XPoints: Extension Interfaces for Multilayered Applications

Mohamed Aly*†, Anis Charfi*, Sebastian Erdweg†, and Mira Mezini†
*Applied Research, SAP AG
firstname.lastname@sap.com
†Software Technology Group, TU Darmstadt
lastname@informatik.tu-darmstadt.de

***Abstract*—Extensibility is a key requirement in modern software applications. In the context of business applications it is one of the major selection criteria from the customer perspective. However, there are some challenges concerning the specification and enforcement of extension interfaces. Extension interfaces define the resources of the base applications 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 hot research topic for "traditional" software constructed using a single programming language, they are completely missing for complex consisting of several abstraction layers. In addition, state-of-the-art approaches do not support providing different extension interfaces for different stakeholders.**

This paper attempts to fill this gap by introducing XPoints, an approach and a language for specifying and enforcing extension interfaces in multilayered applications. An extension interface in XPoints defines the available extension points on the different abstraction layers, controls the access and visibility of the core application to the extension, and constrains the interplay between extension points possibly from different abstraction layers. Several extension interfaces can be overlaid over the same core application, hence, enabling multiple extender views to co-exist. Using an XPoints interface, a software provider can automatically generate the extensibility code infrastructure to provide the extension interface for the core application.

I. Introduction

Applications targeted for a large scale and a wide range of customers such as business applications typically support a set of standard business processes (e.g. sales order processing, recruitment, etc.). Once an organization acquires such an application, it has to customize and / or extend it to match their specific needs. To achieve that, the software provider has to design the software system to support *variability* and *extensibility*. In the context of this paper, we focus on extensibility. We refer to extensibility as the addition of new functionalities to a software system to support new requirements.

In most commercial business software systems, a software provider does not give the source code of his applications to the extension developers. However, the software provider gives the extension developers access to artifacts like, e.g., API libraries, frameworks, etc. along with documentation, tutorials, and other materials to help an extension developer understand what extension possibilities exist, and how to develop and integrate extensions. The extensions are likely to interact with the core software (e.g. access internal data resources) and can as well affect the main execution stream. In the case of business applications, especially those that implement strict legal regulations (e.g. tax calculations), extensibility has to be controlled in a rigorous way. This is required for example to prevent undesirable system behavior, data inconsistencies, and restrict access to sensitive system information. In our previous works [1], [2], we have outlined the challenges and requirements for enabling extensibility for complex multilayered applications.

With respect to applications that require a controlled form of extensibility, there are two perspectives that should be considered: the software provider perspective and the extension developer perspective. From the perspective of the software provider, the application consists of several logical layers (e.g. user interface (UI), business process, business object, database etc.) containing many artifacts that can be made extensible for the extension developer. However, in the context of complex business applications, a software provider can have several kinds of extension developer groups that can build extensions for the software (e.g. internal development teams and external partner companies). In all cases, the software provider has to develop the necessary mechanisms to support extensibility such that they express and support the following for the extension developer.

- *M1. Extension possibilities*: the artifacts that are allowed to be extended (e.g. UI forms, business process activities, database tables, etc.).
- *M2. Interdependencies*: the relationships and constraints that exist between these extensible artifacts.
- *M3. Extension types*: the types of extensions that are allowed to be added to these artifacts (e.g. new methods, attributes, UI elements, process artifacts, new columns in a database table, etc.).
- *M4. Extension method*: the required coding elements and how to extend these artifacts (e.g. inheritance, plug-ins etc.).
- *M5. Extension control*: what underlying application resources are available for the extension code (e.g. variables, methods, etc.) as well as their access rights and usage rules.

- *M6. Extension integration and execution*: when and where will the extension code run.

An *extension interface* specifies the extensibility of a source code artifact according to M1–M6 above. The extension developer, on the other hand, has to understand the extension interface of the system as well as its correct usage to successfully develop and integrate his extension with the core software.

Turning to object-oriented languages (e.g. Java), there are two kinds of mechanisms 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(c.f. [3]). 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 the software system.

In this paper we argue that the state-of-the-art approaches have several limitations for realizing the extension interfaces of complex multilayered applications. First, the technical realization of the extension interface is coupled with the functional code of the core software. Second, these conventional means for controlling extensibility e.g., via Java modifiers, are not expressive enough to enable fine-grained control on what can be extended and how. Third, it is not possible to provide different extension interfaces to different groups of extension developers. Fourth, software applications are nowadays extremely complex and involve several architectural layers, demanding extension interfaces that cut across these layers; support for the latter is also lacking. Moreover, most approaches focus on language or layer-specific extensibility mechanisms and thus do not support the needs of multilayered applications. Last but not least, to generate an extension interface of a complex software with many extensibility constraints, a developer has to be experienced with advanced development techniques.

