An Extension Interface Concept for Multilayered Applications

Dem Fachbereich Informatik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte

Dissertation

von

Mohamed Abdulazim Mohamed Aly, MSc.
geboren in Abu Dhabi, Vereinigte Arabische Emirate

Referent:	Prof. Dr.-Ing. Mira Mezini, Teschnische Universität Darmstadt
Korreferent:	Prof. Dr.-Ing. Mario Südholt, École des Mines de Nantes
Tag der Einreichung:	08.09.2014
Tag der mündlichen Prüfung:	10.11.2014

Erscheinungsjahr 2014
Darmstadt D17

To Soha and the family. . .

Affirmation

Ehrenwörtliche Erklärung

I hereby declare that I have written the following dissertation without the inadmissible assistance of third parties and using only the indicated sources and aids. All instances in which outside sources were used have been marked accordingly. This work has not been presented to any examination authority in its current or in a similar form.

Hiermit versichere ich, die vorliegende Doktorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, September 2014 Mohamed Abdulazim Mohamed Aly

Abstract

Extensibility is an important feature of modern software applications. In the context of business applications it is one of the major selection criteria from the customer perspective. Software extensions enable developers to integrate new features to a software system for supporting new requirements. However, there are many open challenges concerning the software provider and the extension developer.

A software provider must provide extension interfaces that define the software artifacts of the base application that are allowed to be extended, where and when the extension code will run, and what resources of the base application an extension is allowed to access. While concepts for such interfaces are still a challenging research topic for "traditional" software constructed using a single programming language, they are completely missing for complex systems consisting of several abstraction layers. In addition, state-of-the-art approaches do not support providing different extension interfaces for different stakeholders.

To develop an extension for a certain software system, the extension developer has to understand what extension possibilities exist, which software artifacts provide these possibilities, the constraints and dependencies between the extensible software artifacts, and how to correctly implement an extension. For example, a simple user interface extension in a business application can require a developer to consider extensible artifacts from underlying business processes, database tables, and business objects. In commercial applications, extension developers can depend on classical means like application programming interfaces, frameworks, documentation, tutorials, and example code provided by the software provider to understand the extension possibilities and how to successfully implement, deploy, and run an extension.

For complex multilayered applications, relying on such classical means can be very hard and time-consuming for the extension developers. In integrated development environments, various program comprehension tools and approaches have helped developers in carrying out development tasks. However, most of the tools focus on the code level, lack the support for multilayered applications, and do not particularly focus on extensibility.

In this dissertation I aim to provide better means for defining, implementing, and consuming extension interfaces for multilayered applications. I claim that explicit extension interfaces are

required for multilayered applications and they are needed for simplifying the implementation (i.e., the concrete realization) and maintainability of extension interfaces on the side of the software provider as well as the consumption of these interfaces by the extension developers.
To support this thesis, I first analyze problems with extension interfaces from the perspectives of both the software provider through an example business application and an analysis of a corpus of software systems. I then analyze the problems with the consumption of extension interfaces (i.e., extension development) through a user study involving extension developers performing extension development tasks for a complex business application. Next, I present XPoints, an approach and a language for the specification of extension possibilities for multilayered applications. I develop an instantiation of XPoints evaluate it against current state-of-the-art works and its usability through a user study. I finally show how XPoints can be applied to simplify the extension development through the implementation of a recommender system for extension possibilities for multilayered applications. The advantages of the recommender system are illustrated through an example as well through a comparison between the current state-of-the-art tools for program comprehension. Topics like extension validation, monitoring, and conflict detection are left for future work.

Zusammenfassung

Erweiterbarkeit ist eine wichtige Eigenschaft von modernen Softwareanwendungen. Aus der Perspektive der Kunden ist Erweiterbarkeit ein Hauptentscheidungskriterium zur Auswahl von Geschäftsanwendungen. Mithilfe von Erweiterungen können Entwickler neue Anforderungen an ein Softwaresystem unterstützen. Dennoch gibt es für den Softwareanbieter und den Erweiterungsentwickler viele offene Herausforderungen.

Softwareanbieter müssen Erweiterungsschnittstellen zur Verfügung stellen: Die erweiterbaren Softwareartefakte, sowie die Ausführungszeiten und die Ausführungspunkte der Erweiterung, und die verfügbaren Softwareressourcen für die Erweiterung. Während die Konzepte für solche Erweiterungsschnittstellen im Umfeld von „traditionellen“ mit einer einzelnen Programmiersprache entwickelten Anwendungen noch ein anspruchsvolles Forschungsthema sind, fehlen vergleichbare Konzepte für mehrschichtige, mehrsprachige Softwaresysteme.

Um eine Erweiterung für ein bestimmtes Softwaresystem zu entwickeln, muss der Erweiterungsentwickler die angebotene Erweiterungsmöglichkeiten verstehen, die dazugehörenden Softwareartefakte finden, die Abhängigkeiten und Randbedienungen zwischen den Erweiterungsartefakten identifizieren und die richtige Entwicklungsmethode verstehen. Zum Beispiel kann eine einfache Erweiterung der Benutzeroberfläche einer Geschäftsanwendung eine Erweiterung der unterliegenden Geschäftsprozesse, Datenbanktabellen und Businessobjekte erfordern. In kommerziellen Anwendungen benutzen Erweiterungsentwickler die von den Softwareanbietern angebotenen klassischen Mittel wie APIs, Frameworks, Dokumentationen, Anleitungen und Beispielcode, um Erweiterungsmöglichkeiten zu verstehen und Erweiterungen erfolgreich zu entwickeln, auszuführen und einzusetzen. Die Nutzung von diesen klassischen Mitteln zur Entwicklung von Erweiterungen für komplexe Anwendungen kann schwer und zeitaufwändig für Erweiterungsentwickler sein. Obwohl in modernen Entwicklungsumgebungen viele Werkzeuge und Ansätze zum Programverständnis den Entwickler unterstützen, sind die meisten dieser Werkzeuge und Methoden auf die Code-Ebene beschränkt. Außerdem, fehlt eine geeignete Unterstützung von mehrschichtigen Anwendungen und der Fokus auf Erweiterbarkeit.

Ziel dieser Dissertation ist es eine bessere Methode zur Definition, Entwicklung und Nutzung von Erweiterungsschnittstellen in mehrschichtigen Anwendungen zu entwickeln. Diese Arbeit

zeigt, dass explizite Erweiterungsschnittstellen für Softwareanbieter und Erweiterungsentwickler benötigt werden. Durch die Nutzung expliziter Erweiterungsschnittstellen kann die Entwicklung (d.h., die konkrete Implementierung) auf der Seite des Softwareanbieters vereinfacht und beschleunigt werden. Zudem kann der Wartungsaufwand reduziert werden. Mithilfe dieser expliziten Erweiterungsschnittstellen kann, auch auf der Seite der Erweiterungsentwickler, der Entwicklungsprozess einer Erweiterung vereinfacht und beschleunigt werden.
Um dies zu zeigen, werden die Probleme bei der Realisierung von Erweiterbarkeit sowie Schwachstellen von Werkzeugen zum Programmverständnis analysiert. Zuerst werden die Probleme anhand einer beispielhaften Geschäftsanwendung und einer Studie einer Reihe von Softwaresystemen gezeigt. Im Folgenden werden die Probleme bei der Erweiterung komplexer Geschäftsanwendungen anhand einer Nutzerstudie mit mehreren Erweiterungsentwicklern analysiert. Darauf aufbauend wird XPoints, ein Konzept und eine Sprache zur Definition von expliziten Erweiterungsschnittstellen, beschrieben. XPoints wird durch einen Vergleich mit heutigen Ansätzen evaluiert. Anhand einer Benutzerstudie werden die Vorteile von XPoints gezeigt. Auf Basis von XPoints, wird ein Recommender-System entwickelt, das Entwickler bei der Entwicklung von Erweiterungen unterstützt. Die Vorteile dieses Recommender-Systems werden anhand eines Beispiels und Vergleichs mit heutigen Ansätzen zum Programmverständnis aufgezeigt. Weitere Forschungsbereiche wie die Validierung von Erweiterungen, Monitoring sowie die Erkennung von Konflikten bleiben offene Punkte für zukünftige Forschungsarbeiten.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Software systems designed and built for specific purposes are often required to accommodate new functionality to enhance, compliment, or change existing features. This trend in software flexibility is becoming a necessary feature of modern software as it becomes more oriented towards end-user customizations and requirements. Artifacts often referred to as plug-ins, add-ons, apps, and extensions are emerging as popular means for extending the functionality of a software system.

An example of software systems created for a large scale and a wide range of customers are business software systems which typically support a set of standard business processes (e.g., sales order processing, recruitment, etc.). Since business requirements can vary from one organization to the other, after an organization acquires a system, customizations and / or extensions are required to match the specific business requirements of the organization. To achieve that, the software provider has to design the software system to support *variability* and *extensibility*. In the context of this dissertation the focus is on extensibility. As a working definition, extensibility is defined as the addition of new functionality to a software system to support new requirements.

1.1 Motivation

Designing for extensibility is a challenging task [Parnas, 1978]. In the world of proprietary commercial business software systems, most software providers do not offer the source code of their software systems to extension developers. However, the software providers offer the extension developers artifacts like, e.g., API libraries, frameworks, etc. along with documentation, tutorials, and other materials to help an extension developer understand the existing extension possibilities and how to develop and integrate extensions with the core software system.

Extensions are likely to interact with the core software (e.g., access internal data resources) and can also affect its main execution stream. In the case of business software systems, especially those that implement legal regulations (e.g., tax calculations), extensibility has to be rigorously controlled. Controlling extensibility is required, e.g., to prevent undesirable system behavior, data inconsistencies, and restrict access to sensitive system information [Krishnamurthi and Felleisen, 1998].

Such software systems can consist of several logical layers (e.g., user interface, business process, business object, database etc.) [Fowler, 2002] which contain many artifacts that can be made extensible for the extension developer. The realization of these artifacts and their execution logic can be made through, e.g., an object-oriented language (e.g., classes and methods). Figure 1.1 shows an example of these logical layers within a module of a business application supporting a sales order creation process. The business process layer depicts the business process that is supported by the software system, the graphical user interface layer contains the interaction elements that will be used, and business objects hold the data and business logic needed for the process execution.

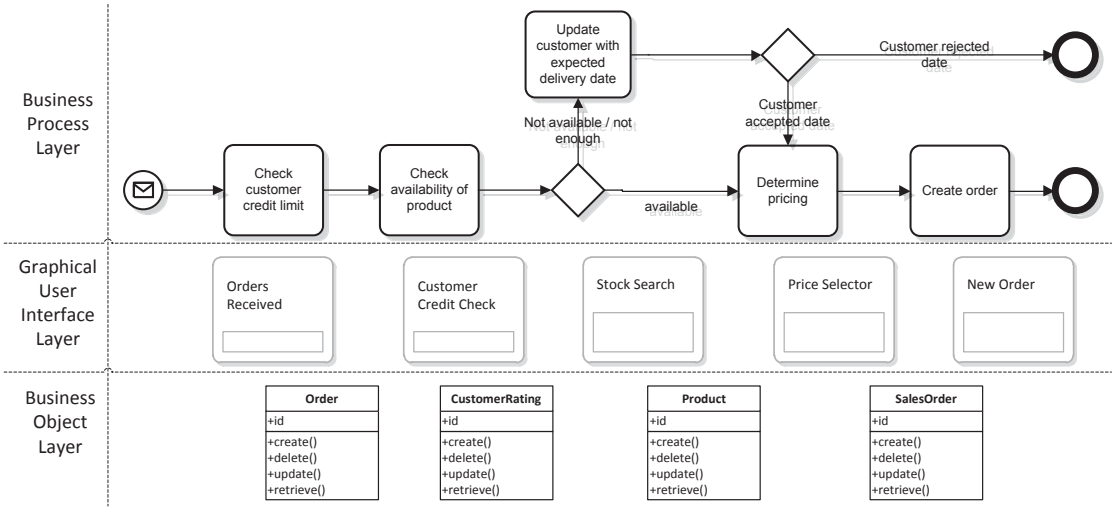


Figure 1.1: Example layers within a business application

In the case of controlled extensibility the software provider has to decide which artifacts within each layer can be extended by the extension developer. Within each layer, the software provider can offer one or more extensible artifacts (e.g., a user interface form, a business process activity, etc.). Following these observations, an extension can cut in general across the application logic along two dimensions as depicted in Figure 1.2. The vertical axis stands for the different layers. Given a layer, the horizontal axis stands for different artifacts of this layer. An extension may cut across different artifacts of the same layer in the sense that, e.g., extending a class `A` may

also require that class `B`, referred to by `A`, is also extended and that the extension of `A` uses the respective extension of `B` (“horizontally co-variant extensions"). To define the extensibility supported by a multilayered application in this case, a software provider has to define and implement an **extension interface** that exposes the extension possibilities as well as the imposed implementation constraints to the extension developer.

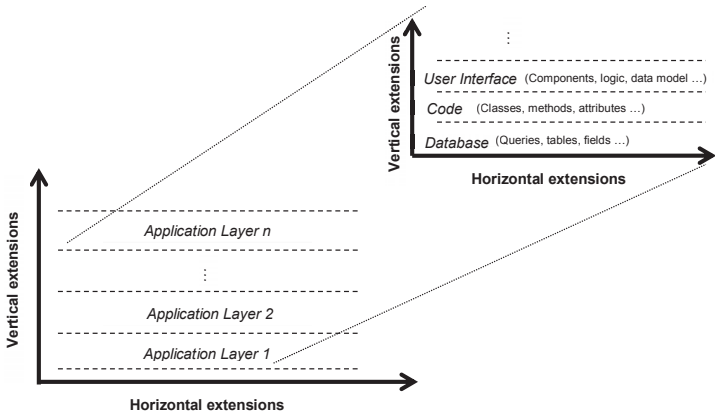


Figure 1.2: Horizontal and vertical Extensions

1.2 Extension Interfaces and Multilayered Applications

Designing and implementing extension interfaces for multilayered applications requiring a controlled form of extensibility is a challenging task. There are several requirements that must be supported by an extension interface.

First, for each extensible artifact within each logical layer, the software provider has to define the types of artifacts (i.e., the extension units) which are accepted as extensions. For example, on the user interface layer, a form made as extensible can accept the addition of new buttons, text labels, and input fields.

Second, several extensible artifacts from one or more layers can be extended by an extension. For example, an extension developer adding a new input field to an existing user interface will most likely extend the corresponding persistency logic and database tables to store the data of the new input field. Moreover, interdependencies between extension possibilities of extensible artifacts from different logical layers can also be imposed as a development constraint by the software provider (i.e., a valid extension is required to implement an extension of several extensible artifacts from different logical layers).

Third, extensions are likely to interact with or use the resources of the software system that also

have to be controlled. Restricted resources must not be made available for extension developers and access rights to the permissible resources must be explicitly defined. For example, an extension can be allowed to call specific methods or access particular variables in read-only mode.

Last, in a controlled extensibility setup, the software provider expects the extensions to be implemented and package them in a particular way. For example, a software provider might expect the software extender to extend particular classes or implement specific interfaces that represent particular extension possibilities. Moreover, the software provider must make explicit when and where will the extension code be executed.

In the previous discussion, the need for extension interfaces for multilayered applications that require controlled extensibility was motivated. In more complex scenarios, the software systems can be potentially extended by several kinds of extension developers which require the software provider to account for. For example, internal developers working on the implementation of new features of the software system are required to have more extension possibilities and access privileges to the internal resources of the software systems than external developers that are required to extend the software for customizing the solution for a particular user. As a consequence, the software provider has to support multiple extension interfaces for the software system to support the different extension developer groups. To summarize, an extension interface for a multilayered software system must define the following:

- *Extension possibilities*: declares the artifacts that are allowed to be extended (e.g., user interface forms, business process activities, database tables, etc.).
- *Extension types*: define the types of extensions that are allowed to be added to these artifacts (e.g., new methods, attributes, user interface elements, process artifacts, new columns in a database table, etc.).
- *Interdependencies*: governs the relationships and constraints that exist between these extensible artifacts.
- *Extension access control*: declares the underlying application resources are available for the extension code (e.g., variables, methods, etc.) as well as their access rights and usage rules.
- *Extension method*: defines how to extend these artifacts (e.g., inheritance, plug-ins etc.).
- *Extension integration and execution*: declares when and where the extension code will be run.

1.3 The Problem in a Nutshell

When defining extension interfaces for multilayered software systems, there are two sides that have to be considered; the side of the software developer that involves the specification and implementation of extension interfaces, and the side of the extension developer that involves the consumption (i.e., through the development of extensions) of these extension interfaces.

Turning to object-oriented languages (e.g., Java), there are two kinds of mechanisms that are related to the implementation of extension interfaces: those geared towards enabling extensibility (e.g., inheritance and overriding), and those geared towards controlling extensibility, e.g., modifiers that enable the developer of a class to control what methods can be overridden or attributes that can be accessed (cf. [Micallef, 1988]). In addition to these mechanisms, a software provider can use advanced means (e.g., design patterns, aspect-oriented programming, plug-in architectures, etc.) to implement the required extension interface for a software system.

In this dissertation I argue that the state-of-the-art approaches have several limitations for realizing the extension interfaces of complex multilayered applications. This is due to the following reasons:

- The technical realization of the extension interface is coupled with the functional code of the core software.
- The conventional means for controlling extensibility, e.g., via Java modifiers, are not expressive enough to enable fine-grained control on what artifacts can be extended and how they are meant to be extended.
- It is not possible to provide different extension interfaces to different kinds of extension developers. Current state-of-the-art techniques provide a one-size-fits-all extension interface and do not handle the different extensibility constraints for different kinds of extension developers.
- Software applications are nowadays extremely complex and involve several architectural layers, demanding extension interfaces that cut across these layers. There is also no support for such layer-crossing extensions. Most approaches focus on language or layer-specific extensibility mechanisms and thus do not support the needs of multilayered applications.
- A developer has to be experienced with advanced development techniques to generate an extension interface of complex software with many extensibility constraints.

The need and the challenges related to providing well-defined extension interfaces for object-oriented systems are documented in the literature [Kiczales and Lamping, 1992, Steyaert et al., 1996, Mezini, 1997, Kiczales and Mezini, 2005]. As a variation on this theme, several proposals for aspect-based extension interfaces have been published recently [Aldrich, 2005, Steimann et al., 2010, Sullivan et al., 2010, Inostroza et al., 2011]. However, as it will be elaborated in a discussion on the state of the art, these approaches do not address the limitations mentioned above.

Extension developers currently rely on different artifacts and methods to understand how to build extensions. Artifacts like APIs, documentation, code examples, and program comprehension tools [Storey et al., 1997] can help an extension developer build an extension. In more complex contexts, developers can further seek assistance by attending special training sessions or tutorials. These means are not always feasible or easy to be learned and used by the developers (cf. [Robillard, 2009] and [Hou and Li, 2011]). The more complex a software system is, the more resources (e.g., time and money) needed to successfully develop an extension are required. In this dissertation I also argue that the current state-of-the-art means that are used for assisting extension developers with developing extensions are not very effective for multilayered applications.

1.4 Contributions

In this dissertation I claim that:

An expressive language is required for simplifying the specification, implementation and consumption of extension interfaces of multilayered applications.

The work in this dissertation supports this thesis by an analysis of the state of the art and the design, implementation, and evaluation of such a language. More specifically this dissertation contributes the following:

- *Definition of Requirements for Extension Interfaces of Multilayered Applications*
  The requirements of extension interfaces supporting multilayered applications are defined through the following. First, interviews that have been conducted on a development team responsible for implementing the necessary support for extensibility at a leading software company that provides business software systems. Second, an example application that is defined to illustrate the limitations of current state-of-the-art object-oriented mechanisms

in supporting the realization of extension interfaces of multilayered applications. Third, a user study that is conducted on extension interfaces of open-source Java-based software systems within the Qualitas corpus [Tempero et al., 2010].

- *Definition of Requirements for Consumption of Extension Interfaces of Multilayered Applications*
  The requirements for supporting extension developers with the implementation of extensions are defined through a study in which the resources and methods that extension developers currently use for accomplishing extension development tasks are investigated and evaluated. The problems and challenges that face extension developers are outlined and requirements that program comprehension tools must support to aid extension developers are defined.

- *A Concept for Extension Interfaces for Multilayered Applications*
  The novel concept XPoints consists of an approach and a language that enables the explicit and declarative expression and control of extensibility by well-defined extension interfaces in multilayered applications, including cross-layer dependencies. XPoints introduces an additional abstraction layer, which separates the declaration of extension interfaces from their realization (e.g., using design patterns or plug-ins). By decoupling the extension interface from the application, XPoints enables different extension interfaces for different groups of extension developers. Moreover, a developer can realize the extensibility interface of a software system by automatically generating the extensibility supporting code from an XPoints interface.

  - *Instantiation of the Concept*
    An implementation of XPoints is reported on in the context of business applications consisting of three logical layers: business object, user interface, and business process. Furthermore, the implementation is used to demonstrate the definition and implementation of extension interfaces for multilayered applications. A more generic Java-based implementation is also reported on and used for the evaluation. The corresponding implementations and toolsets in Eclipse [Eclipse Foundation, 2014a] are described and reported on.
  - *A Tool for Recommending Extension Possibilities*
    Using XPoints as a foundation, a tool for recommending extension possibilities and guiding extension development is proposed and its corresponding implementation in Eclipse is reported on.

- *Evaluation of the Concept*
  The evaluation of the concept is presented through the comparison of XPoints with the related works. A user study on the usability of the concept is also reported on. The tool for recommending extension possibilities is evaluated along with state-of-the-art program comprehension with respect to the fulfillment of the defined requirements. A discussion on the advantages and limitations of the approach is also presented.

The following papers were published within the context of this dissertation [1]:

- Aly, M., Charfi, A., Erdweg, S., and Mezini, M. (2013a). XPoints: Extension interfaces for multilayered applications. In *Proceedings of the 2013 17th IEEE International Enterprise Distributed Object Computing Conference*, EDOC '13, pages 237–246, Washington, DC, USA. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/EDOC.2013.34
- Aly, M., Charfi, A., and Mezini, M. (2013b). Building extensions for applications: Towards the understanding of extension possibilities. In *Proceedings of the 2013 IEEE 21st International Conference on Program Comprehension*, ICPC '13, pages 182–191, Washington, DC, USA. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/ICPC.2013.6613846
- Aly, M., Charfi, A., Wu, D., and Mezini, M. (2013c). Understanding multilayered applications for building extensions. In *Proceedings of the 1st workshop on Comprehension of complex systems*, CoCoS '13, pages 1–6, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2451592.2451594
- Aly, M., Charfi, A., and Mezini, M. (2012). On the extensibility requirements of business applications. In *Proceedings of the 2012 workshop on Next Generation Modularity Approaches for Requirements and Architecture*, NEMARA'12, pages 1–6, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2162004.2162006

1.5 Organization of the Dissertation

This dissertation is comprised of 7 chapters. Chapter 1, **Introduction**, presents the research topic and the motivation of work as well as an overview of the contributions of this dissertation. Chapter 2, **Extensibility and the Software Provider** presents the problems on the side of the software provider, limitations of object-oriented languages for the specification of extension

[1] Some content and text from the listed publications have been reused in this dissertation.

interfaces, a study on the limitations of extension interfaces in Java-based software systems, and the requirements for extension interfaces in multilayered applications. Chapter 3, **Extensibility and the Extension Developer** presents the problems on the side of the extension developer, a user study on the current artifacts that extension developers depend on to realize extensions, and requirements for a program comprehension tool that aids extension developers to perform extension development tasks. Chapter 4, **State of the Art**, presents state-of-the-art approaches on extension interfaces and program comprehension tools. Chapter 5, **XPoints Extension Interface Concept and Implementation**, the proposed concept, XPoints, is presented and the implementation details are reported on. Based on the presented concept, a tool for recommending extension possibilities and guiding extension development is proposed. Chapter 6, **Evaluation of the Approach**, presents the evaluation of XPoints and the recommender tool. Chapter 7, **Conclusion**, presents a summary of the dissertation and sketches future directions of work.

2 Extensibility and the Software Provider

At the early stages of the work presented in this dissertation, several interviews were conducted with a development team that is responsible for designing and implementing the extension interfaces for a business software system at a leading software company. The software system supported critical business processes and extensibility is required to be controlled in a rigorous way. The main goal of these interviews is to understand the limitations on the side of the software provider and identify the strengths and weaknesses of state-of-the-art methods for realizing extension interfaces of multilayered applications.

In the first part of this chapter, an exemplary Java-based multilayered business application is presented through which the identified problems of the current state-of-the-art methods are discussed. The presented exemplary application is used to illustrate the limitations of the state-of-the-art approaches that were identified during the conducted interviews. In the second part of this chapter, a study on several open source Java-based software systems that are part of the Qualitas corpus is presented. Based on the outcome of the study, the discussion in the first part is further extended by emphasizing on the magnitude of the problem with extension interfaces of modern software systems.

2.1 An Example Business Application

In the following a business application spanning three logical layers is presented: the business process layer, the business object layer, and the user interface layer. A business process defines the flow of activities that are required to achieve a specific business objective such as creating a sales order or hiring a new employee. Business objects [Sutherland, 1995, Casanave, 1997] represent entities that are meaningful within a specific business process like sales order, invoice,

customer, and employee. A business object encapsulates attributes, behavior, constraints, and relationships to other business objects. User interfaces provide means to support the end users to accomplish the different activities within a business process.

The following simple sales quotation management module is introduced as an example of a multilayered business application that spans the three layers mentioned above. Figure 2.1 shows the sales quotation business process (layer 1) defined using the Business Process Modeling Notation (BPMN) [(OMG), 2011]. The process starts upon receiving a request of a customer for a quotation for a specific set of products. A sales representative analyzes the request and creates a sales quotation and fills in the necessary data. Then, the sales representative sends the quotation for approval to her manager. The manager can either approve the quotation or request a revision. Based on that decision, the sales representative may have to edit the quotation and resubmit it for approval. At the end, the approved sales quotation is sent to the inquiring customer.

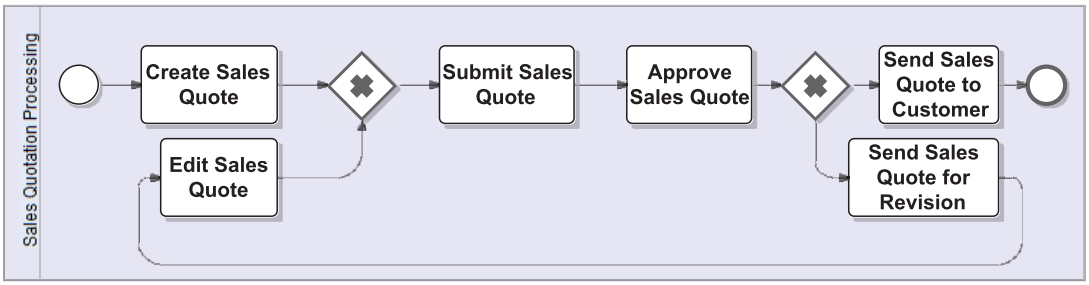


Figure 2.1: Sales quotation business process - ©2013 IEEE

```
1  class SalesQuoteForm extends JPanel {
2   ...
3  private CustomerInfo customerInfo;
4  private double discount;
5  private SalesQuote salesQuote;
6   ...
7  public SalesQuoteForm() {...}
8   ...
9  private void initializeForm () {...}
10 private void onSendToApprovalButtonClick() {...}
11 private void savetoSalesQuoteBusinessObject () { ... }
12  ...
13 }
```

Listing 2.1: Source code of the sales quotation form

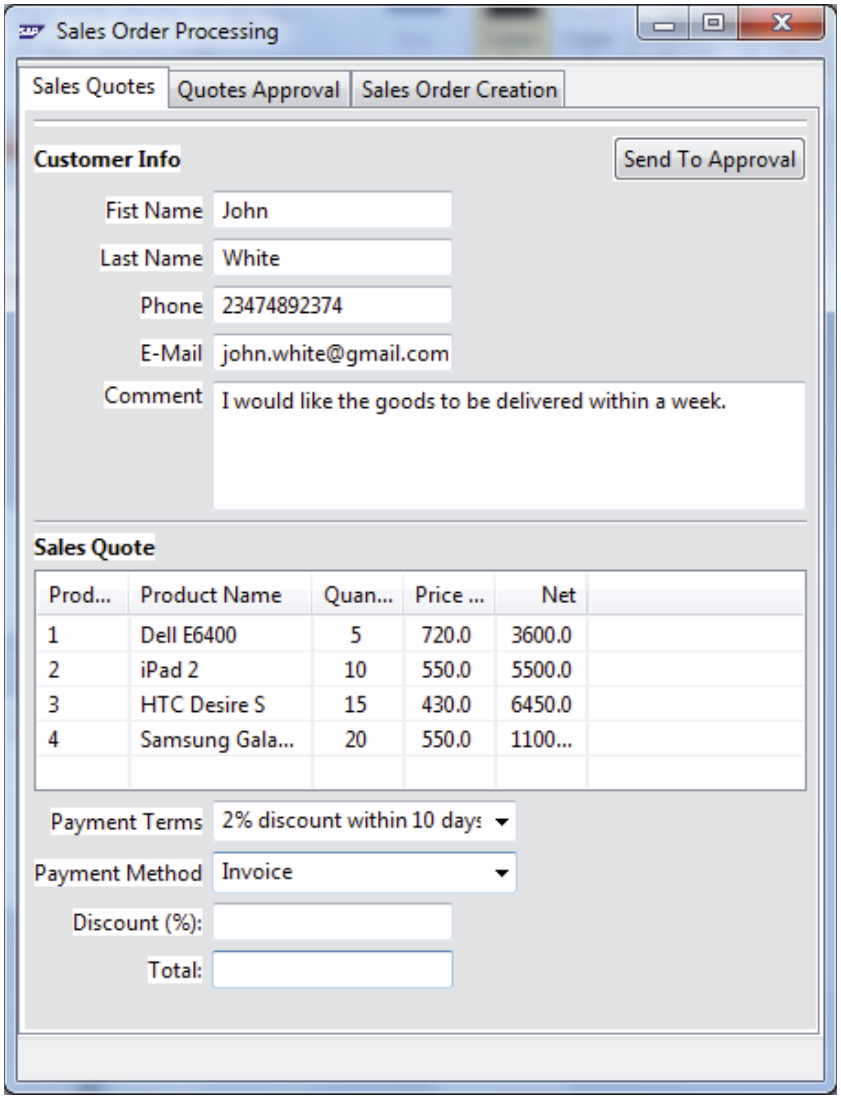


Figure 2.2: User interface for sales quotation creation - ©2013 IEEE

Figure 2.2 shows the user interface (layer 2) associated with the sales quotation creation activity. An excerpt of the source code associated with this user interface is shown in Listing 2.1. Using this user interface, a sales representative can enter the customer information, define the sales quotation, and specify the payment details. An excerpt of the source code of the sales quotation business object (layer 3), which holds the data and business logic of the sales quotation, is shown in Listing 2.2. The sales quotation module involves other user interfaces and business objects, as well as classes that support the execution of the business process, which are not shown for brevity.

```
1  class SalesQuote {
2   protected CustomerInfo customerInfo;
3   private List<ProductQuote> products;
4   protected String comment;
5   private double total ;
6   protected double discount ;
7   private double tax ;
8    ...
9   public final SalesQuote readSalesQuote (...){...}
10  public final SalesQuote createSalesQuote (...){...}
11  private void saveSalesQuote (){...}
12  protected double calculateTotal (){...}
13  protected void calculateDiscount (double discount ){...}
14  protected void sendToApproval (){...}
15   ...
16 }
```

Listing 2.2: Source code of the sales quotation business object

2.2 Problem Definition

Let us first consider the business object layer. Each class in Java has two perspectives: a *usage perspective* and an *extension perspective*. The usage perspective allows a client of the class (e.g., by instantiation or subclassing) to call all methods and access all attributes that are not declared as private. The extension perspective allows an extension class to potentially affect non-final accessible classes by overriding (non-private, non-final, and non-static) methods and through the addition of new attributes and methods. In the following, a discussion of several limitations of the usage and extension perspectives in Java to express complex extension interfaces for software systems is presented.

The first problem is the lack of means to express and constrain the extension types. For example, it is not possible to express that an extension developer is allowed to add new methods to the class *SalesQuote* but not allowed to add any new attributes (e.g., to prevent them from being persisted in the database behind the business object). Further, it is not possible to express that an extension developer is allowed to add custom business logic only if the original method is called by the overriding one. By allowing the extension developer to override a method arbitrarily, this property cannot be guaranteed. While this second example can be realized with other techniques (e.g.,

using the template method design pattern) I argue that it is necessary to have declarative means for the specification of extension possibilities. Such declarative specification is beneficial for both the software provider and the extension developer: the provider can express extension possibilities in a declarative way without thinking about how to enforce them, whereas the extension developer can implement extensions against the extension interface without going through all methods and classes of the software system.

The second problem is the limitation of the usage interface to express fine-grained overriding and access rights to the methods and attributes of the extended class. For example, the modifier *protected* of the attribute *discount* gives the extension developer full access (i.e., read and write) to that attribute. To give the extension developer read-only access to that attribute one could declare it as final and protected. However, in that case the class *SalesQuote* will not be allowed to modify the discount value anymore. Without using, e.g., a protected getter and a private setter method, there is no possibility to restrict the access right of extension developers to the attributes of the parent class. Moreover, by using getters and setters, the extension possibilities are not expressed declaratively and the focus is again shifted from what extension possibilities are available to how these possibilities are enforced.

The third limitation is that Java does not allow the expression of interdependencies between extension possibilities. For example, in order to preserve the correct application logic, the extension developer overriding the method *calculateDiscount()* must also override the method *calculateTotal()*. One solution for this limitation is to use an interface and require the extension developer to implement both methods. However, the realization of this solution to express these interdependencies is complex.

The fourth limitation is that Java provides a one-size-fits-all extension interface. It is not possible to have different extension interfaces for different groups of extension developers, which is often required. For instance one extension developer group (e.g., external developers) can be restricted to only perform a validation of the sales quotation by providing them with read-only access to attributes as well as the possibility to add some custom business logic before the method *saveSalesQuote()*.

Another group of extension developers (e.g., extenders from partner companies) can be allowed to perform validations and, in addition, update selected attributes of the *SalesQuote* class. This second group will have write access to some attributes of the *SalesQuote* in addition to the extension possibilities given to the first group. A third group (e.g., extenders at the software provider side who are building an industry-specific solution on top of the standard application)

can be allowed to realize advanced extensions that go beyond simple validations such as extending the quotation process to include a second approval step, e.g., for sales quotations that exceed a predefined amount.

