Explicit Exception Handling Variability in Component-based Product Line Architectures

Ivo Augusto Bertoncello*
University of Campinas (UNICAMP), Brazil
ivo.bertoncello@students.ic.unicamp.br

Marcelo Oliveira Dias†
University of Campinas (UNICAMP), Brazil
marcelo.dias@students.ic.unicamp.br

Patrick H. S. Brito‡
University of Campinas (UNICAMP), Brazil
pbrito@ic.unicamp.br

Cecília M. F. Rubira§
University of Campinas (UNICAMP), Brazil
cmrubira@ic.unicamp.br

ABSTRACT

Separation of concerns is one of the overarching goals of exception handling in order to keep separate normal and exceptional behaviour of a software system. In the context of a software product line (SPL), this separation of concerns is also important for designing software variabilities related to different exception handling strategies, such as the choice of different handlers depending on the set of selected features. This paper presents a method for refactoring object-oriented product line architecture in order to separate explicitly their normal and exceptional behaviour into different software components. The new component-based software architecture includes variation points related to different choices of exception handlers that can be selected during product instantiations, thus facilitating the evolution of the exceptional behaviour. The feasibility of the proposed approach is assessed through a SPL of mobile applications.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design

*Ivo Bertoncello is supported by FAPESP/Brazil, grant 08/02500-2.

†Marcelo Dias is supported by FAPESP/Brazil, grant 08/02501-9.

‡Patrick Brito is supported by FAPESP/Brazil, grant 06/02116-2.

§Cecília Rubira is partially supported by CNPq/Brazil, grants 301446/2006-7 and 484138/2006-5.

WEH '08, November 14, Atlanta, Georgia, USA

Keywords

Exception Handling, Software Architecture, Component-Based Software Development, Exceptional Behaviour

1. INTRODUCTION

Currently, many efforts are being made for achieving higher levels of reuse when developing software systems. One of the main approaches discussed in the literature for promoting the reuse of software artefacts is called software product line (SPL). A SPL is an approach that systematises the reuse of software artefacts through the exploration of commonalities and variabilities among similar products [1]. One of the main artefacts in the context of a SPL is its product line architecture (PLA). While abstracting away from system details, a PLA provides a global system perspective that is key for identifying commonalities and variabilities in terms of architectural elements and their configurations, as well as planning strategies for software reuse. In a PLA, the commonalities of architectural elements and their configurations are reused in different products, while the variabilities are resolved through design decisions related to the different choices captured by the software architecture through variation points.

According to Bass et al. [2], the software architecture defines the structure or structures of the system, which comprise software components, the relationships between them, and the externally visible properties of the application, which are usually related to its quality attributes. Since the software architecture realises the design decisions associated to quality attributes, such as dependability, when evolving the strategy of error handling, it is desirable to start from the software architecture and then apply the changes to the source code. Moreover, in the context of SPL, it is desirable to keep traceability between the feature model its software architecture, in such a way that SPL evolutions can be easily applied at the software architecture. For providing a high-level abstraction, we adopt a component-based software architecture, where components are deployable units of composition with contractually specified interfaces and explicit context dependencies [19]. It is desirable that component-based software architecture considers explicitly both the normal and exceptional behaviour of the system in order to easily evolve.

When one considers the system's behaviour in erroneous situations, the complexity of its behaviour can be increased, as well as the tangling between the normal behaviour and the behaviour responsible for handling errors. Exception handling [8] was conceived as a way to manage the complexity of systems that should cope with rare situations, such as error recovery. An exception handling mechanism promotes an explicit textual separation between the normal code, responsible for the application functionality and the code responsible for its behaviour in exceptional situations. Exception handling complements other techniques for error recovery, such as atomic transaction [14], and aims to support the construction of programs that are more reliable, reusable, and easy to evolve [17]. The implementation of exception handling mechanisms by several modern object-oriented languages, such as Java, Ada, C#, and C++, and component models, such as CCM, EJB, Ice, and .NET, attest its importance to the current practice of software development. These programming languages provide constructs to indicate the occurrence of an error (*raise* or *throw* an exception), and means to incorporate recovery actions (*handle* the exception), including error handling.

Separation of concerns is one of the overarching goals of exception handling in order to keep separate normal and exceptional behaviour of a software system. This separation promotes both adaptability and reuse of normal and error handling code. Moreover, it is also desirable to consider the exceptional behaviour from a high-level perspective, since the error recovery strategy is a property associated to the system's global structure, not only to a individual component. In the context of PLAs, the separation of concerns between normal and exceptional behaviour is particularly suitable for facilitating both the evolution of the PLA, and the instantiation of different products according to the system's exception handling variabilities. For example, the selection of an optional feature during the instantiation of a product can interfere in the way exceptions are handled.