The need and the challenges related to providing well-defined extension interfaces for object-oriented systems are documented in the literature [4], [5], [6], [7]. As a variation on this theme, several proposals for aspect-based extension interfaces have been published recently [8], [9], [10], [11], [12]. Yet, as we will elaborate in related work, these approaches do not address the limitations mentioned above.

This paper contributes the following. First, we further elaborate on the limitations identified above through a simplified example of a business application consisting of three layers (Section II). Second, we introduce XPoints, an approach and a language that enables an 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 (Section III). Third, we report on one particular instantiation of the approach in business applications consisting of three layers: business object, UI, and business process. We also report on an implementation of XPoints in this context (Section IV). Finally, we discuss the advantages and limitations of our approach (Section V) and compare our approach with related work (Section VI).

II. Problem Statement

In this section, we first introduce an exemplary business application that we use throughout the paper. Then, we analyze the limitation of current works with respect to extensibility and extension interfaces.

Example Business Application. We consider a business application spanning three logical layers: the business process layer, the business object layer, and the UI layer. A business process defines the flow of activities that are required to achieve a specific business objective such as creating a sales order, ordering goods, or hiring a new employee. Business objects [13] represent entities that are meaningful within a specific business process like sales order, invoice, customer, and employee. A business object encapsulates attributes, behaviour, constraints, and relationships to other business objects. UIs provide means to support the end users to accomplish the different activities within a business process via a graphical interface.

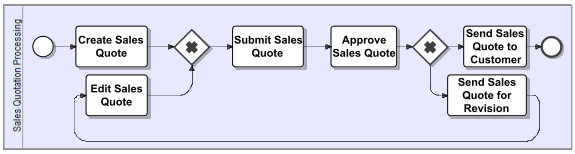

Figure 1. Sales quotation business process

We introduce a simple sales quotation management module as an example of multilayered business applications that spans the three layers mentioned above. Figure 1 shows the sales quotation business process (layer 1) defined in the Business Process Modeling Notation (BPMN) [14]. The process starts upon receiving a request for a quotation for a specific set of products from a customer. A sales representative analyzes the request and creates a sales quotation and fills in the necessary data. Then, she 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

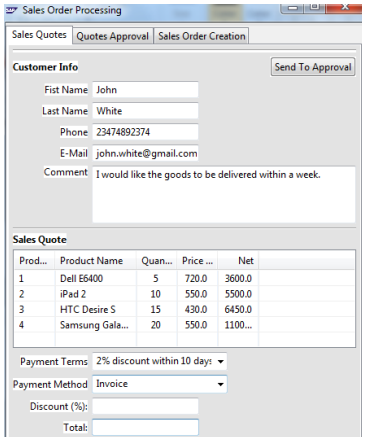

Figure 2. User interface for sales quotation creation

resubmit it for approval. At the end, the approved sales quotation is sent to the inquiring customer.

```
class SalesQuoteForm extends JPanel {
...
 private CustomerInfo customerInfo;
 private double discount;
 private SalesQuote salesQuote;
...
 public SalesQuoteForm() {...}
...
 private void initializeForm() {...}
 private void onSendToApprovalButtonClick() {...}
 private void savetoSalesQuoteBusinessObject() { ... }
...}
```

Listing 1. Sales quotation form source code

Figure 2 shows the UI (layer 2) associated with the sales quotation creation activity. An excerpt of the source code associated with this UI is shown in Listing 1. Using this UI, 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. The sales quotation module involves other UIs and business objects, as well as classes that support the execution of the business process which is not shown for brevity.

```
class SalesQuote
{
 protected CustomerInfo customerInfo;
 private List<ProductQuote> products;
 protected String comment;
 private double total;
 protected double discount;
 private double tax;
 ...
 public final SalesQuote readSalesQuote(...){...}
 public final SalesQuote createSalesQuote(...){...}
 private void saveSalesQuote(){...}
 protected double calculateTotal(){...}
 protected void calculateDiscount(double discount){...}
 protected void sendToApproval(){...}
...}
```

Listing 2. Sales quotation business object source code

Problem Analysis. Let us first consider the business object layer. Each class in Java has two interfaces: a usage interface and an extension interface. The *usage interface* allows a client of the class to call all methods and access all attributes that are not declared as private (including methods and attributes declared as final). The *extension interface*, via subclassing, allows the developer to override all methods that are not declared as private or final (like *calculateTotal()*), and introduce new methods and attributes. Moreover, the extension interface allows the extending subclass to access all attributes and to call all methods of the parent class that are not declared as private. The subclass has a read only access to the final attributes of the parent class. In the following, we discuss several limitations of the usage and extension interface in Java to express complex extension interfaces for software systems.

