F3 - From Features to Framework

Matheus C. Viana[1], Rafael S. Durelli[2], Rosângela A. D. Penteado[1] and Antônio F. do Prado[1]
[1]*Department of Computing, Federal University of São Carlos, São Carlos, SP, Brazil*
[2]*Institute of Mathematical and Computer Sciences, University of São Paulo, São Carlos, SP, Brazil*

Keywords: Reuse, Framework, Pattern, Domain, Feature.

Abstract: Frameworks allow applications to be developed more efficiently and with higher quality since application functionality can be designed and implemented by reusing framework classes. However, frameworks are hard to develop, learn and use, due to their adaptive nature. The effort to develop a framework tends to be greater than an application. In this paper we propose an approach to facilitate the development of white box frameworks. In this approach, denominated From Features to Framework (F3), the domain of the framework is modeled and a set of patterns guides the developer to design and implement the framework according to the elements and rules defined in this model. An example of use of the F3 approach is also presented in this paper.

1 INTRODUCTION

Frameworks support the design and the implementation of applications providing abstract classes with partially implemented functionality. When a framework is reused, developers complete its functionality with application-specific details. Thus, applications are not developed from scratch, reducing the time spent in their development and making use of the quality of framework code (Abi-Antoun, 2007; Lopes et al., 2005; Johnson, 1997).

Due to these advantages, when many closely related applications are developed, like in a software product line, a framework can be used as the core asset (Kim et al., 2004; Weiss and Lai, 1999). Common features of the domain are implemented in the framework and applications can have their specific elements. Moreover, frameworks are often used in the implementation of common non-functional requirements, such as persistence (JBoss Community, 2013) and software architectures (Spring Source Community, 2013).

However, frameworks are hard to develop, because their classes must be abstract enough to be reused by several applications. Developers must determine the domain of the applications able to be instantiated from the framework and how the framework accesses the application-specific elements, despite these elements are unknown during framework development (Parsons et al., 1999; Weiss and Lai, 1999).

Frameworks are also hard to learn and use. Typically, it is a steep learning curve since application developers need to understand the complex design of the framework to know which classes will be reused by the application and the rules to do it. Some of these rules may not be apparent in the framework interface (Srinivasan, 1999). Even developers who are conversant with a specific framework may make mistakes while reusing it to instantiate an application.

In a previous paper we devised an approach for building Domain-Specific Modeling Languages (DSML) to facilitate framework reuse (Viana et al., 2012). A framework DSML could be built by identifying the features of the framework and the information required to instantiate them. Application models created with the DSML were used to generate application code, protecting developers from framework complexities.

In this paper we propose the approach that aims to facilitate the development of white box frameworks that are domain-specific. In this approach, denominated From Features to Framework (F3), the domain of a framework is modeled and a set of patterns guides the developer to design and implement the framework according to the features and rules defined in this model.

The novelty of our research is that F3 provides a set of patterns that can be identified in the feature models and present design and implementation solutions to the development of white box frameworks. In addition to show to the developers

C. Viana M., S. Durelli R., A. D. Penteado R. and F. do Prado A..
F3 - From Features to Framework.
DOI: 10.5220/0004417701100117
In *Proceedings of the 15th International Conference on Enterprise Information Systems* (ICEIS-2013), pages 110-117
ISBN: 978-989-8565-60-0

how they can proceed, the F3 patterns provide a systematic way to develop frameworks.

The remainder of this paper is organized as follows: the background concepts applied in this research are discussed in Section 2; the F3 approach is presented in Section 3; an example of framework development based on the F3 approach is presented in Section 4; some related works are discussed in Section 5; and conclusions and further works are presented in Section 6.

2 BACKGROUND

2.1 Patterns and Frameworks

Reuse is a practice that aims to reduce time spent in a development process and to increase the quality of the final product (Shiva and Shala, 2007). Copy/paste is the simpler form of reuse. Programming languages allow the reuse of operations, classes, modules and other blocks without the need of access to the original code. However, other reuse techniques, such as patterns and frameworks, are more sophisticated and offer reuse at more abstract levels than implementation (Frakes and Kang, 2005).