The contribution of this paper is to provide a method for refactoring object-oriented PLAs in order to separate their normal and exceptional behaviour into different software components. This separation of concerns allows the specification of different exception handlers that can be selected during the architectural configuration, thus facilitating the evolution of the exceptional behaviour and its instantiation into different products.

The rest of this paper is organised as follows. Section 2 provides some background information. Section 3 presents a general method for refactoring object-oriented PLAs. Section 4 describes the case study used to evaluate the proposed approach. Section 5 describes how the target SPL has been evolved in order to better evaluate our solution. Section 6 summarises the overall evaluation by comparing the original and the refactored PLA. Section 7 describes some related work. Finally, Section 8 provides some concluding remarks, and future directions of research.

2. BACKGROUND

2.1 Software Product Line Architectures

Feature modelling is one of the most accepted ways to represent commonalities and variabilities at the requirements phase. At the architecture design phase, product line architecture is an important core asset that should represent the common and variable parts in a product line.

A Software Product Line (SPL) is an approach that systematises the reuse of software artefacts through the exploration of commonalities and variabilities among similar products [1]. Feature model is one of the most accepted ways to represent commonalities and variabilities at the requirements phase. At the architecture design phase, Product Line Architecture (PLA) is an important core asset that should represent the common and variable parts in a product line. One of the main artefacts in the contexts of a SPL is the PLA, which explicitly represents the commonalities and variabilities of architectural elements and their configurations. The commonalities are reused in different products, while the variabilities are resolved through design decisions related to the choices at the PLA.

The reuse obtained with SPL facilitates the the development of similar software systems of a given domain. This approach allows large-scale reuse through a common set of core assets in a prescribed way [7]. The development process of a product line consists of two complementary sub-processes: product line engineering and application engineering. Product line engineering analyses software products with regard to their commonalities and variabilities in order to build a reuse infrastructure that can be used to derive new similar products. Application engineering uses this infrastructure to instantiate particular software products [1].

2.2 Component-based PLAs

The software architecture of a program or computing system is the structure or structures of the system, which comprises software elements, the externally visible properties of those elements (architectural components and connectors), and the relationships among them (architectural configuration) [2]. Since the software architecture represents the structure of the software system, it is the set of significant decisions about the organization of a software. Those decisions concerns the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour as specified in the collaborations among those elements [15]. Moreover, since software architecture deals with the design and implementation of the high-level structure of the software, its decisions have a decisive impact into the quality attributes of the whole system, such as availability, reliability, and testability.

Figure 1 presents an example of an architectural configuration involving two components and a connector. *Architectural components* (e.g., A and B) are primary computational elements or data stores of a system and are defined through the list of operations that they provide (provided interfaces), as well as the operations necessary for executing their functionalities (required interfaces). *Architectural connectors* (e.g., conn) are mediators of the communication and coordination activities among components. That is, they define the rules governing component interaction and specify any auxiliary implementation mechanism required. The way that components and connectors are connected together is defined by the *architectural configuration*.

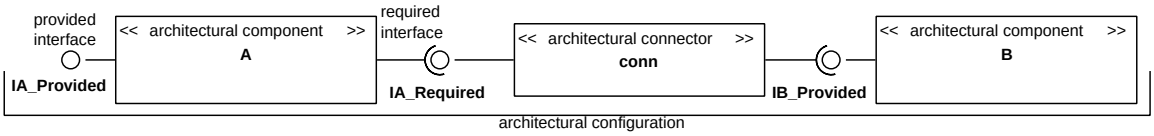

Figure 1: Example of an architectural configuration in UML

In the context of this paper, the normal and exceptional behaviour of the software components are explicitly separated. This explicit separation at the implementation level aims to reduce the coupling between the normal and exceptional behaviour, thus improving the system evolvability and the reuse of the normal part. In the proposed approach, interfaces may contain either operations for handling exceptions (*exceptional interfaces*), or operations for implementing functionalities (*normal interfaces*); normal and exceptional interfaces can be either provided or required. In the same way, components can provide either normal or exceptional interfaces, not both at the same time. In the first case, they are considered *normal components*, while in the second case they are considered *exceptional components*.