The first problem is the lack of means to express and constrain the extension types (M3). For example, it is not possible to express that an extension developer is allowed to add new methods to the class *SalesQuote()* but he is 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, such property cannot be guaranteed (M6). While this second example can be realized with workarounds (e.g., using the template method design pattern) we 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 would be able to express extension possibilities in a declarative way without thinking about how to enforce them (e.g., through applying a design pattern), whereas the extension developer will be able to directly understand the extension interface of the class without going through all its methods and related classes.

The second limitation is the limitation of the usage interface to express fine-grained overriding and access rights to the methods and attributes of the extended class (M5). 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 a workaround like using 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 provides a one-size-fits-all extension interface (M1). 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 without partner status) can be restricted to only perform 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. In addition to the extension possibilities given to the second group, this third group will have the possibility to define new attributes for the class *SalesQuote* and to add custom business logic after the method *sendToApproval()*.

There is no simple workaround for this third 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 extension interface. However, such a realization is very complex. For example, one could just consider the work required to provide three proxy classes for the three extension developer groups mentioned above for this example. The more the number of extensibility offerings and constraints, the more effort and time will be needed for implementing the extension interface.

Using workarounds as suggested in the discussion brings in a lot of disadvantages. First, the extension interface is realized *implicitly* rather than *explicitly*. In other words, the technical realization of the extension interface is coupled with the functional code of the core software (e.g. the design pattern suggested to realize the proxy classes to support multiple groups of extension developers will have to be adhered to by the functional code). The extensibility decisions and intents taken by the application 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., a comprehensive documentation) to find out the exact methods, classes, and interfaces that comprise the extension interface. Second, an extension developer will have a hard time identifying the extension possibilities as they are not expressed directly. Instead he will have to read documentation and tutorials and to understand the whole provided APIs to assess the feasibility of some extension scenario. This gets even more difficult as the functional API of the class and its extensibility API are mixed. Third, the design and implementation complexity for the core software provider increases and high developer expertise becomes necessary (e.g. with design patterns). The more complex the system and the extensibility constraints are, the more difficult the realization of extension interfaces will be.

In the discussion above, we focused on the businss object layer. However, modern software applications such as business applications (e.g. SAP Business Suite [1]) involve multiple layers and multiple artifacts on these layers (e.g. UI models, business process models, code artifacts, database tables, etc.). The extensibility problems discussed above on the code level arise also on the other layers. An extension can typically span several layers which makes it important to support extensibility on all these layers. For example, a software provider can make a certain database table extensible by allowing the addition of new columns. He can make a certain UI form extensible by allowing extension developers to embed their custom UI elements at a predefined location. He can also make a business process model extensible by allowing the extension developer to add custom activities. We 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 UI form extension or some business process extension without diving deeply into the implementation and the provided APIs on the code layer.

Furthermore, when supporting extensibility on different layers, it is necessary to capture the dependencies (M2) between the extension possibilities available on these layers. For example, if the extension interface of the *SalesQuote* on the UI 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, he has to consider the extension possibilities available on the database layer and to extend the table that stores the *SalesQuote* data. As this example illustrates, an extension can span multiple layers within an application. These inter-layer dependencies impose constraints on the way extension possibilities are expressed and also on the way an extension is developed.

[1] http://www.sap.com

III. XPoints

XPoints is a generic approach and a language for expressing extension interfaces of multilayered applications. Using an XPoints interface, a software provider can define and generate an extension interface supporting the mechanisms (M1-M6) outlined in Section I. In an XPoints interface, the software provider separately specifies the required extension possibilities (M1), interdependencies (M2), supported extension types (M3), and control constraints (M5) that are offered by the core software. Several XPoints interfaces can be defined for a software system. The XPoints compiler takes the defined XPoints interfaces and the source code of the core software, and generates the required system extension interface (i.e. extensibility framework and code artifacts (M4, M6)) on the code level using advanced techniques (e.g. design patterns, aspect oriented programming, plug-ins, etc.). Using the XPoints interface, a software extender can identify the available extension possibilities and use it as a guide to identify the right coding elements generated by the XPoints compiler to develop an extension.