Patterns are successful solutions that can be reapplied to different contexts (Johnson, 1997). They provide reuse of the experience to help developers to solve common problems (Fowler, 2003). The documentation of a pattern mainly contains its name, the context it can be applied, the problem it is intended to solve, the solution it proposes, illustrative class models and examples of use. There are patterns for several purposes, such as design, analysis, architectural, implementation, process and organizational patterns (Pressman, 2009).

Frameworks are reusable software that can be instantiated into several applications in a domain (Johnson, 1997). Applications are linked to the framework by reusing its classes. Unlike library classes, whose execution is controlled by the applications, frameworks control the execution flux of the applications accessing the application-specific code (Pressman, 2009).

The fixed parts of the frameworks are called frozen spots. These parts implement common functionality of the domain that is reused by all applications. The variable parts, called hot spots, can adapt according to the specifications of the desired application. What differs one application from another is the way in which the framework hot spots are configured (Srinivasan, 1999).

Frameworks can be classified according to the way they are reused: white box frameworks are reused by class specialization; black box frameworks work like a set of components; and gray box frameworks are reused by the two previous ways (Abi-Antoun, 2007; Johnson, 1997).

There is another classification that takes in consideration the purpose of the frameworks: System Infrastructure Frameworks (SIF) simplify the development of software that controls low-level operations, such as operator systems and graphical window managers; Middleware Integration Frameworks (MIF) help the modularization and the integration of applications; and Enterprise Application Frameworks (EAF) originate applications that are specific to domains of industry, commerce, services, etc. (Abi-Antoun, 2007; Fayad and Schmidt, 1997).

2.2 Modeling Domains

A domain, or family, of software consists of a set of applications that share common features. A feature is a distinguishing characteristic that aggregates value to applications (Jezequel, 2012; Lee et al., 2002; Kang et al., 1990). For example, Rental, Customer, Resource and Payment could be features of the domain of rental applications.

Different domain modeling approaches can be found in the literature (Jezequel, 2012; Gomaa, 2004; Bayer et al., 1999; Kang et al., 1990). Although there are differences in the graphical notation they adopt, the semantic of the elements and rules of their features models are almost the same. The features of a domain can be mandatory or optional, have variations and require or exclude other features.

In a feature model, the features are arranged in a tree-view notation. Usually, the feature that most represents the purpose of the domain is put in the root and a top-down approach is applied to add the other features. For example, the main purpose of the domain of rental applications is to perform rentals, so Rental is supposed to be the root feature. The other features are arranged following it.

Other way to model domains is by using metamodels, such as MetaObject Facility (MOF) (OMG, 2013). They are similar to class models and therefore they are more appropriate to developers accustomed to the UML. While feature models can only define the features that compose the domain and the conditions for these features to take part in the applications, metamodels can specify attributes and operations for their elements and how these elements can communicate to each other. On the other hand, feature models can define dependencies between the features, while metamodels depend on declarative languages to do it (Gronback, 2009).

3 THE F3 APPROACH

The F3 approach has two steps: 1) Domain Modeling, in which a white box framework domain is defined; and 2) Framework Construction, in which the framework is designed and implemented according to the definitions of its domain.

3.1 Domain Modeling

The domain of applications that can be developed with the framework is determined in this step. However, feature models are too abstract to contain information enough to develop frameworks and metamodels depend on other languages to define dependencies between elements. Therefore, a new type of feature model, called F3 model, has been created to define the features of the frameworks in the F3 approach.

F3 models incorporate the characteristics of both feature models and metamodels. The elements represent the features of the framework domain, i.e., the domain of applications that can be developed with the outcome framework. In white box frameworks, an instance of a feature is an application class that reuses the framework class of this feature. As in conventional feature models, the features in the F3 models can also be arranged in a tree-view, in which the root feature is decomposed in other features. However, the features in the F3 models do not necessarily form a tree, since a feature can have a relationship targeting a sibling or even itself, as in metamodels. The elements and relationships in F3 models are:

