On the Extensibility Requirements of Business Applications

Mohamed Aly Anis Charfi
SAP Research Darmstadt, Germany
{mohamed.aly,anis.charfi}@sap.com

Mira Mezini
Software Technology Group, TU Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

Abstract

Business applications play a crucial role for the day-to-day running of a business. These applications typically support a wide range of standard business processes like opportunity-to-order and order-to-cash. Customers using these solutions often demand extensions that will complement the existing functionalities offered by the standard application. The requirements for extensibility can be different for each customer which makes the enablement of business software for extensibility very challenging. In this paper we demonstrate some of these challenges and requirements through an example application and evaluate them against some state-of-the-art works on extensibility.

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design—Methodologies

General Terms Design, Languages

Keywords Business, Requirements, Extensibility, Extensions

1. Introduction

Software systems designed and built for specific purposes are often required to accommodate new functionalities to enhance, compliment, or change existing features. This trend in software flexibility is becoming a necessary part of modern software as it becomes more oriented towards end user customizations and requirements. As an observation, most software is currently delivered with a set of core functionalities and is left open for expansions through different extensibility means. Terms like plug-ins, add-ons, apps, and extensions are emerging as popular means for extending the functionalities of a core software. For example, Eclipse [1], the famous platform for building IDEs, is based on the idea of having different tools that contribute to the development process as independantely developed pluggable components (plug-ins). The Eclipse core platform however is responsible for handling the integration and runtime to ensure that plug-ins seamlessly work together (this is similar to a microkernel). Another example of an extensible application is Firefox [2], a popular web browser. Firefox offers two means for extensions namely add-ons and plugins. Add-ons bring in new functionalities (i.e. a new toolbar, a new button, etc.) whereas a plug-in is used to display content Firefox cannot natively render (like special audio, video, and text formats).

Software extensions has also been adopted in the context of cloud based business software. SAP Business ByDesign [3], Netsuite [4], and SalesForce [5] are examples of cloud based business software products. In this context, business applications, like enterprise resource planning (ERP) and customer relationship management (CRM), are delivered in the software as a service (SaaS) form where all software and data associated are stored centrally and accessed using a thin client over the web. Different tenants can choose relevant core business software modules and further customize their selections with extensions. The software provider has to have a convenient model to enable the core software for extensibility and to facilitate the development of extensions.

Business applications are typically built according to these layers: business process, graphical user interface (GUI), and business objects (BO). The business process layer depicts the underlying business process which this application supports, the GUI layer contains the interaction elements that will be used, and business objects hold the data model and business logic needed for the process execution. Extensions can take place at one or many layers of the software. Several challenges arise to declare extensible artifacts at each layer. Some of these challenges address issues like how to enable different views (white, grey, and black box) on each layer, how define cross-layer extension points, constraints associated with extension points, and how to simplify the understanding and consumption of the underlying extensibility model of an application.

In this paper we outline some of these extensibility requirements through an example business application and analyze them against state-of-the-art approaches to demonstrate the need for better solutions to support the cross-layer extensibility of business applications.

2. Business Software

Enabling business software for extensibility is very challenging. There are several stakeholders involved. The ***business software provider*** is responsible for providing a core business software that typically supports a set of standard business processes for a wide range of ***customers***. The software can be installed locally on premises or offered in a cloud setup in the form of a Software as a Service (SaaS). Each customer can then choose to further extend the core functionalities by integrating more add-ons to the application. These add-ons are developed by ***extension developers*** and can be made available through an online add-on store where they can be downloaded and activated by ***end users***. The extensions can also be tailored and developed offline and integrated manually.

[1] http://www.eclipse.org

[2] http://www.firefox.com

[3] http://www.sap.com/bydesign

[4] http://www.netsuite.com

[5] http://www.salesforce.com

NEMARA'12, March 27, 2012, Potsdam, Germany.