2.3 The COSMOS* Component Implementation Model

The COSMOS* model [13] is a generic and platform-independent implementation model, which uses object-oriented structures, such as interfaces, classes and packages, to implementing component-based software architectures. The main objectives of the COSMOS* model are: (i) to provide traceability between the software architecture and the source code of the application; and thus (ii) to facilitate the evolution of its implementation.

The main advantages of COSMOS*, when compared with other component models such as Corba Component Model (CCM), Enterprise Java Beans, and .NET, is threefold. First, COSMOS* explicitly represents architectural units, such as components, connectors and configuration, thus providing traceability between the software architecture and the respective source code. Second, as CCM, COSMOS* implements software components with explicit required interfaces, which facilitates the implementation of architectural variation points related to both normal and exceptional behaviour. Third, COSMOS* is considered a platform-independent model, since it is based on a set of design patterns. The structure defined by COSMOS* can be used as the basis for model transformation from component-based software architectures to detailed design of software components. For example, the Bellatrix case tool [20] supports model transformation from graphically specified software architectures, to Java source code in COSMOS*.

COSMOS* defines five sub-models, which address different aspects of component-based systems: (i) the *specification model* specifies the components using UML; (ii) the *implementation model* explicitly separates the definition of the provided and required interfaces of the components from the implementation of its provided services; (iii) the *connector model* specifies the link between components using connectors, thus enabling two or more components to be connected in a configuration; (iv) *composite components model* specifies high-granularity components, which are composed by other COSMOS* components; and (v) *system model* defines a software component which can be executed straight forward, thus encapsulating the necessary dependencies. Each of these models is implemented as a set of design pattern which can be automatically translated to source code. Due to space constraints, COSMOS* is exemplified only in Section 4.5, in the context of a case study of a target SPL.

3. A REFACTORING METHOD FOR COMPONENT-BASED PLA

In our approach, the PLA is considered a first-level unit, which guides the instantiation of different products of the SPL. Figure 2 presents an overview of the proposed approach for refactoring object-oriented PLAs. Activity 1 specifies the component-based PLA, which aims to provide a high-level view of the system structure maintaining traceability with feature model. From its feature model and classes of the existing SPL, two artefacts are produced: a component-based PLA, and the existing exceptions and classes related to each architectural element. Activity 2 explicitly separates the normal and exceptional behaviour of the architectural elements. This activity consists on the definition of explicit exceptional components, which are capable of handling exceptions. The objective of such separation of concerns is to group in fewer components the exception handling that was scattered in the code, thus facilitating its evolution and implementation of variability. Activity 3 is the implementation of the system's source code. For this, a specific component implementation model has to be selected. Each one of such activities is further detailed in the paper, in the context of a case study presented in Section 4.

4. CASE STUDY OF A MOBILE SPL

4.1 Target SPL: MobileMedia

In the following, in order to exemplify and evaluate our solution, we present as a case study a real software application, called MobileMedia [10], which is a SPL for mobile applications that manipulates photo, music, and video on mobile devices, such as mobile phones. The system uses various technologies based on the Java ME platform, such as SMS, WMA and MMAPI, and it is representative of how exception handling is used to deal with errors in real software development for two reasons. First, MobileMedia encompasses a large number of exception handlers related to different features of the SPL. Second, it presents heterogeneous crosscutting relationships involving the normal code, the handler code, the clean-up actions, and other crosscutting concerns.

Figure 3 presents a partial view of the feature model of MobileMedia, following the notation proposed by Ferber et al. [9]. The core features of MobileMedia are: Create/Delete