There is no simple solution for this limitation. One solution could be to provide a variation of the proxy pattern, in which different proxy classes are offered for each extension developer group. The proxy provides access only to the methods and attributes that are part of the extension interface. However, such a realization is complex. For example, one could just consider the work required to provide three proxy classes for the three extension developer groups mentioned above. The greater the number of extension possibilities and constraints are, the more effort and time for implementing the extension interface will be required. Furthermore, additional accidental complexity with negative effects on comprehensibility and maintainability will result.

Using the solutions suggested above have a lot of disadvantages for both software provider and extension developer since the extension interface is realized *implicitly* rather than *explicitly*. On one hand, the software provider must encode the extension interface by complex application code (e.g., using design patterns). The more complex the system and the extensibility constraints are, the more difficult the realization of extension interfaces will be. Moreover, the extensibility decisions and intents taken by the software provider are lost. When the complexity of an application increases, more code is required for realizing an extension interface, which leads to maintainability problems. It will be very difficult for the software provider (without, e.g., comprehensive documentation) to find out the exact methods, classes, and interfaces that comprise the extension interface.

On the other hand, an extension developer will have a hard time identifying the extension possibilities and constraints as they are not expressed explicitly. Instead the extension developer will have to read documentation and tutorials and to understand the provided APIs to assess the feasibility of, e.g., some extension scenario. This gets even more difficult as the functional API of the class (i.e., the usage perspective) and its extensibility API (i.e., the extension perspective) are mixed.

In the discussion above, the focus was on the business object layer. The extensibility problems discussed also arise on the other layers. An extension can typically span several layers, which makes it important to support extensibility on all and across layers. For example, a software provider can make a certain database table extensible by allowing the addition of new columns. The software provider can make a certain user interface form extensible by allowing extension developers to embed their custom user interface elements at a predefined location. A business

process model can also be made extensible by allowing the extension developer to add custom activities. I argue that the extension possibilities have to be expressed directly on the different layers of the application. Most state-of-the-art approaches express these possibilities in the implementation (i.e., on the code layer). As a result, an extension developer cannot assess the feasibility of some user interface form extension or some business process extension without diving deeply into the implementation as well as the provided APIs on the code layer. Furthermore, a software provider has to manually encode extensibility in the code.

Furthermore, when supporting extensibility on different layers, it is necessary to capture the dependencies between the extension possibilities available on these layers. For example, if the extension interface of the *SalesQuote* on the user interface layer allows an extension to bring in a new button that triggers a particular function, and a text field to display a new attribute, an extension developer has to also consider the extension possibilities available on the code layer (i.e., the Java class *SalesQuote*) and to add a new attribute to that class and implement the necessary logic. In addition, the extension developer has to consider the extension possibilities available on the database layer and to extend the table that stores the *SalesQuote* data. These inter-layer dependencies impose constraints on the way extension possibilities are expressed and also on the way an extension is developed.

2.3 Extensibility in the Qualitas Corpus

To further emphasize on the magnitude of the problem, a study is conducted on a collection of real-world Java-based software systems within the Qualitas corpus [Tempero et al., 2010]. There are two main goals of the study. The first goal is to estimate and compare the number of classes that are intended for extensibility by the software provider with the number of classes that are potentially extensible for extension developers as well as measure the number of artifacts that can be potentially affected by an extension (i.e., accessible and overridable methods and attributes). The second goal is to analyze how the extension interface is implemented, i.e., what the system expects as an extension unit is identified (i.e., a subclass of particular classes, a plug-in, etc.), what an extension is allowed to contribute (i.e., a generic functionality or domain/application-specific functionality), how are extensions packaged and integrated within the core software system, and what resources are made available for extension developers to help them with the extension development process.

2.3.1 A Study on the Qualitas Corpus

The Qualitas corpus is a collection of open-source Java-based software applications intended for empirical studies. The corpus contains source code of Java-based software systems that cover different application domains like databases, graphics, compiler tools, programming languages, and IDEs. For the purpose of the study, version 20120401r of the corpus which includes the source code releases of 111 software systems is used. The study consists of four phases.

In the first phase, the systems which explicitly document their support for extensibility are separated out. The separation process took place by examining the official documentation, the online websites, and the provided API of the corresponding version for each system in the corpus. The separated systems are the only ones used in the following phases. In the second phase, the number of the potentially extensible and accessible classes, interfaces, methods, and attributes is measured. For this phase the PMD [PMD, 2014] source code analyzer was used and custom XPath rules for finding out the artifacts that are potentially affected by extensions were implemented. In the third phase, the number of classes that are intended by the software provider to be used by the extension developer is estimated. To achieve that, the classes which were explicitly listed in the documentation or API specification as intended for extensibility were manually analyzed. In the last phase, the provided documentation of each software system was analyzed to understand the extension development process as described in the second goal of the study. Two different Java developers were employed for the manual analysis of all of the selected 44 systems and their findings were compared to identify any inconsistencies.

2.3.2 Results

Out of the 111 systems, 44 systems (containing in total 129,827 classes, 1,174,953 methods, and 577,999 attributes) that document extensibility were found. For each of the 44 systems, the findings were documented in a table similar to Table 2.1. In the following the overall findings of the 44 systems are reported and explained. Tables 2.2 and 2.3 show the aggregated results for each system.

Classes The *total number of classes* presents the total number of classes, interfaces, and abstract classes of the software system. The *final classes* present the number of classes that are accessible but not overridable. *Abstract classes, interfaces, and other classes* present the number of abstract classes, interfaces, and classes that can be potentially accessed and overridden by an extension. The *number of classes intended for extensibility* is the number of classes that was

estimated through the manual analysis as being intended for extensibility by analyzing the official documentation resources from the software provider. Out of 129,827 classes a total of 12,526 final classes (i.e., can be accessed but not overridden), 25,599 abstract classes and interfaces (i.e., can be accessed and overridden), and 91,702 other classes that can be potentially accessed and overridden were found. Only 951 classes were identified as intended for extensibility. For some systems it was not possible to identify the classes that were intended for extensibility due to lack of documentation of the classes that are meant for extensibility or that the system did not rely on classes for extensibility.

System	**aoi**
Domain	3D/graphics/media
Total Number of Classes	**562**
- Final Classes	0
- Abstract Classes	32
- Interfaces	32
- Other Classes	498
Total Number of Classes intended for Extensibility	**16**
Total Number of Methods	**6735**
- Private Methods	929
- Accessible Methods	5806
- Final Accessible Methods	515
- Overridable Methods in Classes	4621
- Overridable Methods in Abstract Classes	499
- Methods in Interfaces	171
Total Number of Attributes	**3482**
- Private Attributes	1535
- Accessible Attributes (non-final)	1398
- Accessible Attributes (final)	549
Extension Unit	Plug-ins
Extension Contribution	Renderers, Modelling Tools, Translators, Textures, Texture Mappings, Materials, Material Mappings, Image Filters, Procedural Modules, Generic
Extension Integration	Proprietary XML to point to extension
Code Examples	No
Tutorials	Yes

Table 2.1: Example findings of “Art of Illusion” version 2.8.1

Methods The *total number of methods* presents the number of methods within all classes, interfaces, and abstract classes. The *private methods* present the number of methods that are declared as private (i.e., non-accessible and non-overridable methods). *Accessible methods* present the number of methods that can be potentially called by an extension. *Final accessible methods* present the number of methods that can be potentially called by an extension but not overridden (i.e., the package private, protected, or public methods that are final and not declared as static). The *overridable methods in classes* represent the number of methods that can be accessed and overridden in classes (i.e., the package private, protected, or public methods that are

non-final and non-static). The *overridable methods in abstract classes and in interfaces* present the number of methods that can be accessed and overridden by the extension in abstract classes and interfaces respectively. Out of 1,174,953 methods, 137,859 methods that are non-accessible and not non-overridable (i.e., methods declared as private), 186,330 methods that are accessible but non-overridable (i.e., package private, protected, or public methods that are not declared as final or static), and 850,764 methods that are both accessible and overridable were found.

Attributes *The total number of attributes* present the total number of field attributes within all classes, interfaces, and abstract classes. The *private attributes* present the number of attributes that cannot be accessed by an extension. The *accessible attributes (non-final)* present all class attributes that can be accessed by an extension in read/write mode (i.e., non-final attributes). The *accessible attributes (final)* present all class attributes that can be accessed by an extension in read-only mode. Out of 577,999 attributes, 316,277 attributes that are not accessible, 121,781 attributes that are accessible in read/write mode, and 139,941 that are accessible in read-only mode were found. Table 2.2 shows the aggregated results for the 44 systems.

Extension Units The *extension unit* describes what the software provider expects as an extension. 23 systems require the extension developer to simply extend predefined classes, interfaces, and abstract classes. The compiled classes can then be integrated with the core software system through the means specified by the software provider. 16 systems require the extension developers to deliver their extension in the form of a plug-in. A plug-in consists of one or more classes which are packaged together. The classes can extend a set of predefined classes of the software system and/or provide new classes. 4 systems support extensions as standalone applications. In these systems, the extensions have a separate runtime and can integrate with and use the core system via the usage API. 1 system expects an extension to be developed in another proprietary XML language. The extension written in this language will be interpreted and executed during the runtime of the application.

Extension Contributions The *extension contribution* describes what the software provider expects an extension to do, i.e., what kind of new functionality it can introduce to the software system. Based on the findings, the software system either supports the addition of *domain-specific* or *generic* extensions. Domain-specific extensions provide new features that are relevant to the domain of the software system. Generic extensions can provide new features that can also be non-relevant to the domain of the system. Out of the 44 systems, 24 systems support only domain-specific extensions and 20 systems support generic extensions.

Extension Integration *Extension integration* describes how an extension unit is integrated with the core system. 4 systems require that the extender will simply include the extension binaries within the same class path of the system. The classes will then be loaded and executed during runtime. 21 systems require the definition of a proprietary XML or text file. This file can contain meta-data like paths to the classes of the extension (i.e., for the system to be able to load and execute the extension) and a description of the extension. 14 systems require the extension developer to program the integration with the core system within the source code of the extension. 4 systems use proprietary frameworks for integrating and executing extensions, and 1 system relies on well-defined web service interfaces for integration.

Documentation Besides the official API documentation, software providers usually supply the extension developers with tutorials. These tutorials can contain information like architecture diagrams and extension development instructions. Moreover, the software provider can also provide code examples of extensions. 27 systems provide both code examples and tutorials, 7 systems provide tutorials and no code examples, and 10 systems provide tutorials but no code examples. Moreover, within the 44 systems no technical documentation explaining how extensibility is implemented within the core software system was found (e.g., which classes are meant to support extension development, which methods and attributes are used for extensibility, etc.).

System	Total # Classes	Final Classes	Abstract Classes	Interfaces	Other Classes	Classes Intended for Extensibility	Total # Methods	Non-accessible and Non-overridable Methods	Accessible and Non-overridable Methods	Accessible and Overridable Methods (Classes, Interfaces, Abstract Classes)	Total # Attributes	Non-Accessible Attributes	Accessible Attributes (non-final)	Accessible Attributes (final)
ant	1491	68	75	95	1253	**7**	13277	1344	906	11027	6398	4821	502	1075
aoi	562	0	32	32	498	**16**	6735	929	515	5291	3482	1535	1398	549
argouml	2093	57	142	187	1707	**1**	16549	1840	1587	13122	5603	4109	347	1147
azureus	3711	19	131	1075	2486	**358**	40877	2017	2966	35894	18147	11881	2841	3425
batik	2580	24	216	288	2052	**28**	17707	487	1042	16178	11827	1469	5201	5157
cayenne	3140	178	295	160	2507	**6**	19048	767	1297	16984	6272	1281	3395	1596
checkstyle	998	68	50	70	810	**28**	4218	660	570	2988	1608	1101	261	246
collections	608	56	57	27	468	**59**	6520	344	1171	5005	1263	702	355	206
colt	298	3	42	67	186	**5**	3823	156	788	2879	845	114	394	337
columba	1219	15	36	117	1051	**7**	5837	404	347	5086	3125	1994	737	394
derby	2955	308	191	441	2015	**3**	39620	6315	8678	24627	19034	7615	3793	7626
displaytag	318	17	14	16	271	**4**	1665	95	150	1420	747	557	27	163
eclipse_SDK	24511	1855	1959	3598	17099	**182**	241539	36520	32591	172428	146459	64788	44089	37582
exoportal	2134	8	53	257	1816	-	11405	431	725	10249	4608	3116	654	838
findbugs	1181	17	82	116	966	-	9455	1198	979	7278	5183	3021	1153	1009
fitjava	150	0	1	0	149	**3**	816	52	67	697	426	8	418	0
freemind	579	5	30	73	471	**4**	5685	635	350	4700	2439	1517	630	292
heritrix	609	4	43	51	511	**8**	5216	407	618	4191	3187	1507	542	1138
james	464	17	33	78	336	**3**	3152	319	139	2694	1723	1069	233	421
jasperreports	1815	131	101	292	1291	**18**	17125	794	1298	15033	8033	3492	1942	2599
jboss	7415	196	416	1674	5129	-	50031	2860	3753	43418	17548	10139	4731	2678
jedit	837	3	44	68	722	**14**	7237	1109	1071	5057	4256	2216	1119	921
jena	1051	18	70	190	773	**15**	9451	633	823	7995	3728	654	1333	1741
jext	690	28	13	31	618	**4**	3447	308	695	2444	2199	1420	448	331
jgrapht	273	16	17	33	207	**20**	1283	169	84	1030	735	513	168	54
jgroups	839	5	19	77	738	**8**	9117	992	1147	6978	5043	1933	1874	1236
jhotdraw	658	0	45	60	553	**4**	6977	569	467	5941	2582	1811	366	405
jmeter	953	41	53	80	779	**12**	8132	1248	711	6173	4969	4148	95	726
jre	8776	1310	937	1694	4835	-	88476	9038	16406	63032	45100	18812	8624	17664
jspwiki	541	56	27	36	422	**2**	3972	407	785	2780	2472	1370	338	764
log4j	417	38	15	25	339	**6**	3149	137	705	2307	1538	567	649	322
lucene	2913	518	196	132	2067	**2**	21943	2481	4026	15436	10971	5908	2924	2139
marauroa	237	2	7	14	214	**5**	1584	81	117	1386	665	531	96	38
maven	865	17	24	169	655	**2**	5790	645	209	4936	1872	1442	176	254
nekohtml	66	0	1	5	60	**3**	610	62	58	490	551	93	207	251
netbeans	40376	6742	2355	4417	26862	-	386960	54584	86876	245500	176619	127458	19289	29872
pmd	780	7	38	48	687	**1**	5417	1154	524	3739	2213	1198	247	768
quartz	258	3	9	36	210	**9**	2705	93	106	2506	1095	565	91	439
rssowl	696	71	26	110	489	**21**	7305	1481	780	5044	4054	2275	1390	389
springframework	5035	266	498	638	3633	**5**	36611	1555	4394	30662	10690	8584	904	1202
squirrel_sql	72	1	0	16	55	**13**	652	143	3	506	340	280	42	18
struts	2076	82	81	165	1748	**14**	15096	942	969	13185	6478	3508	2204	766
weka	1469	12	106	147	1204	**51**	18625	834	1884	15907	8725	2851	4740	1134
xalan	1118	244	47	67	760	-	10114	620	2953	6541	13147	2304	814	10029

Table 2.2: Results - Qualitas Corpus - Part I

System	Extension Unit	Extension Contributions	Extension Integration	Code Examples	Tutorials
ant	Subclass	Ant Task	Proprietary XML to point to extension	Yes	Yes
aoi	Plug-in	Renderers, Modelling Tools, Translators, Textures, Texture Mappings, Materials, Material Mappings, Image Filters, Procedural Modules, Generic	Proprietary XML to point to extension	No	Yes
argouml	Subclass	Custom Modules	Update package manifest file	No	Yes
azureus	Plug-in	Generic Plug-in	Proprietary properties file pointing to plug-in	Yes	Yes
batik	Subclass	URL Protocols, Image File Formats, XML Elements, DOM Extension, Bridge Extensions, Script Interpreters	Classpath (Jar in a predefined directory)	No	Yes
cayenne	Subclass	Custom queries	Developer integration from source code	Yes	Yes
checkstyle	Subclass	Modules including new Checks, Filters, Listeners	Proprietary XML File describing extension	Yes	Yes
collections	Subclass	Custom Collection	Developer integration from source code	No	No
colt	Subclass	Custom Histograms	Developer integration from source code	No	No
columba	Plug-in	Generic Plug-in	Proprietary XML	No	No
derby	Subclass	User defined authenticator, User defined types	Developer integration from source code	Yes	Yes
displaytag	Subclass	Custom export view, Custom decorator	Proprietary properties file	No	Yes
eclipse_SDK	Plug-in	Generic Plug-in	OSGi based	Yes	Yes
exoportal	Web application	Web application	Proprietary XML to point to extension	Yes	Yes
findbugs	XML defining new Rules	Custom detector	Proprietary XML declaring extension	Yes	Yes
fitjava	Subclass	Column, Action, and Row Fixtures	Developer integration from source code	Yes	Yes
freemind	Plug-in	Generic Plug-in	Proprietary XML to point to extension	No	No
heritrix	Subclass	Modules	Proprietary text file declaring a module	Yes	Yes
james	Subclass	Custom Matcher and Mailet	Classpath	No	Yes
jasperreports	Subclass + XML Extension + Font extension	Various	Proprietary Framework	No	Yes
jboss	App with web service interface	Web Services	Based on Web Services	No	Yes
jedit	Plug-in	Generic Plug-in	Proprietary XML + Proprietary properties file	Yes	Yes
jena	Subclass + Custom language extension	Various	Classpath	Yes	Yes
jext	Plug-in	Generic Plug-in	Proprietary XML	No	Yes
jgrapht	Subclass	Custom Graph, Nodes, Edges	Developer integration from source code	No	No
jgroups	Subclass	Custom Protocols, Logging, Addresses, Request Handler	Developer integration from source code	Yes	Yes
jhotdraw	Subclass	GUI Extensions	Developer integration from source code	No	Yes
jmeter	Plug-in	Jmeter GUI, Visualizer, Test Elements, Graphs	Developer integration from source code	Yes	Yes
jre	Full app packaged as jar	Java applications	Classpath	Yes	Yes
jspwiki	Plug-in	Generic Plug-in	Proprietary text file to point to plug-in	Yes	Yes
log4j	Subclass	Conversion Characters	Developer integration from source code	Yes	Yes
lucene	Subclass	Custom Query Parser	Developer integration from source code	No	No
marauroa	Subclass	Custom Game	Developer integration from source code	Yes	Yes
maven	Plug-in	Maven Mojo	Proprietary XML	Yes	Yes
nekohtml	Subclass	Custom Filters and Parsers	Developer integration from source code	Yes	Yes
netbeans	Plug-in	Modules	Netbeans Proprietary	Yes	Yes
pmd	Subclass	Custom Rules	Proprietary XML	Yes	Yes
quartz	Plug-in	Generic and / or Custom Job Listener, Trigger Listener, Scheduler Listener	Developer integration from source code	No	No
rssowl	Plug-in	Generic Plug-in	Eclipse based	No	Yes
springframework	Subclass	Spring IoC Container Extension	Proprietary XML	Yes	Yes
squirrel_sql	Plug-in	Generic Plug-in	Proprietary XML	Yes	Yes
struts	Plug-in	Generic Plug-in	Proprietary XML	Yes	Yes
weka	Plug-in	Plug-ins provide Classifiers, Filters, Algorithms, GUI extensions	Proprietary text file to describe plug-in	Yes	Yes
xalan	Separate classes	Generic	Proprietary XML	Yes	Yes

Table 2.3: Results - Qualitas Corpus - Part II

2.3.3 Problems

Based on the findings of the study the following problems are outlined.

The first problem is that in all 44 software systems the intended extension interface (i.e., what the software provider really means to offer as extensible) is much smaller than the potential extension interface (i.e., what can be technically extended). On average, less than 1% of the classes were meant for extensibility whereas 90.4% are potentially overridable and accessible by extensions. The first, third, and sixth problems presented in Section 2.2 show that Java lacks the appropriate means for explicitly expressing extension interfaces and defining fine-grained access rights. Moreover, enforcing an extension interface will cause the code intended to support extensibility to mix with the functional code of the software system. Due to these problems, it was very hard in the study to estimate the number of methods and attributes that are intended to support extension development. To count the number of methods and attributes, a manual analysis of the source code of the software systems will be required.

The second problem is the limitation of the Java language constructs for expressing the extension interface. This is also analogous with the second problem presented in the previous discussion in Section 2.2. The software providers rely on class and method names along with documentation for expressing the extension possibilities rather than on domain-specific constructs. In the results of the study, 24 systems that support domain-specific extensions were identified. However, these systems did not share a standard way for expressing extension interfaces. Without explicit domain-specific extension interfaces or detailed documentation about how to realize such extensions, the extension developer might not be able to implement an extension as expected by the software provider has foreseen it.

Moreover, there is no standard way for documenting extensibility for the extension developer. Without the presence of detailed documentation on the interface as well as the extension development mechanism, the extension developer will have to analyze all classes to understand the extensibility model. For close-source applications this is not feasible both for the software provider and extender. Besides correctly identifying the classes, the extender also has to understand how to correctly extend the software system which is also not explicitly expressed.

The third problem is there is no standard way for implementing extension interfaces and for integrating and loading extensions. In the results of the study, 4 different kinds of extension units a software provider expect as extensions for the software systems and 5 different ways through which extensions are integrated with the core software system were identified.

2.4 Requirements for Extension Interfaces for Multilayered Applications

Problem		Explanation
P1	Expression vs. Enforcement	No support for the explicit expression of extension possibilities. A software provider also has to also think about how to enforce the extension interface.
P2	Expression of Extension Types	There are no high-level / domain-specific constructs to express different kinds of extension types. Extension developer is limited by object-oriented language constructs, extension interface, and usage interface.
P3	Fine-grained Access Rights	It is not possible to directly express access rights that extensions have on resources of the core software (e.g., read only).
P4	Interdependencies	No support for expressing cross-layer interdependencies between extension possibilities.
P5	One-size-fits-all Interface	There is no support for defining different interfaces for different kinds of extension developers.
P6	Mixing of functional and extensibility supporting code	The code supporting the extension interface is coupled with the functional code, i.e., there is no clear separation of concerns. This can cause maintainability issues.
P7	Enforcement, Documentation, and Integration Standards	There are currently no standards for enforcing and documenting extension interfaces, integrating extensions with the core software, and documenting extension interfaces.

Table 2.4: Summary of the identified problems of extension interfaces

In the previous discussions, a set of problems with extension interfaces were outlined. Table 2.4 summarizes the problems identified in this discussion. Based on the identified Problems 1–7 (P1–P7), the following requirements are defined for extension interfaces for multilayered applications.

RSP1: Explicit Extension Possibilities An extension interface must define which artifacts within the different logical layers of a software system are extensible as well as the types of extensions that are permitted (P1, P2).

RSP2: Access to Resources Besides declaring artifacts as extensible, the extension interface must specify what resources of the core software are available and what access rights does an extension have to these resources (P3).

RSP3: Separation of Concerns The development of the code supporting extensibility and the functional code must be separated and the extension interface must allow for better maintainability. An extension interface must allow the definition and enforcement of the extensibility supporting code without polluting the functional code of the software system (P1, P6).

RSP4: Support for Cross-layer Extensions The extension interface should allow for cross-layer extensions at different layers and should express the interdependencies between inter-layer and cross-layer extension possibilities (P4).

RSP5: Multiple Extensions and Extenders The generated extension interface enforcement code must handle multiple extensions. Moreover, the extension interface must account for that the software system can be potentially extended by several kinds of extension developers, e.g., internal development teams and external teams. The software provider can allocate different extension possibilities as well as access privileges to the resources of the core software systems for the different extension developers. This implies that it should be possible to specify and generate multiple extension interfaces for the same software system for the different kinds of extension developers (P5).

RSP6: Enforcement Standard Enforcing the extension interface is also another challenge faced by the software developer. There is currently no common method for enforcing (i.e., implementing) extension interfaces. A common method is needed for enforcing an extension interface. This is important for the software provider for better maintainability and program comprehension (P1, P7).

RSP7: Simplified Consumption of the Extension Interface It is very important to attract developers to build extensions for business applications. The more extensions are available for a certain business software system, the more likely customers will be willing to invest in it. If the underlying extension interface 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 extension developer. 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 have to be presented in a simplified way for an extension developer (P2, P3, P4, P7).

2.5 Summary

Supporting complex software systems that consist of several logical layers can be challenging for the software provider. In this chapter an example business application that consists of several logical layers was presented and the limitations of current object-oriented languages to express and implement extension interfaces for multilayered applications were described.

These limitations are disadvantageous for both the software provider and extension developer. With the current approaches, it is difficult to specify and enforce extension possibilities, express the different types of supported extensions, and control the access of extensions to the core resources of a software system. Furthermore, it is not possible to specify interdependencies between different extension possibilities on different logical layers and to support different kinds of extension developers. The more extension scenarios to be supported, the more complex the code of a software system will become since there are no clear separation of concerns and common enforcement techniques. In addition to the presented example, a study on real open source software systems of the Qualitas corpus confirms the described problems.

Based on these problems, requirements for extension interfaces that support multilayered software systems and different kinds of extensions developers were defined.

3 Extensibility and the Extension Developer

Implementing extension interfaces for multilayered applications is a challenging task. In the previous chapter, the problems with specifying and enforcing extension interfaces were outlined from the perspective of the software provider (see Table 2.4). In addition the specification and implementation of extension interfaces, the software provider has to give the extension developers the necessary means for implementing extensions, i.e., consuming these extension interfaces. From the perspective of the software provider, the easier a software system can be extended, the higher is the potential of attracting more extension developers.

In this chapter, I argue that the current means given by the software provider to the extension developers are not effective in helping them with implementing extensions for multilayered applications. More specifically, extension developers spend a lot of time and effort to identify the extension possibilities that are available, the types of extensions that are supported, and the implementation constraints that exist.

In the following I support this argument by presenting a user study on extension developers performing extension development comprehension tasks for a multilayered application while given some of the means that are provided by a software provider (e.g., API documentation, tutorials, etc.). Based on the outcome of this study, requirements for a tool supporting extension developers of multilayered applications are defined.

3.1 Design of the Study

There are two main goals of this study. The first goal is to identify which means do extension developers prefer to use and what information do they need for extension development. The

second goal is to evaluate the effectiveness of these resources and methods while accomplishing three extension development tasks. To cover these goals, the study is designed in two parts (The concrete tasks can be found in Appendix A.3).

3.1.1 Part 1 - What Means do Extension Developers Prefer and What Information do they Need?

Part 1 consists of a questionnaire that is used to identify and evaluate the resources that extension developers mostly rely on (or expect to have) when developing software extensions. The questionnaire is divided into two sections. In Section 1 the developers are asked to identify the resources that they would use as a good starting point for building extensions in general and they are asked to rank a list of 10 resources and methods. The developers are then asked to rate how favorable each resource or method is on a 7-point Likert scale. The resources and methods are:

- official API documentation,
- tutorials on building extensions,
- extension code examples,
- video tutorials,
- asking an experienced developer,
- web search,
- official online forums,
- IDE tool support (e.g., debuggers, wizards, code recommenders, etc.),
- learning by doing,
- and reading the source code.

In Section 2, the developers are provided with a screenshot of the user interface of the exemplary business application presented in Figure 2.2 in Chapter 2 along with the requirements of an extension that can eventually span several logical layers of the application. The developers are asked to freely report on what they need to know and have in order to implement the extension. The goal of this task is to identify what information do developers need out of these means to implement extensions for multilayered applications.

3.1.2 Part 2 - How Effective are these Means?

In Part 2 of the study, an investigation of the effectiveness of some given resources that support extension developers outlined in first part is done. This is achieved by tracking the resources used and measuring the time spent on each resource while performing three program comprehension tasks during the implementation of an extension. For this part, SAP Business One is chosen as a multilayered application.

The correctness of each task performed is also measured. The evaluation of these resources was focused on three comprehension tasks:

- Identification of the extension mechanisms offered by the software system. In this task the developer is required to express what extension methods and types are supported, the available API libraries for building extensions, and how extensions are integrated, loaded, and executed by the software system.
- Identification of the right API libraries and their correct usage while given the requirements for an extension for a particular extensible artifact belonging to a certain logical layer (e.g., a user interface form).
- Identification of the interdependencies and relationships between the extensible artifacts belonging to different logical layers (e.g., user interface and underlying database tables and business objects).

SAP Business One SAP Business One is an enterprise resource planning application for small and medium enterprises [SAP AG, 2014]. The application is intended to assist companies by providing support for many business processes such as sales, customer relationships, inventory, operations, finance and human resources. This application consists of several logical layers such as the user interface layer, the business object layer, the database layer, and the web services layer. The extensible artifacts of this application include business objects and database tables, user interface forms, and web services.

SAP Business One is built using Microsoft .NET technologies and can be extended via a software development kit (SDK) with C# and VisualBasic libraries. Using Microsoft Visual Studio, the extension developer creates a new extension project and then imports the SDK libraries required for building an extension. Currently, a developer can learn about building extensions through the SDK documentation provided with the system, training materials in the form of tutorials,

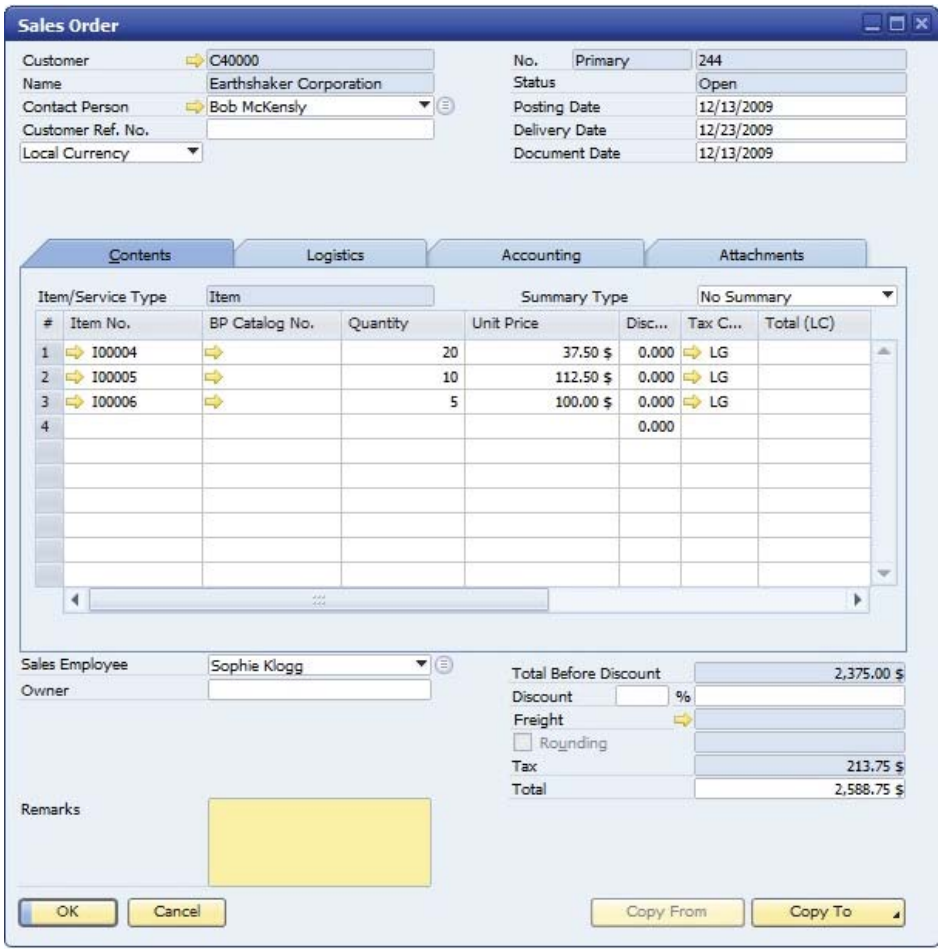


Figure 3.1: User interface for sales order processing in SAP Business One - ©2013 IEEE

code snippets and examples for various extension scenarios, an online development community forum, and video tutorials. Moreover, the developer can use various tools within the application that can provide debug information to help him with carrying out the extension development comprehension tasks.

The Extension Scenario Figure 3.1 illustrates the sales order form, which is part of the sales order module of the application. In this context an extension scenario is considered in which the sales order form needs to be extended with an additional text field to store the credit risk information of the customer and an additional button to save that information. The credit worthiness of the customer can be retrieved through the website of a credit reporting agency. This simple extension requires first an understanding of the available extension mechanisms of the software system and the available extension possibilities of the sales order form as well as those of the related business objects and database tables.

In addition to the new user interface elements that have to be added to the sales order form implementing this extension requires defining an additional attribute in the sales order business object and an additional column in the underlying database table. This requires the extension developer to correctly identify the user interface classes, the business objects, web services and database tables behind the sales order form that are allowed to be extended. Furthermore, the developer has to identify the right method calls to be used to add a new button and a text field to the form. Listing 3.1 shows a code example that adds a button to the sales order form (without the other extensions, i.e., database tables, business objects, and web services).

```
1  ...
2  //Get the Sales Order form (form number 139)
3  SAPbouiCOM.Form oOrderForm;
4  oOrderForm = SBO_Application.Forms.GetFormByType(139);
5
6  //Add a new button
7  SAPbouiCOM.Item oNewItem;
8  oNewItem = oOrderForm.Items.Add("CRATINGB", SAPbouiCOM.BoFormItemTypes.it_BUTTON);
9
10 // Position and define the size of the button
11 oNewItem.Left   = 120;
12 oNewItem.Width = 30;
13 oNewItem.Top    = 35;
14 oNewItem.Height = 10;
15 ...
```

Listing 3.1: Extending the sales order form with a button example

The Concrete Tasks This part consists of three tasks. In the first task, the developer is required to investigate the general architecture, API offerings, and extensible artifacts in the software system. During this task, the developer is expected to find the answer to three questions about what API can be used for extensibility and what high-level artifacts exist in the software system.

In the second task, the developer is required to identify the API coding elements behind the sales order form to realize the button and text field extension of the sales order form and answer three questions. The questions were about finding the right API methods and classes to access the sales order form, and adding a new button and a text field. The developers were also required to identify the correct usage of the API elements as well. In the third task, the developer is required to identify the interdependencies between the sales order form and the other extensible system

artifacts (i.e., database tables, business objects, and web services) that were identified by him in the first task. The developer is also expected to answer two questions about the names and types of the interdependent artifacts (i.e., business objects, database tables, and web services) that he is required to extend.