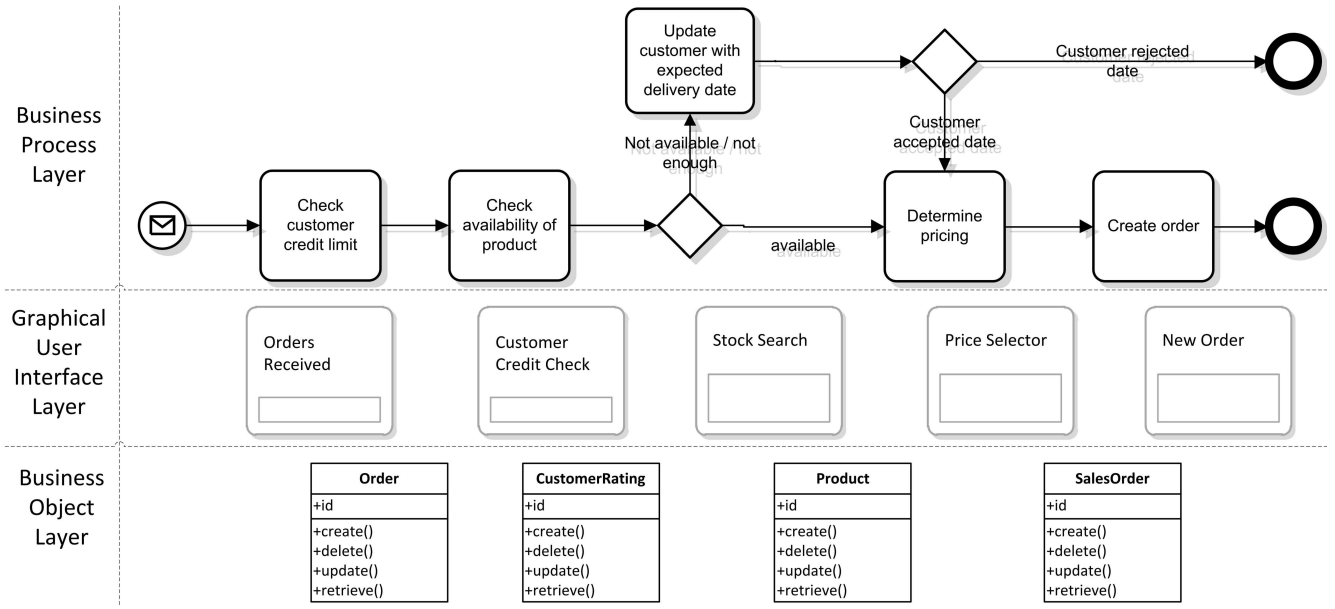

Figure 1. An example application: Simplified order-to-cash

A business software is used to support the execution of a set of business processes (for example, sales order processing, or human resources management process). For the purpose of defining our extensibility requirements, we can abstract a business software to be composed of the following layers: business process, user interface, and business object layer. Of course this is only a simplified abstraction of a business software (for exampkle reporting or persistency have not been taken into consideration). The business process layer depicts underlying business process activities and tasks along with their execution sequence. The user interface layer involves all screen forms, and reports that are used throughout the execution of a certain business process (for example, a Sales Order form and a Billing form). The business object layer contains data and business logic that is performed in the background to ensure the correct execution of the business process as well as to validate the consistency of the data the software is using. More information about business objects can be found in [8]. In Section 2.1, we will give an example of a business application with an emphasis on the three described layers and an extension scenario.

2.1 Example Application

2.1.1 The core application

In an enterprise resource planning (ERP) application, an important business process is the order-to-cash [11]. This process involves activities that aim to receive and process an order initiated by a customer. In this example, we will consider a simplified version of an application implementing the order-to-cash process as well as all artifacts involved (UI and business objects). Figure 1 illustrates our example.

Business process In this application, the underlying business process is as follows. Orders sent by customers are constantly monitored. Once an order arrives, the customer credit limit and the availability of the products ordered are checked internally. If products are not available, the customer is informed about a forecasted delivery date of his order. If the customer accepts the delivery date, the orders are processed normally, otherwise the process ends. Finally, a pricing scheme can be determined and a sales order can then be created.

User interface The end user is expected to interact with several forms as depicted in the figure. User interfaces for checking orders, customer credit, product availability, pricing, and creating a new order are provided.