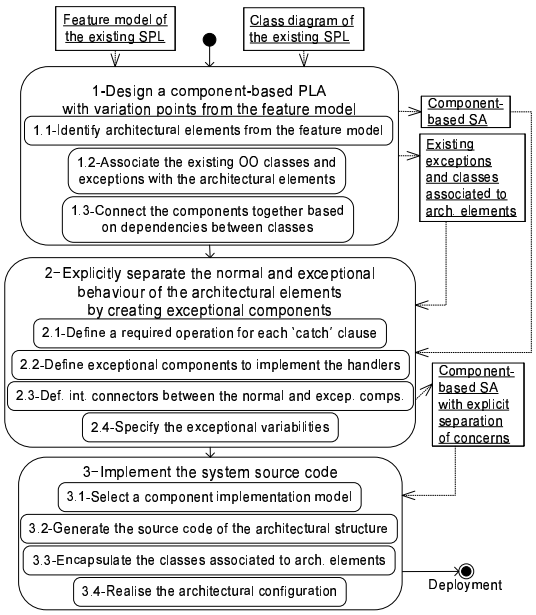

Figure 2: Method for Refactoring Exceptional Behaviour

Media, Persistence (SDcard, RecordStore), Media (photo, music or video), Label Media, and View/Play Media, as well as the types of storage device. The multiple features are the types of media supported: Photo, Music, and/or Video. Finally, the optional features are: send photo via SMS (SMS for short), Copy Media, and set favourite media (Favourites). The core features of MobileMedia are applicable to all the mobile phone devices that are J2ME enabled. The optional and alternative features are configurable on selected mobile phones depending on the API support they provided. MobileMedia was developed for a family of four brands of devices, namely Nokia, Motorola, Siemens, and RIM.

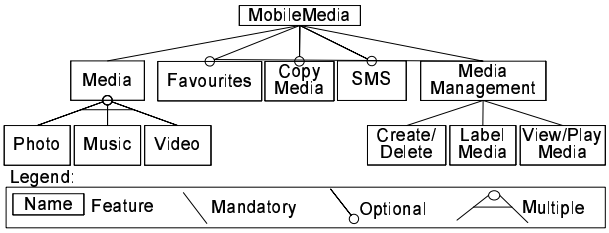

Figure 3: Partial feature model of the MobileMedia SPL

The original implementation of the MobileMedia PLA follows the object-oriented paradigm. The exception handling code for the Java implementation followed the design approach described in detail elsewhere [18]. Regarding the PLA of the MobileMedia, it is mainly determined by the use of the Model-View-Controller (MVC) architectural pattern [4]. Each implementation class represents an architectural element, but a high-level component-based representation of its software architecture is not available. Figure 4 presents a partial view of the original software architecture with a total of 18 architectural components. The four grey boxes encompass components that realise each of the three roles of the MVC pattern, namely model, view, and controller, and an utility layer. Figure 4 also relates the architectural elements with the features in Figure 3. This is done by the circles on the left top of the architectural elements. For instance, the SMS on the top of the SMS Controller (Figure 4) indicates that this element contributes to the implementation of the feature SMS in the feature model (Figure 3). According to Skyperski's definition of software component [19], although the architecture presented in Figure 4 is modularised, it is considered monolithic, since the contextual dependencies between modules are not explicitly represented and each module can not be deployable independently.

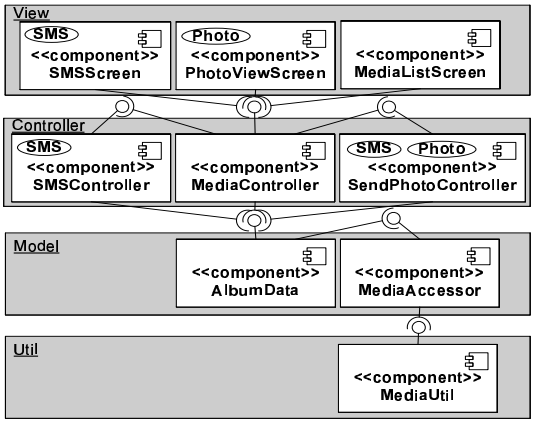

Figure 4: Layers of the Product Line Architecture of the MobileMedia SPL

4.2 Case Study Description

The main goal of the case study was to evaluate the feasibility of our approach, as well as the advantages of having a component-based PLA with explicit separation of concerns between the normal and exceptional behaviour. Such benefits are evaluated on evolvability of the component-based PLA. The execution of the case study is composed of three steps. First, we have refactored the original PLA in order to explicitly separate its normal and exceptional behaviour through a component-based PLA with hierarchical decomposition of software components. Second, we have evolved the feature model of the both PLAs by creating a exception handling variability, that allows the choice of a new optional feature related to raising a sound alarm in case of error. Finally, we evaluate the impact of applying this modification in the original PLA and the refactored PLA. Preliminary results of the benefits of the proposed approach are obtained by comparing the two versions for analysing qualitative and quantitative characteristics related to evolvability, specification of exceptional variability, and separation of concerns.

4.3 Designing a Component-Based PLA

For executing the Activity 1 of Figure 2, we have rafactored the software architecture with two goals: (i) to componentise the PLA in order to specify architectural variation points related to exception handling, and (ii) to reduce the number of architectural elements for improving the system understandability. For designing the component-based software architecture we have followed the UML Components process [6] with some adaptations. First, the layered architec-

tural style, suggested by the UML Components process, was changed by the MVC architectural style, which is adopted by the original PLA. Second, instead of using the use case model as input to identify components, we have used the feature model of the SPL, as defined by the process presented in Section 3.