- **Feature:** represent the entities that compose the domains. graphically represented by a rounded square, it must have a name and it can contain any number of attributes and operations;
- **Decomposition:** a relationship that indicates that a feature is composed of another feature. This relationship specifies a minimum and a maximum multiplicity. The minimum multiplicity indicates whether the target feature is optional (`0`) or mandatory (`1`). The maximum multiplicity indicates how many instances of the target feature can be associated to each instance of the source feature. The valid values to the maximum multiplicity are: `1` (simple), for a single feature instance; `*` (multiple), for a list of a single feature instance; and `**` (variant), for any number of feature instances.
- **Generalization:** a relationship that indicates that a feature is a variation and it can be generalized by another feature.
- **Dependency:** a relationship that define a constraint for a feature to be instantiated. There are two types of dependency: `requires`, when the A feature requires the B feature, an application that contains the A feature has to include the B feature as well; and `excludes`, when the A feature excludes the B feature, no application can include both features at the same time.

3.2 Framework Construction

The F3 approach define a set of patterns to assist developers to design and implement a framework from the domain model. The patterns treat problems that go from the creation of classes for the features to the definition of the framework interface. The name and the purpose of some of the F3 patterns are presented in Table 1.

The documentation of the F3 patterns is organized into topics to assist the developers to identify when a certain pattern should be applied. The topics of this documentation are described as follows and exemplified in Table 2:

- **Name:** identifies each pattern and summarizes its purpose.
- **Context:** describes a desired behavior for the framework/domain.
- **Scenario/Problem:** describes the arrangement of features and relationships in the F3 models that can imply the pattern.
- **Solution:** indicates the code units that should be created to implement the desired behavior.
- **Model:** shows a generic graphical representation of the scenario/problem and the solution.
- **Implementation:** displays a fragment of code, in a programming language, that illustrates how the solution can be implemented.

In addition to provide solutions that indicates the code units that implement the framework functionality, the F3 patterns also determine how the framework can be reused by the applications. For example, the pattern presented in Table 2 suggest the creation of an operation, `getTargetClass`, that must be overridden in the instances of the source feature to indicate which class is an instance of the target feature.

Table 1: Some of the F3 patterns.

Pattern	Purpose
Domain Feature	Indicates the structures that should be created for a feature.
Mandatory Decomposition	Indicates the code units that should be created when there is a mandatory decomposition linking two features.
Optional Decomposition	Indicates the code units that should be created when there is an optional decomposition linking two features.
Simple Decomposition	Indicates the code units that should be created when there is a simple decomposition linking two features.
Multiple Decomposition	Indicates the code units that should be created when there is a multiple decomposition linking two features.
Variant Decomposition	Indicates the code units that should be created when there is a variant decomposition linking two features.
Variant Feature	Defines a hierarchy of classes for features with variants.
Modular Hierarchy	Defines a hierarchy of classes for features with common attributes and operations.
Requiring Dependency	Indicates the code units that should be created when a feature requires another one.
Excluding Dependency	Indicates the code units that should be created when a feature excludes another one.

Table 2: The Mandatory Decomposition pattern.

Name	Mandatory Decomposition
Context	When a target feature is mandatory to a source feature, every instance of the source feature must be associated with a instance of the target feature.
Scenario/ Problem	A feature has a decomposition relationship with minimum multiplicity equals `1`.
Solution	The class that implements the source feature must have an abstract operation that indicates what class implements the target feature in the applications.
Model	The scenario and the design solutions of this pattern are shown in Figure 1
Implementation	`public abstract class Source {` `public abstract` `Class<? extends Target>` `getTargetClass();` `}`

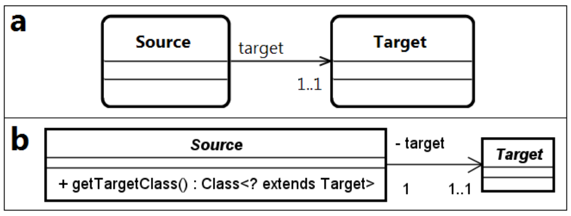

Figure 1: The (a) scenario described by the pattern in the domain models and (b) its design solution.