Business objects Based on the business objects (BO) provided, the application is able to encapsulate different functionalities and access the underlying database to create, retrieve, update, and delete records. The $Order$ BO holds the records of all orders requested by customers. The $CustomerRating$ BO holds credit rating information for each customer. The $Product$ BO holds the information of all products as well as the availability of the products in stock. The $SalesOrder$ BO contains the sales order information generated for each customer order. Each BO stated also contains necessary business logic to ensure the proper execution of the business process and the consistency of the data.

2.1.2 An extension scenario

The core application described provides the necessary basic functionalities for an order-to-cash process. Consider the following extension scenario of a customer who would like to extend his business process to include an additional activity which involves checking a customer credit risk with an external credit reporting agency. For that purpose, the following is required on each of the layers:

- Business process: a new activity should be added for the external check after (or in parallel with) the internal customer credit local check activity and before the product availability check activity.
- GUI: the existing customer credit check user interface should be updated with UI elements that are necessary to display the indicator of the external credit reference agency rating.
- Business object: a new $CustomerRating$ BO should be implemented to communicate with the external credit reporting agency, store queries, and additional business logic to account for that. The $OrderPricing$ BO price calculation method should be extended to account for the external rating.

3. Requirements for Extensibility

Given the example described in Section 2.1, we describe the following requirements that are necessary for enabling extensibility:

R1: Controlled Visibility The developers of the base software would often like to hide their implementation details from extension developers. Giving the source code under the full disposal of an extender is undesirable in the context of business software. There can be parts of the source code that implement functionality which is compliant to certain legal measures. For example, in our example application, the $SalesOrder$ BO contains methods that calculate tax. The calculation is done based on tax calculation law within the country. Another important reason to hide source code is to avoid problems with upgrades. If an extender has the full control to modify the source code of the core system, future upgrades are more likely to fail. The extender, therefore, should only be allowed to extend parts of the source code which are accounted for. Hiding source code can also be important to hide the extender from the underlying complexity of the core system or to protect intellectual property. Three different views on the source code have been widely used in literature; ***white-box***, ***gray-box***, and ***black-box*** view [23]. A white-box view guarantees that full access to the extender is provided. However, giving an extension developer the source code, does not mean that he can change whatever he can to accommodate a new extension. This leads to the further distinguishment of two forms of white-box exposure: ***open-box*** and ***glass-box***. In an open-box view, the extension developer can see the source code and is allowed to modify it and change whatever he wants to accommodate his new extension. The resulting binaries might therefore be totally new. A glass-box view is more restrictive. The extension developer is allowed to see the source code but is not allowed to modify it. Therefore, an extension is separately developed and extends the binaries of the base software. The black-box view is the most restrictive form of views. In this view no implementation details are provided to the extender. The software can be abstracted through particular interfaces that can be used by an extension for integration. The gray-box view provides a compromise between the white-box and the black-box view on the software. Grey-box view provides a restricted form of access to a system artifact. In this view some aspects are hidden and others are exposed. The core software developer should be able to define different views on source code as necessary.

R2: Controlled Extensibility Besides defining what is visible and what is not, a core developer should be able to define which artifacts in the three mentioned layers are extensible or not i.e. what are the extension points in each layer. It should also be possible to define any extensibility constraints that are involved with an extensible artifact. For example, at the business process layer, the core developer should have means to declare certain activities within a business process as extensible as well as constraints that exist (e.g. on data or in relevancy of execution order). The developer should also be able to define which user interface forms and form elements are extensible (for example, where can an extender add a new field or a new panel). It should also be possible for the developer to declare extensible business object artifacts. Furthermore, besides declaring artifacts as extensible, the core developer should be able to specify what data is available and how it can be accessed (e.g. read and write permissions) at each extension point. In our application example, the activity associated with the internal customer rating check should be declared as extensible and appropriate customer data should be made available to be able to use it for the forecasted extension. The relevant user interfaces should also be declared as extensible and appropriate data connectors should be made possible. In conclusion, appropriate constructs are required to define and control extension points.

R3: Stable Contract The development of extensions and the core software should be separated. There should be no dependencies between the base system development and the extension development. Parallel development of core and extensions should be possible without any problems. This implies that a stable contract must be provided by a core developer for an extender. The core software should be also able to anticipate and support most extension scenarios for each artifact at each layer.