Language Concepts. Within an XPoints *extension interface*, several layers can be defined corresponding to the *logical layers* of the base application. Each layer consists of one or more *extensible artifacts* (M1) that are made available to an extender. 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* (M3, M6). Extension artifacts can be seen as containers of extension points. Each extension point has a type and a set of parameters, which specify the base class artifacts that are needed to generate the appropriate extension interface. With this concept, we declare extension possibilities as first class entities and hence we can explicitly express extension possibilities.

Listing 3 shows an example of a very simple extension interface on the business object layer. The interface declares the *SalesQuote* business object as an extensible artifact with the extension point *EXP1* of type *afterMethodCall* that allows the extender to insert his custom logic after the execution of the *sendToApproval* method.

```
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    }
10  }
11 }
```

Listing 3. XPoints interface example

Extension points can be further grouped within the same or a different layer via *extension point groups* (M2). 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* (M5) on extensible artifacts and extension points restrict the access, visibility, and usage of the base application artifacts by the extenders. The purpose of this concept is to provide a fine grained access control of the extensions to the core application resources. The example in Listing 3 shows a control constraint for *EXP1* in the form of a permission set *per* that allows the extender a *READ* access to the *total* attribute and hides all methods of the class from him.

The control constraints can also be defined on a group to (M2) 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. UI and business object), a valid extension can require the implementation of several extension points from the same or multiple layers. Figure 3 summarizes the language concepts.

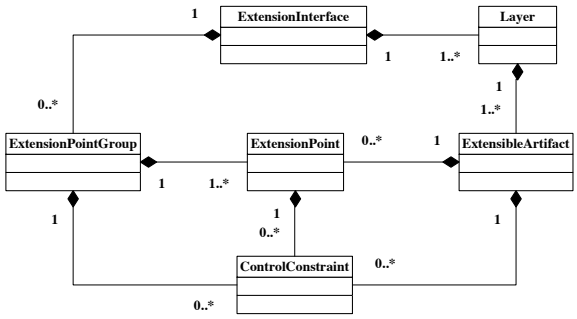

Figure 3. Language concepts of XPoints

Example. In the following, we demonstrate how XPoints can be applied in a simplified context of business applications (see Section II). The concrete instantiation of the concepts is described later in Section IV. We first consider two extension scenarios for two kinds of extension developer groups, then we show how XPoints can be used to specify the extension interface to implement the requirements of the two scenarios.

Scenario 1: External Developer, Let us consider external developers, who are allowed to perform some custom logic before the *SalesQuote* business object is saved, but are not allowed to modify any attribute. This group of extenders is also 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. Further, this group does not see any method of the *SalesQuote* business object.

Listing 4 shows the specification of the extension interface in XPoints for this extender group. This extension interface spans two layers (business object and UI). Line 1 declares the external developer extension interface. Line 3 declares the business object and Line 12 declares the UI as the containers of extensible artifacts. In this example, there are two artifacts declared as being extensible; *com.sap.SalesQuote*

and *com.sap.SalesQuoteForm* (Line 4 and Line 13). 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 5 shows the declaration of the extension point EPBO1 of type *beforeMethodCall* and Line 14 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 extender 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 7-10). This permission set declares that all attributes should be available only in *READ* mode and methods should be hidden to all extension points within the artifact. The *SalesQuoteForm* UI 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 extender.

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

Listing 4. Extension interface in XPoints for the external developer group

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 should extend both extension points.

Scenario 2: Internal Developer. In this scenario we consider a group of extenders, who are working on the provider side to realize industry-specific solutions on top of the standard application. These extenders are allowed to define extensions that span multiple layers. More specifically these extenders are allowed to extend the business process after the approval step for example to realize a second approval step (c.f. Section II). Thereby only some relevant business process activities should only be made visible while hiding the rest of the process details. Further, these extenders are also allowed to extend the *SalesQuote* business object with new attributes and extend the business object logic after it has been sent for approval. The extenders should also be allowed to read and write values to the attributes *products* and *customerInfo* as well as to call the method *calculateTotal*. Listing 5 shows the XPoints implementation.

```
1  extensioninterface internaldeveloper{
2
3  layer BusinessObject {
4    extensibleartifact "com.sap.SalesQuote" permission=defview{
5   allowBOAttributes EPBO1 ("String",10);
6   afterMethodCall EPBO2 ("void sendToApproval()")permission=intdev;
7
8   permissionset intdev{
9    attributepermission ("products",READWRITE);
10   attributepermission ("customerInfo",READWRITE);
11   methodpermission ("calculateTotal",CALLABLE);
12  }}
13
14  permissionset defview {
15  attributepermission ("*",READ);
16  methodpermission("*",HIDDEN);
17  }}
18
19 layer UserInterface {
20   extensibleartifact "com.sap.SalesQuoteForm" permission=defview{
21    allowUIComponent EPUI1 ("JPanel","approvalPanel");
22  }
23
24  permissionset defview{
25    attributepermission("*",HIDDEN);
26    methodpermission("*",HIDDEN);
27  }}
28
29 layer BusinessProcess {
30   extensibleartifact "sales_quotation.bpmn" permission=defview {
31   afterActivity EPBP1 permission = view
32   ("Approve Sales Quote","com.sap.SQProcessing"
33                          ,"void approveQuote()");
34
35   permissionset view{
36    activitypermission("Create Sales Quote",VISIBLE);
37    activitypermission("Approve Sales Quote",VISIBLE);
38    activitypermission("Send Sales Quote",VISIBLE);
39   }}
40
41  permissionset defview{
42   lanepermission("Sales Quotation Processing",HIDDEN);
43  }}
44
45 Group extensionScenario {(EPUI1,EPBP1,EPBO2),ExtendAll};}
```