The step of Framework Construction has as output a white box framework for the domain defined in the step of Domain Modeling. Although the F3 patterns support many aspects of framework development, they have limitations. They do not assist the implementation of the internal code of the operations defined in the domain model. They also do not assist the implementation of non-functional requirements, such as data persistence, logging and graphical user interface. However, frameworks specific for these purposes can be reused together the frameworks developed with the F3 approach.

4 APPLING THE F3 APPROACH

In this section it is presented an example of use of the F3 approach for the development of a framework in the domain of rental and trade transactions. The requirements of this domain are:

1. One or more resources can be traded or rented by a destination party. Both rental and trade transactions have the following attributes: number, date and total value. Rental transactions also includes a ending date and a return date.
2. A resource has three attributes: ID, description and value. When a resource participates in a transaction, it is regarded as transaction item and it is necessary to specify its quantity and value.
3. A resource type defines a classification for a resource based on a perspective. For example, in an movie rental application, a Movie can be a resource classified by the types Category, Genre, Director and so on. Thus, there may be any

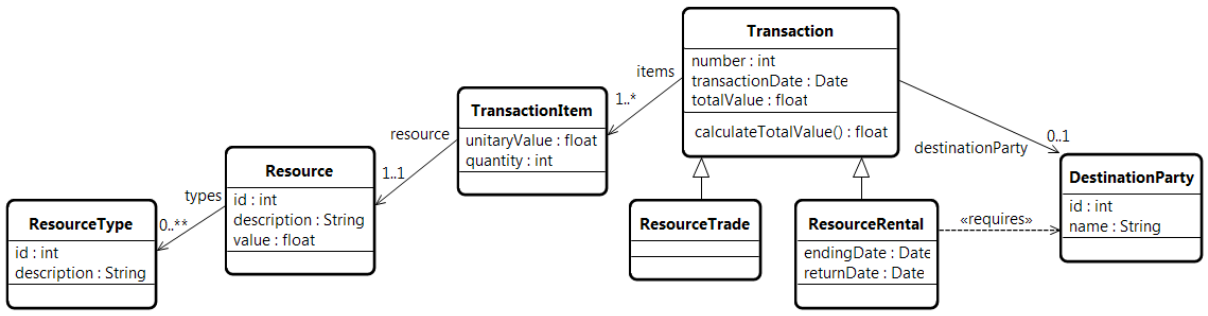

Figure 2: F3 model for the domain of rental and trade transactions.

number of resource types for a resource. The attributes of resource type are ID and description.

4. A destination party has ID and name as attributes and it is mandatory for rental transactions.

The following sections describe the steps of Domain Modeling and Framework Construction for the framework that deal with rental and trade transactions.

4.1 Rental and Trade Transaction Domain Modeling

The F3 model of the domain of rental and trade transactions was created based on the rules described in Section 3.1 and the requirements of this domain. This model is shown in Figure 2.

`Rental` and `Trade` are variations of the `Transaction` feature. Since destination party is optional for trade transactions, but not for rental transactions, a dependency of the type `requires` was established between the features `Rental` and `DestinationParty`.

`Transaction` also has a mandatory one-to-many decomposition with `TransactionItem`. Each transaction item register a resource that is involved in a transaction and the value that must be paid for it, hence `Resource` is also mandatory to `TransactionItem`.

`Resource` has a decomposition relationship with variant maximum multiplicity (`**`), because there may be many classes in the applications that implement the `ResourceType` feature to classify the resources defined in these applications. The minimum multiplicity of this decomposition relationship indicates that `ResourceType` is optional for `Resource`.

4.2 Rental and Trade Transaction Framework Construction

The step of Framework Construction for the domain of rental and trade transactions has been carried out applying the F3 patterns listed in Table 3.

The Domain Feature pattern was used to define a class for each feature in the F3 model of Figure 2. The Variant Feature pattern was applied to implement `Trade` and `Rental` as classes which extend the `Transaction` class. The Simple Decomposition and the Multiple Decomposition patterns were applied to define the whether an association is implemented as a single attribute or a list in the framework classes. For example, the attributes `destination` and `items` in the `Transaction` class were implemented as follows:

```
public abstract class Transaction {
```

Table 3: F3 patterns applied in the step of Framework Construction of the domain of rental and trade transactions.