Figure 5 presents the refactored PLA, which is composed by three layers: an interface layer, an operation layer, and a data layer. The PresentationDialog component represents the union of the Presentation and Dialog layers proposed by the UML Components process, and it is responsible for manipulating menus and user actions. The operation layer is composed of four components: VideoOperations, which implements functionalities for playing and managing video files, PhotoOperations, which implements functionalities for viewing and managing image files, AudioOperations, which implements functionalities for playing and managing audio files and MessagingOperations, which implements functionalities for sending and receiving messages. Finally, the data layer contains the services related to the management and access of storage devices (e.g. SD cards and the internal file system). This layer has a single component, called MediaManager, which is responsible for managing the media and albums data. Since the features of Audio, Photo, Video, and Messaging are optional features, the existence of different required interfaces into the PresentationDialog component allows the specification of architectural variation points. That is, during the architectural configuration, the dependencies of the PresentationDialog component can be either totally or just partially satisfied.

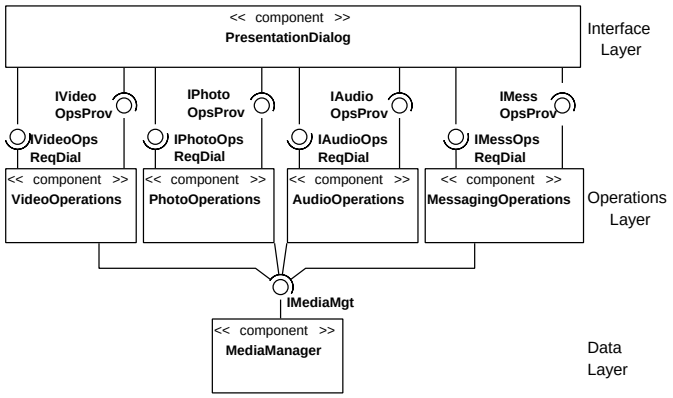

Figure 5: Component-based PLA for MobileMedia

Since architectural elements should be self-contained entities, we have defined data types, which are classes with public attributes and no operations, containing only the information needed by other components. Instances of data types are used to exchange information between architectural elements, thus providing information hiding of the implementation classes. Examples of data types are: AlbumDataDT and MediaDataDT.

4.4 Creation of Exceptional Components

According to Activity 2 of Figure 2, in order to facilitate the evolution of the exceptional behaviour, as well as the specification of variable exception handlers, our approach states that the normal and exceptional behaviour should be explicitly separated. Moreover, the architectural elements of the refactored PLA are hierarchically decomposed in terms of its normal and exceptional parts, by grouping the handlers defined for the PLA in exceptional components. The exception detectors, which depends on the system state, stays in the normal component. The association between the normal and exceptional components is realised by an internal (not architectural) connector, as presented in Figure 6. In this figure, the normal behaviour is realised by the MessagingOperations component, while the exceptional behaviour is realised by WarningHandlers. The association between the error detection (into MessagingOperations) and the exception handlers (into WarningHandlers) is realised by the InternalConnector. The MessagingOperationsIFTC represents an architectural component, which is characterised by combining both normal and exceptional components. Figure 6 also presents the connection involving architectural elements. In this example, the MessagingOperationsIFTC and MediaManager components were connected via the MsgMediaConn architectural connector.

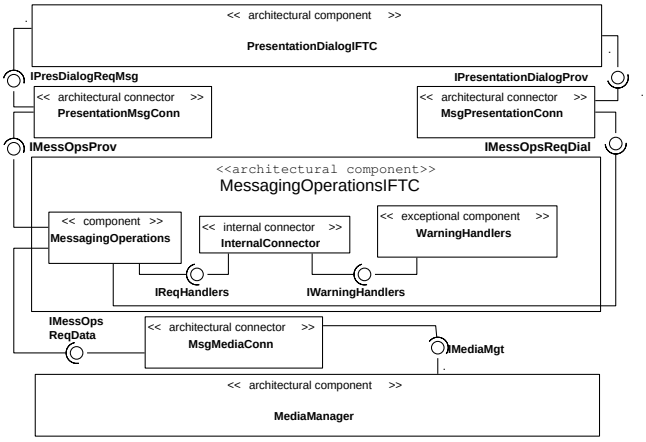

Figure 6: Refactored PLA with exceptional components

4.5 Implementation of PLA

For implementing the refactored PLA (Activity 3 of Figure 2), we have chosen COSMOS* component implementation model, as presented in Section 2.3. First, the software architecture presented in Figure 5 was graphically modelled using the Bellatrix case tool [20]. Second, the skeleton source code of its structure was automatically generated in Java. Then, following the process presented in Activity 3.3 of Figure 2, the existing classes of the object-oriented version of MobileMedia were reused and encapsulated into architectural components. The criterion we have used to group classes was their relations with the features present into the feature model. Some classes, which were used for more than a single feature, have been replicated into different components.