3.2 Participants and Execution

A total of 14 developers were recruited for the experiment sessions. The developers reported between 2 to 20 years of experience (μ=10, σ=6) of software development. 13 developers reported to have used a business software system before. 11 developers reported that they developed software extensions for different types of applications, 5 of them have reported that they developed extensions for business applications. All developers have confirmed that they have either programmed with Visual Basic or C# before and that they have never programmed an extension for (or used) SAP Business One.

The total amount of time given for each session was 90 minutes (the time was thought to be adequate for the experiment execution during the pilot study). The developers were allowed 30 minutes to complete Part 1 of the study and 60 minutes to complete Part 2. Before the execution of the main study, 2 developers (not part of the 14 developers of the study) were recruited for a pilot study in order to evaluate the setup, design, and time constraints defined.

For Part 2, a workstation was set up with internet access and a running copy of the software system. The developers were provided with a 15 minute introduction to the system and to the sales order module (both from a user perspective). The developers were then provided with the official SDK documentation (containing code examples), tutorials and learning materials, links to the official development forum of the product, and video tutorials for developing different extension scenarios. Moreover, the developers were shown the application resources that provide debug information which can be used to help them develop extensions. All resources were provided to the developer on the same workstation running the software system.

Points	Explanataion
0	No answer / Wrong answer
1	Incomplete with incorrect answers
2	Correct but incomplete
3	Fully correct and complete

Table 3.1: The point-based scheme for grading the tasks

The developers were given 15 minutes to complete each task, and they were given the option to move on to the next task when they thought that they completely and correctly answered the questions. During the execution of each task, the activities done by the developers (i.e., browsing the application, reading the documentation, web search, searching forums, etc.) as well as the time spent for each activity were tracked by TasksShow [Schmidt and Godehardt, 2011]. The TasksShow tool monitors the user system interaction and creates an interaction history. The interaction history contains detailed information about the accessed content, the used functionality and the duration of the user system interaction.

After the sessions were concluded the answers provided by the developers for each task were graded according to the following scheme: 0 points are awarded if the developer was not able to answer or provided a wrong answer to the question, 1 point if the solution was incomplete and contained incorrect answers, 2 points if the solution is correct but incomplete, 3 points if the solution is complete but contains incorrect answers in addition, and 4 points if the solution is fully correct and complete. Table 3.1 summarizes the scheme. The maximum score for Tasks 1 and 2 is 12 points and 8 points for Task 3.

3.3 Results

3.3.1 Part 1

For Section 1, Figures 3.3 and 3.2 report on the average and standard error of the rankings and ratings of the resources and methods respectively. No significant differences between the ratings and the rankings of each of the resources and methods were noticed in the results. The highly ranked resource was the code examples for extension scenarios. Extension tutorials and IDE tools almost shared the same rank and asking an experienced developer comes next. API documentation and forums were almost close in ranking. Video tutorials, learning by doing, and web search were almost on the same rank. The least ranked resource was using the source code of the application.

In Section 2, the responses of the developers were analyzed and categorized into groups of recurring themes and requests. The groups are ranked based on the frequency of the responses and are reported on them in the following.

The first group (G1) of responses reflected that the developers are focused on the technical realization of an extension. The developers wanted to know more about which API methods they must use, which classes they must extend, what are the accepted extension types. Responses in

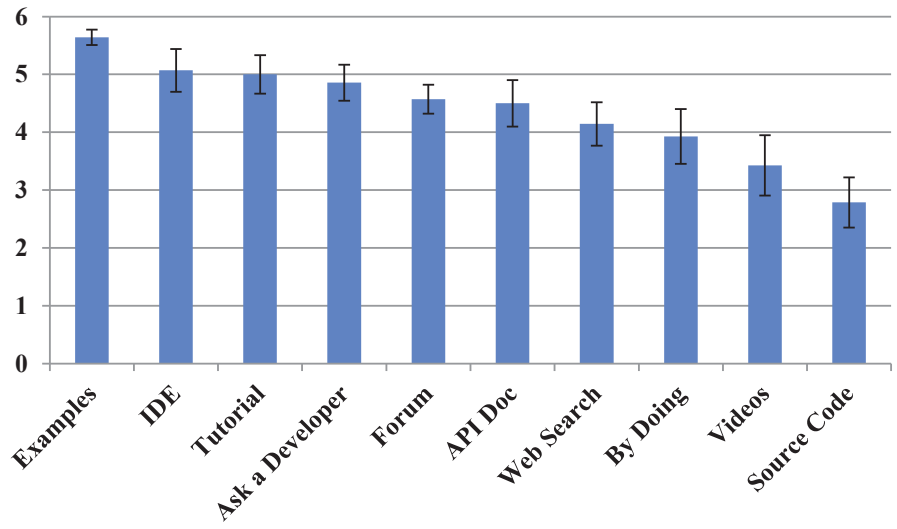


Figure 3.2: Ratings on a 7-point Likert scale (mean and standard error) - ©2013 IEEE

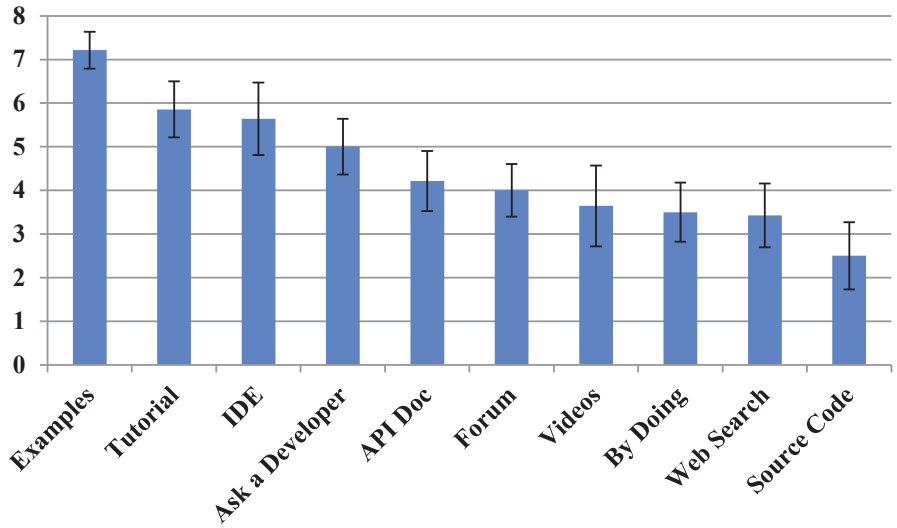


Figure 3.3: Rankings (mean and standard error) - ©2013 IEEE

this group included requests from developers like: *"What is the name of the class behind this user interface?"*, *"What methods are needed for adding new user interface elements?"*, and *"What types of user interface elements are supported?"*.

The second group (G2) of responses was focused on the technologies and frameworks that are used by the software system. The responses reflected that the developers wanted to try and find out if the software system was built with a technology or a framework that they are already familiar with. This can help understand better what they will have to do to implement an extension. Responses in this group included statements like: *"Is the user interface based on the Java Swing library?"* and *"What is the name of the persistence framework used by this software?"*.

The third group (G3) of responses included the extensibility concepts and the application architecture. In this group, the developers wanted to understand concepts like how the extension code is packaged and integrated (e.g., via plug-ins), loaded, executed, and managed by the core software. Responses in this group included statements like *"Do I have to implement a plug-in?"* and *"Where do I have to place the implemented extension code?"*.

The fourth group (G4) of responses included application logic, side effects, and dependencies involved with other software artifacts when implementing the extension scenario. The responses of this group showed that developers are aware that building extensions can affect and cut through multiple logical layers and artifacts in the core software. Responses in this group included statements like: *"What is the name of the database table that stores the data for this module?"* and *"Which business objects implement the logic for this module?"*

The fifth group (G5) of responses included questions about the availability of documentation, tutorials, code examples, and the availability of the source code. Table 3.2 summarizes the groups of responses.

Group	Need to Know / Have
G1	Names of the available API classes and methods.
G2	The frameworks and technologies that the software system is based on.
G3	Packaging and integration of extensions.
G4	Interdependencies between extensible artifacts.
G5	IDE tools, documentation, tutorials, code examples, source code.

Table 3.2: Categorized responses of the developers in Part 1, Section 1

3.3.2 Part 2

The following reports on the time spent on each resource and then reports on the scores achieved in each task during the sessions.

Using the data provided by TasksShow, the time spent on each resource and the search queries that were input by the developers during the session were extracted. On average the developers spent 34 minutes (rounded to the nearest minute) to solve the tasks. 38.3% of the time was spent on API documentation, 23.6% on tutorials, 17.5% using the application debug information tool, 10.5% on forums, 8.1% on web search, and 2% on video tutorials. Figure 3.4 reports how the resources were used over time by each developer during the session. The number shown on each bar indicates the total time spent (rounded to the nearest minute) by the developer to solve the given tasks. Figure 3.5 reports the number of points awarded for each task for each developer. On average, the scores were 93.5%, 51.8%, and 18.5% for Tasks 1, 2, and 3 respectively.

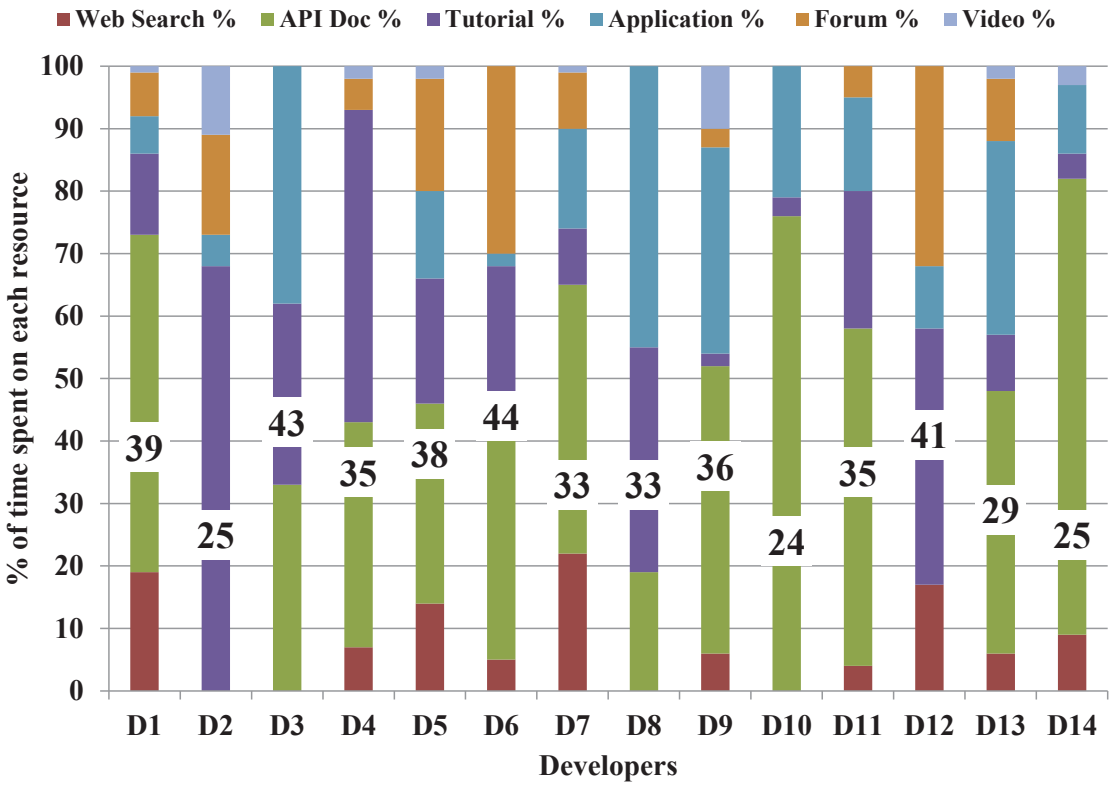


Figure 3.4: Time spent by each developer on resources - ©2013 IEEE

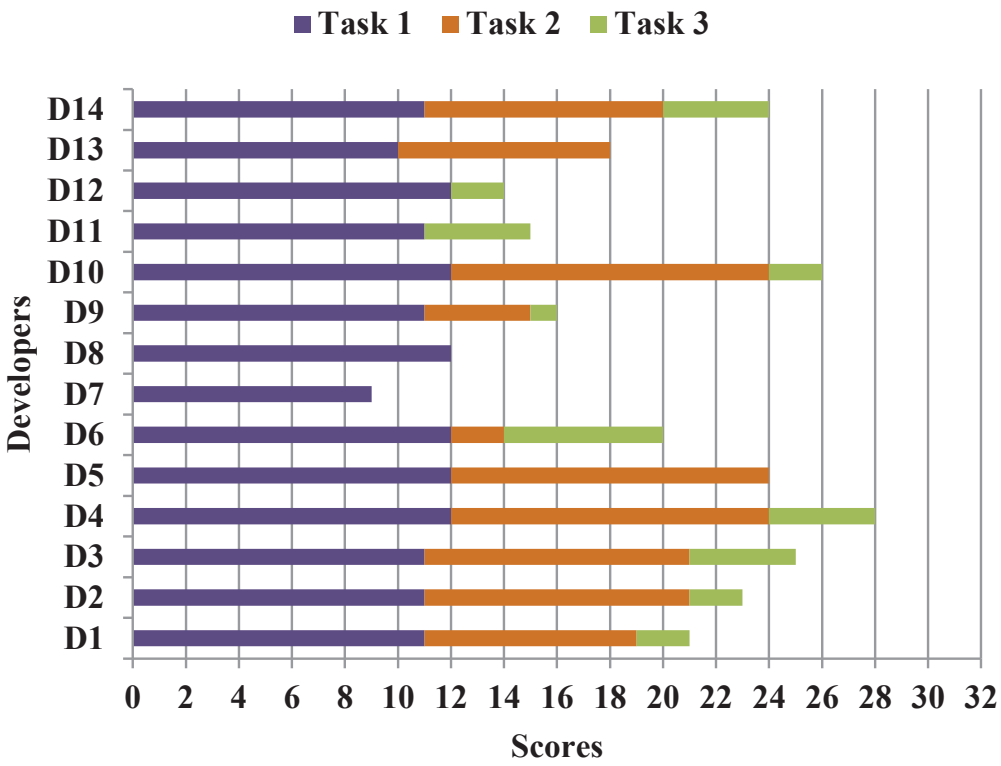


Figure 3.5: Scores for each task - ©2013 IEEE

Threats to Validity In the following some of the threats to validity are outlined. The first threat to validity is caused by the choice of the subjects. Almost all of the subjects have a long experience in software development and extension development. However, all of the subjects have no experience with the software system used in the study. The results however might be different with developers who are already experienced with the software.

The second threat to validity is caused by the choice of the software system. It may be the case with other software systems that the results may be different due to factors like documentation and tutorials designed in another way, different resources and forums on the web, and different tool support for building extensions. To control this threat as much as possible, the developers were shortly briefed about the software system and were allowed to get acquainted with the different resources given to them before performing the tasks.

The third threat to validity is caused by the design of the tasks and the time given to the developers. With other tasks and more time, the results may be different. However, this threat has been controlled appropriately by carrying out two pilot sessions before executing the study.

3.4 Discussion and Problem Definition

In Part 1, the results of Section 1 reflected that the top 3 preferred methods were extension code examples, IDE tool support, and extension development tutorial, whereas the least preferred technique is reading the source code of the software system. In Section 2, the responses confirmed that the software developers are aware of the development challenges that can arise while implementing extensions for multilayered applications. Based on the categorized responses (G1–G5), there are several concerns that need to be addressed by the extension developers. The effectiveness of the current means in addressing the concerns of the developers can be demonstrated by the results of Part 2. In the following these concerns are outlined and discussed.

The first concern is about identifying the right API classes and methods that are required for implementing the extension, i.e., mapping the high-level artifact (e.g., user interface) to the corresponding coding elements. This can be reflected by the responses categorized in the groups G1 and G2. In Task 2, the developers were required to identify the right methods for adding a button and a text field. The scores of Task 2 (51.8%) reflect that most of the developers did not find it easy to identify the right API elements for realizing the extension for the user interface. In order to understand the extension possibilities offered in the user interface, the extension developers depend on resources such as tutorials, documentation, and source code examples rather than more "natural" explicit (i.e., domain specific) resources or runtime artifacts. As a result, more time and effort is required by the developer to understand the extensibility offerings.

The second concern is about packaging and integrating the extension with the core software system (reflected in group G3). The analysis however of the scores for Task 1 (93.5%) reflects that the resources provided to the developers for the given time frame were very effective in understanding the general architecture and extensibility concepts of the software system.

The third concern is about identifying the interdependencies between extensible artifacts (G4). The scores of Task 3 (18.5%) reflect that the developers found it extremely difficult to use the given resources to find the relationship between the extensible artifact (sales order form) and other extensible artifacts belonging to other layers (web services, business objects, and database tables).

The last concern is about acquiring the relevant API documentation, examples, tutorials, and source code (G5). During the execution of Part 2, 61.9% of the time was spent on average by the developers reading documentation and tutorials and 28% on web resources which is much more than the time spent on the tools provided by the application (17.5%).

To summarize, there are several problems that have to be addressed to help support extension developers more effectively. The first problem is that the developers need to understand the different logical layers, the extensible artifacts in each layer, and their interdependencies that exist in the application. This is very important especially in the early stages of the extension development process where the extension developer has to assess the feasibility to realize the extension requirements. The second problem is that the relationship between the provided extensibility possibilities and the low-level API coding elements through which these possibilities can be used are not explicit and intuitive for the extension developer to find out. The third problem is that if the API coding elements were correctly identified by the extension developer, the extension developer still has to be able to correctly use the API libraries as expected by the software provider. The fourth problem is that the developers find it very difficult to identify the relationship between extensible artifacts belonging to different logical layers. The fifth problem is that the developers still spend a lot of time reading tutorials and documentation and searching forums and the web in order to understand what extension possibilities exist, what libraries to use, and what extension mechanisms must be used. Table 3.3 summarizes the outlined problems.

Problem		Explanation
P1	Explicit Extension Possibilities and Interdependencies	The extension developer has to be able to identify which extension possibilities exist on each logical layer and what interdependencies and constraints exist.
P2	Mapping of High-level Extensible Artifacts to Low-level Coding Artifacts	Once an extensible artifact in a logical layer is identified, the corresponding coding elements (e.g., Classes and Methods) have to be identified.
P3	Correct Usage of Coding Artifacts	The identified coding elements have to be correctly used by the extension developer (i.e., as expected by the software provider).
P4	Time	The extension developer spends a lot of time relying on the classical means to be able to develop an extension.

Table 3.3: Summary of the identified problems of extension developers

3.5 Requirements of the Extension Developer

The previous discussion outlined several problems that extension developers face when implementing extensions for multilayered applications. To aid extension developers with the implementation of extensions, the following requirements for a recommender system that aids extension developers with developing extensions for multilayered applications are proposed:

- **RPC1: Explicit expression of extension possibilities** The system is required to explicitly express the extension possibilities of the different extensible artifacts in the different logical layers without the extension developer getting access to the source code of the core software.
- **RPC2: Expression of interdependencies between extension possibilities** Besides the visualization of extension possibilities, the system must represent the interdependencies between the extension possibilities within the same or the other logical layers.
- **RPC3: Mapping of extension possibilities to coding artifacts** The system must recommend the relevant API libraries, code examples (or stubs), and methods that are required to implement an extension possibility.
- **RPC4: Recommendation of documentation** The system should recommend the relevant documentation required for the realization of a specific extension possibility.
- **RPC5: Reduce time needed for development** Ideally, the system must reduce or eliminate the need or time for a developer to access other helping aids like search engines, forums, training, etc. for performing extension development related tasks.

3.6 Summary

In this chapter a study on extension developers performing extension development comprehension tasks was presented. The study shows that the current means that extension developers use for implementing multilayered extensions are not effective. To perform a simple cross-layer extension, the developers still spend a lot of time reading documentation and tutorials as well as searching the web. Besides the time consumption, there are several problems with these classical methods. First, they do not provide an explicit representation of extension possibilities and their interdependencies on the different logical layers of an application. Second, these methods do not provide the means to map the extension possibilities of high-level coding artifacts (e.g., user

interfaces and business processes) to low-level coding artifacts that are required for the concrete realization of the extension. Third, these methods do not guide the developer with the correct usage of the low-level coding artifacts during the implementation process of an extension.

As it will be emphasized in Chapter 4 various program comprehension tools have assisted developers with accomplishing various development tasks. However, these tools do not provide the necessary support for multilayered extensibility. Based on these problems and the findings of the study, requirements for program comprehension tools that support extension developers of multilayered applications have been proposed.

4 State of the Art

Extensibility is considered as a very important software reuse mechanism for software providers who do not give their source code to their customers. A broad range of research has been directed towards specifying and enforcing extension interfaces as well as aiding extension developers with the extension development process. In the previous chapters, the problems and challenges associated with supporting extensibility for multilayered applications have been discussed. Most of these problems emerge due to the shortcomings of the state-of-the-art approaches in supporting multilayered applications.

The need and the challenges related to providing well-defined extensibility interfaces for object-oriented systems are well-documented in the literature [Kiczales and Lamping, 1992, Steyaert et al., 1996, Mezini, 1997, Bloch, 2008]. Furthermore, the need for tools that assist developers to identify, comprehend, evaluate and use these interfaces has also been acknowledged by researchers [Tichy, 1992, Krueger, 1992].

In this chapter several works on supporting the specification and enforcement of extension interfaces are presented. Moreover, recent works on developer tools for program comprehension are described. The strengths and weaknesses for each of the approaches is discussed.

4.1 Extensibility and Extension Interfaces

Several lines of research have been targeted towards evolution (e.g., extensibility, variability, etc.) of software systems. In the following an analysis of the strengths and weaknesses of each approach is presented. First, a discussion on the approaches supporting extensibility for object-oriented frameworks is presented. Second, the view on extensibility and extension interfaces

by some recent state-of-the-art programming paradigms is discussed. Third, recent works on language-based mechanisms that enable extensibility are described. Last, the discussion on language-based mechanisms is discussed by a focus on recent works on aspect-oriented interfaces that are relevant for defining extension interfaces. Tables 4.1, 4.2, 4.3, and 4.4 present the major strengths and weaknesses for each approach respectively.

4.1.1 Object-Oriented Frameworks

Object oriented frameworks [Fayad and Schmidt, 1997] provide generic bases for application developers that can be extended or specialized to support a particular domain. To support extensibility, a framework must provide explicit interfaces (e.g., through hooks and predefined methods) to facilitate the implementation and integration of new functionality. The realization of such frameworks highly depends on advanced implementation strategies like design patterns. Two models of extensibility are typically supported by these frameworks; white-box and black-box views [Zenger, 2004] extensibility. White-box extensibility relies on extension developers inheriting from predefined classes or interfaces. The extension code is then integrated with the core software through dynamic binding. Black-box extensibility typically depends on coding extensions to comply with predefined extension interfaces or through the use of (domain-specific) scripting languages. Extensions are typically integrated using object-composition.

The main strength of object-oriented frameworks is that they provide a generic base for developers to extend or customize to fit their particular domain of application, saving development time, effort, and cost. However, there are several drawbacks of this approach. First, the code enforcing the extension interfaces is tightly coupled with the functional code of the framework. As a framework evolves, evolving the extension interfaces will not be an easy task. Second, the frameworks provide a one-size-fits-all interface for all developers. There is no possibility to define different interfaces for different kinds of extension developers.

Reuse Contracts and Smart Composition [Steyaert et al., 1996, Mezini, 1997] are approaches both on the design and language levels for documenting reusable assets of the base software. Using a contract, the software provider can document the design of the class (e.g., method interdependencies) and how it can be reused. Using predefined operators, the extension developer can declare in what way the declared classes will be reused. Using such contracts, it is possible to detect if any conflicts will take place as the base code and extension classes evolve. There are several limitations of these approaches. First, these contracts are not only specialized for defining extensibility, but also other kind of modifications. Moreover, each contract is defined

on a single class, and it is not possible to define extension possibilities spanning several classes, interdependencies, and advanced extensibility constraints.

Design patterns [Gamma, 1995] are informally specified patterns in software design that aim to solve reoccurring problems. Each pattern can either have a creational, structural, or behavioral purpose. Patterns are usually documented and described in terms of purpose, motivation, structure, and relations to other patterns. There are several design patterns that support extensibility through the addition of new behavior or structures, e.g., visitor, strategy, template, and extension object pattern [Gamma, 1997].

Although design patterns can support developers with the enforcement of extension interfaces, their informal semantics can be misleading [Krishnamurthi and Felleisen, 1998]. To apply the correct design pattern, a developer has to be experienced with the selection and implementation of the relevant patterns. Moreover, documenting and maintaining the implemented patterns requires a lot of efforts at the provider side, since the realization of the patterns is done on the code level.

Plug-in systems abstract the data and functionality of an application through an application programming interface that acts as hooks or extension points [Ivar et al., 1997, Clements and Northrop, 2007]. Extenders can then write applications and package them in the form of plug-ins that conform to the API. The plug-in platform manages the integration and execution of plug-ins. Examples of plug-in systems are the OSGi [OSGi Alliance, 2003] based Eclipse [Eclipse Foundation, 2014a] and the Microsoft Managed Extensibility Framework (MEF) [Microsoft Corporation, 2014]. Extension points are dependent on the interface definitions declared by the base plug-in developer. These interface definitions indicate how the contributing plug-in should be called and what data it can get.

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 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.

Plug-ins improve the usage of object-oriented frameworks by adding a meta-layer on top of the concrete code-level interfaces that has to be implemented by the extension developer. In addition, a plug-in based framework must document for the extension developer the life cycle of a plug-in (i.e., integration, loading, and execution). In comparison to object-oriented frameworks, the approach can help developers identify extension possibilities on the code level, however there are several drawbacks. First, the implementation effort is increased for the software developer as the interfaces on the code level as well as the plug-in mechanism has to be manually developed and the corresponding metadata describing the requirements for plug-ins and development constraints has to be specified as well. Second, the specification of the extension possibilities is still done on the code-level and not using domain-specific terms. Third, an extension developer has to go through documentation and tutorials to understand how his plug-in will be integrated, loaded, and executed.

Scripting Based Approaches The main essence of scripting based approaches is to provide a language to the extension developers through which they can develop extensions that will be interpreted by the core software. For example, Mozilla Firefox provides a language called XUL [Feldt, 2007] that allows extension developers to implement graphical user interfaces that are interpreted by the browser. The text editor Emacs provides a dialect of Lisp (Emacs Lisp) [Glickstein, 1997] for allowing extension developers to extend its features.

Scripting approaches offer the most rigorous form of controlled extensibility [McVeigh, 2009]. However on the side of the software provider they require high development skills, efforts, and cost. The software provider will have to define a new language along with its semantics on top of the application as well as implement the necessary support for interpreting the extensions that are implemented using this language. Moreover, in a commercial setup, the software provider will have to provide the extension developer with the necessary integrated development environment support (e.g., code editors, debuggers, documentation, tutorials, etc.) which will also contribute to the development effort and cost. From the perspective of the extension developer, this can be advantageous, since the core software can support extensions which are developed using a domain-specific extension language. However, the extension developer will also need the time to learn that language.

Approach	Strengths	Weaknesses
Object Oriented Frameworks	+ Abstracts generic functionality. + Supports extensibility through black-box and white-box reuse. + Provides developers with points for variability and extensibility.	- The code required for realizing the functionality of the system is mixed with the code of the extension interface. - Implementation of extension points requires appropriate design and coding skills as well as high development efforts. - For closed-source systems developers must have extensive knowledge on the internal workings of the framework. - High cognitive load for identifying the extension points and underlying coding artifacts which are required for realizing an extension.
Reuse Contracts / Smart Composition	+ Specify the design of the class and how it can be reused.	- Specified for a single class. - Not possible to specify extension possibilities covering several classes or advanced extensibility constraints.
Design Patterns	+ Provides documented coding patterns for variability and integrating extensions. + Generic; can be applied to any object oriented language.	- Design patterns are informally specified. - Informal specification and semantics can be misleading for software developers. - Cannot be automatically enforced. - Prone to error if software developer is not experienced with implementation. - Extension developer has to understand the underlying pattern to be able to implement an extension.
Plug-in Systems	+ Provide developers a framework for developing and integrating extensions with core systems. + Manages extension lifecycle (discovery, validation, and execution). + Support black-box reuse	- High investment and development efforts are required to build a plug-in framework. - Dependent on the language of development of the core software. - Functional source code will be polluted with the code required to discover, load, and execute extensions. - The extension developer has to understand the extension lifecycle (i.e., how extensions are managed) of the plug-in framework to develop an extension. - More effort is required for the implementation of extensions as plug-ins have an overhead of metadata for specifying contributions and used extension points.
Script Based Approaches	+ Provide domain-specific constructs for realizing extensions. + Great control for extensibility.	- Software provider has to foresee all extension scenarios. - When new extension scenarios need to be supported, the language has to be extended. - High costs and development effort to support the extension language by the software provider. - Extension developers might need to learn a new language to implement their extensions.

Table 4.1: Object-oriented approaches supporting extensibility - Strengths and weaknesses

4.1.2 Extensibility and Programming Paradigms

Component-Based Software Engineering In [Szyperski et al., 2002] the authors define software components as *"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. Introduction of extensions requires an extension developer to provide a new component that provides new functionality and interacts with existing components. 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 [Crnković et al., 2011]. Components offer a great concept for black-box reuse and separation of concerns.

However, extending a component based system can be difficult [Zenger, 2004]. Building extensions are highly dependent on interface definitions, which imply that the extension of structural or behavioral attributes of an existing component might not always be feasible. Changes to an existing interface of a core component can also adversely affect extensions. The composition of an extension component with existing components requires the understanding of the current composition model (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.

Aspect Oriented Programming The main motivation behind Aspect Oriented Programming (AOP)[Kiczales et al., 1997] is to reduce the scattering and tangling of cross-cutting concerns that interfere with the core concerns of a base system. AOP allows the modularization of cross-cutting concerns by abstracting their logic into advices that get executed at certain join points within the base system. An extension of a software system requires the extension developer to implement an advice that can consist of the behavioral and/or structural additions as well as join points which define where the advice should run. Join points can be chosen 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 [Brichau and Haupt, 2005].

AOP assumes a white-box view on source code. This is not suitable for commercial applications which do not provide the source code to the customers. The knowledge of the extension developer of the source code is very important to specify the right pointcuts that the advice will extend. Moreover, there are several other limitations. First, the specified pointcuts might capture unintended join points. Second, there is no control over the advice on what changes or effects it

can cause to the main execution stream of the software. Third, the evolution of the core software might lead to breaking the specified pointcuts or advice code (e.g., pointcuts might be no longer valid or the advice code performs an incorrect function). Fourth, it is not possible to express extension interdependencies between different extension possibilities specified by the pointcuts. There are many approaches as they will be later elaborated on which address these disadvantages, however they also have their limitations.

Subject Oriented Programming Subject oriented programming (SOP) [Harrison and Ossher, 1993] is an improvement on object-oriented programming that allows a class to be defined in a decentralized way [Tarr et al., 1999, Ossher et al., 1995]. It can also be seen as complementing AOP [Kiczales et al., 1997]. Each subject specifies the particular data and operations that it expects from the class. The system then combines the different subjective views and generates the corresponding class definitions. Composition takes place at the binary level. This implies that different subjects can be written and maintained separately. Composition rules govern how the composition should take place. SOP claims to provide more structuring to software artifacts as well as independent and non-invasive software development [Ossher and Tarr, 1999]. The most famous implementation of SOP is Hyper/J [Ossher and Tarr, 2000].

While subject-oriented programming promotes the introduction of unforeseen extensions, there are several limitations when implementing extensions. First, the software provider will have to extensively specify composition rules which govern how extensions will be integrated with other subjects of the core software. Second, the composition can lead to unforeseen effects on the resulting system behavior if not correctly handled. Third, SOP does not provide a way to enforce constraints on different extension possibilities. Fourth, the extension developer still has to identify the relevant subjects as well as the existing interdependencies and constraints to be able to realize his extension.

Feature Oriented Programming Feature oriented programming (FOP) is a programming paradigm that supports the production of large software systems [Apel and Kästner, 2009]. The paradigm is most famous for its support of software product lines [Lee et al., 2002]. A feature represents a requirement or a functionality that is expected in the software. Extending a software system implies the introduction of a new feature. In FOP, three key areas play an important role: feature modeling, feature interaction, and feature implementation. Feature models [Kang et al., 1990] provide means to describe relationships and constraints between different features. Feature interaction [Calder et al., 2003] is important to analyze if features can possibly interfere when

combined together. Feature implementation involves the transformation of feature models to concrete programs.

The advantage 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. As a consequence, the approach promotes feature reuse. However several problems can arise. A single feature model is maintained for a certain family of software products, which makes independent extensibility very difficult. Also the maintenance of feature models for large product lines can be very tedious. Furthermore, FOP approaches only modularize hierarchical features and they do not support capturing crosscutting features [Mezini and Ostermann, 2004].

Change-oriented Programming Change-oriented programming (ChOP) [Robbes and Lanza, 2007, Ebraert et al., 2007] defines first-class change entity objects to model a program evolution. The approach consists of monitoring the activities of a developer while implementing a software system and recording them in change objects. Examples of such activities include the creation, removal, and modification of packages, classes, methods, variable, and statements. ChOP can be used to improve FOP by adding improved composability and increased expressiveness [Ebraert and Merino, 2008]. For example, it can be used to verify whether a certain feature composition is valid or not. Another advantage is that it allows the application to be developed in an incremental way in contrast to FOP.

There are several drawbacks of this approach. Tracking changes depends on the granularity of what is considered as a change. Since it is developer oriented, changes required to come to a target software might be incorrectly specified or result in conflicts with existing changes. In order to extend a software system, the right set of changes has to be composed with the existing software. This implies that the current knowledge of all changes the software has been through should be known to indicate the right sequence of changes needed to further extend the software. Furthermore, since it follows an incremental approach and assumes that developers have access to the source code of the software system, it can be very difficult to perform independent and black-box development.

