 4. PaperService with Domain Annotations

It is important to notice that all the annotations are related to the domain and do not directly reference any infrastructure service.

4.3 Non-Functional Requirements

After the domain annotations definition, it is important to know what each one means for the other concerns, usually related to infrastructure services and crosscutting concerns. Follow the meaning of each annotation for transaction management, logging and security:

- @DataManagement: It must execute inside a transaction because data is modified; the information must be logged in a file to be audited; and those functionalities can only be accessed by the administrators.
- @FreeQuery: As the data is only accessed, it must not be executed in a transaction context; there is no need for logging; and any user, even without authentication, can access it.
- @PaidQuery: As the data is only accessed, it must not be executed in a transaction context; the logging must be made in the database because it is used for charging; and can be accessed by regular users as well by administrators.

4.4 Annotation Mapping

The last step is to map the domain annotations to the framework annotations. The Metadata-based Logger support natively the definition of indirect annotations and is only necessary to put the @LogMarker annotating each domain annotation.

The EJB 3 annotations do not support the indirect definition and Daileon [12] is used for this mapping. The Daileon annotations indicate that the annotations defined in a method must be copied to the methods that have that domain annotation. The domain annotations definition is presented in Figure 5 and the class with the annotation template methods is presented in Figure 6.

```
@LogMarker(logInfo={METHOD,PARAMETER},
           logLocation={FILE})
@DomainAnnotation
@MethodTemplate(annotatedClass="AnnotationsHome",
                method="dataManagement")
public @interface DataManagement {}

@DomainAnnotation
@MethodTemplate(annotatedClass="AnnotationsHome",
                method="freeQuery")
public @interface FreeQuery {}

@LogMarker(logInfo={METHOD,PARAMETER,RETURN},
           logLocation={DATABASE})
@DomainAnnotation
@MethodTemplate(annotatedClass="AnnotationsHome",
                method="paidQuery")
public @interface PaidQuery {}
```

Figure 5. PaperService with Domain Annotations

```
public class AnnotationsHome{

   @TransactionAttribute(REQUIRED)
   @RolesAllowed({"admin"})
   public void dataManagment(){}

   @TransactionAttribute(NOT_SUPORTED)
   @PermitAll
   public void freeQuery(){}

   @TransactionAttribute(NOT_SUPORTED)
   @RolesAllowed({"admin","user"})
   public void paidQuery(){}

}
```

Figure 6. PaperService with Domain Annotations

5. Evaluating Modularity

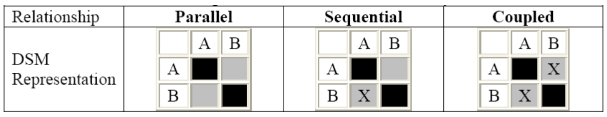
The objective of this section is to present a comparison between two approaches used by Metadata-based frameworks [5]. It is compared metadata-based frameworks that use traditional annotations like EJB 3 [7] and Metadata-based Logger [5] against the ones that use domain annotations like the framework presented in this paper called Daileon [12]. The following subsections introduce the concepts related to Design Structure Matrices (DSMs) and display two DSMs for the case study presented in last section. The first DSM shows the usage of traditional annotations and the second one shows the usage of domain annotations in order to evaluate modularity.

5.1 Design Structure Matrix and Design Rules

In simple terms a Design Structure Matrix can be defined as a square matrix that relates its constituent elements. More specifically, DSMs show the dependencies (relations) among design parameters. A design parameter corresponds to a design decision made about an aspect of a design. Design parameters can be represented in various abstraction levels. Commonly used software design parameters include classes, packages, methods, type signatures and annotations. Non-functional software requirements can also be represented in DSMs such as concurrency, transaction management and logging. DSMs are a good tool not only for visualizing dependencies relationships but also for the evaluation of software modularity.

Design parameters are disposed both in the rows and the columns of a matrix. One parameter depends on another one when there is a mark (typically an “X”) relating the row to the column. That is, if there is an “X” marked on row B and column A then it means that B depends on A.

There are basically three types of configurations that characterize the parameters. Design parameters can have: (i) no dependencies among them, that is, they are can be developed in parallel; (ii) a sequential dependency, which means that they have a pre-defined sequence for development; and (iii) a mutual dependency in which case it is said they are coupled. Figure 7 illustrates these situations.



Relationship	Parallel			Sequential			Coupled		
DSM Representation		A	B		A	B		A	B
	A	■	▨	A	■	▨	A	■	X
	B	▨	■	B	X	■	B	X	■

Figure 7. Relationships among design parameters

Sequential and coupled design parameters represent a problem in terms of modularity. The sequential relationship in Figure 7 means that the B cannot be developed until the development of A has completed. Another drawback is that a change in A can lead to one or more changes in B. The coupled relationship in Figure 7 represents a cyclical dependency. That is, A is linked to B and vice-versa.