Figure 7 illustrates the internal structure of the MessagingOperations component, which implements the messaging functionality. As defined by COSMOS*, its specification package (`messagingoperations.spec`) contains its provided (`spec.prov`) and required (`spec.req`) interfaces, as well as

the respective data types (`spec.dataTypes`) and exceptions (`spec.dataTypes.excep`). The IMessOpsProv interface contains the operations provided by the MessagingOperations component. The provided and required interfaces, as well as data types and exceptions, are represented as part of the software component, which makes it a self-contained and deployable element.

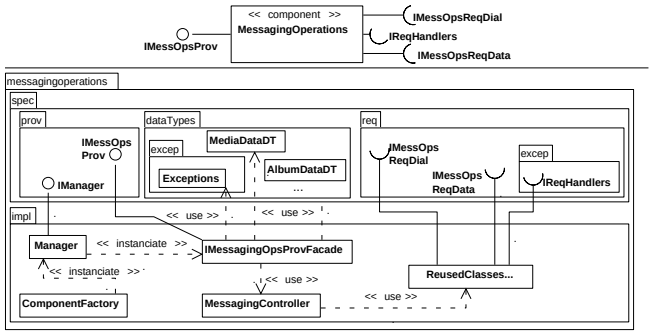

Figure 7: Internal structure of the MessagingOperations component based on COSMOS*

Regarding the required interfaces, the interfaces IMessOpsReqDial and IMessOpsReqData contain the required operations related respectively to the PresentationDialog and MediaManager components. The IPresReqHandlers declares the exception handlers which are executed by the implementation classes of the MesagingOperations component. This interface is the link between the normal and exceptional behaviour, and is also responsible for providing an explicit separation of concerns between them, as we have presented in Section 4.4. This separation facilitates the specification of variabilities related to the choice of different versions of exception handlers, that is, the choice of different exceptional components.

In the package `presentationdialog.impl`, we have reused the object-oriented implementation of MobileMedia classes responsible for the user interface functionality (`ReusedClasses`). The façade class IPresDialogProvFacade implements the provided interface IPresDialogProv in order to abstract away from the real internal controllers that actually realises it. Since the reused classes implement the functionality of the component, the IPresDialogProvFacade only propagates the requests received through the component's provided interface to the reused controllers.

Since each architectural component is a deployable entity of the application, the main program needs to implement the binding between the required and provided interfaces to create an executable configuration. For this, it is necessary to use the management operations defined by the COSMOS* model (IManager interface). Figure 8 exemplifies the assembling between the component `MessagingOperations`, which depends on its `IMessOpsReqData` required interface, and the component `MediaManager`, which provides the `IMediaMgt` interface. This communication is intermediated by the `MsgMediaConn` connector: $MessagingOperations \rightarrow MsgMediaConn \rightarrow MediaManager$. The same principle is applied to any assembly between two components. Lines 2 and 3 of Figure 8 present the instantiation of the COSMOS* components, using the factory method design pattern [12], while Line 4 presents the instantiation of the connector class. Finally, Line 6 shows the association between a required interface of the `MessagingOperations` and the object that provides it.

```
1 ...
2 msgoperations.spec.prov.IManager msgOps =
      msgoperations.impl.ComponentFactory.
      createInstance();
3 mediamanager.spec.prov.IManager mediaMgr =
      mediamanager.impl.ComponentFactory.
      createInstance();
4 msgoperations.spec.req.IMessOpsReqData mmConn =
      new MsgMediaConn();
5
6 msgOps.setRequiredInterface("IMessOpsReqData",
      mmConn);
7 ...
```

Figure 8: Example of binding between architectural elements in COSMOS*

5. EVOLVING THE TARGET SPL

After refactoring the PLA, we have evolved the feature model of the SPL, presented in Figure 3, in order to evaluate the benefits of the proposed approach related to the evolvability of the exceptional behaviour and the choice of different exception handlers according to different product instantiations. The evolution of the feature model has been conducted by adding a new optional feature, called Alarm, which impacts the system exceptional behaviour. Depending on the selection of this feature, all handlers can either present a single message (as they did originally) or present a message followed by an alarm beep.