Pattern	**Applied to**
Domain Feature	All features.
Variant Feature	`Transaction`, `Trade` and `Rental`.
Simple Decomposition	`Transaction` to `DestinationParty` and `TransactionItem` to `Resource`.
Multiple Decomposition	`Transaction` to `TransactionItem`.
Optional Decomposition	`Transaction` to `DestinationParty` and `Resource` to `ResourceType`.
Mandatory Decomposition	`Transaction` to `TransactionItem` and `TransactionItem` to `Resource`.
Variant Decomposition	`Resource` to `ResourceType`.
Requiring Dependency	`Rental` to `DestinationParty`.
Modular Hierarchy	`number`, `id`, `description` and `name` in `Transaction`, `Resource`, `ResourceType` and `DestinationParty`.

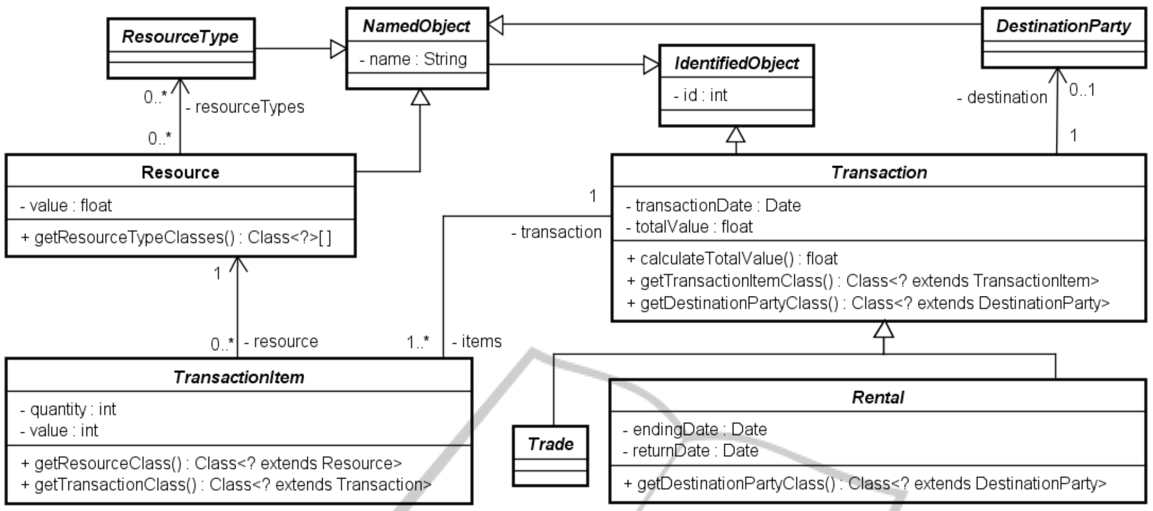
ResourceType
NamedObject
- name : String
IdentifiedObject
- id : int
DestinationParty
0..*
- resourceTypes
0..*
- destination
0..1
1
Resource
- value : float
+ getResourceTypeClasses() : Class<?>[ ]
1
- transaction
Transaction
- transactionDate : Date
- totalValue : float
+ calculateTotalValue() : float
+ getTransactionItemClass() : Class<? extends TransactionItem>
+ getDestinationPartyClass() : Class<? extends DestinationParty>
1
0..*
- resource
1..*
- items
TransactionItem
- quantity : int
- value : int
+ getResourceClass() : Class<? extends Resource>
+ getTransactionClass() : Class<? extends Transaction>
Trade
Rental
- endingDate : Date
- returnDate : Date
+ getDestinationPartyClass() : Class<? extends DestinationParty>

Figure 3: Design class model of the framework in the domain of rental and trade transactions.

```
   private DestinationParty destination;
   private List<TransactionItem> items;
   ...
}
```

The Mandatory Decomposition and the Optional Decomposition patterns suggest the creation of an operation that allows the framework to identify application classes. However, in the Mandatory Decomposition pattern this operation is abstract to force applications to inform which class implements the mandatory feature, while in the Optional Decomposition pattern this operation has a default implementation which returns null to indicate that the optional feature is not being implemented. For example, in the `Transaction` class the operations that identify which application classes extend `TransactionItem` and `DestinationParty` were implemented as follows:

```
\\Mandatory Decomposition
public abstract
  Class<? extends TransactionItem>
    getTransactionItemClass();

\\Optional Decomposition
public Class<? extends DestinationParty>
  getDestinationPartyClass() {
    return null;
}
```