Listing 5. Extension interface in XPoints for the internal developer group

In this extension interface, there are three layers defined (business object, UI, and business process). In business object layer (Lines 3-17), the *SalesQuote* business object is declared as extensible. The permission set *defview* expresses that the extender cannot call any method, and has a read only access to all attributes (Lines 14-17). There are two extension points defined (Lines 5-6) 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. EPBO2 has a reference to permission set *intdev* (that refines the permission set of the parent), which allows a read / write access to the attributes *products* and *customerInfo*, and allows the method *calculateTotal()* to be called (Lines 8-12).

The next part of the interface (Lines 19-27) declares the

SalesQuoteForm as extensible with the *allowUIComponent* extension possibility EPUI1 that allows the extender 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 extender. The following part (Lines 29-43) 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 41-43). The permission set *view* referenced by EPBP1 makes the main activities of the business process visible to the extender.

Similarly to the previous scenario, the last part of the interface (Line 45) declares a group called *extensionScenario* that contains three extension points EPUI1, EPBP1, and EPBO2. This requires then the extender to extend all extension points.

IV. Extension Interface Generation

In the following, we will first describe the concrete instantiation of the general language concepts described in Section III for business applications consisting of the three logical layers described in Section II, assuming that the underlying classes are 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. Then, we show how XPoints can be used to generate the extension interface code.

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 respectively.

Extension Points: On the business object layer, the following types are supported. *AfterConstructor* enables to define 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. *AfterBOAttributeChange* enables to define extension-specific logic to be executed after the value of a certain business object attribute changes. *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. *AllowBOAttributes* enables the extension of a business object with a maximum number of attributes with a certain type.

On the UI layer, the following types are supported. *beforeForm* and *afterForm* enable to extend the form flow of a certain application; it can be used to insert a custom UI before or after a certain displayed UI. *beforeUIEventHandler* and *afterUIEventHandler* enables to define custom logic to be inserted before or after a certain event handler is called. *allowUIAttributes* enables to extend the data model of a UI with a maximum number of attributes of a certain type.

On the business process layer, more types are supported. *BeforeActivity*, *AfterActivity*, and *ParallelActivity* declare the possibility of extending an activity before, after, or parallel to its execution. *BeforeEvent* and *AfterEvent* allow the extender to insert his extension before or after an event. *AfterDecision* defines the possibility of inserting an extension after a certain decision result from a gateway. *ExtensibleMessage* allows the extension of the message content or type used in the process (data extension). *ExtensibleDecision* allows extending the result set of a gateway.

Control Constraints: In the concrete instantiation, control constraints are realized as *permissionset*s which restrict the access, visibility, and usage rights of the base application resources (i.e. supports the principle of least privilege [15]) to the extender. 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 UI layers, permission sets support method and attribute permissions of the extensible artifact. Attributes can be declared as either *READ, WRITE, READWRITE* or *HIDDEN*. Methods can be declared as *CALLABLE or HIDDEN*. Extensible artifacts that do not declare a permission set get the default extension and usage interface 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 extender.

Group Control Constraints: The current instantiation supports one control constraint, *ExtendAll*, requiring that a valid extension should provide an extension for all extension points within the group. For example, it can be required that an extender extends the data model of the business object when adding a new input text field for a UI.

Interface Generation. 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 (M3, M4), a *proxy class* that 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 extender implementing the interface and will be initialized once an extension is loaded), and an *aspect code* (implemented in AspectJ [16]), which injects 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 necessary to load and initialize an implemented extension in a plug-in

like fashion).