Approach	Strengths	Weaknesses
Component-Based Software Engineering	+ Well defined interfaces. + Promotes black-box extensibility. + Good separation of concerns.	- Interface definitions might not allow appropriate extensions of behavioral and structural aspects. - There is no standard for expressing interfaces. - There is no standard for composing components. - Composition of components can result in unwanted behavior if not explicitly defined. - Evolution of the core software can require updating the corresponding extension interfaces. - Highly dependent on the component implementation technology.
Aspect Oriented Programming	+ Provides better modularization of the software system and separation of concerns. + Reduces the development effort for tackling extensions spanning cross-cutting concerns.	- Only white-box extensions are supported. - Does not support independent extension development. - Extensions are limited by the pointcut language. - Extension developer is required to understand the source code of the core software to implement an extension.
Subject Oriented Programming	+ Allows for black-box and independent extension development. + Composition is managed by composition rules to prevent conflicts.	- Software provider must carefully define composition rules. - Extension developer has to identify the relevant subjects to be able to implement a valid extension. - Composition must be carefully carried out. - No constraints on extension possibilities.
Feature Oriented Programming	+ Incremental evolution of a software system by addition of features. + Supports product families of related features.	- Additional development effort by defining feature models. - Single feature model is maintained for a family of products (no multiple views). - Independent extensibility is difficult. - Only modularization of hierarchal features and no support for cross-cutting (horizontal) features – inadequate for multilayered extensibility.
Change Oriented Programming	+ Models extensibility as a first-class change object. + Validates if extensions (changes) are valid.	- Requires monitoring of development activities. - Allows only white-box extensibility. - Extension developer has to understand the internals of the core software to implement the right changes. - Changes are only tracked on the source code level.

Table 4.2: Programming paradigms supporting extensibility - Strengths and weaknesses

4.1.3 Language-level Approaches

Mixins A mixin [Bracha and Cook, 1990, Findler and Flatt, 1999] 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 functionality specified by the mixins. Mixins use single inheritance as means of composition. Mixins are limited in many aspects. 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. Given the resulting inheritance chains with glue code, the introduction of new mixins to an existing parent class can be very tedious. Furthermore, the modification of a mixin that is being used can be difficult as dependencies may exist.

Traits A trait [Schärli et al., 2003, Ducasse et al., 2006] 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. 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. 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. 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 [Madsen and Møller-Pedersen, 1989, Ernst et al., 2006] 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. 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.

Difference Based Modules Inspired from the assumption that collaborations are better units of reuse, difference based modules [Ichisugi and Tanaka, 2002] define a module based mechanism to support collaborations as units of reuse instead of classes. A module is described to be the difference between the original program and the extended program. Several modules (differences) can be added up to obtain a target program. A module consists of class implementations. Modules can be inherited and support the addition of new classes, new attributes and methods to existing classes, and overriding existing methods. Furthermore, difference based modules support the separate compilation of modules, allowing black-box and white-box reuse.

Approach	Strengths	Weaknesses
Mixins	+ Separate functional modules. + Support reuse of implemented functionality.	- Only one composition method: via inheritance - Might require complimentary code to ensure the integration of multiple mixins. - Order of inheritance can influence structural and behavioral properties, and thus can alter the intended behavior of the core software. - Extension developer must be aware of the inherited mixin classes to avoid any conflict.
Traits	+ Fine-granular methods for composition (not only inheritance). + Conflicts upon composition have to be supported. + Composition order is not relevant.	- Code is required to glue traits together. - Each conflict must be resolved separately.
Virtual Classes	+ Provides a contract for extension developers. + Support different types of extensions for a class.	- Statically bound and require planning ahead by the software provider. - Complex realization through nested classes. - Access to enclosing class is not always convenient (possible with workarounds). - Covariant types; not type safe, needs runtime checks. - Extension developer must have the source code of the enclosing class to understand how to implement an extension.
Difference Based Modules	+ Support black-box reuse. + No code required for integrating modules. + Customizable application through module integration.	- Do not allow dynamic loading of extensions. - Multiple versions of the same class cannot exist.

Table 4.3: Code-level approaches supporting extensibility - Strengths and weaknesses

4.1.4 Aspect-Oriented Approaches

Open modules [Aldrich, 2005] use modules that contain functions and pointcuts to expose advisable join points of a particular module. Clients of the module are allowed to advise the external calls of the functions and the exposed pointcuts of the module, but they are not allowed to advice the internal calls of the functions within the module. The exposed pointcuts are promised to be maintained as the module evolves. A major limitation of open modules is that pointcuts are tightly coupled with the definition of the module, and therefore it is not possible to express crosscutting concerns across several modules.

Crosscutting interfaces (XPIs) [Sullivan et al., 2010] partially address the limitations of open modules, by defining the crosscutting interfaces independently of both the advised code and the advice. XPIs use AspectJ pointcuts to expose the join points in the base modules along with informally defined contracts relying on design rules. Furthermore, the design rules contracts used in XPIs are informally defined and no means are provided for enforcing them.

Extension Join Points (EJPs) [Kulesza et al., 2006] uses XPI-like pointcuts to support the modularization of object-oriented frameworks. The main goal of EJPs is to facilitate the integration of a framework with other software components, offer possibilities of extension to the core of the framework, and support variability. EJPs define two types of contracts; internal contracts and extension contracts. Internal contracts assure that the evolution of the framework will not affect the extension aspects. Extension contracts assure that extensions do not violate the constraints and invariants of the framework. In contrast to XPIs, the contracts are partially enforced using AspectJ.

Model-based pointcuts [Kellens et al., 2006] specify pointcuts on the conceptual model of the software and not on the source code in comparison to the traditional pointcut specification in AOP. This is advantageous since the source code can evolve without harming predefined pointcuts as they are not expressed on the source code.

Explicit join points [Hoffman and Eugster, 2007] offer an explicit representation of bidirectional communication channels between aspects and base code in the form of abstract join points. The main idea of the approach is to make the core software aware of the aspects. The base code can "invoke" abstract join points declared in aspect interfaces to denote a concrete join point. Abstract joint points are used in the pointcut definitions of concrete aspects.

Join point types [Steimann et al., 2010] and *join point interfaces* (JPIs) [Inostroza et al., 2011] introduce an additional layer to serve as an interface between join points and advice. These approaches enrich pointcuts with a “type” (syntactically in a method signature like fashion) that specifies information passed between the base code and the aspect. This is advantageous since the advice code can only access the elements within the declared type as a specific join point.

Although these approaches work towards explicitly defining extension possibilities, there are several limitations that make them inadequate for controlling extensibility. First, the extension

possibilities are expressed using a pointcut language and not using domain-specific terms. Second, there is no fine grained access control to the elements specified in the type. Third, it is not possible to express whether the extender has a read / write access to certain attributes. In addition to that, there is no possibility to restrict an advice code from calling certain methods. Fourth, It is not possible to constrain the interplay within extension possibilities (i.e., from different logical layers). Last, it is not possible to support multiple extenders with different access rights to the base code resources.

Approach	Strengths	Weaknesses
Open Modules	+ Explicit pointcuts indicating extension possibilities. + Preserve intended behavior of a module by allowing advices only to pointcuts and to external calls of the functions of the modules.	- Definition of pointcuts is closely coupled with the definition of the module. - There is no possibility for expressing crosscutting concerns across different modules. - Classes are not aware of the exposed pointcuts
Crosscutting Interfaces	+ Explicit representation of extension possibilities on the code level. + Separates extension possibilities from the functional code.	- No way to validate an extension, since contracts are specified informally. - Conflicts can arise within multiple extensions since the pointcuts are not typed. - Extension possibilities are not explicit since they depend on the language constructs of the pointcut. - No support for expressing interdependencies between join points.
Extension Join Points	+ Modularizes extensibility, variability and integrability of object-oriented-frameworks.	- Based on XPIs (same limitations). - Fine-grained extensibility is not supported on framework high-level artifacts (e.g., user interfaces, persistency, etc.) - Interdependencies between the defined categories (extensibility, variability, and integrability) cannot be expressed.
Model-Based Pointcuts	+ Expression of pointcuts on the conceptual level allows source code to evolve independently	- Lack of domain specific constructs to express extension possibilities. - Extension developer has to understand the semantics of the model-based pointcut language to know how his advice will be integrated and executed. - No possibility for expressing interdependencies.
Explicit Join Points/ Join Point Types / Join Point Interfaces	+ Separate extension possibilities from the functional code. + Allow extensions to be separately compiled from the core software. + Can validate extensions to a certain extent since contracts are formally specified.	- Cannot constraint multiple join points from different layers of abstraction. - No support for domain-specific constructs to specify extension possibilities.

Table 4.4: Aspect-oriented approaches supporting extensibility - Strengths and weaknesses

4.2 Program Comprehension Tools

When the decision is made to develop an extension for a particular software system, the extension developer will have to be able to realize the requirements through code. With integrated development environments being popular, the use of program comprehension tools that help developers with various development tasks (e.g., design, implementation, maintenance, etc.) is becoming more popular. According to Storey [Storey, 2005], program comprehension tools can be roughly categorized into extraction, analysis, and presentation tools. In this section, recent works on program comprehension tools that adopt one or more approaches of these categories are presented. Table 4.5 summarizes the major strengths and weaknesses for each approach.

4.2.1 Search Engine Approaches

Developers tend to use web search get assistance in using and understanding APIs. There are several tools that help improve the search engine usage experience for developers. Mica [Stylos and Myers, 2006] is a web search tool targeted at helping programmers with using web resources to learn software libraries. Mica uses the Google Web APIs to retrieve its web search results. The initial search results are analyzed further in order to get programming-relevant information such as API methods, class and field names, so that the programming-oriented results will be later presented to the developers. Keywords related to programming such as Java class names and method names will be selected from the search results and listed in a tree structure. Assieme [Hoffmann et al., 2007] is designed to provide search for programming related tasks. Based on the search query, the results contain information grouped by packages, types and members from JavaDocs as well as pages with code examples which can be filtered by the packages specified by the user.

4.2.2 Code Recommendation Approaches

Recommender systems for software engineering provide developers with information within a particular context that assist them accomplishing particular tasks [Robillard et al., 2010]. A common example for recommender systems are code recommendation systems that assist developers while coding. For example, when the programmer tries to invoke a method on an object in Eclipse, a pop up window will show up, listing all the methods available for the class of that object. Bruch et al. [Bruch et al., 2009] proposed an intelligent code completion system (also known as Eclipse Code Recommenders) that proposes the most useful methods on top of the list. The recommender system proposed ranks the most useful methods on top thus saving

the time required by the developers to go through all the possible method calls. The method rankings are based on usage patterns that are extracted from a certain code base with different machine learning algorithms. Other approaches like PROSPECTOR [Mandelin et al., 2005] and XSnippet [Sahavechaphan and Claypool, 2006] support developers looking for code examples to accomplish an implementation task by mining and recommending sample code snippets from a code repository.

4.2.3 Tracking Based Approaches

These approaches are targeted at helping programmers finding related and relevant software artifacts by tracking the development activities of the developers (e.g., by visiting particular project files). Mylar [Kersten and Murphy, 2005] is a plug-in tool for Eclipse that shows programmers the relevance of a file to the active task in Java or AspectJ programs. The relevance depends on a degree-of-interest model. The model stores a value for each program element and when a certain part of the program has been selected or modified (e.g. a variable or a method), the corresponding value is increased. If it is not being visited for a certain amount of time, then the value is decreased. In the IDE, the files will be covered with different shades. The darkest shade means the highest interesting value.

Teamtracks [DeLine et al., 2005] collects interactive data from all the members in a development team to reveal navigation patterns. It provides functionality such as favorite classes and related items. In the favorite class view, less visited classes, methods and members will be hidden from the class hierarchy. When a class or a method is selected, the related items will be shown to programmers. Similarly to Mylar, the tool also uses a degree-of-interest model to capture the related artifacts.

NavTracks [Singer et al., 2005] targets understanding the high-level architecture of a software system. The authors discussed the problem with the conceptual organization of software elements, that is, they are usually organized by the hierarchical relationships between files such as class and subclasses. However such an organization fails to reveal other meaningful relationships between files. NavTracks groups the files according to their relevance by analyzing the browsing history of the user. Each file selection will be placed into an event stream which will be examined to generate associations for a browsing pattern. This pattern will be stored in a repository. The tool will recommend to the users the related classes when they select a class.

SmartGroups [Rothlisberger et al., 2011] is a tool similar to NavTracks. Compared to NavTracks, it analyzes evolutionary data related to version and dynamic data such as the number of invoca-

tions, memory usage or execution time in addition to the file browsing history. It further defines 3 different types of tasks: defect correction, feature implementation, and system understanding. For each type of task, information will be extracted from the data it collects in order to build groups. Artifacts that are closely related to a specific type will be in the same group. It also allows programmers to find groups in a specified range, e.g., within certain packages.

4.2.4 Visualization Approaches

Ishio et al. [Ishio et al., 2012] have implemented a plug-in tool in Eclipse for visualization of inter-procedural data-flow paths. A single functionality of a modern software system is usually realized through the cooperation of many modules. According to the authors, programmers often apply control-flow and data-flow techniques to analyze the dependencies between modules. The tool consists of two components: data-flow analyzer and visualizer. With the analyzer, links will be built between the assign statements and the reference statements of variables. The visualizer draws the data-flow paths graph to the user, when a method name or a variable name in the text editor is clicked. The intraprocedural contents of a method are represented as an edge between vertices in the graph.

FEAT [Robillard and Murphy, 2003] is a plug-in tool for Eclipse that also deals with the problem that program parts implementing a certain functionality of a system are scattered across different modules. The tool depends on concern graphs [Robillard and Murphy, 2002] that present relations of program elements. Each concern in a graph is only a name for the aspect of the program that is important to the user, and the leaf nodes can hold any classes, methods, or fields. Each leaf node will be analyzed to build a relation with other concerns in the graph. While browsing through concerns, the tool also shows the original code associated to a concern.

4.2.5 Documentation Approaches

Documentation still plays an important role in understanding APIs. However, going through documentation can be a tedious and a time-consuming task. The following approaches help solve some of the limitations of documentation.

Emoose [Dekel and Herbsleb, 2009] improves the API function documentation by forwarding the important information to the developers. The important information includes rules and caveats about how the function should be used. For example, some Java methods may need the programmer to invoke the super method when overriding. According to the paper, this

information (referred to as directives) is contained within long texts of documentation for particular functions. Programmers are believed to tend to overlook this information which will likely cause their program to fail at runtime. eMoose assists developers to correctly use particular method invocations documented in an API by means of directives from documentation. Based on the development context, the directives inform the developer with constraints, side effects, and other important information that are crucial for the correct usage of a particular method invocation.

CriticAL [Rupakheti and Hou, 2012] is a tool that is similar to eMoose. Its purpose is also to help programmers with API learning. However, CriticAL does not generate API usage rules from API documentation but rather depends on a predefined input of a set of rules by an experienced developer. The tool is based on the idea of a critic that is able to explain API element interactions, criticize inappropriate use, and recommend relevant API elements for usage. Within the source code viewer of an IDE, a developer can find critiques attached to certain lines of code that are written. By selecting a critic, a developer gets an explanation of the current context, alternative solutions and relevant API elements, and criticisms if the developer violates implementation conditions in a certain context.

JTourBus [Oezbek and Prechelt, 2007] is targeted at saving the developer the time needed to read extensive design documentation. The idea is to organize the design documentation into a set of tours. A tour goes in order through different stops and each stop marks an important part in the source code and a documentation fragment related to it. The approach is complemented by a tour browser (JTourBrowser) to enable the navigation of marked stops. The authors claim the following advantages. First, a tour is much easier to create. Secondly, the programmer can direct browse through the tour within the code, and since the programmer creates the tours at the same place where the code is located, there is no need to open a new editor window.

eXoaDocs [Kim et al., 2010] are example oriented API documents (eXoaDocs). The official JavaDoc is used for the search and the results are enhanced with indicators for the popularity of a method and along with several examples.

4.2.6 Code Query Approaches

With code query approaches, a developer can query source code repositories to get help with accomplishing a certain development task and / or to understand the existing code. ARA-BICA [Noguera et al., 2012] is a tool used to query Java code by using UML class and sequence diagrams. The goal is to find specific patterns in code such as different design patterns. UML

diagrams are used for querying because most programmers are well acquainted with these diagrams. JQuery [Janzen and De Volder, 2003] provides user-specific navigation by letting them query the code. The tool uses its predefined or user-defined query to generate an initial result in form of a tree in its viewer. Then the user can further explore each branch with new queries to expand the tree, i.e., users can explore further with their own concerns. The tree always maintains its structure, so that the users will not lose the overview. Strathcona [Holmes and Murphy, 2005] is an approach allowing the user to retrieve structure-matching relevant code from code repositories. This helps a developer to quickly learn the usage of a certain API from the code examples. CodeGenie [Lazzarini Lemos et al., 2007] is a tool for search and reuse of code from large scale code repositories. The tool can assist a developer to search and find existing code from repositories and then integrate it to the local task. MAPO [Zhong et al., 2009] allows the user to mine code repositories and return the usage pattern for a chosen method. A pattern describes how the method is used, usually together with other method calls.

4.2.7 Annotation Approaches

The idea of annotation approaches is to allow developers to share important information to help understanding the source code when visited by other developers. TagSEA [Storey et al., 2009] is a plug-in for the Eclipse IDE that allows developers to create semantically rich annotations. Its main goal is to help programmers revisiting a part of the program. The tool defines a waypoint simply by typing @tag at a comment location. A waypoint can be associated with one or more tags. The user can define hierarchical tags using Java-like syntax, e.g., @tag A.B. Thus, it is possible to provide tours like JTourBus for the user to travel through. Pollicino [Guzzi et al., 2011] uses collective code bookmarks the approach for passing knowledge between developers. The goal of the tool is to let the programmers document their discoveries while browsing through source code. This knowledge can then be passed to another programmer. Pollicino allows programmers to create bookmarks at any locations within the code to document their findings. Each bookmark can store different references to resources like comments, documents, websites, and other information that can assist other developers to be able to comprehend software artifacts in an easier way. Bookmarks can also be placed in groups to further increase the readability.

Approach	Strengths	Weaknesses
Search Engine	+ Can contain more information since it uses web based resources rather than depending on local code repositories. + Reduce the web search time by displaying only development-relevant results.	- Requires the programmer to search rather than automatically recommending the relevant coding elements. - The amount of information on the web for a particular closed-source commercial software system might not be sufficient.
Code Recommendation	+ Reduces the time needed to write code by recommending relevant code. + Reduces the time needed to seek documentation and tutorial resources for accomplishing a specific task.	- Recommendation is highly dependent on the mining algorithm and the available source code repository. - Can only be used during the implementation phase and not for planning for extensions. - Availability for large source code repositories for extensions for commercial applications might not be available.
Tracking-based	+ Reduces the time needed to get to relevant source code artifacts during the software development process.	- Requires team collaboration to be able to track the relevant source code artifacts. - Focused in general on all kinds of development activities and not extensibility in particular. - Cannot be used at the early stages of extension development.
Visualization-based	+ Improves the understanding of the developer by representing features and interactions of different software components. + More explicit representation of features and interactions between the different components rather than code.	- The visualizations generated are only used to visualize data flow or function calls. - Not suitable for extension developers since it assumes a white-box view on the source code. - Cannot support the expression of interdependencies between extension possibilities.
Documentation-based	+ Reduces the time required to find the right documentation for a development task or for understanding a software system. + Supports the extension developer during coding. + Can support software provider with the simplification and understanding of the design documentation.	- Cannot be used at the early stages of extension development. - Requires enhancing the documentation with metadata (that can also be linked to the source code) for recommendation. - Requires more effort for maintenance for the software provider.
Code Query	+ Saves development time by promoting code reuse through querying existing code repositories.	- Requires the availability of source code, which is not suitable for commercial applications. - The developer has to learn a new query language.
Annotation-based	+ Assists developers with understanding and browsing source code through means of rich annotations. + Better documentation support than plain text inline source code comments.	- Require the availability of source code (not suitable for commercial applications). - Maintaining these annotations with the source code can be an overhead for the developer. - Requires team collaboration for code annotation.

Table 4.5: Program comprehension approaches - Strengths and weaknesses

5 XPoints: Extension Interface Concept and Implementation

The previous chapters motivated the need for language mechanisms for defining extension interfaces that explicitly specify extension possibilities and development constraints for artifacts of different levels of abstraction (e.g., user interfaces, business processes, business objects, etc.). Moreover, a more convenient way for enforcing extension interfaces while reducing the development complexity and improving the maintainability is required while simplifying and supporting the consumption of these extension interfaces by the extension developer.

This chapter introduces XPoints, a generic approach expressing and enforcing extension interfaces for multilayered applications. An instantiation of XPoints for business applications based on the exemplary application introduced in Section 2.1 in Chapter 2 is described. Using XPoints as a foundation, a recommender tool for assisting extension developers with the extension development process is described.

5.1 The Approach in a Nutshell

XPoints is an approach for expressing and enforcing extension interfaces for multilayered applications. The approach consists of 3 main components that are depicted in Figure 5.1: the XPoints language through which extension interfaces are implemented, the XPoints compiler that generates the necessary code to enforce the extension interface, and the recommender tool that uses an XPoints extension interface as basis for guiding extension developers with the extension development process.

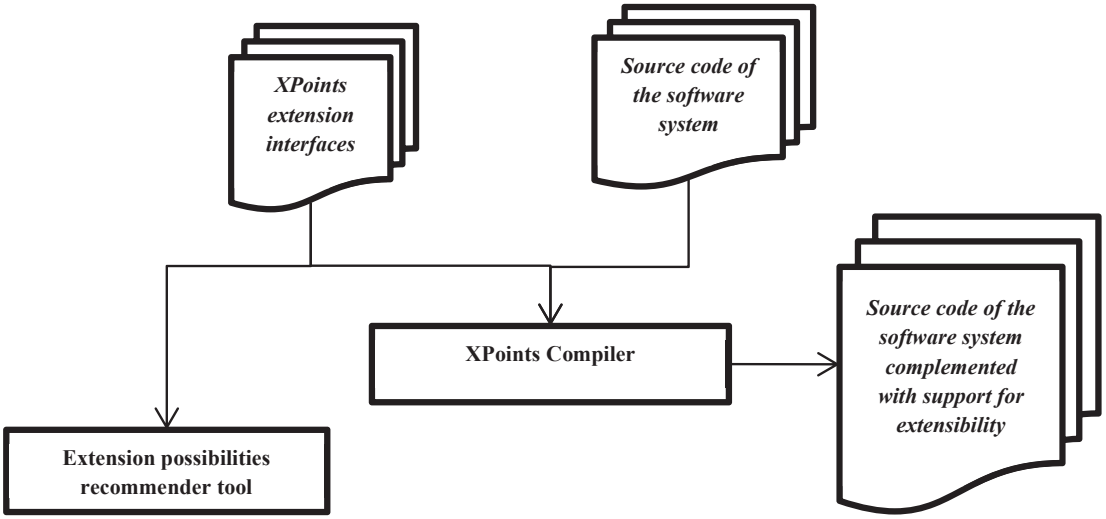


Figure 5.1: The approach in a nutshell

XPoints Extension Interfaces In an XPoints interface, the software provider separately specifies the extension possibilities as explicit first-class entities (i.e., using domain-specific constructs), interdependencies, supported extension types, and control constraints that are offered by the core software. XPoints interfaces are separately defined from the core software system. This provides better modularity and separation of concerns allowing the software provider to separately implement the core functional code and the extension interface. Several XPoints interfaces can be defined for a software system and hence several kinds of extension developers can be supported.

XPoints Compiler The XPoints compiler takes the XPoints interfaces and the source code of the core software as input, and enforces the extension interface by generating the required code for supporting extensibility. On the code level, the generated extension interface can be implemented using advanced techniques like design patterns, aspect oriented programming, plug-ins, etc. The generated code provides the coding elements (i.e., classes, interfaces, and methods) for the extension developers, that are necessary for implementing the extension. Moreover, the generated code is responsible for integrating, validating, and executing the extension code with the core software system.

Recommender Tool The IDE-based recommender tool uses the defined XPoints extension interfaces to guide an extension developer with the implementation of extensions. The tool uses each defined extension possibility along with their interdependencies and control constraints and visualizes them on the corresponding logical layer. The extension developer can browse through the logical layers and directly see the extension possibilities within the considered artifacts. Moreover, the extension developer can use the tool to bookmark the interesting extension possibilities and automatically generate an extension development project with the code skeletons that are necessary to implement the extension.

5.2 Language Concepts

The language concepts of XPoints can be summarized in the meta-model shown in Figure 5.2. Within an XPoints *extension interface*, several logical layers can be defined corresponding to the *logical layers* of the core software. Each layer consists of one or more *extensible artifacts* that are made available to an extension developer. This concept declares the base code artifacts that are extensible (e.g., classes, methods, components, etc.). Extension possibilities within each artifact are declared through *extension points*. Extension artifacts can be seen as containers of extension points. Each extension point has a type and a set of parameters, which specify the artifacts of the core software that are needed to generate the appropriate extension interface. With this concept, extension possibilities are declared as first class entities and are used to explicitly express extension possibilities.

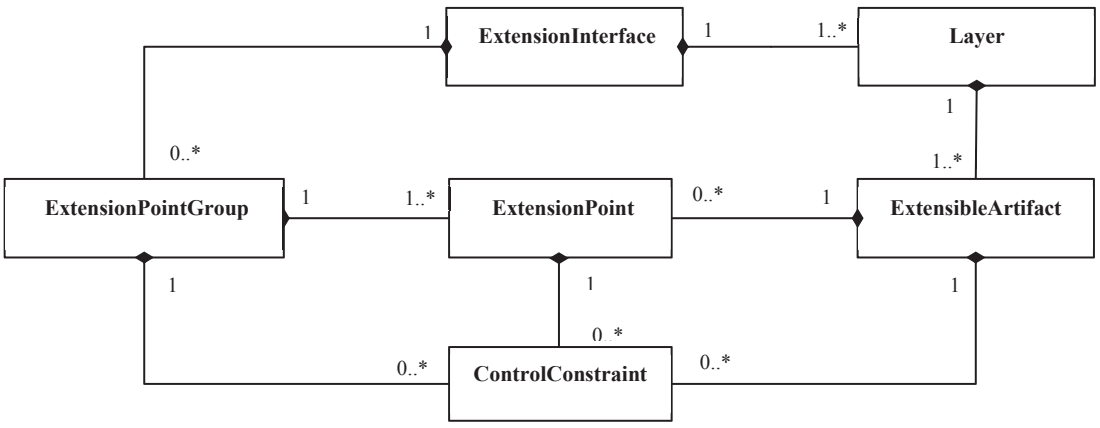


Figure 5.2: Language concepts of XPoints

Extension points can be further grouped within the same or a different layer via *extension point groups*. A group of extension points simply implies that the extension possibilities offered by these extension points are related. Groups can be used in XPoints with or without control constraints. The *control constraints* on extensible artifacts and extension points restrict the access, visibility, and usage of the base application artifacts by the extension developers. The purpose of this concept is to provide fine grained access control of the extensions to the resources of the software system.

The control constraints can also be defined on a group to control how an extension realizing the member extension points within a group should be implemented. In some extension scenarios, where an extension spans several layers (e.g., user interface and business object), a valid extension can require the implementation of several extension points from the same or multiple layers.

5.3 Instantiation of the Concepts

In Section 2.1 in Chapter 2 an exemplary business application was described. In the following an instantiation of the language concepts of XPoints for the exemplary application is presented. The underlying classes of the user interface and business process artifacts are assumed to be implemented in Java. The instantiated concepts only present example constructs that can exist in business applications (i.e., the extensible artifacts, extension point types, etc.). However, in other multilayered application domains, the concepts can be instantiated accordingly to cover all possible constructs.

5.3.1 Supported Scenarios

In the following, a selection of extension scenarios for artifacts of the different logical layers for the purpose of this instantiation is presented. The scenarios present a set of typical extension scenarios that are foreseen by a business software provider. However, in real business software applications more extension scenarios can exist.

Business Objects In this instantiation business object data extensions and logic extensions are supported. Data extensions are meant for extending the data model of the business object. These extensions take place when new attributes of certain data types are added. In real business applications the software provider can restrict the number or types of the attributes added to the business objects, e.g., due to design, performance, and space considerations. Logic extensions are meant for allowing the extension developer to extend existing logic (e.g., discount calculation in a sales order business object) or to introduce new custom logic.

User Interface User interfaces are assumed to be implemented following the model-view-controller [Krasner and Pope, 1988] pattern. Three types of extension scenarios are supported in this instantiation: user interface extensions (view), data extensions (model), and logic extensions (controller and model). User interface extensions are meant for allowing the extension developer to introduce new user interface elements (e.g., buttons, text fields, forms, panels, etc.) to the existing user interfaces or completely new user interfaces to the software system. Data extensions are allowed to introduce extensions to the underlying model objects that are associated with a user interface. Logic extensions enable the extension of the logic of the underlying model objects as well as to the controller of the user interface.

Business Process A business process presents the set of business-related activities and their logical sequence that are supported by the software system [Aguilar-Saven, 2004]. In this instantiation extensions are supported for activities and messages while considering the events and decisions that can take place during the execution of the process. It is assumed that the business process elements are modelled using BPMN on the modeling level and implemented Java classes on the code level.

5.3.2 Informal Semantics

Based on the previously described extension scenarios for each of the logical layers, the language constructs of this instantiation are described. These constructs are used for explicitly expressing the extension possibilities. The grammar of this instantiation is listed in Figures A.1 and A.2.

Extensible Artifacts The extensible artifacts supported by the instantiation for business applications are Java business object classes, Java Swing classes, and BPMN business process models (cf. Figure 2.1, Figure 2.2, Listing 2.1, and Listing 2.2).

Extension Points Extension points have a type representing an extension possibility, a unique identifier, and an optional reference to a defined permission set that act as a control constraint for the extension point (permission sets are discussed later in this section). In the following the supported types of extension points on each of the logical layers are described.

On the business object layer, the following types are supported. *afterConstructor* allows defining extension-specific logic to be executed after the constructor of a business object.
beforeMethodCall and *afterMethodCall* enable the definition of extension-specific logic before or after a certain method is called. These constructs require as an input the constructor / method signature. *allowNewBOLogic* enables the definition of new business logic, e.g., a new custom method that is not associated with the core logic of the business object. This construct expresses that the extension developer is allowed to extend the set of methods of a business object by a new custom method.

afterBOAttributeChange enables defining extension-specific logic to be executed after the value of a certain business object attribute changes. As an input this construct requires the name of the attribute to be monitored after which the extension code will be executed. *allowBOAttributes* enables the extension of a business object with a maximum number of attributes with a certain type. As an input this construct expects an integer representing the maximum number of attributes (or * for no limits) and the type of attribute. Listing 5.1 summarizes the supported constructs for business objects.

```
<construct> <ep_id> <parameters> <permission=pset_id>?
afterConstructor   <ep_id>   (<constructor_name>)
beforeMethodCall <ep_id> (<method_signature>)
afterMethodCall    <ep_id> (<method_signature>)
allowNewBOLogic <ep_id>
afterBOAttributeChange <ep_id> (<attribute_name>)
allowBOAttributes <ep_id> (< attribute_type >,  <number>)
```

Listing 5.1: Extension point types for the business object layer

On the user interface layer, the following extension point types are supported. *allowUIComponent* defines the possibility to extend an existing user interface component through the addition of new user interface elements (e.g., allowing the addition of a new button or a text field to an existing form). As an input the construct expects the type of component that is allowed to be added and a reference to the parent form. *beforeForm* and *afterForm* enable to extend the form flow of a certain application; it can be used to insert a custom user interface (e.g., a form) before or after a certain displayed interface.

The *beforeForm* construct requires the reference to the form, the display method of the form, and the dispose method of the previous form (or *null* if it does not exist). The *afterForm* construct requires the reference to the form, the dispose method of the form, and the display method of the previous form (or *null* if it does not exist). *beforeUIEventHandler* and *afterUIEventHandler* allow defining custom logic to be inserted before or after a certain event handler is called. The constructs expect the type of the event raised and a reference to the event handling method. *allowUIAttributes* enables to extend the data model of a user interface with a maximum number of attributes of a certain type. This construct requires an integer representing the maximum number of attributes (or * for no limits) and the type of the attribute. Listing 5.2 summarizes the supported constructs.

<construct> <ep_id> <parameters> <**permission**=pset_id>?
allowUIComponent <ep_id> (<type_of_component>, <parent_form_name>)
beforeForm <ep_id> (<form_name>, <form_display_method> <prev_form_dispose_method>)
afterForm <ep_id>)(<form_name>, <next_form_display_method>, <form_dispose_method>)
beforeUIEventHandler <ep_id> (<event_type>, <handler_method)>)
afterUIEventHandler <ep_id> (<event_type>, <handler_method>)
allowUIAttributes <ep_id> (< attrib_type >, <number>)

Listing 5.2: Extension point types for the user interface layer

On the business process layer, the following extension point types are supported. *beforeActivity*, *afterActivity*, and *parallelActivity* declare the possibility of extending an activity before, after, or in parallel to its execution. *beforeEvent* and *afterEvent* allow the extension developer to insert an extension before or after an event. *afterDecision* defines the possibility of inserting an extension after a certain decision result from a gateway. All of the constructs expect as an input an identifier of the BPMN process element, a reference to the underlying class that realizes the element, and a reference to the main execution method. Listing 5.3 summarizes these constructs.

```
<construct> <ep_id> <parameters> <permission=pset_id>?
beforeActivity <ep_id> (<act_id>, <class_name>, <main_exe_method>)
afterActivity <ep_id> (<act_id>, <class_name>, <main_exe_method>)
parallelActivity <ep_id> (<act_id>, <class_name>, <main_exe_method>)
beforeEvent <ep_id> (<event_id>, <class_name>, <main_exe_method>)
afterEvent <ep_id> (<event_id>, <class_name>, <main_exe_method>)
afterDecision <ep_id> (<dec_id>, <dec_value>, <dec_exe_method>)
```

Listing 5.3: Extension point types for business process layer

Control Constraints In this concrete instantiation, control constraints are realized as *permissionset*s which restrict the access, visibility, and usage rights of the base application resources (i.e., they support the principle of least privilege [Mayfield et al., 1991]) to the extension developer. The sets can be defined on the extensible artifact level (i.e., container level) and / or on the extension point level. Extension points inherit the permission set of their container. An extension point that declares its own permission set, can further override or refine the permission set of its container.

For the business object and user interface layers, permission sets support method and attribute permissions of the extensible artifact. Attributes can be declared as either *READ, WRITE, READWRITE* to restrict the read / write operations to the attribute or *HIDDEN* to make the attribute unavailable for any kind of read/ write operation for the extender. A permission is declared using the *attributepermission* construct that expects a reference to the attribute (or * if the permission is to be applied to all attributes of the extensible artifact) and a permission.

Methods can be declared as *CALLABLE* to be available for being called by the extension developer or *HIDDEN* to be prevented from being called by the extension developer. The *methodpermission* construct is used for defining a method permission, and expects as an input a reference to the method (or * if the permission is to be applied to all methods of the extensible artifact) and a permission. Extensible artifacts that do not declare a permission set get the default modifier offered by Java.