The Variant Decomposition pattern was applied to allow each application class that extends `Resource` to be associated with several subclasses of `ResourceType`. To identify which classes extend ResourceType in the applications, the following operation was implemented in the `Resource` class:

```
public abstract
  Class[]<? extends ResourceType>
    getResourceTypeClasses();
```

Although destination party is optional for transactions in general, the `requires` dependency makes it mandatory for rental transactions. It implies in the use of the Requiring Dependency pattern. Thus, the operation `getDestinationPartyClass` is overridden in the `Rental` class to become abstract.

Finally, to enhance the design of the framework, the Modular Hierarchy pattern was applied to organize common attributes and operations in reusable abstract classes. Then the `IdentifiedObject` and `NamedObject` classes were created to contain the attributes `id` and `name` (`description` in some classes), respectively, and all operations related to these attributes.

After applying all patterns listed in Table 3, the design class model of the framework that was created with the support of the F3 patterns is shown in Figure 3. Constructors and the getter/setter operations has been omitted in this model to simplify it.

5 RELATED WORKS

Keepence and Mannion (1999), Almeida et al. (2007) and Loo and Lee (2010) proposed the use of design patterns to model variabilities in domains. Srinivasan (1999) also discussed the applicability of design patterns for the development of frameworks, but she did not work with feature models. Lopes et al. (2009) and Stanojevic et al. (2011) performed an research that analyzed framework reuse difficulties and described some programming and design techniques that positively impact on framework reusability. These works served as basis for the F3 approach, specially regarding the creation of the F3 patterns (Keepence and Mannion, 1999; Almeida et al., 2007; Loo and Lee, 2010; Srinivasan, 1999; Lopes et al., 2009; Stanojevic et al., 2011).

Xu and Butler (2006) proposed an cascaded refactoring method which addresses the identification of variability and framework development. In this method, a framework is specified by different models, sorted from the most abstract (feature model) to the least abstract (source-code). A set of refactorings is performed sequentially on the models and alignment maps are defined to maintain the traceability amongst the models by linking correspondent elements. In the F3 approach the domain variabilities are documented in the F3 models. Moreover, the F3 patterns can provide a traceability between the features in these models and the elements of the design and the implementation of the framework (Xu and Butler, 2006).

Amatriain and Arumi (2011) also proposed a method for the development of a framework through iterative and incremental activities. In their method, the domain of the framework could be defined from existing applications and the framework could be implemented through a series of refactorings over these applications. The advantage of this method is a small initial investment and the reuse of the applications. Although it is not mandatory, the F3 approach can also be applied in iterative and incremental activities, starting from a small domain and then adding features. Applications can also be used to facilitate the identification of the features of the domain. However, the advantage of the F3 approach is the fact that the design and the implementation of the frameworks are performed with the support of patterns specific for framework development (Amatriain and Arumi, 2011).

6 CONCLUDING REMARKS AND FUTURE WORK

This paper proposed the F3 approach for the development of white box frameworks from feature models. The F3 approach uses a kind of feature model that combines characteristics from conventional feature models and metamodels to define the domain of the framework. Then, to design and implement the framework, the approach offers patterns that indicates the classes, properties and operations that should be created based on the elements and relationships found in the domain model of the framework.

The F3 model allows the developers to define the domain of the framework regardless design and implementation details. It can reproduce different domain scenarios that involve decompositions, dependencies, and variabilities of the features.

The F3 patterns are independent of programming language, although their documentation contains examples of code implemented in Java. Their design solution can be used to develop many versions of the same framework implemented in different programming languages.

More F3 patterns are being created to deal with data persistence in the frameworks. Moreover, a tool with a F3 model editor and a code generator based in the F3 patterns are being developed. In other future works we also intend to create patterns to provide a Model-View-Controller architecture to the frameworks created with the F3 approach.