The change of the feature model was propagated to both original and the refactored PLAs and the source codes. A new component has been added called AlarmOperations, which is responsible to receive the requests for raising an alarm and propagate it to the infrastructure of the mobile device. In the case of the target SPL, the mobile device is represented by the standard API of Java Micro Edition. Besides adding a new component, the architectural configuration has been also modified. The elements which have exceptional components may request services to the AlarmOperations component.

New exceptional components have been added in the refactored PLA for implementing the exception handlers with alarm. Moreover, external architectural connectors has been defined for the new exceptional components. Figure 9 presents the new internal structure of the MessagingOperations architectural component. Notice that in this case, the selection of the Alarm feature is associated with VP-EH variation point, which consists on choosing either WarningHandlers or WarningAlertHandlers.

Regarding the original PLA, a total of nine classes had to be modified in a total of 39 different places, since all the "catch" blocks which originally provided an error message had to be changed. In this case, the variability of the exceptional behaviour has been implemented through conditional compilation using the `#if` and `#endif` directives.

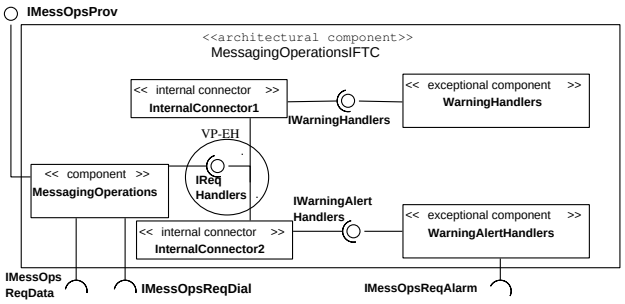

Figure 9: Architectural variation point of exception handlers

6. DISCUSSION AND EVALUATION

The claim being made in this paper is that the system componentisation brings a better separation of concerns between the normal and exceptional behaviour and allows the specification of architectural variabilities related to the use of different exception handlers. Moreover, the traceability between the software architecture and the feature model facilitates the evolution of the normal and exceptional behaviour. Our discussion is separated into two parts: exception handling evaluation, presented in Section 6.1, and component-based PLA evaluation, presented in Section 6.2.

6.1 Exception Handling Evaluation

As discussed in Section 5, in order to assess the evolvability of the exceptional behaviour using the proposed solution, we have added an optional feature which impacts the exceptional behaviour. In order to verify the benefits of our approach, we have compared the number of modifications related to exception handling made for evolving the original PLA with the number of modifications for evolving the refactored PLA. To apply this evolution on the original PLA we had to modify 39 "catch" blocks in nine classes in order to implement the new handler, while in the refactored PLA we have just added two new exceptional components, which were implemented as single classes. This happened because, in the original PLA, the classes that were changed also implements the normal behaviour of the system, while in the refactored PLA the normal components were not affected by the change.

Moreover, the instantiation of products with different exception handlers can also be facilitated by the implementation of exception handling variability at the architectural level. In the original PLA the exception handler varibility is implemented using directives of conditional compilation, which are spread throughout the normal components, while in the refactored PLA the selection of a feature just implies on the selection of the architectural elements that implement that feature.

For assessing the overhead of our approach, we have measured the memory usage related to the exceptional behaviour. As presented in Section 4.4, internal connectors have been created to link the normal behaviour to the exception handlers. Moreover, since the software components presented in our solution are self-contained, it was necessary to define exception classes in each component. In the refactored PLA, we measured a total of 5.4 extra KB of '.class' files, compared to the original PLA, which corresponds to a growth of 18% on the size of the exceptional behaviour.

6.2 Component-Based PLA Evaluation

The modelling and understanding of the refactored PLA was made easier because the building blocks of its architecture were considered in a high-level abstraction. This higher abstraction has reduced the number of architectural elements in 67%. As presented in Section 4.1, the original PLA had a total of 18 classes (as architectural elements), against 6 components in the refactored PLA.

Besides the simplified view of the system's structure, the adoption of a systematic process for identifying architectural components from the feature model has provided an important categorisation of components according to the features they refer to. In such a way, this case study has also shown that the component-based modularisation of the SPL allows an architecture-centred perspective of the development, thus facilitating the instantiation of products based on exception handling variability of the PLA. In other words, once the features have been selected, it is possible to know how to instantiate the PLA through its variation points.