Neto et al [14] have analyzed two different types of dependencies named as syntactic and semantic coupling. Syntactic coupling occurs when one component contains a direct reference to some other component, such as inheritance, method calls, composition and so on. Semantic coupling is a dependency that is not syntactically defined in the code, so that there is no direct reference among the components [14].

Baldwin and Clark [15] have introduced the notion of design rules for obtaining higher software modularity. Design rules can be defined as a contract among design parameters. It is worth mentioning that they must not be considered as recommendations or development guidelines but rigorously obeyed. This way, sequential or coupled design parameters can be treated independently. Design rules establish a hierarchy among other software parameters. As they establish the communication among the system modules they must be firstly addressed. As an example, interfaces can be used to concretize the abstraction of design rules allowing two or more different modules be developed in parallel.

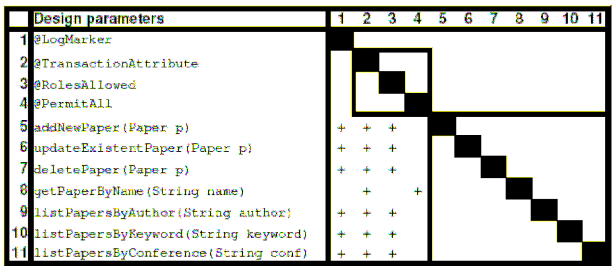
Other elements can be used to represent design rules besides interfaces. Annotations, for instance, can be used to represent design rules as well. That is the case of metadata-based frameworks, in which some specific behavior is defined in the framework that interacts with some base code via annotations.

The following subsections are going to assess modularity in the direct usage of framework annotations against the usage of domain annotations.

5.2 Evaluating Modularity with Annotations

Metadata-based frameworks take decision based on metadata related to the base code. The use of this approach has represented an improvement in the modularity of the development of frameworks [5]. Figure 8 presents a DSM containing the dependencies of the PaperService. As this section evaluates modularity of code that uses regular annotations, the PaperService class presented in the DSM of Figure 8 does not contains any domain annotations. It represents how the PaperService class would be implemented without domain annotations.

The coupling implied by the use of metadata is considered to be semantic in the same fashion that the coupling resulted implied by the use of java reflection is considered to be semantic. Therefore, the plus sign "+" is going to be used to represent semantic relationship, instead of the traditional "X" sign.



	Design parameters	1	2	3	4	5	6	7	8	9	10	11
1	@LogMarker	■										
2	@TransactionAttribute		■									
3	@RolesAllowed			■								
4	@PermitAll				■							
5	addNewPaper(Paper p)	+	+	+		■						
6	updateExistentPaper(Paper p)	+	+	+			■					
7	deletePaper(Paper p)	+	+	+				■				
8	getPaperByName(String name)		+		+				■			
9	listPapersByAuthor(String author)	+	+	+						■		
10	listPapersByKeyword(String keyword)	+	+	+							■	
11	listPapersByConference(String conf)	+	+	+								■

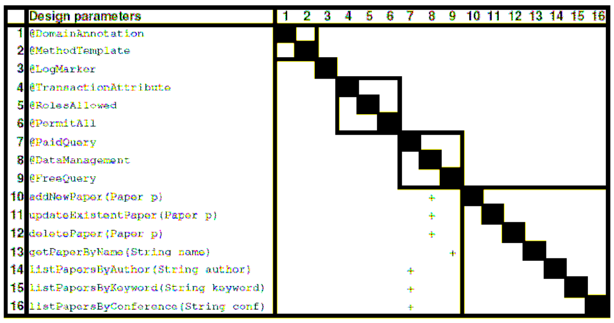
Figure 8. Paper Service with regular annotations

It is important to notice that there is a difference between a base code that depends on framework annotations and a base code that depends on domain annotations. Domain annotations are related to the business and therefore allow the base code remain oblivious of any aspects other than its own. Framework annotations are specific to the framework and as such do not make the base code oblivious to these matters.

This approach presents basically three limitations: (i) business classes lose their obliviousness because they contains code specific to the framework. Although the coupling in this case is semantic, there still code specific not related to the domain in the class; (ii) marking the code with lots of any kind of metadata may lead business classes to a clutter difficult to manage; (iii) annotated methods present patterns in the way they depend on framework annotations for some specific behavior. A change in this pattern would cause all methods that depend on it to change.