The permission sets defined on the business process layer define the visibility of the business process elements (activity, tasks, lanes, and data are currently supported). Each element can be declared as *HIDDEN* or *VISIBLE* for an extension developer. These constructs however only have an effect on the business process model (i.e., the marked elements of the BPMN model will be either hidden or visible for the extension developer). The constructs *datapermission*,

activitypermission, *taskpermission*, and *lanepermission* are used to declare permissions for data, activity, task, and lanes respectively. The constructs expect the identifier of the BPMN element and a permission.

Groups and Control Constraints Groups are defined using the *group* construct which expects a list of extension point identifiers. Adding a control constraint to the list will enforce the constraint on the group. The instantiation supports one control constraint on groups, *ExtendAll*, requiring a valid extension to provide an implementation for all extension points within the group. This is essential for enforcing cross-layer or inter-layer implementation constraints of extensions. For example, it can be required that an extension developer extends the data model of the business object when adding a new input text field to a user interface.

Hello World Listing 5.4 shows an example of a very simple extension interface on the business object layer. This interface declares the *SalesQuote* business object as an extensible artifact with the extension point *EXP1* of type *afterMethodCall* (Line 4) that allows the extension developer to insert some custom logic after the execution of the method *sendToApproval()*.

The example in Listing 5.4 shows a control constraint for *EXP1* in the form of a permission set *per* (Lines 6–9) that allows the extension developer *READ* access to the *total* attribute and hides all methods of the class *SalesQuote*.

```
1 extensioninterface Example{
2  layer BusinessObject{
3    extensibleartifact "com.sap.SalesQuote"{
4    afterMethodCall EXP1 ("void sendToApproval()") permission=per;
5
6    permissionset per{
7     attributepermission("double total ",READ);
8     methodpermission("*",HIDDEN);}}}
9          }
```

Listing 5.4: XPoints interface example

5.4 Generation of the Enforcement Code

The XPoints compiler plays an important role with the enforcement of extension interfaces. As an input the compiler expects the XPoints extension interfaces and the source code the core software. The extension interface generation strategy depends on the implementation of the compiler. In this instantiation the compiler enforces the extension interface through the generation of a code framework that employs Java interfaces, the proxy design pattern, and aspects to support a plug-in like extensibility mechanism. However, it is also possible to use other techniques for the generation and enforcement of extension interfaces (i.e., by implementing a different XPoints compiler).

The generated code framework consists of two parts: an extension-developer specific part and an extensibility-supporting part. The extension-developer-specific part is responsible for providing the extension developer with an “entry point" for developing his extension and providing a controlled access to the underlying resources of the core software. The part of the software provider is responsible for discovering extensions at runtime, loading the extensions, validating the extensions, and executing the extensions. The following describes the enforcement of the extension interface. The simple example shown in Listing 5.4 will be used along to explain the generated code framework.

5.4.1 Extension Developer-Specific Code

For the extension developer, the extension point is meant as an entry point for developing an extension. For each extension point a Java interface and a proxy class is generated by the compiler. The Java interface contains a set of methods that must be implemented to integrate the new functionality with the core software. The generated methods in the interface depend on the type of the extension point.

The generated proxy class controls the access to the attributes and methods of the underlying class that contains the extension possibilities. The generation depends on the defined permission sets that are enforced on the extension point (i.e., on the extensible artifact or the extension point level). The control is done by generating getter methods for attributes declared as read-only, setter methods for attributes declared as write-only, and both getters and setters for attributes declared as read/write. Methods that are declared as callable will get a method with an identical signature in the proxy class that simply forwards the call to the method in the core class. If no permission sets are defined, then the default modifiers of the class, methods, and attributes are

used to generate the proxy class. The proxy class is initialized by the extensibility-supporting code framework. An object of the proxy class is also passed to the class implementing the interface by the extensibility-supporting code framework.

Listing 5.5 shows the generated interface and proxy class for the exemplary XPoints interface in Listing 5.4. Lines 1–4 show the interface and Lines 6–19 show the code of the generated proxy class. The interface consists of an *init()* method that takes as a parameter a proxy object. The proxy class contains a getter method *getTotal()* that returns the *double total* attribute of the *SalesQuote* class. The proxy class is initialized by the extensibility-supporting code framework. An object of the proxy class is also passed to the class implementing the interface by the extensibility-supporting code framework.

```
1  public interface ExampleEXP1Interface {
2    public void init (EXP1Proxy p1);
3    public void yourEXP1Logic();
4  }
5
6  public class ExampleEXP1Proxy {
7
8          private SalesQuote salesquote ;
9
10   // Proxy object is created by the extensibility framework
11         public ExampleEXP1Proxy(Salesquote salesquote){
12           this . salesquote = salesquote ;
13         }
14
15         // Getter method for the total attribute declared as READ only.
16         public double getTotal (){
17           return salesquote . getTotal ();
18         }
19 }
```

Listing 5.5: Generated Java interface for the XPoints example

5.4.2 Extensibility-Supporting Code

The generated extensibility-supporting code is used for loading, initializing, validating, and executing the extensions. To avoid mixing functional code and extensibility-supporting code, aspects are used to complement the code of the core software with these functionalities.

Loading the Extension Extensions are loaded into the core software using the class loader mechanism [Liang and Bracha, 1998]. An extension developer is expected to compile his extension code and place it in a particular directory with the binaries of the core software. The generated aspect code contains inter-type declarations that enrich the core classes with data structures and methods that are necessary to load the extensions in a plug-in like fashion. The loading process of an extension takes place during runtime before the instantiation of the corresponding extensible class. Listing 5.6 shows an excerpt of the generated Java code for loading an extension that is injected to the core class by the aspect code.

```
1  ArrayList<ExampleEXP1> EXP1Extensions;
2
3  private void loadExtension(){
4   try{
5           EXP1Extensions = new ArrayList<ExampleEXP1Interface>();
6           //Discover extensions of this type
7           File [] extensions = discoverExtensions("/plugins/");
8
9           for(File extension: extensions){
10
11          //Load the extensions
12          URL url = extension.toURI().toURL();
13          URLClassLoader loader = new URLClassLoader(new URL[]{url});
14          ServiceLoader<ExampleEXP1> loader = ServiceLoader.load(ExampleEXP1Interface.class, loader);
15
16          for (ExampleEXP1Interface x: loader) {
17                  // Validate and execute the extensions
18                  validateExtension(x); //Throw an execption if not valid
19            executeExtension(x); // Handle the execution logic of the extension
20           }
21          }
22  }
23 catch(Exception e){
24          //HANDLE ERRORS / DISPLAY WARNING
25  }
26 }
```

Listing 5.6: Generated Java code for loading an extension

Initializing the Extension Once the extension code is loaded, the classes implementing the interfaces are passed objects of the proxy classes. The generated aspect code contains inter-type declarations that enrich the core classes with the necessary helper methods to support the proxy classes. This is only necessary to override the default Java modifiers of attributes and methods that are not accessible by the proxy class due to their implemented modifiers in the core class. For example, if an attribute in the core class is declared as *private* but marked as *READ* in a permission set, a getter method will be generated in the core class that returns a copy of the attribute to the proxy class. The proxy class will use this method to return a copy of the attribute to the extender.

Validating the Interdependencies In this instantiation the extension points declared within a group constrained with the *ExtendAll* constraint will require a valid extension to provide an implementation of all extension points within this group. The generated extensibility-supporting code does a simple check on this constraint by checking if the provided extension provides classes that implement all interfaces that correspond to the extension points of groups with this constraint. This check takes place once during the loading of the extension. If a complete implementation is provided the extension is initialized and executed, otherwise the extension will not be loaded and a warning is issued to the user.

Executing the Extension The execution is supported by the generated aspect code. The place and the time of the execution of the extension code are determined by the pointcut of the aspect code and depend on the type of the extension point. For example, extension point *EXP1* in Listing 5.4 has the type *afterMethodCall*. The generated aspect code in this case will execute the extension code after a call is made to the *sendToApproval()* method. Listing 5.7 shows an excerpt of the aspect code generated for running extensions. In the first part, the join point is specified by the pointcut *extension()* that selects the calls to the method *sendToApproval()* in the class *com.sap.SalesQuote*. In the next part the advice code runs the extensions after the defined pointcut.

```
1 //Join point method call
2 pointcut extension (): call (void com.sap.SalesQuote.sendToApproval());
3
4 //Advice after the joinpoint
5 after(com.sap.SalesQuote x): extension () && target(x){
6 if(x.EXP1Extensions != null){
7             for(ExampleEXP1Interface i : Extensions ){
8                                 i.yourEXP1Logic();
9           }}}
```

Listing 5.7: Aspect code for executing an extension

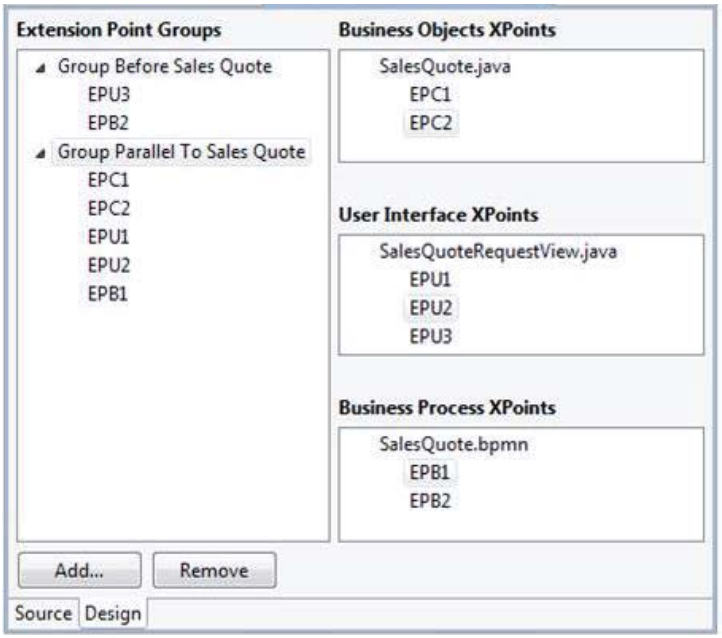


Figure 5.3: XPoints extension point group editor

5.4.3 Implementation

The general concepts of XPoints and the instantiation for business applications are implemented as a domain-specific language using XText [Eysholdt and Behrens, 2010] in Eclipse. The software provider has two possibilities to implement an XPoints extension interface. The first possibility is to use the generated code editor from XText and directly code XPoints interfaces in a script form as in Listing 5.4. This option is recommended for advanced developers who are familiar with the syntax of XPoints. The second possibility is to use an annotation-based tool for XPoints in Eclipse. With this tool, the developer can annotate the extensible artifacts with the constructs of XPoints using a drag-and-drop interface. A corresponding XPoints extension interface script will be generated. Figures 5.4, 5.6, and 5.5 show a screenshot of the XPoints business object, user interface and business process annotation tools respectively. Furthermore, using a group editor (Figure 5.3), the software provider can define extension point groups with the defined extension points. The implementation of the annotation tool supports the STP BPMN process editor [Eclipse Foundation, 2014c] for eclipse, WindowBuilder Java swing editor [Eclipse Foundation, 2014b], and the Eclipse Java code editor. The generation of the interfaces, proxy classes, and aspects (i.e., the XPoints compiler) is implemented using XTend. For weaving of the aspects and the core software AspectJ [Kiczales et al., 2001] is used.

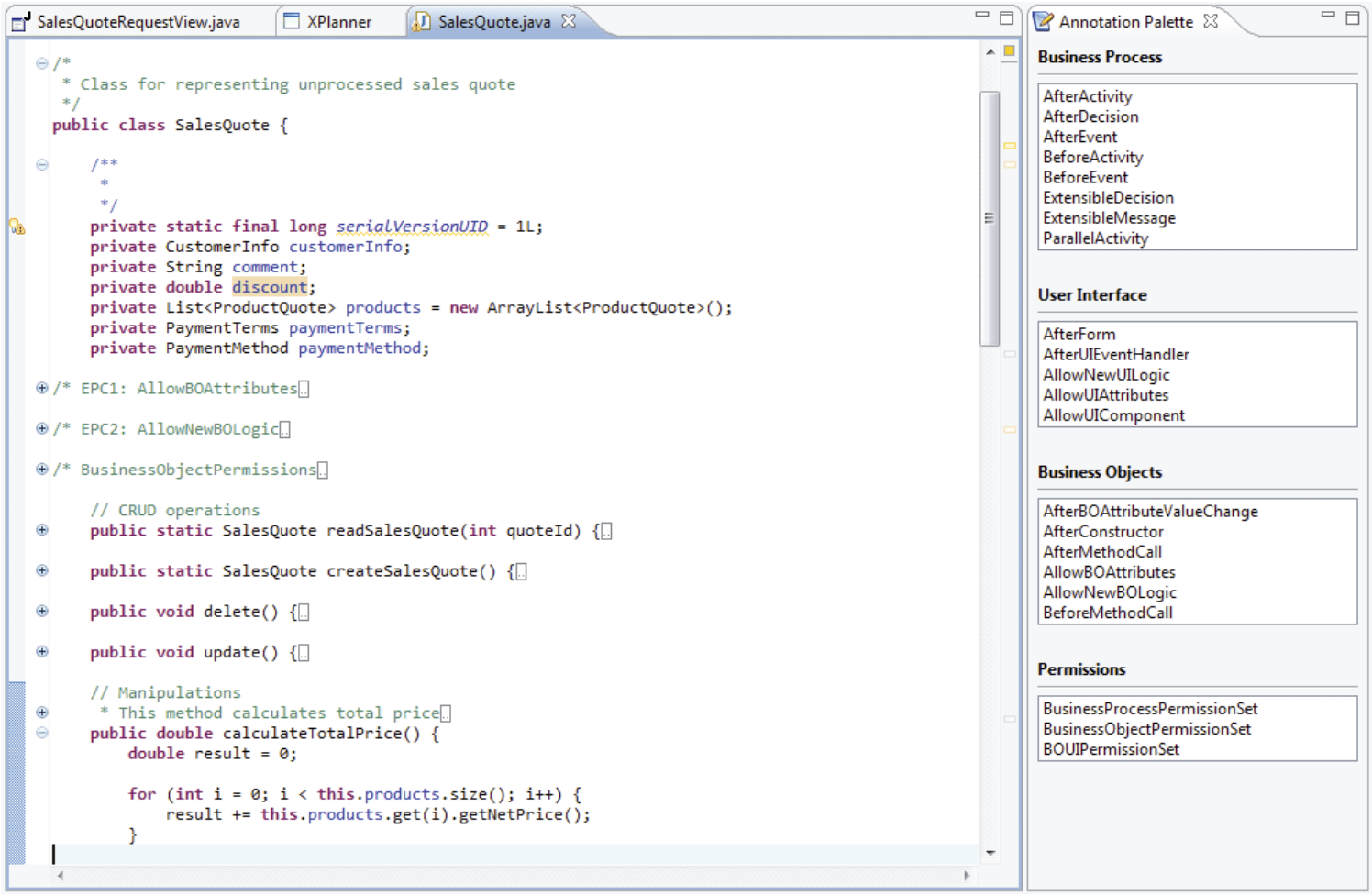


Figure 5.4: Example XPoints business object annotation

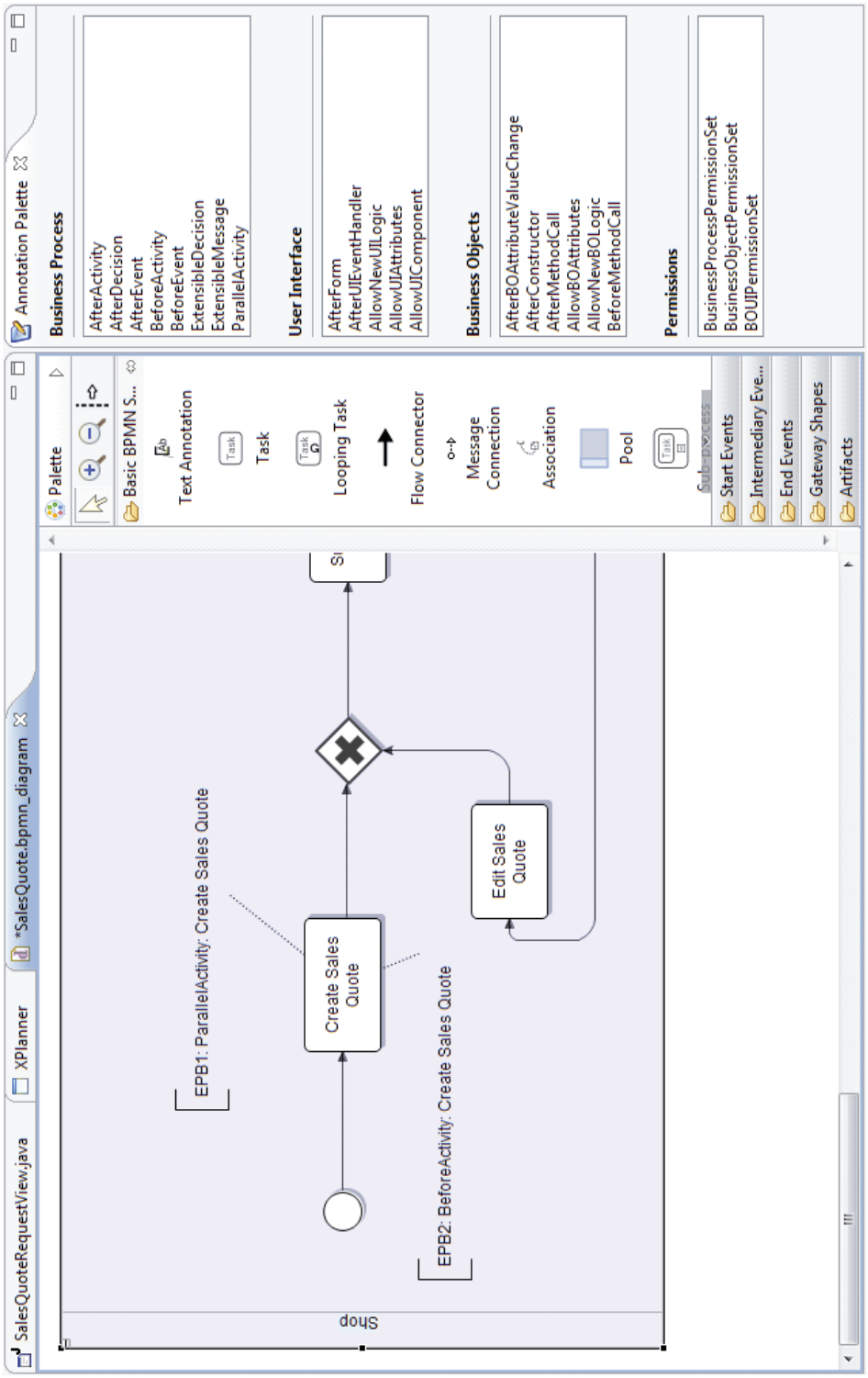


Figure 5.5: Example XPoints business process annotation

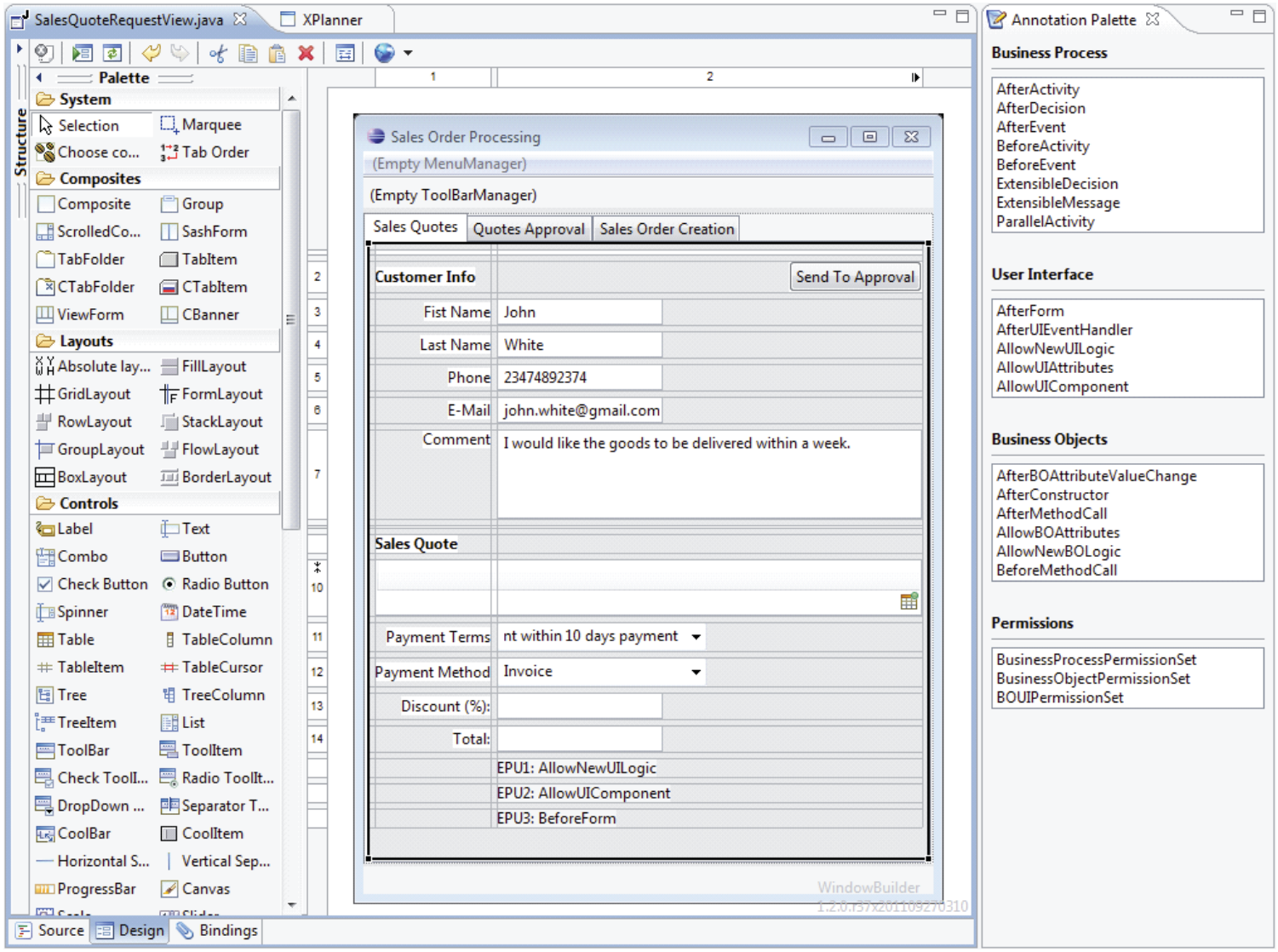


Figure 5.6: Example XPoints user interface annotation

5.5 Guiding the Extension Developer

Based on the XPoints specification, a tool is implemented to support extension developers with the extension development process. The tool aims at providing an explicit visual representation of extension possibilities of the extensible artifacts of the different logical layers. The tool is implemented as a plug-in for Eclipse and supports the visualization of extension possibilities of business object Java classes, STP BPMN business processes [Eclipse Foundation, 2014c], and WindowBuilder Java Swing user interfaces. Using the tool, the extension developer can bookmark extension possibilities of interest, and generate a code skeleton of an extension project. Figure 5.7 shows a screenshot of that tool. The figure is marked with Regions 1-6 and the following explains the main components of that tool.

Visualizing Extension Possibilities The tool depends on the defined extension points, permission sets, and groups to visualize the extension possibilities. As an input, the tool gets the XPoints interface for visualizing the extension possibilities. The extension developer has the possibility to first select the logical layer of interest in Region 1, i.e., user interface, business object, and business process to start with. The offered extensible artifacts within this layer will be listed in Region 2 and the extension developer can browse through the artifacts. Extension points defined within artifacts with visual elements like user interfaces and business processes are visualized as they will appear in an editor. This is meant to help the extension developer quickly identify the intended extension possibility without having to read a lot of documentation text. The current implementation of the tool depends on the source code of the extensible user interfaces and the business process models to visualize the extension possibilities. However, the implementation can be extended in the future to take as an input an encrypted version of the source code (i.e., to protect the source code).

Cross-Layer Browsing Once an extensible artifact of interest is identified, the tool visualizes the interdependent artifacts within the same and different logical layers by analyzing the groups containing the extension points. In Figure 5.7, Regions 3 and 4 show the underlying business process and business objects that are associated with the selected user interface. Artifacts related within the same logical layers will be visualized as icons in Region 2.

Bookmarking of Extension Possibilities The tool provides the facility for the extension developer to bookmark extension possibilities of interest by simply dragging an extension points from Regions 2, 3, and 4 and dropping it to the favorites list in Region 5. Dragging extension points that have interdependencies with other extension points will automatically drag all other related extension points along.

Generating the Extension Project Out of the list of bookmarked extension points, the extension developer can create development tasks that are associated with the selected extension points and place them in a to-do list in Region 6. The tasks can be named according to the preference of the extension developer. Once a task is clicked, an extension project with the code skeleton required to implement a valid extension for the bookmarked extension points is generated.

Integrating the Extension with the Core Software After the extension developer has implemented the required interfaces for his extension, the project is exported as a Java archive and placed within the plug-ins folder of the core software. During runtime, the core software will load and execute the extensions.

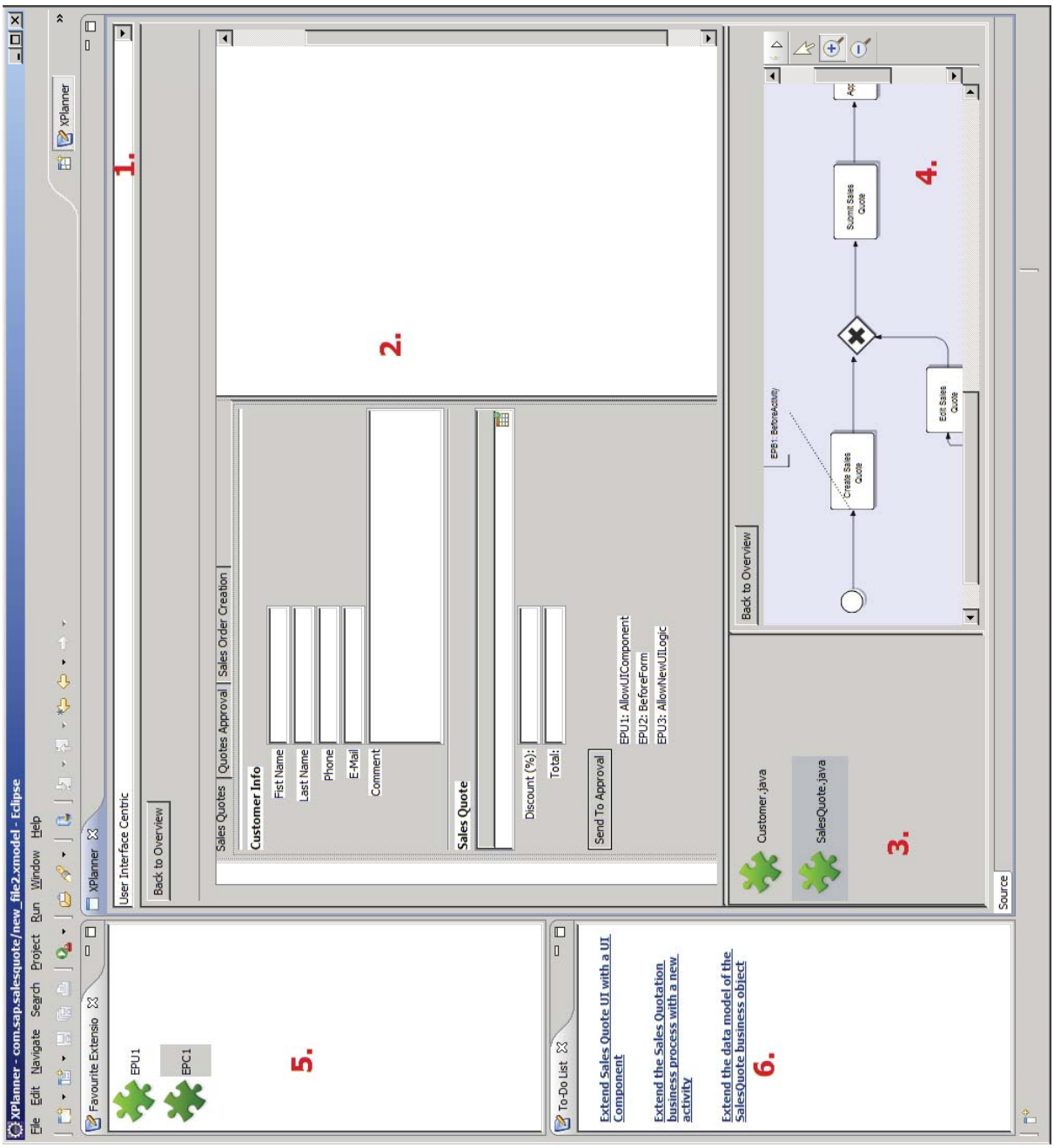


Figure 5.7: Recommender tool for the extensibility of multilayered applications - ©2013 IEEE

5.6 Summary

This chapter presented XPoints, a language and an approach for the specification and enforcement of extension interfaces of multilayered applications. In XPoints extension possibilities and their interdependencies are declared as domain-specific first-class entities. Moreover, XPoints provides the possibility for controlling access of extensions to the resources of a core software system. XPoints interfaces are defined separately from the core implementation of the software system allowing multiple extension interfaces to coexist to support different kinds of extension developers with different constraints.

Based on XPoints, a recommender tool for guiding the extension developer with implementing cross-layer extensions is proposed. Using that tool, the extension developers can identify the extension possibilities and their interdependencies directly on artifacts from any layer. In addition, the tool supports the extension developer by generating the necessary coding stubs for implementing an extension.

6 Evaluation of the Approach

Finding the best strategy for evaluating new software engineering methods and tools is a challenging task. Several evaluation strategies have been recommended in literature for evaluating new approaches in software engineering [Kitchenham et al., 1997, Pfleeger, 1995, Zelkowitz and Wallace, 1998, Shull et al., 2007, Juristo and Moreno, 2010]. This chapter presents a threefold evaluation of the proposed approach. First, a qualitative part in which a case study is described through which the approach is applied and the advantages of the approach are discussed. Second, the approach is compared with the related work of the state-of-the-art approaches presented in Chapter 4 in terms of satisfaction of the requirements defined in Chapters 2 and 3. Third, a quantitative part is presented, which involves a user study of developers implementing an extension interface using XPoints and Java for an open source business software.

6.1 Case Study

This case study shows the advantages of XPoints in comparison to the state-of-the-art approaches in realizing requirements for extension interfaces of multilayered applications. Based on the example presented in Chapter 2, two scenarios are considered in which extension interfaces are implemented for two kinds of extension developers. For each scenario, the extension possibilities and the corresponding constraints are described. In the first part of the case study, the implementation of both scenarios is done using XPoints and the extension interface enforcement is described. In the second part, the usage of the recommender tool to implement extensions for the implemented extension interface is described. In the last part, a discussion on the advantages of XPoints and the recommender tool for extension development over the related work is presented.

6.1.1 Scenario 1: External Developer

In this scenario external developers are considered. The external developers are allowed to perform some custom logic before the *SalesQuote* business object is saved, but they are not allowed to modify any attribute. Furthermore, the external developers are allowed to read all attributes of the *SalesQuote* and display a message in a label with the outcome of their logic in the *SalesQuotation* form. As a constraint, the external developers must not see any method of the *SalesQuote* business object.

Listing 6.1 shows the specification of the extension interface in XPoints for this extension developer group. This extension interface spans two layers (business object and user interface). Line 1 declares the external developer extension interface. Line 2 declares the business object and Line 11 declares the user interface as the container of extensible artifacts. In this example, there are two artifacts declared as being extensible; *com.sap.SalesQuote* and *com.sap.SalesQuoteForm* (Line 3 and Line 12). Extension possibilities are defined through extension points. Each extension point has a type, a unique identifier (e.g., EPBO1), a set of parameters, and an optional reference to a permission set.

Line 4 shows the declaration of the extension point *EPBO1* of type *beforeMethodCall* and Line 13 shows the extension point *EPUI1* of type *allowUIComponent*. The parameters of *EPBO1* declare the extension possibility before the method *saveSalesQuote()*. The parameters of *EPUI1* specify that the extension developer can add a new component of type *JLabel* on the parent component *salesQuotePanel*. The *SalesQuote* business object artifact has a reference to the artifact permission set *default1* (Lines 6-9). This permission set declares that all attributes should be available only in *READ* mode and that all methods should be hidden to all extension points within the artifact. The *SalesQuoteForm* user interface artifact has a reference to the artifact permission set *default2* (Lines 16-19). This permission set declares all attributes and methods to be hidden from the extension developer.

The last part of the interface (Line 21) declares a group called *extensionScenario* that contains two extension points *EPBO1* and *EPUI1*. This implies that the two extension points are related. At the end of the group declaration, an *ExtendAll* constraint is declared, which means that a valid extension must implement both extension points.

```
1  extensioninterface   externaldeveloper {
2  layer  BusinessObject{
3    extensibleartifact   "com.sap.SalesQuote"  permission=default1{
4    beforeMethodCall EPBO1 ("void saveSalesQuote()");
5   }
6   permissionset   default1 {
7    attributepermission ("*", READ);
8    methodpermission("*",HIDDEN);
9   }
10 }
11 layer  UserInterface  {
12   extensibleartifact   "com.sap.SalesQuoteForm" permission=default2{
13              allowUIComponent EPUI1 ("JLabel","salesQuotePanel");
14            }
15
16  permissionset   default2   {
17     attributepermission   ("*", HIDDEN);
18     methodpermission("*",HIDDEN);
19  }
20 }
21  Group extensionScenario{(EPBO1,EPUI1),ExtendAll};
22 }
```

Listing 6.1: Extension interface in XPoints for the external developer group

6.1.2 Scenario 2: Internal Developer

In this scenario a group of extension developers who are working on the provider side to realize industry-specific solutions on top of the standard application is considered. These extension developers are allowed to define extensions that span multiple layers. More specifically these extension developers are allowed to extend the business process after the approval step, e.g., to realize an additional approval. Thereby only some relevant business process activities should be made visible while hiding the rest of the process details. Further, these extension developers are also allowed to extend the *SalesQuote* business object with new attributes and extend the business object logic after the sales quote has been sent for approval. The extension developers should also be allowed to read and write values to the attributes *products* and *customerInfo* as well as to call the method *calculateTotal*. Listing 6.2 shows the XPoints extension interface for this scenario.