The general concepts of XPoints and the business application extension are implemented as a domain specific language (DSL) using XText [17] in Eclipse. To generate the interfaces, proxy classes, and AspectJ programs as well as the validations of the XPoints interface and references to the core application source code, we have used XTend.

For illustration, we schematically present the generated code framework that realizes the extension interface of the software for the external developer scenario (see Listing 4). Listing 6 shows the generated code framework. Lines 3-9 show the generated interface. The interface includes two parts. The first part is needed by the code framework to initialize the extension (Line 6). Moreover, a reference to the corresponding proxy classes is 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.

```
1  //*************Generated Interface**************
2
3  public interface ExtensionScenarioInterface{
4
5   //these are the methods the extender has to implement
6   public void init(EPBO1Proxy p1, EPUI1Proxy p2);
7   public void yourEPBO1Logic();
8   public JLabel yourEPUI1JLabel();
9  ...}
10
11 //**********Generated proxy classes*************
12
13 public class EPBO1Proxy{
14  private SalesQuote salesquote;
15 ...
16 //getter methods for the READ attributes
17  public CustomerInfo getCustomerInfo(){
18   return salesquote.getCustomerInfo(this);
19  }
20  public List<ProductQuote> getProductQuote(){...}
21  public String getComment(){...}
22  public double getDiscount(){...}
23 ...}
24
25 public class EPUI1Proxy{
26 //empty since no access has been granted
27 }
28
29 //************Generated Aspects*****************
30
31 public privileged aspect EPBO1Aspect {
32
33  /*Datastructure to hold
34  extensions of type ExtensionScenarioInterface*/
35  private ArrayList<ExtensionScenarioInterface>
36                          SalesQuote.EPBO1Extensions;
37
38 //New method in SalesQuote class to add the extensions
39  private void SalesQuote.loadExtensionScenarioExtensions(){
40 ...
41 //loads the extensions with class loader
42 ...
43    extensions.init(this.getEPBO1Proxy(),this.getEPUI1Proxy());
44    EPBO1Extensions.add(extension);
45 ...}
46
47 //New method in SalesQuote class to perform
48 //EPBO1 extension sanity checks
49  private void SalesQuote.sanityChecksEPBO1(){...}
50
51 //New method in SalesQuote class to get the EPBO1 proxy
52  private EPBO1Proxy SalesQuote.getEPBO1Proxy(){
53   return new EPBO1Proxy(this);}
54
55 //New methods to support the proxy access to the base class
56  public CustomerInfo
57            SalesQuote.getCustomerInfo(EPBO1Proxy proxy){
58         //validate the proxy and return
59        if(isLegalProxy(proxy)) return this.customerInfo;
60        else return null;
61       }
62
63  public List<ProductQuote>
64            SalesQuote.getProducts(EPBO1Proxy proxy){...}
65 //Similarly for the rest of the attributes ...
66
67 //load the extensions and
68 //perform sanity checks in constructor constructor
69  pointcut onload(): execution(* SalesQuote.new(..));
70  after(SalesQuote s): onload() && this(s){
71  s.loadExtensionScenarioExtensions();
72  s.sanityChecksEPBO1();}
73
74 //Pointcut and advice for running the EPBO1 extension
75  pointcut extension(): execution(* SalesQuote.saveSalesQuote(..));
76  before(SalesQuote s): extension() && this(s) {
77
78  if(s.EPBO1Extensions != null)
79  {
80   for(int i=0; i<s.EPBO1Extensions.size(); i++)
81   {
82    s.EPBO1Extensions.get(i).yourEPBO1Logic();
83   }
84  }}...}
85
86 public privileged aspect EPUI1Aspect {
87 ...
88  //Aspect body similar to the EPBO1Aspect
89 ...
90  //Pointcut and advice for running the EPUI1 extension
91  pointcut extension(): execution(* SalesQuoteForm.new(..));
92  after(SalesQuoteForm s): extension() && this(s) {
93  if(s.EPUI1Extensions != null)
94  {
95   for(int i=0; i<s.EPUI1Extensions.size(); i++)
96   {
97    JLabel j = s.EPUI1Extensions.get(i).yourEPUI1JLabel();
98    s.salesQuotePanel.add(j);
99   }
100 }}}...
```

Listing 6. Generated code framework for the external developer

The EPBO1 proxy class (Lines 13-23) 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 5, Lines 7-10). The proxy class generated for EPUI1 is empty since all methods and attributes were declared as hidden by the permission set *default2* (Listing 5, Lines 16-19). The last part of the code framework generated is the aspect code for EPBO1 (Lines 31-84) and EPUI1 (86-100).