5.3 Evaluating Modularity With Domain Annotations

Domain annotations enhance application modularity. The extra layer of indirection solves the problems arisen from the use of regular annotations. As stated earlier, the technique consists of defining annotations related to the business – named domain annotations – that will be mapped to other annotations and define some specific behavior. Figure 9 depicts the PaperService class using domain annotations.



	Design parameters	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	@DomainAnnotation	■															
2	@MethodTemplate		■														
3	@LogMarker			■													
4	@TransactionAttribute				■												
5	@RolesAllowed					■											
6	@PermitAll						■										
7	@PaidQuery							■									
8	@DataManagement								■								
9	@FreeQuery									■							
10	addNewPaper(Paper p)								+		■						
11	updateExistentPaper(Paper p)								+			■					
12	deletePaper(Paper p)								+				■				
13	getPaperByName(String name)									+				■			
14	listPapersByAuthor(String author)							+							■		
15	listPapersByKeyword(String keyword)							+								■	
16	listPapersByConference(String conf)							+									■

Figure 9. Paper Service with Domain Annotations

Since there is no framework annotation in the class, an enhancement is gained in the class modularity – there are less "+" signs marked in the matrix. As a consequence, the code is cleaner, more comprehensible as well as more manageable. Modular applications allow the parallel development, since annotations may be considered as design rules.

Other advantage of the domain annotations technique is that the business application can be oblivious to elements related to any other concerns related to the framework.

6. Conclusions

This paper presented the use of domain annotations as an alternative to the metadata modularization in applications that uses metadata-based frameworks. It also presented a case study that illustrates the use of the technique and evaluates its modularity using DSMs. The following are considered by the authors the main contributions of this work:

- The development of the Daileon framework that enable the use of domain annotations for frameworks that do not support it.
- The development of a case study that suggests a development process for the use of domain annotations and enables its evaluation.
- The evaluation of the technique using DSMs, present how the metadata modularity and obliviousness can be achieved using the proposed technique.

As a future work to this research line, many improvements can be made to the Daileon framework to support a more flexible metadata handling. Some examples are the use of annotations attributes and the support of implicit metadata like name conventions and interface implementation. Other alternatives to the annotation mapping definition, like the use of XML documents, also should be explored.

References

[1] Wada, H.; Suzuki, J. "Modeling Turnpike Frontend System: a Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming". In Proc. of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Sytems (MoDELS/UML 2005), 2005.

[2] "JSR 175 - A Metadata Facility for the Java Programming Language", 2003. Available at: http://www.jcp.org/en/jsr/detail?id=175.

[3] Guerra, Eduardo; Souza, Jerffeson; Fernandes, Clovis "A Pattern Language for Metadata-based Frameworks", 16th Conference on Pattern Languages of Programming, 2009.

[4] "Convention over Configuration", Available at: http://softwareengineering.vazexqi.com/files/pattern.html

[5] Guerra, Eduardo; Silva, Jefferson; Silveira, Fábio; Fernandes, Clovis "Using metadata in aspect-oriented frameworks", in 2nd Workshop on Assessment of Contemporary Modularization Techniques (AcoM.08). OOPSLA, October 2008.

[6] Bauer, C.; King, G. "Hibernate in Action". Manning Publications, 2004.

[7] "JSR 220: Enterprise JavaBeans 3.0", 2006. Available on http://www.jcp.org/en/jsr/detail?id=220.

[8] "Spring Framework", Available at http://www.springframework.org/.

[9] Evans, E. "Domain-Driven Design: Tackling Complexity In the Heart of Software". Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] Tansey, W.; Tilevich, E. "Annotation Refactoring: Inferring Upgrade Transformations for Legacy Applications", The International Conference on Object Oriented Programming, Systems, Languages and Applications - OOPSLA 2008, Nashville, USA, 2008.

[11] E. Doernenburg. "Domain Annotations" In The ThoughtWorks Anthology: Essays on Software Technology and Innovation, chapter 10. Pragmatic Bookshelf, Raleigh, NC, USA, March 2008.

[12] Perillo, José Roberto; Guerra, Eduardo; Fernandes, Clovis "Daileon: A Tool for Enabling Domain Annotations", 6th ECOOP'2009 Workshop on Reflection, AOP and Meta-Data for Software Evolution, Genova, Italy, 7th of July 2009

[13] ASM "ASM Engineering Library", 2005.

[14] Neto, A. C., de Medeiros Ribeiro, M., D´osea, M., Bonifácio, R., Borba, P., and Soares, S.. Semantic Dependencies and Modularity of Aspect-Oriented Software. In 1st Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07), in conjunction with the 29th International Conference on Software Engineering, 2007

[15] Baldwin, C. Y. and Clark, K. B. Design Rules, Vol. 1: The Power of Modularity. The MIT Press, 2000.