ACKNOWLEDGEMENTS

We would like to thank CAPES and FAPESP for financial support.

REFERENCES

Abi-Antoun, M. (2007). Making Frameworks Work: a Project Retrospective. In *Companion to the 22nd ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 1004–1018, New York, NY, USA. ACM.

Almeida, E. S., Alvaro, R., Garcia, V. C., Nascimento, R., Meira, S. L., and Lucrdio, D. (2007). A systematic approach to design domain-specific software architectures. *Journal of Software*, 2(2).

Amatriain, X. and Arumi, P. (2011). Frameworks Generate Domain-Specific Languages: A Case Study in the Multimedia Domain. *Software Engineering, IEEE Transactions on*, 37(4):544–558.

Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., and DeBaud, J.-M. (1999). Pulse: a methodology to develop software product lines. In *Proceedings of the 1999 symposium on Software reusability*, pages 122–131. ACM.

Fayad, M. and Schmidt, D. C. (1997). Object-Oriented Application Frameworks. *Communications of ACM*, 40(10).

Fowler, M. (2003). Patterns. *IEEE Software*, 20(2):56–57.

Frakes, W. and Kang, K. (2005). Software reuse research: Status and future. *Software Engineering, IEEE Transactions on*, 31(7):529–536.

Gomaa, H. (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley.

Gronback, R. C. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley.

JBoss Community (2013). Hibernate. http://www.hibernate.org.

Jezequel, J.-M. (2012). Model-Driven Engineering for Software Product Lines. *ISRN Software Engineering*, 2012.

Johnson, R. E. (1997). Frameworks = (Components + Patterns). *Communications of ACM*, 40(10):39–42.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute.

Keepence, B. and Mannion, M. (1999). Using patterns to model variability in product families. *Software, IEEE*, 16(4):102 –108.

Kim, S. D., Chang, S. H., and Chang, C. W. (2004). A systematic method to instantiate core assets in product line engineering. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 92–98.

Lee, K., Kang, K. C., and Lee, J. (2002). Concepts and guidelines of feature modeling for product line software engineering. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, ICSR-7, pages 62–77, London, UK. Springer-Verlag.

Loo, K. N. and Lee, S. P. (2010). Representing Design Pattern Interaction Roles and Variants. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 6, pages 470–474.

Lopes, S., Afonso, F., Tavares, A., and Monteiro, J. (2009). Framework characteristics - a starting point for addressing reuse difficulties. In *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, pages 256 –264.

Lopes, S. F., Silva, C. A., Tavares, A., and Monteiro, J. L. (2005). Application development by reusing object-oriented frameworks. In *International Conference on Computer as a Tool (EUROCON'05)*, pages 583–586.

OMG (2013). OMG's MetaObject Facility. http://www.omg.org/mof.

Parsons, D., Rashid, A., Speck, A., and Telea, A. (1999). A ldquo;framework rdquo; for object oriented frameworks design. In *Technology of Object-Oriented Languages and Systems, 1999. Proceedings of*, pages 141 –151.

Pressman, R. S. (2009). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science, 7th edition.

Shiva, S. G. and Shala, L. A. (2007). Software reuse: Research and practice. In *Information Technology, 2007. ITNG '07. Fourth International Conference on*, pages 603–609.

Spring Source Community (2013). Spring Framework. http://www.springsource.org/spring-framework.

Srinivasan, S. (1999). Design Patterns in Object-Oriented Frameworks. *Computer*, 32(2):24 –32.

Stanojevic, V., Vlajic, S., Milic, M., and Ognjanovic, M. (2011). Guidelines for Framework Development Process. In *Software Engineering Conference in Russia (CEE-SECR), 7th Central and Eastern European*, pages 1–9.

Viana, M., Penteado, R., and do Prado, A. (2012). Generating Applications: Framework Reuse Supported by Domain-Specific Modeling Languages. In *14th International Conference on Enterprise Information Systems (ICEIS'14)*.

Weiss, D. M. and Lai, C. T. R. (1999). *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley.

Xu, L. and Butler, G. (2006). Cascaded refactoring for framework development and evolution. *Software Engineering Conference, Australian*, pages 319–330.