In this extension interface, there are three layers defined (business object, user interface, and business process). In the business object layer (Lines 2-10), the *SalesQuote* business object is declared as extensible. The permission set *defview* expresses that the extension developer cannot call any method, and has read-only access to all attributes (Lines 11-14). There are two extension points defined (Lines 4-5) *EPBO1* and *EPBO2*, which declare two extension possibilities to allow the addition of a maximum of 10 new attributes of type *String* (that will be persisted in the database) and to extend the logic after the *sendToApproval()* method respectively. *EPBO2* has a reference to the permission set *intdev* (that refines the permission set of the parent, i.e., *defview*), which allows read / write access to the attributes *products* and *customerInfo*, and allows the method *calculateTotal()* to be called (Lines 6-10).

The next part of the interface (Lines 15-22) declares the *SalesQuoteForm* as extensible with the *allowUIComponent* extension possibility *EPUI1* that allows the extension developer to add a new panel in the sales quote approval panel. The artifact permission set *defview* hides all methods and attributes of the class from the extension developer. The following part (Lines 23-35) defines the business process layer and the sales quotation business process as an extensible artifact. The *EPB1* extension point declares the possibility of adding an activity after the sales quote approval activity and the underlying class *SQProcessing* that processes the logic of the activity through the method *approveQuote()*. The *defview* permission set declares the whole lane that contains the sales quotation business process as hidden (Lines 33-35). The permission set *view* referenced by *EPBP1* makes the main activities of the business process visible to the extension developer.

Similarly to the previous scenario, the last part of the interface (Line 36) declares a group called *ExtensionScenario* that contains three extension points *EPUI1*, *EPBP1*, and *EPBO2*. This requires then the developer to implement all three extension points.

```
1  extensioninterface   internaldeveloper {
2  layer  BusinessObject {
3    extensibleartifact   "com.sap.SalesQuote"  permission=defview{
4            allowBOAttributes EPBO1 ("String",10);
5            afterMethodCall EPBO2 ("void sendToApproval()") permission=intdev;
6            permissionset  intdev{
7             attributepermission  (" products ",READWRITE);
8             attributepermission  (" customerInfo ",READWRITE);
9             methodpermission (" calculateTotal ",CALLABLE);
10           }}
11   permissionset  defview  {
12           attributepermission   ("*",READ);
13           methodpermission("*",HIDDEN);
14   }}
15 layer  UserInterface{
16    extensibleartifact   "com.sap.SalesQuoteForm" permission=defview{
17    allowUIComponent EPUI1 ("JPanel","approvalPanel");
18   }
19   permissionset  defview{
20    attributepermission ("*", HIDDEN);
21    methodpermission("*",HIDDEN);
22   }}
23 layer  BusinessProcess{
24    extensibleartifact     " sales_quotation .bpmn" permission=defview {
25     afterActivity  EPBP1 permission
26                              = view("Approve Sales  Quote","com.sap.SQProcessing","void  approveQuote ()");
27    permissionset  view{
28      activitypermission (" Create  Sales  Quote",VISIBLE);
29      activitypermission (" Approve Sales  Quote",VISIBLE);
30      activitypermission (" Send Sales  Quote",VISIBLE);
31    }}
32
33   permissionset  defview{
34    lanepermission(" Sales  Quotation  Processing ",HIDDEN);
35   }}
36 Group ExtensionScenario {(EPUI1,EPBP1,EPBO2),ExtendAll};
37 }
```

Listing 6.2: Extension interface in XPoints for the internal developer group

6.1.3 Enforcement of the Extension Interface

The code generated from an XPoints interface consists of three main parts; a generated Java *interface* acts as an entry point for the extension developer, a *proxy class* controls the access, visibility, and usage rights of the methods and attributes of the base class (the proxy class will be passed to the class of the extension developer implementing the interface and will be initialized once an extension is loaded), and *aspects* (implemented in AspectJ [Kiczales et al., 2001]), which inject into the base application the necessary logic for supporting the execution of the implemented extension (i.e., the aspect code enriches the base class with methods and data structures to load and initialize an implemented extension in a plug-in like fashion).

For reasons of brevity, the enforcement of the extension interface of the external developer is described. Listing 6.3 presents an excerpt of the generated code framework that realizes the extension interface of the software for the external developer scenario (see Listing 6.1). Since the two extension points are placed in one group, the compiler will generate a single interface for the extension developer that has to be implemented to realize the extension scenario. Lines 2-8 show the generated interface *ExtensionScenarioInterface*. The interface includes two parts. The first part is needed by the code framework to initialize the extension (Line 5). Moreover, references to the corresponding proxy classes are provided that will be used by the developer during the implementation of the extension. The second part is the extension point specific part: The extension developer has to implement the method *yourEPBO1Logic()* for the extension point *EPBO1* and the method *yourEPUI1JLabel()* for the extension point *EPUI1*.

The *EPBO1* proxy class (Lines 11-21) contains the generated list of getter methods required to provide a *READ* access to the *SalesQuote* class attributes. Note that no setter methods have been generated and no methods have been exposed as defined in the permission set *default1* (Listing 6.1, Lines 6-9). The proxy class generated for *EPUI1* is empty (Lines 23-25) since all methods and attributes were declared as hidden by the permission set *default2* (Listing 6.1, Lines 16-19). The last part of the code framework generated is the aspect code for *EPBO1* (Lines 28-75) and *EPUI1* (Lines 77-90).

In the *EPBO1* aspect, the first part (Lines 30-42) contains inter-type declarations, which enrich the base class with data structures and methods necessary to load the extensions implementing the *ExtensionScenarioInterface* in a plug-in like fashion (the extensions of type *ExtensionScenarioInterface* are loaded with a class loader and they are passed an instance of the proxy). The second part of the aspect code (Lines 44-57) enriches the base class in a similar fashion with methods to support the proxy class *EPBO1Proxy* calls. The last part of the aspect (Lines 59-75)

generates the advice that will load the extension after the constructor (i.e., trigger the plug-in load mechanism) of the *SalesQuote* business object, and the *saveSalesQuote()* method pointcut within the base class where the extension code will run as well as the advice that will run the extension code. The *EPUI1* aspect contains a similar body to the *EPBO1* aspect. However, the generated pointcut and advice (Lines 82-90) will add the *JLabel* component from the extension to the *salesQuotePanel*.

6.1.4 Tool Support for the Software Provider

Once the extension interface is compiled, the interfaces, proxy classes, and aspects are generated for each extension point by the XPoints compiler. To package the extension interface that will be delivered for a target software extender, the software provider can simply use the tool support of XPoints within Eclipse. The provider has to simply right click on the extension interface and choose to generate the extensibility API for the target extension developers. Figure 6.1 shows a screenshot of the tool support. The tool generates a Java archive containing the Java interfaces and proxy classes of the corresponding XPoints interface. The extension developer is expected to program his extension against using these artifacts along with the XPoints interface that will be used to guide him to the extension possibilities and the code-level artifacts (i.e., interfaces and proxy classes) using the recommender tool.

```
1  /**********Generated Interface**********/
2  public interface ExtensionScenarioInterface {
3
4   // these are the methods the extension developer has to implement
5   public void init (EPBO1Proxy p1, EPUI1Proxy p2);
6   public void yourEPBO1Logic();
7   public JLabel yourEPUI1JLabel();
8  ...}
9
10 /**********Generated Proxy Classes**********/
11 public class EPBO1Proxy{
12  private SalesQuote salesquote ;
13  ...
14 // getter methods for the READ attributes
15  public CustomerInfo getCustomerInfo(){
16   return salesquote .getCustomerInfo( this );
17  }
18  public List<ProductQuote> getProductQuote (){...}
19  public String getComment(){...}
20  public double getDiscount (){...}
21  ...}
22
23 public class EPUI1Proxy{
24 //empty since no access has been granted
25 }
26
27 /**********Generated Aspects**********/
28 public privileged aspect EPBO1Aspect {
29
30  // Datastructure to hold extensions of type ExtensionScenarioInterface
31  private ArrayList< ExtensionScenarioInterface > SalesQuote. ExtensionScenarioExtensions ;
32
33 //New method in SalesQuote class to add the extensions
34  private void SalesQuote. loadExtensionScenarioExtensions (){
35  ...
36 //load the extensions with class loader
37  ...
38       extensions . init ( this .getEPBO1Proxy(),this.getEPUI1Proxy());
39       ExtensionScenarioExtensions .add( extension );
40  ...}
```

```
41 //New method in SalesQuote class to perform EPBO1 extension sanity checks
42  private void SalesQuote.sanityChecksEPBO1(){...}
43
44 //New method in SalesQuote class to get the EPBO1 proxy
45  private EPBO1Proxy SalesQuote.getEPBO1Proxy(){
46            return new EPBO1Proxy(this);
47           }
48
49 //New methods to support the proxy access to the base class
50  public CustomerInfo SalesQuote.getCustomerInfo(EPBO1Proxy proxy){
51           // validate the proxy and return
52           if(isLegalProxy(proxy)) return this.customerInfo;
53           else return null;
54  }
55
56  public List<ProductQuote> SalesQuote.getProducts(EPBO1Proxy proxy){...}
57 // Similarly to the rest of the attributes ...
58
59 //load the extensions and perform sanity checks in constructor constructor
60  pointcut onload(): execution(* SalesQuote.new(..));
61  after(SalesQuote s): onload() && this(s){
62           s.loadExtensionScenarioExtensions();
63           s.sanityChecksEPBO1();}
64
65 // Pointcut and advice for running the extensionScenario
66  pointcut extension(): execution(* SalesQuote.saveSalesQuote(..));
67  before(SalesQuote s): extension() && this(s) {
68
69           if(s.ExtensionScenarioExtensions != null)
70           {
71                    for(int i=0; i<s.ExtensionScenarioExtensions.size(); i++)
72                    {
73                             s.ExtensionScenarioExtensions.get(i).yourEPBO1Logic();
74                    }
75            }}...}
76
77 public privileged aspect EPUI1Aspect {
78  ...
79 //Aspect body similar to the EPBO1Aspect
80  ...
```

```
81
82 // Pointcut and advice for running the EPUI1 extension
83 pointcut extension(): execution(* SalesQuoteForm.new(..));
84 after(SalesQuoteForm s): extension() && this(s) {
85  if(s.ExtensionScenarioExtensions != null)
86          {
87           for(int i=0; i<s.ExtensionScenarioExtensions.size(); i++){
88              JLabel j = s.ExtensionScenarioExtensions.get(i).yourEPUI1JLabel();
89                          s.salesQuotePanel.add(j);
90                  }
91          }}}...
```

Listing 6.3: Generated code framework for the external developer

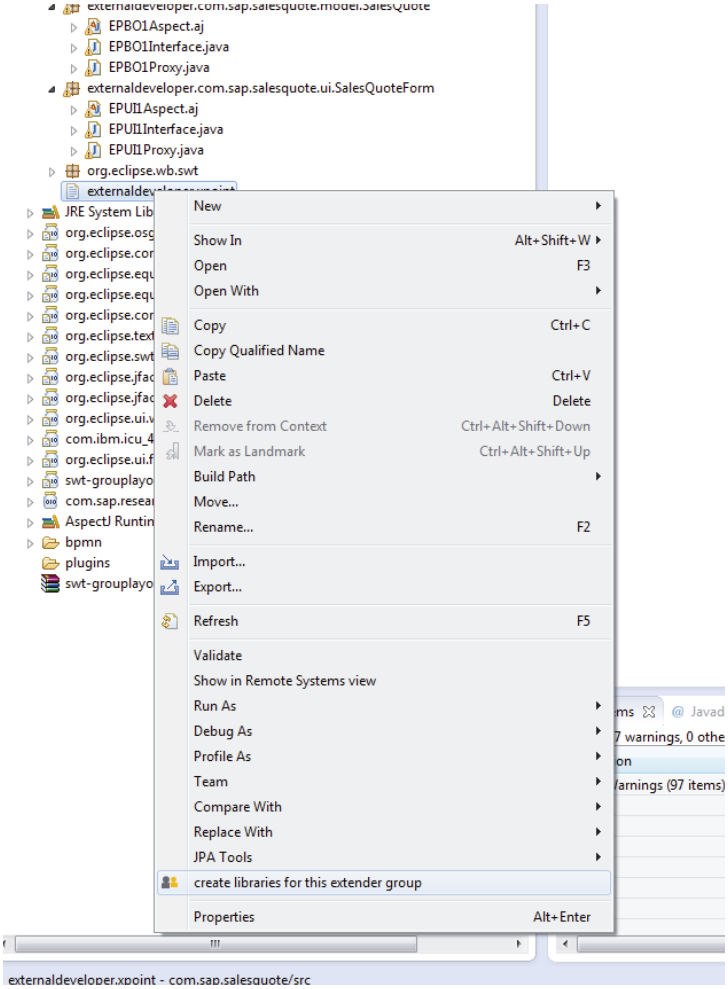


Figure 6.1: Software provider: Generation of the extensibility API

6.1.5 Guiding the Extension Developer

To develop an extension for the software, the developer goes through four phases. First, the developer uses the tool to identify the extension possibilities that exist. Second, the developer adds the extension possibilities of interest to the bookmark list. Third, out of the bookmarked extension possibilities, the developer generates a new extension project. Fourth, the extension developer uses the generated skeleton to develop the extension.

To illustrate these phases, an example extension is described. In this extension the sales quotation form is extended with a customer rating module that facilitates retrieving the rating of the credit worthiness of a customer from an external credit rating agency. Using this module, the sales representative can assess possible risks that are associated with a particular customer and adjust the terms of the sales quotation before issuing it.

In the first phase, the developer loads the provided XPoints interface. Using this interface, the tool renders the available extension possibilities on the different logical layers of the software. Figure 6.2 shows a screenshot of the recommender tool showing the browsing of the extensible artifacts. Since the XPoints extension interface does not include extension possibilities on the business process layer, the view is not displayed by the tool. In the sales quotation creation user interface form, the defined extension point *EPOUI1* and its type *allowUIComponent* is rendered by the tool on the user interface model. The corresponding extension possibility depicted by *EPBO1* within the *SalesQuote* business object is illustrated below. Using the recommender tool, the extension developer can simply drag and drop the recommended extension possibilities to the favorite list on the left side. Furthermore, the developer can link the extension possibilities to extension development tasks in the to-do list.

Switching to the implementation perspective, the extension developer can then use the bookmarked extension possibilities to generate an extension project skeleton. Figure 6.3 shows the implementation perspective and the extension plug-in generation wizard. The bookmarked extension possibilities are displayed in the XPoints explorer view. By dragging and dropping the extension possibilities to the plug-in project view and starting the extension project creation wizard, the extension developer will be prompted to define a path to the Java archive of the extensibility API. After that, the extension developer will get a standard Eclipse Java project that contains a class with a code template that implements the *ExtensionScenarioInterface*.

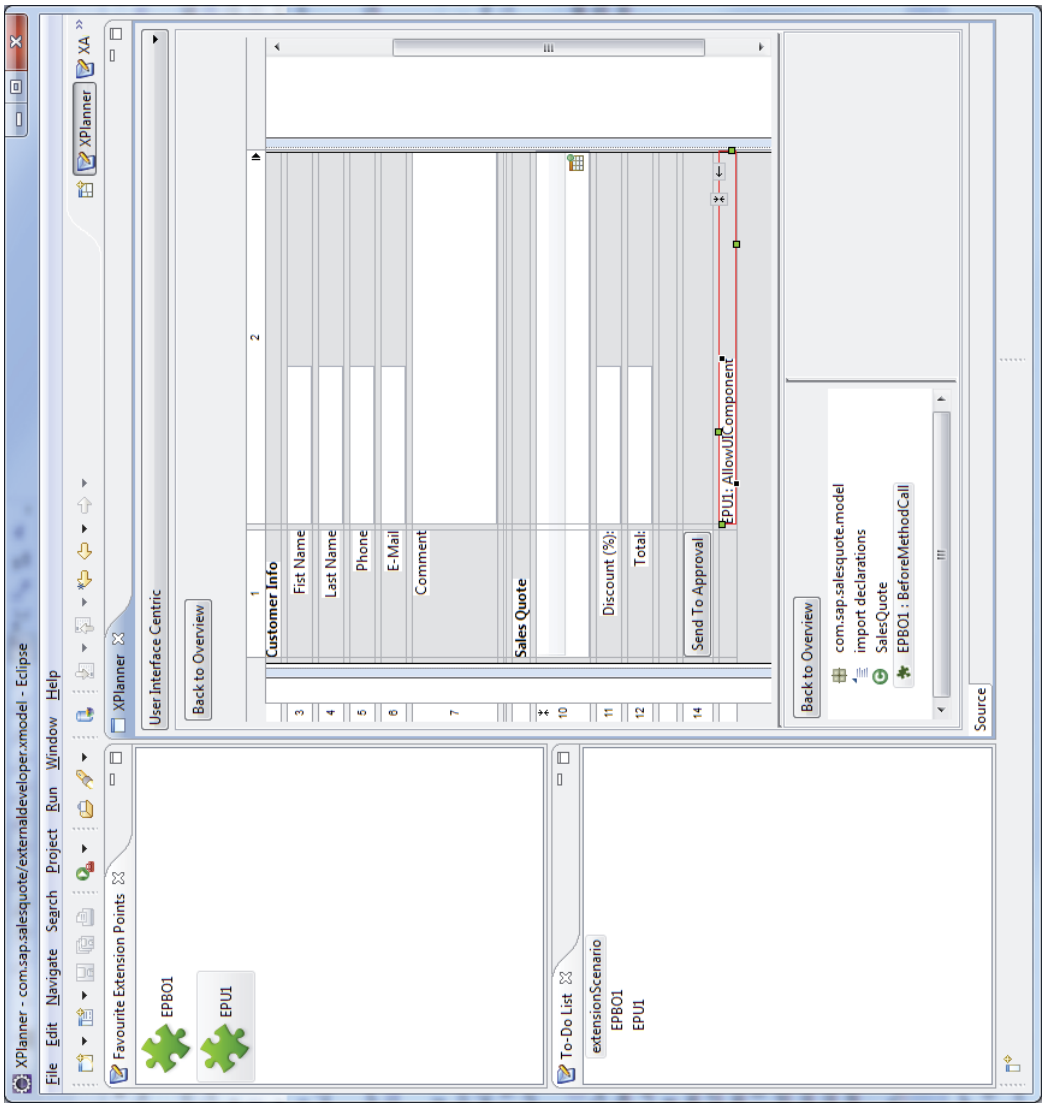


Figure 6.2: Browsing extension possibilities using the recommender tool.

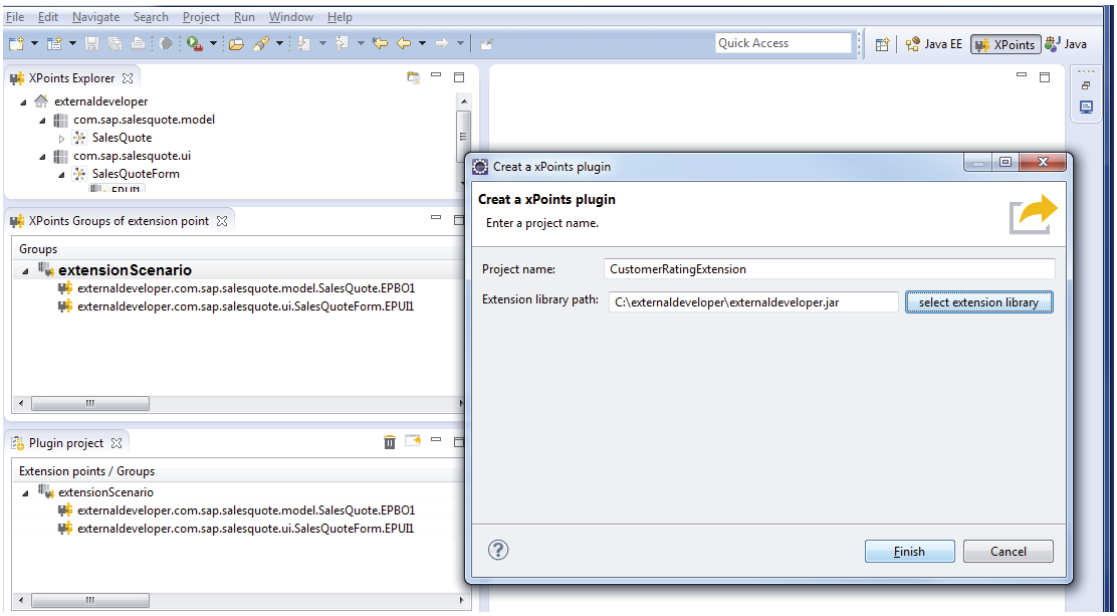


Figure 6.3: External developer: Plug-in creation wizard for the core software.

Using the generated code template, the extension developer can start developing his extension. Listing 6.4 shows an example implementation of the described customer rating extension. The *CustomerRating* class is the custom class of the extension developer that implements the necessary logic to query an external credit rating agency and retrieve the customer rating of a given customer based on an identification number. The implementation of the method *yourEPBO1Logic()* uses the proxy *p1* to retrieve the customer information from the sales quote class. Based on the information a query to the customer rating agency is instantiated and the text results are displayed using the label. The method *yourEPUI1JLabel()* returns the *JLabel* object that is created by the extension class to the core software. The compilation of the extension project is done separately from the core software. The compiled classes are exported as a Java archive and placed in the plug-ins directory of the core software which will be discovered, loaded, and executed during runtime.

```
1  class CustomerRatingExtension implements ExtensionScenarioInterface {
2
3   private CustomerRating cr;
4   private String rating ;
5   private JLabel customer_rating_label = new JLabel("" );
6   private EPBO1Proxy p1;
7   private EPUI1Proxy p2;
8
9   public void init (EPBO1Proxy p1, EPUI1Proxy p2){
10    this .p1 = p1;
11    this .p2 = p2;
12  }
13
14  public void yourEPBO1Logic(){
15   CustomerInfo ci = p1.getCustomerInfo ();
16   cr = new CustomerRating(ci. getCreditID ());
17            rating = ci . getRating (). toText ();
18            customer_rating_label . setText ("Customer rating : "+ rating );
19  }
20
21  public JLabel yourEPUI1JLabel(){
22   return this . customer_rating_label ;
23  }
24 }
```

Listing 6.4: Customer rating extension.

6.1.6 Discussion

To highlight the advantages of XPoints, in absence of XPoints the code in Listing 6.3 would have to be written manually by the developer of the base application in addition to the implementation of the core application functionality. By comparing Listing 6.3 with Listing 6.2, it becomes clear that XPoints significantly reduces design complexity. The XPoints interface provides a declarative mechanism for supporting extensibility, higher level of abstractions, and separation of concerns. While the developer could employ other programming patterns and techniques rather than those used for code generation, the resulting application will not be of lower complexity. This is because the developer will always have to adapt the functional code to support extensibility.

The more distinct ways of extending a software system, the more complicated it would be to mix functional code with aspects, proxy classes, and interfaces that are concerned with governing different extension scenarios. This will lead to an overly complex design with maintainability problems and loss of design intent. As the base application evolves (e.g., more extension scenarios have to be supported), the base application developers will have to implement the extensibility enforcement code through new aspects, interfaces, and proxy classes. The huge number of classes and aspects that have to be written makes the technical realization of the extension interface very hard. The technical realization complexity of the extensibility possibilities is simplified by XPoints since it automatically generates the required (boilerplate) code of the extension interface and avoids polluting the core design with infrastructure for simulating extension interfaces, and results in a less complex design, better class maintainability, and better preservation of the design intent for the software provider.

In addition to the pointed out limitations, XPoints supports defining extension possibilities at different logical layers that have not been handled so far by the current state-of-the-art approaches. The approaches outlined in Chapter 4 only focus on the code level, however XPoints can further support other abstractions like UI and business processes. XPoints also aims at simplifying the base code developer task of designing for extensibility. The developer simply has to specify the extension possibilities for each extension scenario that exist without worrying much about how the extension interface will be realized on the code level.

Providing the classes and interfaces to an extender without proper documentation of the extension possibilities and usage instructions can make the comprehension of the extension possibilities and the identification of the coding artifacts to be used (e.g., interfaces, proxy classes, etc.) very hard. The proxy classes and interfaces provided to the extender in Listing 6.3 are not sufficient to be able to identify whether they are used as a part of the core functionality of the software or they are used for extensibility. On the other hand, an XPoints interface declares extension points and their constraints as first class entities and hence explicitly defines the extension possibilities. Using an XPoints interface as a contract, the developer can see the layer specific extension possibilities and their dependencies and can use it as a pointer to the low-level coding elements that are required to realize an extension. For example, the XPoints interfaces in Listing 6.2 can be used to identify the right interfaces and proxy classes required to realize a particular extension.

Nevertheless, XPoints comes with the necessary tool support to guide the extension developer throughout the phases of the extension development. Using the XPoints interface, the tool visualizes the extension possibilities explicitly on the software artifacts which are easier to comprehend (e.g., user interfaces, business processes, etc.). On one hand, this can support

developers based on their requirements and level of expertise. For example, if a developer wants to implement an extension on the user interface layer will show the relevant extension possibilities using user interface elements. Moreover, the developer will be able to identify the interdependencies between extension possibilities on other layers that he might not have been aware of (e.g., business objects, and business processes). This feature makes the task of the extension developer much simpler and reduces the dependency of the developer on documentation and other resources. Moreover, the tool simplifies the extension development based on the selected extension possibilities by generating code templates in an extension project. This saves the time needed by the extension developer by mapping the high-level extension possibilities (e.g., on the user interface layer) to the low-level coding elements that are required to implement an extension (e.g., the right classes to extend or interfaces to implement).

6.2 Revisiting the Requirements

In Chapters 2 and 3 the problems and challenges that face the software provider were outlined. The requirements for extension interfaces and program comprehension tools to support extensibility (i.e., the definition and consumption of extension interfaces) of multilayered applications were proposed. There are two main concerns that have to be addressed by the software provider; interface specification and interface enforcement. In the following discussion, a selection of the related work from the presented state-of-the-art works in Chapter 4 on extension interfaces and program comprehension tools is compared with the presented approach in terms of how they satisfy the proposed requirements. Table 6.1 lists the related works chosen for comparison with XPoints and Table 6.2 lists the related works chosen for comparison with the XPoints recommender tool. Each work is evaluated as satisfying, not satisfying, or partially satisfying the requirement. These are denoted by the symbols **+**, **-**, and **P** respectively.

6.2.1 XPoints Concept and Implementation

The first requirement **RSP1** specifies that the extension interface should explicitly define the extension possibilities. XPoints supports the explicit definition of extension possibilities as first class entities and using domain-specific constructs (e.g., user interfaces, business objects, business processes, etc.). Object-oriented frameworks do not satisfy this requirement since they highly depend on coding elements like interfaces and abstract classes to present the extension possibilities. Design patterns provide guidelines to enforce extension possibilities, however they are not explicitly expressed. In aspect-oriented programming extension possibilities can

theoretically exist everywhere and is only limited by the pointcut language. There are no constructs for defining explicitly the extension possibilities of a software system.

Related Work	RSP1	RSP2	RSP3	RSP4	RSP5	RSP6	RSP7
Object-Oriented Frameworks	-	P	-	P	-	-	P
Design Patterns	-	P	-	-	P	+	-
Plug-in Systems	P	P	-	P	-	-	P
Script-Based Approaches	P	P	-	P	-	-	-
Aspect-Oriented Programming	-	-	+	-	-	-	-
Open Modules	P	-	-	-	-	-	-
Cross Cutting Interfaces	P	-	+	P	-	-	P
Extension Join Points	P	-	+	P	-	P	P
Model-Based Pointcuts	P	-	+	P	-	-	P
Explicit Join Points / Join Point Types /Join Point Interfaces	P	P	+	P	-	P	P
XPoints	+	+	+	+	+	P	P

Table 6.1: Related work on extension interfaces: Satisfaction of requirements

Plug-in systems enhance on these approaches by providing meta-data which can specify extension points. However, the extension points are usually specified in terms of classes and interfaces and not in domain-specific or layer-specific terms. Scripting-based approaches provide an extension developer with a (domain-specific) language through which extensions can be realized. However, these approaches do not provide language constructs to express extension possibilities. The aspect-oriented interfaces (i.e., open modules, cross cutting interfaces, extension join points, model-based pointcuts, and EJP, JPT, JPIs) provide an explicit representation of extension possibilities on the code-level, but they also do not express these possibilities using layer-specific terms.

The second requirement **RSP2** specifies that the extension interface must define the access of the extensions to the resources of the core software. Besides the specification of the types of resources made available for the extension developer, XPoints provides control constraints which are used to provide a fine-grained access to the resources of the core software. Object-oriented frameworks, design patterns, and plug-in systems rely on the underlying language mechanisms (e.g., types and encapsulation) as well as implementation patterns to specify and control the access to the resources of the core software. However, these approaches do not explicitly express the access rights of extensions to the core resources of the software and do not provide fine-grained control constraints. Scripting-based approaches provide libraries that can be used by the extensions to access the core software. The implementation of such libraries is not simple and analogously to

the other approaches the control constraints have to be implemented manually by the software provider.

Aspect oriented programming and aspect-oriented interfaces do not provide any control on what an advice can access from the core software. EJP, JPT, and JPIs improve on the aspect-oriented interfaces by defining the resources that will be made available to an advice. However, these approaches do not provide the means to control how the access to these resources will be done and it will have to also be manually implemented by the software provider.

The third requirement **RSP3** states that the code enforcing the extension interface must be separated from the functional implementation of the core software. Almost all approaches except of the aspect-oriented approaches do not provide a way to separately enforce the extension interface from the functional code. Object-oriented frameworks, design patterns, plug-in systems, and scripting based approaches all require the developer to complement the functional code with the extensibility enabling code. Open-modules use point-cut declarations that are specified within each module. XPoints shares the idea of aspect-oriented interfaces with decoupling the interface specification from the functional code and therefore supports the modularization of extensibility.

The fourth requirement **RSP4** states that an extension interface has to express the extension possibilities on different logical layers and express the interdependencies between them. Almost all of the approaches allow the expression of extension possibilities on different logical layers. However, the specification is usually realized using code-level artifacts and there is no support for expressing interdependencies on different logical layers. In comparison to these approaches, XPoints provides the language constructs for expressing and enforcing extension interfaces for supporting cross-layer extensions.

The fifth requirement **RSP5** states that an extension interface must support multiple extensions and extenders. By decoupling the extension interface from the functional code, XPoints allows the definition of multiple extension interfaces for different kinds of extension developer groups. In comparison to XPoints, the other approaches do not support for the specification and enforcement of different extension developer groups (i.e., they provider a one-size-fits-all interface). Design patterns can provide a solution to support this on the implementation level. However, the realization is not explicit and very complex.

The sixth requirement **RSP6** requires an extension interface to provide a standard for the low-level (implementation) enforcement of the extension interface. XPoints does not require a developer on the side of the software provider to be an expert with the state-of-the-art approaches

(e.g., design patterns) to realize the required extension interface. The XPoints compiler will automatically complement the core software using the adequate code-level approaches (e.g., like mixins [Bracha and Cook, 1990], virtual classes [Madsen and Møller-Pedersen, 1989], difference based modules [Ichisugi and Tanaka, 2002], traits [Schärli et al., 2003] etc.) or design patterns and generate the required extensibility code framework. From that perspective, XPoints can be seen as introducing a new layer above these approaches and can further make use (depending on the implementation of the compiler) of these approaches or other advanced techniques for the realization of extension interfaces on the code level. Some approaches on aspect-oriented interfaces provide some enforcing mechanisms through the usage of types for predefined join points. However, there is no standard for the enforcement of extension interfaces. Other approaches require the developer to manually develop the interfaces as well as handle them in the implementation of the core software to support extensibility.

The last requirement **RSP7** specifies that the extension developer can simply consume the extension interface that is specified by the software provider. Similar to the other approaches, XPoints uses high-level domain-specific constructs to express extension possibilities. An XPoints interface can be used by the extension developer as a contract to identify the extension possibilities. Plug-in systems define the extension possibilities in terms of classes and interfaces that have to be implemented by the extension developer by metadata constructs. Aspect-oriented interfaces separately express the code-level extension possibilities in a separate contract using aspect language constructs. Without the appropriate guidance as well the recommendation of relevant development artifacts XPoints and these approaches partially satisfy this requirement. XPoints comes with a recommender tool that complements it to simplify the extension development task.

6.2.2 XPoints Recommender Tool for Guiding the Extension Developer

Tracking-based approaches are primarily based on tracking the activities of a development team collaborating on the implementation of a particular project. This is not possible as a solution for black-box extensibility. Visualization-based approaches depend on visualizing interactions and data-flow between different methods and classes for simplifying the understandability of a source code. These approaches are also not possible for supporting extension development as they require the availability of the source code of the software system on the side of the extension developer. The most appropriate related approaches to be used in the context of black-box extensibility on the side of the extension developer are the search engine, code recommendation, documentation-based, and code-query approaches. In the following discussion, a comparison of these approaches and the recommender tool is presented.

Approach	RPC1	RPC2	RPC3	RPC4	RPC5
Search Engine	-	-	-	+	P
Code Recommendation	-	-	-	+	+
Documentation-based	P	P	-	+	P
Code Query	-	-	P	-	P
XPoints Recommender Tool	+	+	P	-	P

Table 6.2: Related work on program comprehension: Satisfaction of requirements

The requirement **RPC1** states that the extension possibilities must be explicitly expressed for the extension developer. The XPoints tool visualizes the extension possibilities on high-level domain-specific artifacts like user interfaces and business process models. Moreover, the extension developer has the possibility to browse through the different artifacts. In comparison to the search engine approaches, the extension developer will have to use text-based search to find out the possible extension possibilities that exist. Most of these approaches only support the search for low-level coding libraries. The code recommendation and code-query approaches can help assisting the developer with the coding process. However, they cannot help him with the identification of the extension possibilities. Similarly, documentation-based approaches recommend documentation through the coding process.

The second requirement **RPC2** requires the tool to present the interdependencies between extension possibilities. The XPoints tool visualizes the interdependent extensible artifacts from different layers while the extension developer is browsing for extension possibilities. Code recommendation and code query approaches can only be used during the coding process to highlight the coding elements that are specific within a particular coding context between the extension possibilities. Documentation-based approaches can forward important parts of the documentation to extension developers while coding allowing the developer to realize important interdependencies to other libraries or coding elements. These approaches can be seen however as complementing XPoints since, they can be used to help the extension developer code an extension based on a particular extension possibility.

Requirement **RPC3** states that the tool must map the high-level extension possibilities to the low-level coding artifacts. The XPoints tool generates code-level stubs that extend the main coding entry points for building an extension based on the bookmarked extension possibilities. The extension developer can further use these code stubs as a foundation to continue building his extension. All approaches do not provide a mapping from high-level artifacts to low-level coding elements. A developer has to read documentation, tutorials, or search the web in order to understand how to implement an extension assuming that extension possibilities of interest

were identified. The next requirement **RPC4** requires the tool to recommend the relevant documentation for implementing an extension. The XPoints tool does not provide this feature. Code recommendation and documentation based approaches can be combined with the XPoints recommender to tool to support the extension developer with all of the development phases.