R4: Support for Extensions on Multiple Layers The core software should allow for extensions at different layers and granularity levels ranging from fine grained to coarse grained. Extensions at the process, UI, and BO layer should be supported. This implies that extensions can be as big as an activity within a business process or as small as a simple formula that calculates a new field. Proper abstractions for structural and behavioral aspects within the software should be provided to the extender at different granularity levels.

R5: Composition Approach Composition [19] refers to the integration of the extension with the core software. Business software, especially when offered as a SaaS, is usually provided with certain service level agreements that have to be achieved. The uptime of the software is a very important factor which imposes a constraint on the composition of the extension with the base software. Composition should can take place at ***compile time***, ***load time***, or at ***runtime***. Ideally an extension should be composed and activated within the least downtime possible (at runtime). The core and the extensions should be composed and provide a seamless runtime view to the end user.

R6: Invasiveness Sometimes changes are required in the base software to accommodate an extension. Ideally, such changes should not exist. With an increase in number of customers building extensions, it will be impossible to manage different versions of the core software. For the business software provider, the software upgrade cycles can be adversely affected as new releases of the core might not be compatible with the extensions existing. The effort and costs required to adapt the software for new core upgrades will increase. Therefore, non-invasiveness is crucial and the software should be able to integrate an extension without any modifications to the core.

R7: Multiple Extensions The core software should be able to integrate and manage multiple extensions. It should also be able to reflect the current state of the underlying business process running with all the extensions in place as well as all artifacts that have been affected. To support the composition of multiple extensions, means for resolving conflicts must also exist. For example, there should be ways to introduce ordering constraints and dependency constraints (e.g. to indicate which extension should come first).

R8: Simplified Consumption of the Extensibility Model It is very important to attract developers to build extensions for business applications. The more developers that exist for a certain business software, the more likely customers will be willing to invest in it. If the underlying extensibility model is complicated, it would be less likely that many developers would contribute to develop extensions. Given the large number of artifacts at each layer of the software, the possibility for extensibility can be overwhelming for an extender. The developer will have to go through a lot of documentation and understand how different artifacts are related. The relationships between extension points, constraints, and extension methods has to be presented in a simplified way for an extender. For example, in our scenario, the developer might find it helpful to understand the relationship between the business activity associated with the customer credit limit check and the user interface and business object associated with it.

4. Related Work

In the following we summarize a part of our survey on the related work on extensibility of software briefly evaluate them with respect to our requirements. We categorize our findings to code level, programming paradigms, and frameworks / other approaches.

Table 1. Evaluation of the related work: + satisfies, - does not satisfy, P partially satisfies

	R1	R2	R3	R4	R5	R6	R7	R8
Mixins	-	-	-	-	-	-	-	-
Traits	-	P	-	-	-	-	P	-
Virtual Classes	-	P	P	-	-	-	-	-
Design Patterns	+	+	-	-	+	+	P	-
FOP	+	-	-	-	-	-	P	-
AOP	-	-	-	-	-	-	P	-
CBSE	+	+	+	-	P	+	+	P
Plug-ins	+	P	+	-	P	+	+	-

4.1 Code Level Approaches

Mixins A mixin [5, 15] is an abstract subclass that defines a particular functionality without specifying the intention of usage. A parent class can be composed of multiple mixins and thus inherits all functionalities specified by the mixins. Mixins use single inheritance as means of composition (R5). The order at which mixins are inherited can influence the structural and behavioral properties of the target class. It might also be required to introduce complimentary code to ensure the correct integration of multiple mixins (R6). Given the resulting inheritance chains with glue code, the introduction of new mixins to an existing parent class can be very tedious (R7). Furthermore, the modification of a mixin that is being used can as well be difficult as dependencies can exist.

Traits A trait [12, 13] is a set of methods and act as a composable unit of behavior. A trait provides a collection of methods that implement behavior and requires a set of methods that parameterize the provided behavior (R2). Each trait has a state which is only accessible via its methods. The resulting class is made up of a state, a set of traits, and complimentary code (glue code) that connects the traits and implements the class logic and interface (R6). There are rules and operators defined for the composition of traits. Operators include sum, exclusion, and aliasing. There are several rules that are used for composition (R7). The order of composition does not matter as the resulting class is flattened. Methods defined within a class takes precedence over those defined within traits. Conflicting methods are excluded from the composition and an overriding method is placed in the parent class.