In the EPBO1 aspect, the first part (Lines 35-53) of the aspect code are 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 56-65) enriches the base class in

a similar fashion with methods to support the proxy class *EPBO1Proxy* calls. The last part of the aspect (Lines 69-84) 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 91-100) will add the *JLabel* component from the extension to the *salesQuotePanel*.

V. Discussion

To highlight the advantages of XPoints, we would like to emphasize that in lack of XPoints, the code in Listing 6 would have to be manually written by the developer of the base application in addition to the implementation of the core application functionality. By comparing Listing 6 with Listing 5, it becomes clear that XPoints significantly reduces design complexity. The XPoints interface provides a declarative mechanism for the implementation of extensibility, higher level of abstractions, and separation of concerns. While the developer could employ other programming patterns and techniques rather than those we 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 different ways of extending a software, 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 created 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.

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 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 5 can be used to identify the right interfaces and proxy classes required to realize a particular extension.

Limitations of the Approach: First, the extension interface generation strategy depends on the implementation. In the presented example implementation for business applications, we used aspects, proxy classes, and Java interfaces for the generation of the extension interface. However, it is also possible to use other techniques for the generation and enforcement of extension interfaces. Second, if the core application code changes, the old generated extension interfaces can become invalid. 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 (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. Last, the extension point types and semantics depend on the implementation of XPoints. The presented implementation of XPoints for business applications is only an example instantiation of the concepts presented (i.e. we do not claim that these are all the possible extension point constructs for business applications). We are currently working on a more generic instantiation for XPoints to support defining extension interfaces for Java.

VI. Related Work

Several works have proposed interfaces enabling modules that are advisable while preserving modularity and controlling internal implementation details. In this section we show how XPoints relate and compare it with different language-level state-of-the-art approaches.

Open modules [8] use pointcuts to expose advisable join points of a particular module. The pointcuts are tightly coupled with the definition of the module, and therefore it is not possible to express crosscutting concerns across several modules. In contrast to our approach, XPoints expresses the extensibility possibilities separately from the base classes. The base class developer has to only focus on defining what extensibility possibilities exist, rather than writing pointcut expressions. Moreover, it is possible to associate different extension possibilities with extension point groups unlike open modules.

Crosscutting interfaces (XPIs) [11] 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. Although the approach enhances on the decoupling of the extension possibilities from the base code and slightly shares our concept of separating the extension possibilities from the base code, the rest of the drawbacks previously described for open modules are not addressed. Furthermore, the design rules contracts used in XPIs are informally defined and no means are provided for enforcing them. Unlike XPoints, the constraints defined on the extensible artifacts, extension points, and extension point groups are enforced by generating code. Furthermore, there is no way to restrict the access to the base class resources to the advice code.

Join point types [10] and *join point interfaces* (JPIs) [12] introduce an additional layer to serve as an interface between join points and advices. These approaches enrich pointcuts with a "type" (syntactically in a method signature like fashion) that specify 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. XPoints share the idea of restricting access of an extension to the base class resources. However, there are also several limitations that are not addressed by these approaches. The first limitation is that there is no fine grained access control to the elements specified in the type. 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. The second limitation is that it is not possible to support multiple extenders with different access rights to the base code resources.

Design patterns [18] are 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. In XPoints a developer does not have to be an expert in design patterns to realize the required extension interface. The XPoints compiler will automatically complement the core software using the adequate design patterns (if the compiler supports that technique as a generation strategy) and generate the required code framework.

Plug-in systems abstract the data and functionalities of an application through an application programming interface that act as hooks or extension points. 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-in. An example of a plug-in system is the OSGi [19] based Eclipse [20]. Each plug-in contributes to a set of extension points and can provide a set of extension points (a manifest file describes the extension points it contributes to, dependencies to other plug-ins, and extension points it provides). 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. XPoints and Eclipse share the idea of explicitly defining extension points as well as their dependencies. However, in XPoints a developer does not have to manually develop the interfaces as well as handle them in the implementation of the core software to support extensibility. XPoints generates the extension handling framework, classes, and interfaces automatically from the XPoints interface specification.

In addition to the limitations pointed out in all of these approaches, XPoints further 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 above 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. 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 (e.g. like mixins [21], virtual classes [22], difference based modules [23] dynamic routines [24], traits [25] etc.) for the realization of extension interfaces on the code level.