The last requirement **RPC5** expects that the tool reduces the time required by extension developers to implement an extension. Using the explicit presentation of extension possibilities and generation of extension code stubs, the XPoints tool is expected to reduce the time required for an extension developer to implement an extension. The tool will cut down the time required for the extension developer to identify the extension possibilities on high-level artifacts and will not require the extension developer to search the web or read documentation. Moreover, the tool generates the code stubs that are ready for usage by the extension developer for the implementation of an extension.

6.3 User Study

In the following user study the usability of XPoints over Java at the side of the software provider is evaluated by measuring the time required for implementing the requirements of an extension interface both in XPoints and Java for an open-source multilayered Java application by experienced Java developers. In addition, the code delivered by both developer groups is compared and evaluated. The developers are also required to evaluate the difficulty of implementing the requirements of the extension interface both in XPoints and Java.

6.3.1 A Generic-Java Instantiation

For the purpose of the user study (to make XPoints more Java-developer-friendly), a more generic instantiation of the concepts of XPoints for Java is implemented. The grammar of this instantiation is listed in Figures A.3 and A.4. The *Layer* concept resembles a Java *package* and the *Extensible Artifact* concept resembles a Java *class*.

On the *class* level the following types instantiate the *Extension Point* concept. *beforeMethodCall* and *afterMethodCall* allow the developer to insert extension code after or before a call to a method is being called. *beforeMethodExe* and *afterMethodExe* allow the developer to insert extension code after or before the execution of a method (call and execution have the same semantics as in AspectJ). *override* requires that a valid extension overrides the implementation of a particular method. *beforeConstructor* and *afterConstructor* allow the developer to insert extension code

before and after the specified constructor of the class is executed. *addItemAfterInitialization* provides the possibility for a developer to extend an object data structure that has an *add()* method (e.g., List) with more elements after a new instance of the object has been created. *addItemAfterMethod* and *addItemBeforeMethod* provide the possibility for a developer to extend an object data structure that has an *add()* method with more elements after or before the execution of a particular method. On the *ExtensionPoint* level, the implementation realizes the same *permissionset* implementation for business objects described in.

The *group* construct instantiates the generic concept *ExtensionPointGroup*. Adding extension points to a group, will automatically apply a constraint requiring a valid extension to implement all the group members. The extension interface generation is done in a similar way as described for the implementation in (i.e., Java interfaces as entry points, proxy classes for controlling access to base code artifacts, aspects for integrating and executing extensions as well as validating constraints).

6.3.2 Setup and Execution

In the context of this study JAllInOne [Carniel, 2007], a Java-based open source business application, is used. The application is made up of 201 packages with a total of 1,731 classes and 13,588 methods (204,549 lines of code) that implement the GUI, business objects, business processes, and database persistency. The module used in this study is the sales order processing module shown in Figure 6.4 which is implemented using the model-view-controller pattern.

In the following the requirements for an extension interface that spans the GUI layer are described. In this scenario the first extension possibility allows the extension developer to add a new label and a new text field (i.e., *JLabel* and *JTextField*) to the *SalesOrderHeader* panel. The extension code providing the new label and field must be executed after all of the main panel components have been initialized. The second extension possibility allows the extender to insert custom logic to perform validation that might be required for the new text field. As a requirement for the extension interface, read-only access to the attributes of the *controller* object of the panel must be granted and the existing public method *loadDataCompleted()* of the panel should not be callable via an extension. Moreover, a valid extension must provide an implementation for both of the first and second extension possibilities (i.e., a valid extension must provide a new label and text field as well as new custom logic).

The study is comprised of two parts (the detailed tasks are available in Appendix A.4). In Part I, the XPoints and Java developer implement the requirements of the extension interface. In this part

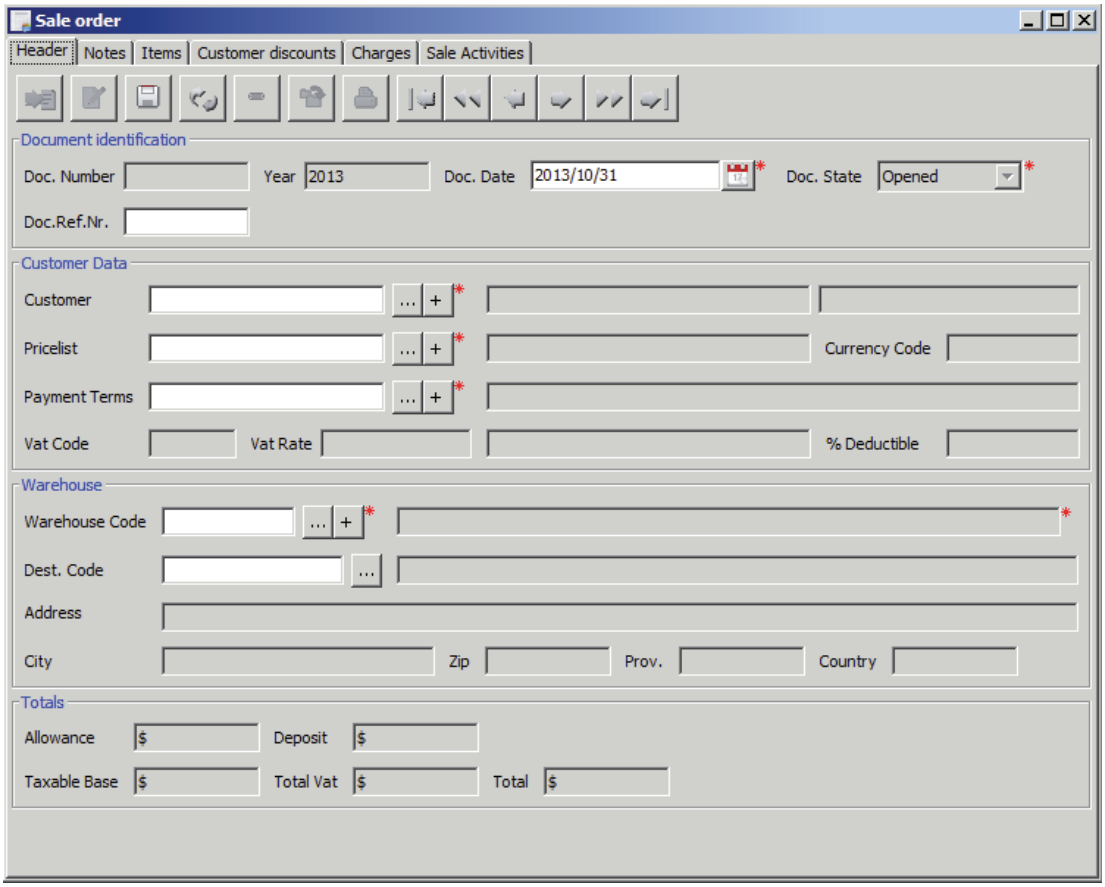


Figure 6.4: JAllInOne: Sales order creation form.

the extension interface developers assume that the extension developers will not get the source code of the system. Before implementing the solution, the developers are required to define what artifacts the extension developer will get for implementing extensions (i.e., the extension interface) and how the extension code will be loaded and executed. For simplicity reasons it is assumed that only a single extension will be executed.

Part I is comprised of three tasks. In the first task (Task I) the developer is required to support the addition of new graphical user interface elements (Task Ia) and the custom logic of the extension developer (Task Ib). In the second task (Task II), the extender is required to provide a read-only access to the *controller* object and make the public method *loadDataCompleted* non-callable by an extension. In the third task (Task III), the developer is required to verify the validity of an extension implementing the provided extension interface (i.e., ensure that the developer provides an implementation for both extension possibilities).

In Part II of the study, the XPoints and Java developers are required to rate on a 5-point Likert scale the difficulty of implementing each of the given tasks and the overall extension scenario (1 being the least difficult and 5 being the most difficult). They are also required to rate the difficulty to understand, to maintain, and to modify their provided solution in Part I. The developers were required to rate the difficulty of understanding the given source code of the *SalesOrderHeader* panel of JAllInOne.

In the study 16 developers were recruited. The developers reported between 3 to 13 years of Java development experience during which they have been exposed to various software development activities including refactoring and extension development. The participants were split into two even groups for solving the tasks in Java (Group I) and XPoints (Group II). The total amount of time given for each participant was 90 minutes. Both groups were given 20-minute guided introduction to the software system, the sales order form and the underlying source code, and the requirements of the extension interface. Group I was given a maximum of 60 minutes to solve Part I and 10 minutes for Part II. Group II were given a 20-minute guided introduction to XPoints (i.e., constructs and semantics) including a small example to implement, 40 minutes to solve Part I, and 10 minutes for Part II. The setup, design, and time constraints defined were validated through a pilot study before the execution of the study.

For the execution of the study, a workstation was setup for each participant with an Eclipse IDE with the copy of the source code of JAllInOne and access to the internet. The time for solving each of the tasks was measured. After the conclusion of the experiments, the solutions provided by the developers were collected and analyzed. The analysis involved the implementation method and code metrics.

6.3.3 Results

The XPoints and Java developers provided on average 20 lines and 88 lines of code respectively to implement the extension interface. Figure 6.5 shows the average and standard deviation of the time spent by each developer for each task and Figure 6.6 shows the average and standard deviation of the Likert values of the self-report on difficulty by the developers. The time for the completion of each task was measured once the developer started coding till the developer was done with the solution. The overall results show that the developers using XPoints were about 4 times faster than those using Java for solving the tasks. The difficulty of understanding the provided source code of the panel of JAllInOne was rated almost the same by both the Java and the XPoints developers. However, the developers using XPoints reported it to be much

easier to implement the tasks in comparison to the developers using Java. Moreover, the XPoints developers also thought that their provided solutions were easier to understand, maintain, and change than the Java developers.

The Java developers provided different solutions. The provided solutions are analyzed to identify what has to be done to implement an extension, how the extension code will be loaded and executed, and the modifications that were done to the base code to implement the requirements of the extension scenario. All of the solutions provided by the developers were invasive (i.e., they modified the provided source code of JAllInOne). In the following the solutions provided by the developers to implement the requirements of the tasks are summarized.

In Task I, the following comprised the Java developer solutions. 5 developers provided interfaces that must be implemented by an extension developer and 3 developers provided abstract classes. The loading and integration of the extension code to the functional code was also realized in different ways by the developers. 4 developers manually implemented a class loader mechanism that recognizes implementations of the provided interfaces or abstract classes during runtime. These developers expect the extension developer to place the extension class in a particular directory during runtime. 2 developers required the extension developer to define a separate properties file with metadata where they declare the names of the extension classes and methods to help them discover and load their extensions. 2 developers used the built-in Java service loader class for loading extension classes.

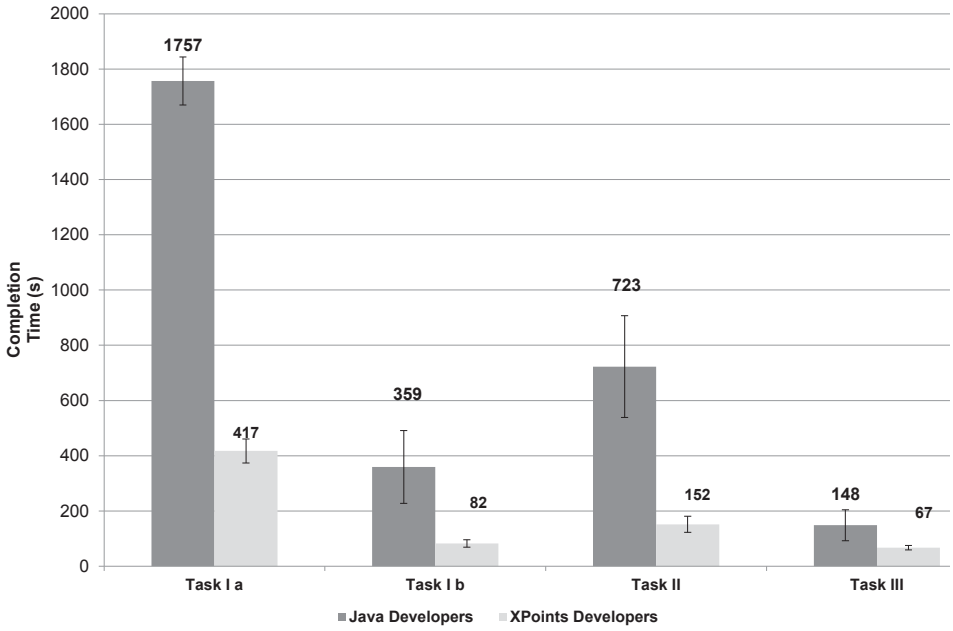


Figure 6.5: Mean and standard deviation of the time spent by the developers for each task

In Task II, all of the developers provided invasive solutions. The developers changed the existing modifier of the public method *loadDataCompleted()* to protected or private. The read-only access to the attributes of the *controller* object was mostly provided by returning a copy of the object to the extender. Moreover, the developers had to trace and modify the existing dependencies from the same and other classes to the method and object. In Task III, the developers depended on the assumption that the extension developers will have to implement a single Java interface or abstract class for the whole extension scenario. However, all developers did not implement a way to validate the implementation of the extension to ensure that it implements all extension possibilities of the scenario.

The XPoints developers provided the same solution for the tasks. Listing 6.5 shows the XPoints implementation of the extension scenario. The solutions provided by all developers had almost the same structure as presented in this listing. In the first part the developer declares the class *SaleOrderDocFrame* as an extensible artifact with three extension points representing the three extension possibilities described in the scenario. The following part allows read-only access to the *controller* object. The last part defines the group *SOImplementAll* that includes the three extension points. This requires a valid extension to implement all three extension points.

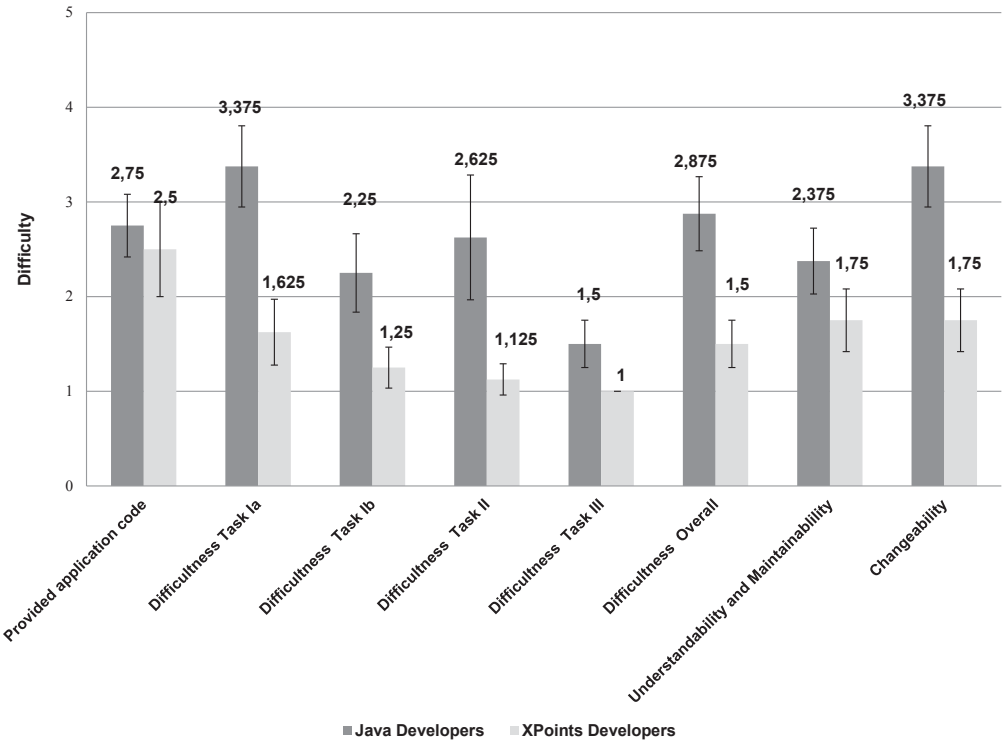


Figure 6.6: Mean and standard deviation of the self-report on the difficulty by the developers

```
1  extensionInterface  SalesOrderFormExtensionScenario{
2
3  package org. jallinone . sales .documents. client {
4
5    class  SaleOrderDocFrame {
6     addItemAfterMethod AddSalesTxtField = saleIdHeadPanel1 <JTextField>, void   jbInit  ();
7     addItemAfterMethod AddSalesLabel = saleIdHeadPanel1 <JLabel>, void jbInit ();
8     beforeMethodCall CustomLogic = void confirmButton_actionPerformed(ActionEvent);
9    }
10   permissionset  (SaleOrderDocFrame){
11    attributepermission( controller ,  READ);
12    methodpermission(loadDataCompleted(), HIDDEN);
13   }
14 }
15   group SOImplementAll{
16    org. jallinone . sales .documents. client .SaleOrderDocFrame.AddSalesTxtField,
17    org. jallinone . sales .documents. client .SaleOrderDocFrame.AddSalesLabel,
18    org. jallinone . sales .documents. client .SaleOrderDocFrame.CustomLogic
19   }
20 }
```

Listing 6.5: Solution in XPoints

Threats to Validity The first threat to validity is caused by the choice of the software system. It may be the case that the results may be different for other software systems. The second threat to validity is caused by the design of the tasks and the time given to the developers. With other tasks and more time, the results may be different. However, this threat is appropriately controlled by validating the setup through the pilot study. The third threat to validity is the number and level of expertise of the developers. However, this threat is controlled by carefully selecting developers who were experienced with Java development and in particular extension development.

6.3.4 Discussion

By comparing the solutions provided by the Java developers with the solutions of the XPoints developers, the following advantages are outlined. First, it becomes clear that XPoints significantly reduces the implementation time. The amount of time spent by the Java developers was 4 times more than the time spent by the XPoints developers. Moreover, the solution of the Java developers was invasive (i.e., required changes to the source code of the provided software), whereas the XPoints developers separately implemented the extension interface without modifying the source code of the system. The XPoints interface provides a declarative mechanism for the specification and implementation of extension interfaces, higher level of abstraction, and separation of concerns. This allows the developers to focus on the definition of the extension interface of the software system without focusing on how it will be enforced.

In the study, the Java developers had to worry about what extension units the extension developers have to provide as well as how the extension code will be integrated and executed (i.e., the technical realization of the interface). On the other hand, the XPoints developers did not have to handle the technical realization complexity of the extension interface as it is simplified by XPoints by automatically generating the required (boilerplate) code of the extension interface. This also avoids polluting the core design with code for implementing extension interfaces. The XPoints compiler handles the generation of the appropriate extension units for the extension developer, integrate, and run the extension code with the functional code as well as validate the extension interface constraints while preserving the base code.

Second, the XPoints approach improves the maintainability of the software systems. As shown in this simple scenario, XPoints provided a reduction of about 85% of the amount of lines of code that were required to implement the scenario. The more versatile the extensibility of the software system is, the more complicated it would be to mix functional code with code that is concerned with governing different extension scenarios. As the base application evolves (e.g., more extension scenarios and kinds of extension developers have to be supported), the huge number of classes and methods, and adaptation of the functional code that must be created makes the realization of the extension interface very hard. This will lead to an overly complex design with maintainability problems and loss of design intent. In the study, the XPoints developers reported that their solution is much easier for other developers to maintain and change than the Java developers.

Last, using an XPoints interface as a contract, the developer can see the layer specific extension possibilities and their dependencies and can use it to find the required artifacts (i.e., the generated

interfaces and proxy classes) to implement an extension. On the other hand, the solutions provided by the Java developers are not sufficient to be able to identify whether they are used as a part of the core functionality of the software or they are used for extensibility without proper documentation and usage instructions. This can make the comprehension of the extension possibilities and the identification of the coding artifacts to be used for both the software providers and the extension developers very hard. In the study, the Java developers reported that it was much harder for other developers to understand their implementation in contrast to the developers using XPoints.

6.4 Limitations of the Approach and Implementation

In the following the limitations of XPoints are described. First, the generated extension interfaces can become invalid if the code of the core application changes. To address this limitation, once the XPoints interface is compiled, the XPoints compiler validates the XPoints interface and the source code of the core application and will output errors and warnings if there are any inconsistencies on the syntactic level (e.g., references to nonexistent classes or methods) in the interface specification. Once the developer updates the XPoints interface, the compiler will generate a new extension interface for the application.

Second, the extension point types and enforcement semantics depend on the implementation of XPoints. Since the instantiation of the approach can vary based on the domain of the application, the semantics has to be defined by the software provider. Moreover, the developers on the software provider side have to learn a new language in addition to the native development language in order to be able to implement an extension interface. On the side of the extension developers, the developers relying only on XPoints have to understand the semantics of the language as well. However, the XPoints recommendation tool can be used to simplify the consumption of the XPoint interfaces.

Third, in case of multiple extensions, the order of the loading and execution of extensions is done randomly. Moreover, there is currently no support for detecting conflicts between extensions or conflicts while accessing the resources of the core software.

Fourth, the current implementation allows the software provider to control the access to the resources of the core software, however it does not allow the software provider to express advanced constraints like performance or access to system calls.

Fifth, the current implementation controls the access of extensions during runtime to the resources of the core application, however it does not provide methods for statically checking extensions.

For example, there is no check whether the extension code will terminate or block the main stream of execution of the core application. Furthermore, the current approach does not support monitoring extensions during runtime for performance (e.g., memory usage, CPU usage, etc.).

Last, developing a custom recommendation tool based on the available logical layers of an application requires a lot of time and effort on the side of the software provider. For a commercial software system covering a large scale of customers, the tool can be worth the investment for attracting more developers in comparison to small commercial solutions. Nevertheless, the investment in such a tool has to be carefully assessed by the software provider.

6.5 Summary

In this chapter the evaluation of XPoints and the recommender tool was presented. In the first part, the implementation of the requirements of extension interfaces for two different kinds of extension developers was presented. As it is shown, the amount of code generated for these simple scenarios is huge. Although a software developer can implement that manually using other techniques, the resulting code will not be of lower complexity. Using the XPoints specification as an input and the recommender tool, the extension developer can navigate and bookmark the interesting extension possibilities using high-level visualizations and generate code stubs that can be used for implementing an extension.

In the second part, a comparison of the approach and the related work with respect to the satisfaction of the defined requirements in Chapters 2 and 3 is presented. On the side of the software provider, XPoints provides a better high-level specification of cross-layer extension possibilities, separation of concerns, multiple extension developer interfaces, and enforcement of extension interfaces. On the side of the extension developer, the recommender tool is more tailored towards supporting the cross-layer extensibility multilayered applications in comparison to the state-of-the-art approaches. Using XPoints and the recommender tool, the approach provides an end-to-end solution for both the software provider and the extension developer.

In the last part, a study is presented that shows the advantages of using XPoints over Java for specifying and enforcing extension interfaces for a Java-based multilayered business application. XPoints developers were 4 times faster than Java developers to implement a given set of requirements for an extension interface and rated their solution to be easier to understand, maintain, and change. At the end, the limitations of the presented approach are discussed.

7 Conclusion

7.1 Summary

In a world that requires software providers to react to market trends and changing requirements to deliver the latest features to customers, a software provider needs to have the necessary capabilities to rapidly develop, evolve, and maintain his software system. Extensibility is a crucial feature of modern software systems that empower customers of a particular software system to integrate new additions that support their organizational requirements. An off-the-shelf software system that cannot be adapted to meet the demands of an organization is useless. Nevertheless, with the increasing complexity of software systems comprising of several logical layers extensibility has to come with the necessary means to support the software provider and extension developer. These means must simplify the specification, enforcement, and maintenance as well as ease the consumption of extension interfaces while controlling access to the underlying resources of the software system.

The study on the software systems on the Qualitas corpus and the review of the state-of-the-art approaches on extensibility showed the necessity for new means for the specification and enforcement of extension interface for multilayered applications. The study on extension developers and the literature review showed the current methods and tools are not effective for assisting an extension developer throughout the different phases of extension development.

The main goal of this dissertation is to provide better means for supporting controlled extensibility of multilayered software systems both for the software provider and extension developer. XPoints declaratively specifies extension possibilities as first-class entities using domain-specific constructs and gives the software provider the possibility to control extension possibilities and their

interdependencies. With the integrated enforcement of the extension interface on the code-level, XPoints raises the burden from the software provider to implement the code necessary for loading, validating, and executing extensions as well as for controlling the underlying resources of the software system. By decoupling the specification of the extension interface and the functional code, XPoints supports the coexistence of multiple extension interfaces for different kinds of extension developers with different extensibility requirements and constraints. The approach proposed in this dissertation also comprises of a recommender tool dedicated for extension development. By navigating through high-level artifacts, an extension developer can explicitly see the extension possibilities that are offered by the software system. To implement a particular extension based on selected extension possibilities, the tool supports the generation of code stubs that can be used to implement the extension without having to go through overheads like documentation, tutorials, and web search.

In contrast to the related work, XPoints fills the gaps between specification and enforcement of extension interfaces and can be seen as introducing a new layer on top of the state-of-the-art approaches which are mainly targeted at the concrete realization of extensibility. XPoints is generic in the sense that it can be applied to different domains and use different techniques for the enforcement of extension interfaces. The usability study of XPoints and Java developers show the potential of this approach for the implementation of extension interfaces for multilayered applications. Along with the recommender tool, the approach provides support for both the extension developer and software provider.

7.2 Future Work

In the context of this work two instantiations of the concepts of XPoints were described. One instantiation is for a conceptual business application consisting of user interfaces, business processes, and business objects and the other one was a generic instantiation for Java-based applications. In the following two main lines of future work are discussed. The first line is to address the current limitations of the described approach. The second line of work is directed towards widening the scope of the work to tackle unaddressed challenges in software extensibility.

7.2.1 Addressing the Limitations

There are several limitations of the approach that were described in Chapter 6 that can be addressed in future work. One limitation of XPoints is that an XPoints interface can become invalid if the code of the software system evolves. In the current implementation the XPoints

compiler generates warnings and errors in case references to the defined artifacts of the core software change (i.e., on the syntactic level). However, the compiler does not check for logical changes in the base code. A direction for future work is to find better means to manage the evolution of the XPoints interface and the base code.

Another limitation is that the syntax and semantics of the language depend on the domain of the application and the enforcement method depends on the implementation of the compiler. One direction of future work is to investigate portable generic set of constructs that can cover several domains of application and offer them as an extensible base for the software provider. Another direction of future work is to implement several extension interface enforcement strategies and offer them to the software provider to select the preferable ones for his domain of application.

The same limitation applies to the proposed recommender tool. While in this work it is argued that the recommender tool improves on the state-of-the-art approaches to support extension development, a more generic instantiation of the tool as well as applications to other domains is still required.

7.2.2 Widening the Scope of Work

In the context of this dissertation the specification and enforcement of extension interfaces for multilayered application were considered. However, there are still a lot of challenging topics that have to be addressed to offer a fully-fledged framework for supporting extensibility.

One direction for future work is targeted towards runtime monitoring and conflict detection of extensions. Runtime monitoring of allows the software system to be aware of the behavior and performance of extensions. This is beneficial, e.g., to detect non-efficient implementations of extensions which can degrade the overall performance of a software system. Conflict detection is targeted towards the detection of possible conflicts between different extensions (e.g., while accessing resources of the software system) that are integrated with a software system. This can prevent possible collisions and unpredictable behavior of the software system during runtime.

In the scope of this work XPoints was mainly considered for the domain of commercial closed-source business software system. Another direction of future work is to investigate other domains of applications and instantiate the concepts of XPoints for them. Moreover, user studies are required to validate the usability and feasibility of the approach in other domains.

A An appendix

A.1 Grammar of XPoints for Business Applications

⟨*ExtensibilityModel*⟩ ::= ‘`extensioninterface`’ ⟨*ID*⟩ ‘`{`’ ⟨*Layer*⟩+ ⟨*Group*⟩* ‘`}`’;

⟨*Layer*⟩ ::= ‘`layer`’ (⟨*BusinessProcess*⟩ | ⟨*UserInterface*⟩ | ⟨*BusinessObject*⟩);

⟨*UserInterface*⟩ ::= ‘`UserInterface`’ ‘`{`’ ⟨*UIExtensibleArtifact*⟩+ ⟨*UIBOConstraint*⟩*‘`}`’;

⟨*UIExtensibleArtifact*⟩ ::= ‘`extensibleartifact`’ ⟨*QualifiedName*⟩ (‘`permission`’ ‘`=`’ ⟨*ID*⟩)? ‘`{`’ ⟨*UIExtensionPoint*⟩+ ⟨*UIBOConstraint*⟩*‘`}`’;

⟨*UIExtensionPoint*⟩ ::= ⟨*UITYPE1*⟩ | ⟨*UITYPE2*⟩ | ⟨*UITYPE3*⟩;

⟨*UITYPE1*⟩ ::= ⟨*ID*⟩ (‘`permission`’ ‘`=`’ ⟨*ID*⟩)? ‘`(`’ ⟨*QualifiedName*⟩ ‘`,`’ ⟨*ID*⟩ ‘`)`’ ‘`;`’;

⟨*UITYPE2*⟩ ::= ⟨*ID*⟩ (‘`permission`’ ‘`=`’ ⟨*ID*⟩)? ‘`(`’ ⟨*QualifiedName*⟩ ‘`,`’ ⟨*INT*⟩ ‘`)`’ ‘`;`’;

⟨*UITYPE3*⟩ ::= ⟨*ID*⟩ (‘`permission`’ ‘`=`’ ⟨*ID*⟩)? ‘`(`’⟨*ID*⟩ ‘`,`’ ⟨*ID*⟩ ‘`,`’ ⟨*ID*⟩‘`)`’ ‘`;`’;

⟨*BusinessProcess*⟩ ::= ‘`BusinessProcess`’ ‘`{`’ ⟨*BPExtensibleArtifact*⟩+ ⟨*BPConstraint*⟩* ‘`}`’;

⟨*BPExtensibleArtifact*⟩ ::= ‘`extensibleartifact`’ ⟨*QualifiedName*⟩ (‘`permission`’ ‘`=`’ ⟨*ID*⟩)? ‘`{`’ ⟨*BPExtensionPoint*⟩+ ⟨*BPConstraint*⟩* ‘`}`’;

⟨*BPExtensionPoint*⟩ ::= ⟨*BPEPTYPE*⟩ ⟨*ID*⟩ (‘`permission`’ ‘`=`’ ⟨*ID*⟩)? ‘`(`’ ⟨*QualifiedName*⟩ ‘`,`’ ⟨*QualifiedName*⟩ ‘`,`’ ⟨*QualifiedName*⟩ ‘`)`’ ‘`;`’;

⟨*BusinessObject*⟩ ::= ‘`BusinessObject`’ ‘`{`’ ⟨*BOExtensibleArtifact*⟩+ ⟨*UIBOConstraint**⟩ ‘`}`’;

⟨*BOExtensibleArtifact*⟩ ::= ‘`extensibleartifact`’ ⟨*QualifiedName*⟩ (‘`permission`’ ‘`=`’ ⟨*ID*⟩)? ‘`{`’ ⟨*BOExtensionPoint*⟩+ ⟨*UIBOConstraint*⟩* ‘`}`’;

⟨*BOExtensionPoint*⟩ ::= ⟨*BOEPTYPE1*⟩ | ⟨*BOEPTYPE2*⟩ | ⟨*BOEPTYPE3*⟩ | ⟨*BOEPTYPE4*⟩;

Figure A.1: Grammar of XPoints for business applications

⟨*BOTYPE1*⟩ ::= ⟨*BOEPTYPE1*⟩ ⟨*ID*⟩ (‘permission’ ‘=’ ⟨*ID*⟩)? ‘(’ ⟨*QualifiedName*⟩ ‘)’ ‘;’;

⟨*BOTYPE2*⟩ ::= ⟨*BOEPTYPE2*⟩ ⟨*ID*⟩ (‘permission’ ‘=’ ⟨*ID*⟩)? ‘(’ ⟨*QualifiedName*⟩ ‘,’ ⟨*QualifiedName*⟩ ‘)’ ‘;’;

⟨*BOTYPE3*⟩ ::= ⟨*BOEPTYPE3*⟩ ⟨*ID*⟩ (‘permission’ ‘=’ ⟨*ID*⟩)? ‘;’;

⟨*BOTYPE4*⟩ ::= ⟨*BOEPTYPE4*⟩ ⟨*ID*⟩ (‘permission’ ‘=’ ⟨*ID*⟩)? ‘(’ ⟨*QualifiedName*⟩ ‘,’ ⟨*INT*⟩ ‘)’;

⟨*BPConstraint*⟩ ::= ‘permissionset’ ⟨*ID*⟩ ‘{’ (⟨*BPPermissionType*⟩ ‘(’⟨*QualifiedName*⟩ ‘,’ ⟨*BPPermission*⟩‘)’ ‘;’)+ ‘}’;

⟨*UIBOConstraint*⟩ ::= ‘permissionset’ ⟨*ID*⟩ ‘{’ (⟨*MethodPermission*⟩ | ⟨*AttributePermission*⟩)+ ‘}’;

⟨*MethodPermission*⟩ ::= ‘methodpermission’ ‘(’ ⟨*QualifiedName*⟩ ‘,’ ⟨*UIBOMethodPermission*⟩ ‘)’ ‘;’;

⟨*AttributePermission*⟩ ::= ‘attributepermission’ ‘(’ ⟨*QualifiedName*⟩ ‘,’ ⟨*UIBOAttributePermission*⟩ ‘)’ ‘;’;

⟨*Group*⟩ ::= ‘Group’ ⟨*ID*⟩ ‘{’ ⟨*ExtensionPointList*⟩ (‘,’ ⟨*GroupConstraint*⟩)? ‘}’;

⟨*ExtensionPointList*⟩ ::= ⟨*ID*⟩ | ⟨*ExtensionPointList*⟩ ‘,’ ⟨*ID*⟩;

⟨*QualifiedName*⟩ ::= ⟨*ID*⟩ | ⟨*QualifiedName*⟩ ‘.’ ⟨*ID*⟩;

⟨*BOEPTYPE1*⟩ ::= ‘afterConstructor’ ‘afterMethodCall’ | ‘beforeMethodCall’;

⟨*BOEPTYPE2*⟩ ::= ‘afterBOAttributeChange’;

⟨*BOEPTYPE3*⟩ ::= ‘allowNewBOLogic’;

⟨*BOEPTYPE4*⟩ ::= ‘allowBOAttributes’;