Virtual Classes Virtual classes [14, 21] offer language mechanisms to specify a certain class pattern which can then be inherited and specified. Virtual classes are defined as inner classes. The concept is similar to virtual functions, however in contrast to virtual functions, the whole class with its methods and attributes can be specified (R2,R3). During runtime, the type of the object of the outer class decides which virtual class implementation should be used. With this approach, extension points have to be preplanned ahead and type safety problems can exist.

Design Patterns Design patterns [16] are patterns in software design that aim to solve reoccurring problems. Each pattern can either have a creational, structural, or behavioral purposes. Patterns are usually documented and described in terms of purpose, motivation, structure, and relations to other patterns. An example of a structural pattern is the decorator pattern can be used to add properties to object dynamically. An example of a behavioral pattern is the visitor pattern. This pattern provides a way for separating an algorithm from an object structure on which it operates. As a consequence, this allows the introduction of new behavioral aspects without modifying an existing object structure. Design patterns provide a good solution for software design challenges, however there are a lot of issues and concerns regarding traceability (R8), reusability, writability, and maintainability (R7) that have been pointed out [1, 4].

4.2 Programming Paradigms

Feature Oriented Programming Feature oriented programming is a programming paradigm that supports the production of large software systems [3]. The paradigm is most famous for its support of software product lines [20]. A feature represents a requirement or a functionality that is expected in the software. In FOP, three key areas play an important role: feature modeling, feature interaction, and feature implementation. Feature models [17] provide means to describe relationships and constraints between different feature. Feature interaction [7] is important to analyze if features can possibly interfere when combined together. Feature implementation involves the transformation of feature models to concrete programs. The advatange of FOP is that it supports product families with common features. A product can be constructed by adding features to the feature model (if necessary) then selecting relevant features that are needed from the model. The advantage of this approach is that it promotes feature reuse. However several problems can arise. A single feature model is maintained for a certain family of software products(R3), which makes independent extensibility very difficult (R7). Also the maintenance of feature models for large product lines can be very tedious .

Aspect Oriented Programming The main motivation behind Aspect Oriented Programming [18] is to reduce the scattering and tangeling of cross-cutting concerns that interfere with the core concerns of a base system . AOP allows the modularization of crosscutting concerns by abstracting them into advices (R7) that get executed at certain join points within the base system. An advice consists of the behavioral and/or structural additions and the join points define where the advice should run. Several join points can be targeted and refined using pointcuts. The composition of advices with the base system is known as weaving. A good survey on existing AOP languages and their models can be found in [6]. AOP assumes a white-box view on source code. The modification of code aspects highly depend on the pointcuts specified. The knowledge of the extension developer of the source code is very important (R8) to specify the right pointcuts that his advices will extend. It is also possible that the extension developer will have to modify the base code to accomodate his new advice (R6).

Component-Based Software Engineering In [22] the authors define software components as *"software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system"*. Each component encapsulates a particular functionality and the interaction of components is ensured through well defined interfaces (R3). Component models specify properties like interface types, languages used, packaging, deployment methods, and interaction styles. A recent survey on component models with a good taxonomy can be found in [10]. The idea of components offer a great concept for black-box reuse and separation of concerns. However, extending a component based system can be difficult. Building extensions are highly dependent on interface definitions, which implies that the extension of structural or behavioral attributes of an existing component might not always be feasible. Also any changes to an existing interface of a core component might adversely affect extensions. The composition of an extension component with existing components require the understanding of the current composition model (R8)

(for example data driven or event driven compositions) of an existing software. This might not be explicitly defined by an implemented system, and therefore composing a new component might lead to undesirable interactions (R5,R7).

4.3 Frameworks and Other Approaches

Plug-in Systems In plug-in systems, a core application is abstracted in terms of data and functionalities through an application programming interface (API) that act as hooks or extension points. An extender can then write applications and package them in the form of plug-ins that conform to the API. The core application contains an integrated plug-ins manager (R5) that registers a plug-in to the core application and manages its runtime. Popular plug-in systems include the OSGi [2] based Eclipse [9] and the Microsoft Managed Extensibility Framework (MEF) [6].