VII. Summary and Outlook

Defining and realizing extension interfaces for multilayered applications is a very challenging task. In this paper we outlined the limitations of state-of-the-art approaches in supporting extensibility for multilayered applications and introduced XPoints, an approach and a language for addressing these limitations. An XPoints extension interface defines the extension possibilities in multilayered applications, controls the available resources of the core application to the extension, and relates extension possibilities from different layers. The interface is defined separately from the core software resulting in less complex code and better software maintainability and also enabling multiple extension interfaces for different kinds of extenders to co-exist. The XPoints compiler automatically generates the necessary code framework to realize the extension interface on the code level without having the core developer being an expert in advanced programming techniques.

Currently, we are investigating a generic realization of XPoints for Java as well as possibilities for generating the extension interface using other approaches. We plan to apply XPoints to other application domains, and to investigate advanced topics like extension validation, monitoring, and conflict detection.

REFERENCES

[1] M. Aly, A. Charfi, and M. Mezini, "On the extensibility requirements of business applications," in *Proceedings of the 2012 workshop on Next Generation Modularity Approaches for Requirements and Architecture*, ser. NEMARA'12. New York, NY, USA: ACM, 2012, pp. 1–6.

[2] M. Aly, A. Charfi, D. Wu, and M. Mezini, "Understanding multilayered applications for building extensions," in *Proceedings of the 1st workshop on Comprehension of complex systems*, ser. CoCoS'13. New York, NY, USA: ACM, 2013, pp. 1–6.

[3] J. Micallef, "Encapsulation, reusability, and extensibility in object-oriented programming languages," *Journal of Object-Oriented Programming*, vol. 1, no. 1, pp. 12–36, 1988.

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

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

[6] M. Mezini, "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*, ser. OOPSLA'97. New York, NY, USA: ACM, 1997, pp. 1–21.

[7] G. Kiczales and M. Mezini, "Aspect-oriented programming and modular reasoning," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 49–58.

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

[9] K. Hoffman and P. Eugster, "Bridging java and aspectj through explicit join points," in *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, ser. PPPJ'07. New York, NY, USA: ACM, 2007, pp. 63–72.

[10] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner, "Types and modularity for implicit invocation with implicit announcement," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 1, pp. 1:1–1:43, Jul. 2010.

[11] K. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari, "Modular aspect-oriented design with XPIs," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 2, pp. 5:1–5:42, Sep. 2010.

[12] M. Inostroza, É. Tanter, and E. Bodden, "Modular reasoning with join point interfaces," Center for Advanced Security Research Darmstadt, Tech. Rep. TUD-CS-2011-0272, 2011.

[13] J. Sutherland, "Business objects in corporate information systems," *ACM Computing Surveys*, vol. 27, pp. 274–276, June 1995.

[14] Object Management Group (OMG), "Business Process Model and Notation (BPMN) Version 2.0," Object Management Group (OMG), Tech. Rep. formal/2011-01-03, January 2011. [Online]. Available: http://www.omg.org/spec/BPMN/2.0

[15] T. Mayfield, J. E. Roskos, S. R. Welke, J. M. Boone, and C. W. McDonald, "Integrity in automated information systems," National Security Agency, Tech. Rep. 79-91, 1991, iDA Paper P-2316.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP'01*, ser. LNCS, J. Knudsen, Ed. Springer Berlin Heidelberg, 2001, vol. 2072, pp. 327–354.

[17] M. Eysholdt and H. Behrens, "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*, ser. SPLASH '10. New York, NY, USA: ACM, 2010, pp. 307–309.

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

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

[20] S. Shavor, J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy, *The Java Developer's Guide to Eclipse*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[21] G. Bracha and W. Cook, "Mixin-based inheritance," *ACM SIGPLAN Notices*, vol. 25, no. 10, pp. 303–311, Oct 1990.

[22] O. L. Madsen and B. Moller-Pedersen, "Virtual classes: a powerful mechanism in object-oriented programming," in *Conference proceedings on Object-oriented programming systems, languages and applications*, ser. OOPSLA'89. New York, NY, USA: ACM, 1989, pp. 397–406.

[23] Y. Ichisugi and A. Tanaka, "Difference-based modules: A class-independent module mechanism," in *ECOOP'02*, ser. LNCS, B. Magnusson, Ed. Springer Berlin Heidelberg, 2006, vol. 2374, pp. 62–88.

[24] C. Heinlein, "Vertical, horizontal, and behavioural extensibility of software systems," Universität Ulm, Fakultät für Informatik, Tech. Rep., 2003. [Online]. Available: http://vts.uni-ulm.de/doc.asp?id=5344

[25] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black, "Traits: Composable units of behaviour," in *ECOOP'03*, ser. LNCS, L. Cardelli, Ed. Springer Berlin Heidelberg, 2003, vol. 2743, pp. 248–274.