⟨*UIEPTYPE1*⟩ ::= ‘allowUIComponent’ | ‘afterUIEventHandler’ | ‘beforeUIEventHandler’;

⟨*UIEPTYPE2*⟩ ::= ‘allowUIAttributes’;

⟨*UIEPTYPE3*⟩ ::= ‘afterForm’ | ‘beforeForm’;

⟨*BPEPTYPE*⟩ ::= ‘afterActivity’ | ‘beforeActivity’ | ‘parallelActivity’ | ‘afterEvent’ | ‘beforeEvent’ | ‘afterDecision’;

⟨*UIBOAttributePermission*⟩ ::= ‘read’ | ‘readwrite’ | ‘write’ | ‘hidden’;

⟨*UIBOMethodPermission*⟩ ::= ‘callable’ | ‘hidden’;

⟨*BPPermissionType*⟩ = ‘datapermission’ | ‘activitypermission’ | ‘taskpermission’ | ‘lanepermission’

⟨*BPPermission*⟩ ::= ‘visible’ | ‘hidden’;

⟨*GroupConstraint*⟩ ::= ‘ExtendAll’;

⟨*INT*⟩ ::= is the standard Java definition of an integer;

⟨*ID*⟩ ::= is the standard Java naming convention of variables;

⟨*MethodSignature*⟩ ::= is the standard Java method signature without the modifiers;

⟨*ConstructorSignature*⟩ ::= is the standard Java constructor signature without the modifiers;

Figure A.2: Grammar of XPoints for business applications (continued)

A.2 Grammar of XPoints for Java

⟨*ExtensionInterface*⟩ ::= ‘`extensionInterface`’ ⟨*ID*⟩ ‘`{`’ (⟨*PackageLayer*⟩ | ⟨*ExtensionPointGroup*⟩)* ‘`}`’;

⟨*PackageLayer*⟩ ::= ‘`package`’ ⟨*QualifiedName*⟩ ‘`{`’ (⟨*ExtensibleArtifact*⟩ | ⟨*ExtensibleArtifactPermissionSet*⟩)* ‘`}`’;

⟨*ExtensibleArtifact*⟩ ::= ‘`class`’ ⟨*ID*⟩ ‘`{`’ (⟨*ExtensionPoint*⟩ | ⟨*ExtensionPointPermissionSet*⟩)+ ‘`}`’;

⟨*ExtensionPoint*⟩ ::= ⟨*ConstructorExtensionPoint*⟩ | ⟨*MethodExtensionPoint*⟩ | | ⟨*AttributeExtensionPoint*⟩ | ⟨*AttributeMethodExtensionPoint*⟩;

⟨*ExtensionPointGroup*⟩ ::= ‘`group`’ ⟨*ID*⟩ ‘`{`’ [⟨*QualifiedNameList*⟩] ‘`}`’;

⟨*ExtensibleArtifactPermissionSet*⟩ ::= ‘`permissionset`’ ‘`(`’ ⟨*ID*⟩ ‘`)`’ ‘`{`’ (⟨*AttributePermission*⟩ | ⟨*MethodPermission*⟩)+ ‘`}`’;

⟨*ExtensionPointPermissionSet*⟩ ::= ‘`permissionset`’ ‘`(`’ ⟨*IDList*⟩ ’)’ ‘`{`’ (⟨*AttributePermission*⟩ | ⟨*MethodPermission*⟩)+ ‘`}`’;

⟨*MethodPermission*⟩ ::= ‘`methodpermission`’ ‘`(`’ (⟨*MethodSignature*⟩ | ’*’) ‘`,`’ ⟨*MethodPermissionModifier*⟩ ‘`)`’ ‘`;`’;

⟨*AttributePermission*⟩ ::= ‘`attributepermission`’ ‘`(`’ (⟨*ID*⟩|‘`*`’) ‘`,`’ ⟨*AttributePermissionModifier*⟩ ‘`)`’ ‘`;`’;

⟨*ConstructorExtensionPoint*⟩ ::= ⟨*ConstructorExtensionPointType*⟩ ⟨*ID*⟩ ‘`=`’ ⟨*ConstructorSignature*⟩ ‘`;`’;

⟨*MethodExtensionPoint*⟩ ::= ⟨*MethodExtensionPointType*⟩ ⟨*ID*⟩ ‘`=`’ ⟨*MethodSignature*⟩ ‘`;`’;

⟨*AttributeExtensionPoint*⟩ ::= ⟨*AttributeExtensionPointType*⟩ ⟨*ID*⟩ ‘`=`’ ID ‘`<`’ ⟨*QualifiedName*⟩ ‘`>`’ ‘`;`’;

⟨*AttributeMethodExtensionPoint*⟩ ::= ⟨*AttributeMethodExtensionPointType*⟩ ⟨*ID*⟩ ‘`=`’ ⟨*ID*⟩ ‘`<`’ ⟨*QualifiedName*⟩ ‘`>`’ ‘`,`’ ⟨*MethodSignature*⟩ ‘`;`’;

⟨*IDList*⟩ ::= ⟨*ID*⟩ | ⟨*IDList*⟩ ‘`,`’ ⟨*ID*⟩;

⟨*ParameterList*⟩ ::= ⟨*Parameter*⟩ | ‘`..`’ ;

⟨*Parameter*⟩ ::= ⟨*ID*⟩ | ⟨*Parameter*⟩ ‘`,`’ ⟨*ID*⟩ ;

Figure A.3: Grammar of XPoints for Java

⟨*QualifiedNameList*⟩ ::= ⟨*QualifiedName*⟩ | ⟨*QualifiedNameList*⟩ ‘,’ ⟨*QualifiedName*⟩ ;

⟨*QualifiedName*⟩ ::= ⟨*ID*⟩ | ⟨*QualifiedName*⟩ ‘.’ ⟨*ID*⟩ ;

⟨*MethodExtensionPointType*⟩ ::= ‘afterMethodCall’ | ‘afterMethodExe’
| ‘beforeMethodCall’ | ‘beforeMethodExe’ | ‘override’;

⟨*ConstructorExtensionPointType*⟩ ::= ‘afterConstructor’ | ‘beforeConstructor’;

⟨*AttributeExtensionPointType*⟩ ::= ‘addItemAfterInitialization’;

⟨*AttributeMethodExtensionPointType*⟩ ::= ‘addItemAfterMethod’ |
‘addItemBeforeMethod’;

⟨*AttributePermissionModifier*⟩ ::= ‘HIDDEN’ | ‘READWRITE’ | ‘READ’ | ‘WRITE’;

⟨*MethodPermissionModifier*⟩ ::= ‘HIDDEN’ | ‘CALLABLE’;

⟨*ID*⟩ ::= is the standard Java naming convention of variables;

⟨*MethodSignature*⟩ ::= is the standard Java method signature without the modifiers;

⟨*ConstructorSignature*⟩ ::= is the standard Java constructor signature without the modifiers;

Figure A.4: Grammar of XPoints for Java (continued)

A.3 Questionnaire: User Study on Extension Developers

Case Study Registration (max 5 minutes)

Name

Age

Software development experience (years)

Have you ever used a business software system (e.g. ERP, CRM, HR, etc.)?

If yes, please mention the name of a business software system you have used as well as any particular module that you can currently recall.

PART I (max 15 minutes)

Understanding the Behavior of an Extension Developer

- Have you ever developed a plug-in or an extension for any software?

(Hint: Google Chrome, Firefox, Eclipse, Emacs, etc.)

YES NO

- If yes, what were the resources you used to learn about developing the extension?

_____

- Have you ever extended a business software and / how did you learn about developing the extension?

YES NO

_____

- You are given an application for which you are required to develop an extension for. You have only used the application as a user. What would be a good starting point for you to learn about building an extension?

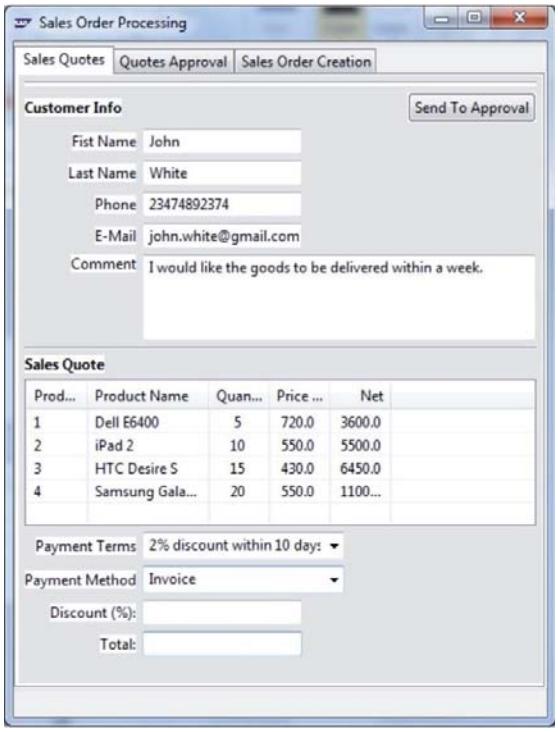
How likely would you prefer using the following methods to learn about building extensions if they were offered to you (please circle a number)?

	Less likely Most Likely
____ API Documentation	-3 -2 -1 0 1 2 3
____Training material in form of PowerPoint slides	-3 -2 -1 0 1 2 3
____Training material in textual form (e.g. online webpage / tutorial)	-3 -2 -1 0 1 2 3
____Code examples of different extension scenarios	-3 -2 -1 0 1 2 3
____YouTube video tutorials	-3 -2 -1 0 1 2 3
____Ask an experienced friend / colleague	-3 -2 -1 0 1 2 3
____General web search	-3 -2 -1 0 1 2 3
____Official product specific forum / search	-3 -2 -1 0 1 2 3
____IDE Support / Extension Wizard / Building Tool	-3 -2 -1 0 1 2 3
____ Learning by doing (i.e. trial and error)	-3 -2 -1 0 1 2 3
____ Looking at the source code of the system	-3 -2 -1 0 1 2 3

Appendix A. An appendix

- If you have to find a solution for a specific extension development task and you were given the methods mentioned above, in what order will you go through them? Please rank the methods in the order you will visit them by placing a number to the left of the method.

- You are given the following user interface:



Your customer wants you to extend this user interface with a new text field that shows the credit rating of the customer. The credit rating will be manually entered by the end user each time he fills in the customer information.

After getting this requirement, can you describe what will you want to learn about the application as an extension developer to successfully build the extension required (i.e. what do you expect to be given to you).

_____
_____
_____
_____

PART II

BusinessOne Tasks (max 10 minutes briefing)

For this part you will be given a real business software system and you will be required to extend the system. You will be briefed about the software system as then get the requirements for the extension.

For the following tasks you will get the **SDK documentation**, **training PowerPoint slides**, **a link to BusinessOne forums (SAP SCN)**, **sample code examples**, and access to **Google search**.

Task 1 – General Architecture and Design (**max 15 minutes**)

In this task you are required to answer some questions about BusinessOne.

- Which BusinessOne API can be used for **building UI** extensions?

_____

- Which BusinessOne API can be used to **access and extend the data** of BusinessOne?

_____

- Can you mention the types of the **high-level artifacts (that you can probably extend)** that exist in Business One (e.g. user interfaces, database tables, etc.)

_____

Task 2 – From Layer to Code (**max 15 minutes**)

In this task you are required to add **a new button** to the **Sales Order form** shown to you. You can choose either Visual Basic or C# as a programming language.

- Can you identify which **class** is required to get a Sales Order form object?

_____

- Which **method** is required to add a new button on the Sales Order form?

_____

- Can you identify how to position the button on the user interface?

_____

Task 3 Interlayer Dependencies (**max 15 minutes**)

In this task you are required to add **a new text field** to the **Sales Order form** shown to you.

- Which high-level software artifacts (e.g. database tables) does this form depend on? If possible, give a concrete example of each artifact that you find out(e.g. database table name).

_____

- Which of the dependent artifacts should be extended to hold the data of the new text field?

_____

A.4 Questionnaire: User Study on Extension Interface Developers

Pre-Study

Age _____

Are you familiar with object-oriented programming? _____

Years of software development experience. _____

Have you ever implemented an extension for any kind of software? If yes please name the software system(s) / technologies used.

_____

Are you familiar with any kind of design patterns that support extensibility?

_____

Have you ever done any kind of software refactoring activities?

_____

If yes, how do you keep track of your modifications (i.e., how do you maintain your modifications)?

_____

Task I – Implementation of a Simple Extensibility Scenario

a) You are given the following Sales Order form class `org.jallinone.sales.documents.client.SaleOrderDocFrame`. In this task you are required to make the "Documentation Identification" part of the Sales Order form extensible. The extension developer will be given the possibilities to extend the `saleIdHeadPanel1` with a custom `JTextField` and a `JLabel`. You are required to modify the code of the core classes appropriately to support the extensibility scenario. For this task we assume the following:

1. The extender will not get source code of the system, therefore an appropriate code "entry point" should be provided (e.g., an interface or an abstract class).
2. The extension code must be loaded and executed after the initialization of the main panel.
3. You should explain how will the extension code will be loaded and executed.
4. For simplicity assume that only a single extension will be executed.

b) Before the confirmation of the sales order, the extension developer will be allowed to add his custom logic. The extension code should be executed before the execution of the method `confirmButton_actionPerformed(ActionEvent ae)`.

Task II – Implementation of Extensibility Constraints

a) This constraint requires that make some methods hidden and some attributes as read only. The extension developer will not be allowed to call the method `loadDataCompleted()` and has a read only access to the `SaleOrderDocController controller` object. The valid extension should only be able to access these resources as described.

Task III – Implementation of Advanced Extensibility Constraints

a) This constraint requires that a valid extension should implement both extension possibilities that were supported in the first task (i.e., the extender should implement the GUI extension and add his custom logic before the confirmation method of the sales order form). The system should check for the validity of the extension during runtime.

Appendix A. An appendix

Solution Evaluation

1. What would be the best way to document the changes that you did to the core application (e.g., for fellow developers)?

_____

2. What code artifacts (i.e., classes, interfaces, methods, examples, etc.) will you give the extender as an entry point to develop his extension for the software after you did the refactoring?

_____

3. Do you need to supply other complementary material to an extension developer to understand how to build an extension (assuming that the extension developer is already an expert in Java)?

_____

Developer Experience

	Simple				**Complex**
1. How would you rate the provided code of the application?	1	2	3	4	5
2. How easy was it to implement the extensibility requirements in Task I. a?	1	2	3	4	5
3. How easy was it to implement the extensibility requirements in Task I. b?	1	2	3	4	5
4. How easy was it to implement the extensibility requirements in Task II?	1	2	3	4	5
5. How easy was it to implement the extensibility requirements in Task III?	1	2	3	4	5
6. How easy was it to implement the extensibility requirements (overall rating)?	1	2	3	4	5
7. How easy do you think it will be for another developer to maintain and understand your modifications?	1	2	3	4	5
8. If the requirements for extensibility are to change, how easy do you think it will be to integrate these new requirements with your implementation?	1	2	3	4	5

Bibliography

[Aguilar-Saven, 2004] Aguilar-Saven, R. S. (2004). Business process modelling: Review and framework. *International Journal of Production Economics*, 90(2):129–149.

[Aldrich, 2005] Aldrich, J. (2005). Open modules: modular reasoning about advice. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 144–168, Berlin, Heidelberg. Springer-Verlag.

[Aly et al., 2013a] Aly, M., Charfi, A., Erdweg, S., and Mezini, M. (2013a). XPoints: Extension interfaces for multilayered applications. In *Proceedings of the 2013 17th IEEE International Enterprise Distributed Object Computing Conference*, EDOC '13, pages 237–246, Washington, DC, USA. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/EDOC.2013.34.

[Aly et al., 2012] Aly, M., Charfi, A., and Mezini, M. (2012). On the extensibility requirements of business applications. In *Proceedings of the 2012 workshop on Next Generation Modularity Approaches for Requirements and Architecture*, NEMARA'12, pages 1–6, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2162004.2162006.

[Aly et al., 2013b] Aly, M., Charfi, A., and Mezini, M. (2013b). Building extensions for applications: Towards the understanding of extension possibilities. In *Proceedings of the 2013 IEEE 21st International Conference on Program Comprehension*, ICPC '13, pages 182–191, Washington, DC, USA. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/ICPC.2013.6613846.

[Aly et al., 2013c] Aly, M., Charfi, A., Wu, D., and Mezini, M. (2013c). Understanding multilayered applications for building extensions. In *Proceedings of the 1st workshop on Comprehension of complex systems*, CoCoS '13, pages 1–6, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2451592.2451594.

[Apel and Kästner, 2009] Apel, S. and Kästner, C. (2009). An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84. Refereed Column.

[Bloch, 2008] Bloch, J. (2008). *Effective Java*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition.

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

[Brichau and Haupt, 2005] Brichau, J. and Haupt, M. (2005). Survey of aspect-oriented languages and execution models. Technical Report AOSD-Europe-VUB-01, European Network of Excellence in AOSD.

[Bruch et al., 2009] Bruch, M., Monperrus, M., and Mezini, M. (2009). Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA. ACM.

[Calder et al., 2003] Calder, M., Kolberg, M., Magill, E. H., and Reiff-Marganiec, S. (2003). Feature interaction: A critical review and considered forecast. *Comput. Netw.*, 41(1):115–141.

[Carniel, 2007] Carniel, M. (2007). JAllInOne. Available from: http://jallinone.sourceforge.net/ [cited 10.04.2014].

[Casanave, 1997] Casanave, C. (1997). Business-object architectures and standards. In Sutherland, J., Casanave, C., Miller, J., Patel, P., and Hollowell, G., editors, *Business Object Design and Implementation*, pages 7–28. Springer London.

[Clements and Northrop, 2007] Clements, P. and Northrop, L. M. (2007). *Software Product Lines: Practices and Patterns*. Addison-Wesley, 6 edition.

[Crnković et al., 2011] Crnković, I., Sentilles, S., Vulgarakis, A., and Chaudron, M. R. (2011). A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615.

[Dekel and Herbsleb, 2009] Dekel, U. and Herbsleb, J. D. (2009). Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 320–330, Washington, DC, USA. IEEE Computer Society.

[DeLine et al., 2005] DeLine, R., Czerwinski, M., and Robertson, G. (2005). Easing program comprehension by sharing navigation data. In *Proceedings of the 2005 IEEE Symposium on*

Visual Languages and Human-Centric Computing, VLHCC '05, pages 241–248, Washington, DC, USA. IEEE Computer Society.

[Ducasse et al., 2006] Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28:331–388.

[Ebraert and Merino, 2008] Ebraert, P. and Merino, L. (2008). Software variation by means of first-class change objects. In *Software Variability: a Programmers' Perspective Symposium*.

[Ebraert et al., 2007] Ebraert, P., Vallejos, J., Costanza, P., Van Paesschen, E., and D'Hondt, T. (2007). Change-oriented software engineering. *Proceedings of the 2007 international conference on Dynamic languages in conjunction with the 15th International Smalltalk Joint Conference 2007 - ICDL '07*, page 3.

[Eclipse Foundation, 2014a] Eclipse Foundation (2014a). Eclipse Integrated Development Environment. Available from: http://www.eclipse.org/ [cited 10.04.2014].

[Eclipse Foundation, 2014b] Eclipse Foundation (2014b). Eclipse WindowBuilder. Available from: http://www.eclipse.org/windowbuilder/ [cited 10.04.2014].

[Eclipse Foundation, 2014c] Eclipse Foundation (2014c). STP/BPMN Component/STP BPMN Presentation Hands on tutorial. Available from: http://wiki.eclipse.org/STP/BPMN_Component/STP_BPMN_Presentation_Hands_on_tutorial [cited 10.04.2014].

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

[Eysholdt and Behrens, 2010] Eysholdt, M. and Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309, New York, NY, USA. ACM.

[Fayad and Schmidt, 1997] Fayad, M. and Schmidt, D. C. (1997). Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38.

[Feldt, 2007] Feldt, K. (2007). *Programming Firefox: Building Rich Internet Applications with Xul*. O'Reilly Media, Inc.

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

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

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

[Gamma, 1997] Gamma, E. (1997). The extension objects pattern - third conference on patterns languages of programs - PLoP '96. Technical Report #wucs-97-07, Dept. of Computer Science, Washington University Department of Computer Science, Monticello, Illinois. Available from: http://www.cs.wustl.edu/~schmidt/PLoP-96/gamma.ps.gz [cited 10.04.2014].

[Glickstein, 1997] Glickstein, B. (1997). *Writing GNU Emacs Extensions*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[Guzzi et al., 2011] Guzzi, A., Hattori, L., Lanza, M., Pinzger, M., and Deursen, A. v. (2011). Collective code bookmarks for program comprehension. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 101–110, Washington, DC, USA. IEEE Computer Society.

[Harrison and Ossher, 1993] Harrison, W. and Ossher, H. (1993). Subject-oriented programming: a critique of pure objects. In *ACM Sigplan Notices*, volume 28, pages 411–428. ACM.

[Hoffman and Eugster, 2007] Hoffman, K. and Eugster, P. (2007). Bridging Java and AspectJ through explicit join points. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, PPPJ'07, pages 63–72, New York, NY, USA. ACM.

[Hoffmann et al., 2007] Hoffmann, R., Fogarty, J., and Weld, D. S. (2007). Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST '07, pages 13–22, New York, NY, USA. ACM.

[Holmes and Murphy, 2005] Holmes, R. and Murphy, G. C. (2005). Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 117–125, New York, NY, USA. ACM.

[Hou and Li, 2011] Hou, D. and Li, L. (2011). Obstacles in using frameworks and APIs: An exploratory study of programmers' newsgroup discussions. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 91–100, Washington, DC, USA. IEEE Computer Society.

[Ichisugi and Tanaka, 2002] Ichisugi, Y. and Tanaka, A. (2002). Difference-based modules: A class independent module mechanism. In *In Proceedings ECOOP 2002, volume 2374 of LNCS, Malaga*, pages 62–88. Springer Verlag.

[Inostroza et al., 2011] Inostroza, M., Tanter, É., and Bodden, E. (2011). Modular reasoning with join point interfaces. Technical Report TUD-CS-2011-0272, Center for Advanced Security Research Darmstadt.

[Ishio et al., 2012] Ishio, T., Etsuda, S., and Inoue, K. (2012). A lightweight visualization of interprocedural data-flow paths for source code reading. In *Proceedings of the 2012 IEEE 20th International Conference on Program Comprehension*, ICPC '12, pages 37–46, Washington, DC, USA. IEEE Computer Society.

[Ivar et al., 1997] Ivar, J., Martin, G., and Patrik, J. (1997). *Software reuse: Architecture, process and organization for business success*. Addison-Wesley Publishing Company.

[Janzen and De Volder, 2003] Janzen, D. and De Volder, K. (2003). Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, pages 178–187, New York, NY, USA. ACM.

[Juristo and Moreno, 2010] Juristo, N. and Moreno, A. M. (2010). *Basics of Software Engineering Experimentation*. Springer Publishing Company, Incorporated, 1st edition.

[Kang et al., 1990] 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 CMU/SEI-90-TR-21, DTIC Document.

[Kellens et al., 2006] Kellens, A., Mens, K., Brichau, J., and Gybels, K. (2006). Managing the evolution of aspect-oriented software with model-based pointcuts. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 501–525, Berlin, Heidelberg. Springer-Verlag.

[Kersten and Murphy, 2005] Kersten, M. and Murphy, G. C. (2005). Mylar: a degree-of-interest model for IDEs. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 159–168, New York, NY, USA. ACM.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK. Springer-Verlag.

[Kiczales and Lamping, 1992] Kiczales, G. and Lamping, J. (1992). Issues in the design and specification of class libraries. In *conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA'92, pages 435–451, New York, NY, USA. ACM.

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

[Kiczales and Mezini, 2005] Kiczales, G. and Mezini, M. (2005). Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 49–58, New York, NY, USA. ACM.

[Kim et al., 2010] Kim, J., Lee, S., won Hwang, S., and Kim, S. (2010). Towards an intelligent code search engine. In Fox, M. and Poole, D., editors, *AAAI*. AAAI Press.

[Kitchenham et al., 1997] Kitchenham, B., Linkman, S., and Law, D. (1997). DESMET: a methodology for evaluating software engineering methods and tools. *Computing Control Engineering Journal*, 8(3):120–126.

[Krasner and Pope, 1988] Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49.

[Krishnamurthi and Felleisen, 1998] Krishnamurthi, S. and Felleisen, M. (1998). Toward a formal theory of extensible software. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '98/FSE-6, pages 88–98, New York, NY, USA. ACM.

[Krueger, 1992] Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv.*, 24(2):131–183.

[Kulesza et al., 2006] Kulesza, U., Alves, V., Garcia, A., Lucena, C., and Borba, P. (2006). Improving extensibility of object-oriented frameworks with aspect-oriented programming. In Morisio, M., editor, *Reuse of Off-the-Shelf Components*, volume 4039 of *Lecture Notes in Computer Science*, pages 231–245. Springer Berlin Heidelberg.

[Lazzarini Lemos et al., 2007] Lazzarini Lemos, O. A., Bajracharya, S. K., and Ossher, J. (2007). CodeGenie: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 917–918, New York, NY, USA. ACM.

[Lee et al., 2002] Lee, K., Kang, K., and Lee, J. (2002). Concepts and guidelines of feature modeling for product line software engineering. In Gacek, C., editor, *Software Reuse: Methods, Techniques, and Tools*, volume 2319 of *LNCS*, pages 62–77. Springer Berlin / Heidelberg.

[Liang and Bracha, 1998] Liang, S. and Bracha, G. (1998). Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 36–44, New York, NY, USA. ACM.

[Madsen and Møller-Pedersen, 1989] Madsen, O. L. and Møller-Pedersen, B. (1989). Virtual classes: A powerful mechanism in object-oriented programming. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 397–406, New York, NY, USA. ACM.

[Mandelin et al., 2005] Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. (2005). Jungloid mining: helping to navigate the API jungle. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 48–61, New York, NY, USA. ACM.

[Mayfield et al., 1991] Mayfield, T., Roskos, J. E., Welke, S. R., Boone, J. M., and McDonald, C. W. (1991). Integrity in automated information systems. Technical Report 79-91, National Security Agency. IDA Paper P-2316.

[McVeigh, 2009] McVeigh, A. (2009). *A Rigorous, Architectural Approach to Extensible Applications*. PhD thesis, Imperial College London.

[Mezini, 1997] Mezini, M. (1997). Maintaining the consistency of class libraries during their evolution. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'97, pages 1–21, New York, NY, USA. ACM.

[Mezini and Ostermann, 2004] Mezini, M. and Ostermann, K. (2004). Variability management with feature-oriented programming and aspects. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 127–136, New York, NY, USA. ACM.

[Micallef, 1988] Micallef, J. (1988). Encapsulation, reusability, and extensibility in object-oriented programming languages. *Journal of Object-Oriented Programming*, 1(1):12–36.

[Microsoft Corporation, 2014] Microsoft Corporation (2014). Managed extensibility framework. Available from: https://mef.codeplex.com/ [cited 10.04.2014].

[Noguera et al., 2012] Noguera, C., Roover, C. D., Kellens, A., and Jonckers, V. (2012). Code querying by UML. In *Proceedings of the 2012 IEEE 20th International Conference on Program Comprehension*, ICPC '12, pages 229–238, Washington, DC, USA. IEEE Computer Society.

[Oezbek and Prechelt, 2007] Oezbek, C. and Prechelt, L. (2007). JTourBus: Simplifying program understanding by documentation that provides tours through the source code. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 64–73.

[(OMG), 2011] (OMG), O. M. G. (2011). Business process model and notation (BPMN) version 2.0. Technical Report formal/2011-01-03, Object Management Group (OMG). Available from: http://www.omg.org/spec/BPMN/2.0 [cited 10.04.2014].

[OSGi Alliance, 2003] OSGi Alliance (2003). *OSGi service platform, release 3*. IOS Press, Inc.

[Ossher et al., 1995] Ossher, H., Kaplan, M., Harrison, W., Katz, A., and Kruskal, V. (1995). Subject-oriented composition rules. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 235–250, New York, NY, USA. ACM.

[Ossher and Tarr, 1999] Ossher, H. and Tarr, P. (1999). Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 687–688, New York, NY, USA. ACM.

[Ossher and Tarr, 2000] Ossher, H. and Tarr, P. (2000). Hyper/J: Multi-dimensional separation of concerns for Java. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 734–737, New York, NY, USA. ACM.

[Parnas, 1978] Parnas, D. L. (1978). Designing software for ease of extension and contraction. In *Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, pages 264–277, Piscataway, NJ, USA. IEEE Press.

[Pfleeger, 1995] Pfleeger, S. L. (1995). Experimental design and analysis in software engineering, part 5: Analyzing the data. *SIGSOFT Software Engineering Notes*, 20(5):14–17.

[PMD, 2014] PMD (2014). PMD. Available from: http://pmd.sourceforge.net/ [cited 10.04.2014].

[Robbes and Lanza, 2007] Robbes, R. and Lanza, M. (2007). A change-based approach to software evolution. *Electron. Notes Theor. Comput. Sci.*, 166:93–109.

[Robillard et al., 2010] Robillard, M., Walker, R., and Zimmermann, T. (2010). Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86.

[Robillard, 2009] Robillard, M. P. (2009). What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34.

[Robillard and Murphy, 2002] Robillard, M. P. and Murphy, G. C. (2002). Capturing concern descriptions during program navigation. In *Workshop on Tools for Aspect-Oriented Software Development (OOPSLA 2002)*. ACM.

[Robillard and Murphy, 2003] Robillard, M. P. and Murphy, G. C. (2003). FEAT: a tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 822–823, Washington, DC, USA. IEEE Computer Society.

[Rothlisberger et al., 2011] Rothlisberger, D., Nierstrasz, O., and Ducasse, S. (2011). Smart-Groups: Focusing on task-relevant source artifacts in IDEs. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 61–70, Washington, DC, USA. IEEE Computer Society.

[Rupakheti and Hou, 2012] Rupakheti, C. R. and Hou, D. (2012). CriticAL: A critic for APIs and libraries. In *Proceedings of the 2012 IEEE 20th International Conference on Program Comprehension*, ICPC '12, pages 241–243, Washington, DC, USA. IEEE Computer Society.

[Sahavechaphan and Claypool, 2006] Sahavechaphan, N. and Claypool, K. (2006). XSnippet: mining for sample code. *SIGPLAN Notices*, 41(10):413–430.

[SAP AG, 2014] SAP AG (2014). SAP Business One. Available from: http://www.sap.com/solution/sme/software/erp/small-business-management/overview/index.html [cited 10.04.2014].

[Schärli et al., 2003] Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A. (2003). Traits: Composable units of behaviour. In Cardelli, L., editor, *ECOOP'03*, volume 2743 of *LNCS*, pages 248–274. Springer Berlin Heidelberg.

[Schmidt and Godehardt, 2011] Schmidt, B. and Godehardt, E. (2011). Interaction data management. In König, A., Dengel, A., Hinkelmann, K., Kise, K., Howlett, R., and Jain, L., editors, *Knowlege-Based and Intelligent Information and Engineering Systems*, volume 6882 of *LNCS*, pages 402–409. Springer Berlin Heidelberg.

[Shull et al., 2007] Shull, F., Singer, J., and Sjøberg, D. I. (2007). *Guide to Advanced Empirical Software Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Singer et al., 2005] Singer, J., Elves, R., and Storey, M.-A. (2005). Navtracks: Supporting navigation in software maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 325–334, Washington, DC, USA. IEEE Computer Society.

[Steimann et al., 2010] Steimann, F., Pawlitzki, T., Apel, S., and Kästner, C. (2010). Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering Methodologies*, 20(1):1:1–1:43.

[Steyaert et al., 1996] Steyaert, P., Lucas, C., Mens, K., and D'Hondt, T. (1996). Reuse contracts: managing the evolution of reusable assets. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'96, pages 268–285, New York, NY, USA. ACM.

[Storey, 2005] Storey, M.-A. (2005). Theories, methods and tools in program comprehension: Past, present and future. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 181–191, Washington, DC, USA. IEEE Computer Society.

[Storey et al., 2009] Storey, M.-A., Ryall, J., Singer, J., Myers, D., Cheng, L.-T., and Muller, M. (2009). How software developers use tagging to support reminding and refinding. *IEEE Trans. Softw. Eng.*, 35(4):470–483.

[Storey et al., 1997] Storey, M.-A. D., Wong, K., and Muller, H. A. (1997). How do program understanding tools affect how programmers understand programs? In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, WCRE '97, pages 12–, Washington, DC, USA. IEEE Computer Society.

[Stylos and Myers, 2006] Stylos, J. and Myers, B. A. (2006). Mica: A web-search tool for finding API components and examples. In *Proceedings of the Visual Languages and Human-Centric Computing*, VLHCC '06, pages 195–202, Washington, DC, USA. IEEE Computer Society.

[Sullivan et al., 2010] Sullivan, K., Griswold, W. G., Rajan, H., Song, Y., Cai, Y., Shonle, M., and Tewari, N. (2010). Modular aspect-oriented design with XPIs. *ACM Trans. Softw. Eng. Methodol.*, 20(2):5:1–5:42.

[Sutherland, 1995] Sutherland, J. (1995). Business objects in corporate information systems. *ACM Computing Surveys*, 27:274–276.

[Szyperski et al., 2002] Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component software: beyond object-oriented programming*. Addison-Wesley Professional.

[Tarr et al., 1999] Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr., S. M. (1999). N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 107–119, New York, NY, USA. ACM.

[Tempero et al., 2010] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). Qualitas corpus: A curated collection of Java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345.

[Tichy, 1992] Tichy, W. F. (1992). Programming-in-the-large: Past, present, and future. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 362–367, New York, NY, USA. ACM.

[Zelkowitz and Wallace, 1998] Zelkowitz, M. and Wallace, D. (1998). Experimental models for validating technology. *Computer*, 31(5):23–31.

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

[Zhong et al., 2009] Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. (2009). MAPO: Mining and recommending API usage patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 318–343, Berlin, Heidelberg. Springer-Verlag.

Academic Résumé

- *2011 - 2014*
  Technische Universität Darmstadt, Germany
  Ph.D. Student at the Software Technology Group of Prof. Mira Mezini
  Research Associate at SAP Research, Darmstadt

- *2008 - 2009*
  Johannes Kepler Universität Linz, Austria
  MSc. in Informatics:Engineering and Management.

- *2003 - 2008*
  German University in Cairo, Egypt
  BSc. in Media Engineering and Technology, Major: Computer Science and Engineering

- *2000 - 2003*
  Al Nahda National School Abu Dhabi, United Arab Emirates
  IGCSE (Cambridge Examination Boards) and GCE (Edexcel - London Examination Boards)

- *June 15, 1987*
  Born in Abu Dhabi, United Arab Emirates