A plug-in in Eclipse is the smallest unit of function. Each plug-in contributes to a set of extension points and can provide a set of extension points. Each plug-in is described by a manifest file (`plugin.xml`) which describes the extension points it contributes to, dependencies to other plug-ins, and extension points it provides. The Eclipse Platform Runtime is responsible for handling the discovery, matching of extensions with extension points, and the runtime of the plug-in (for example activation when required).

In MEF, *parts* specify their dependencies (imports) and capabilities (exports) declaratively. The developer then defines a composition container with all relevant parts of his application. Based on these declarations, the MEF composition engine then discovers these parts (via catalogs) and assembles the application.

In both frameworks, extension points are dependent on the interface definitions declared by the base plug-in developer (R3,R2). These interface definitions indicate how the contributing plug-in should be called and what data it can get. The abstraction level provided by these frameworks is similar to components. The extension developer is therefore limited by the interface definitions. Extensions affecting many layers that might exist in a component are only made on the code level. There are no extensibility constructs provided by these frameworks to separately express extension points that exist on different layers (e.g. user interface).

5. Conclusion and Outlook

Business applications typically consists of several layers (business process, user interface, and business objects). Customers using these applications demand extensions to the existing features to support their business needs. Enabling business applications for extensibility can be very challenging. In this paper we have presented several requirements that are essential for the extensibility of business applications and evaluated them against some related works. Table 1 summarizes our findings. We are currently working on an approach that will support these requirements.

References

[1] E. Agerbo and A. Cornils. How to preserve the benefits of design patterns. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 134–143, New York, NY, USA, 1998. ACM. ISBN 1-58113-005-8.

[2] O. Alliance. *OSGi service platform, release 3*. IOS Press, Inc., 2003.

[3] S. Apel and C. Kstner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):4984, 2009.

[4] J. Bishop. Language features meet design patterns: raising the abstraction bar. In *Proceedings of the 2nd international workshop on The role of abstraction in software engineering*, ROA '08, pages 1–7, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-028-9.

[5] G. Bracha and W. Cook. Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, Oct 1990.

[6] J. Brichau and M. Haupt. Survey of aspect-oriented languages and execution models. *European Network of Excellence in AOSD*, 2005.

[7] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115141, Jan 2003.

[8] C. Casanave. Business-object architectures and standards. In *OOPSLA Workshop on Business Object Design and Implementation. Springer*, October 1995.

[9] E. Clayberg and D. Rubel. *Eclipse Plug-ins*. Addison-Wesley Professional, 2009.

[10] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593 –615, sept.-oct. 2011. ISSN 0098-5589.

[11] K. Croxton, S. Garcia-Dastugue, D. Lambert, and D. Rogers. The supply chain management processes. *The International Journal of Logistics Management*, 12(2):13–36, 2001.

[12] S. Ducasse and O. Nierstrasz. Traits: Composable units of behaviour. In *ECOOP 2003 – Object-Oriented Programming*, pages 327–339, July 2003.

[13] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28:331–388, March 2006. ISSN 0164-0925.

[14] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. *ACM SIGPLAN Notices*, 41(1):270–282, Jan. 2006. ISSN 03621340.

[15] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. *ACM SIGPLAN Notices*, 34(1):94–104, Jan. 1999. ISSN 03621340.

[16] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. ISBN 0201634988.

[17] K. Kang. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer-Verlag, June 1997.

[19] I.-G. Kim, T. Marew, D.-H. Bae, J.-E. Hong, and S.-Y. Min. Dimensions of composition models for supporting software evolution. In W. Löwe and M. Südholt, editors, *Software Composition*, volume 4089, chapter 14, pages 211–226. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-37657-6.

[20] K. Lee, K. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In C. Gacek, editor, *Software Reuse: Methods, Techniques, and Tools*, volume 2319 of *Lecture Notes in Computer Science*, pages 62–77. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-43483-2.

[21] O. L. Madsen and B. Mø ller pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Object-oriented programming systems, languages and applications OOPSLA '89*, pages 397–406, October 1989. ISBN 0897913337.

[22] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*. Addison-Wesley Professional, 2002. ISBN 0201745720.

[23] M. Zenger. *Programming Language Abstractions for Extensible Software Components*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2004.

[6] `http://mef.codeplex.com/`