Regarding the effort of implementation, we have noticed that the adoption of the COSMOS* model has facilitated the reuse of existing classes for implementing the architectural components. This internal reuse was achieved by using the Façade design pattern for implementing the component's provided interfaces. Each provided interface is realised by an intermediate class (Façade) which propagate the calls to the reused controllers. When reusing controllers, the entity classes referred by them are also automatically reused. Ignoring the data types and interfaces, which contain respectively attributes and method's signatures, approximately 78% of the classes used for implementing the architectural components have been reused either straightforward, or with minimal changes.

Regarding the memory usage of the mobile device, the implementation of the refactored PLA have presented an overhead of 173% in size: approximately 156 KB of '.class' files of the original PLA implementation, against 426 KB of the refactored PLA implementation using COSMOS*. This overhead is a consequence of extra interfaces, data types and explicit connectors of the COSMOS* model. At runtime the overhead is not significant, since the refactored PLA implementation used approximately 1.5% of extra RAM memory, when assessed with the memory monitor tool of the Sun Microsystems Wireless Tool-Kit (WTK). This happens because all components of the refactored PLA implement a specific feature, which is not entangled with another component, hence as they are not used at the same time, the Java Virtual Machine can instantiate each one when necessary.

7. RELATED WORK

Managing exceptions is a well-known way of structuring crosscuting concerns for improving the software reuse and reduce the coupling between modules. In previous work [3, 11], we have proposed a development method for specifying the exceptional behaviour of component-based software systems. In that work, the separation of concerns between the

normal and exceptional behaviour was achieved since the beginning of the development process, and remained at the source code using the COSMOS* model. The present paper focuses on the evolution of existing software product lines, and allows the specification of exceptional variabilities.

Other contributions have proposed aspect-oriented solutions for explicitly separating the exception handlers of object-oriented systems. Lippert and Lopes [16] have employed re-engineering on an OO Framework, called JWAM. The authors have refactored JWAM's exception handling code to verify if the use of Aspects brings any benefits. They found that, when applied to a reusable infrastructure that implements general exception handling policies (as in the case of JWAM), the modularisation of exceptions brought advantages in reuse and a decreased number of LOC. Besides allowing the reuse of general exception handlers, our work also aims to allow the specification of variabilities related to exception handling in SPL's and to improve the traceability between the software architecture and the feature model.

Recent work by Cacho et al. [5] proposes a novel exception model that allows explicit representation of exception control flows and handlers. This work is different from ours, since we focus on an architecture-centred solution for component-based software systems. The work by Cacho et al. focuses in a lower level of abstraction, related to the detailed design and implementation of software systems. The approach proposed in our paper starts with the architectural design, thus allowing the specification of exceptional variabilities in earlier phases of the software development.

8. CONCLUSIONS AND FUTURE WORK

This paper presents a method for refactoring object-oriented PLAs in order to separate explicitly their normal and exceptional behaviour using different software components. This separation of concerns allows the specification of different exception handlers that can be selected during the architectural configuration, thus facilitating the evolution of the exceptional behaviour and its instantiation into different products. The feasibility of the proposed approach was assessed using a SPL of mobile applications, which showed that it is possible to separate explicitly the normal and exceptional behaviour of a system, and to specify architectural variation points related to the exceptional behaviour. For implementing the case study, we have used the COSMOS* component implementation model, which maintains traceability between the component-based product-line architecture and its source code.

A limitation of the proposed solution concerns its evaluation when refactoring critical SPLs. Furthermore, we intend to overcome a limitation of our solution regarding the explicit representation of exception variability at the implementation-level. For this, one possibility is the use of aspect-oriented programming to define variable internal connectors realising the binding between the normal component and the exception handlers used by it.

9. REFERENCES

[1] C. Atkinson et al. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, December 1997.

[3] P. H. S. Brito, C. R. Rocha, F. Castor Filho, E. Martins, and C. M. F. Rubira. A method for modeling and testing exceptions in component-based software development. In *Proc. of the 2nd Latin American Symposium on Dependable Computing (LADC 2005)*, LNCS 3747, pages 61–79, 2005.

[4] F. Buschmann et al. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., 1996.

[5] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo. Ejflow: Taming exceptional control flows in aspect-oriented programming. In *7th Int. Conf. on Aspect-Oriented Software Development (AOSD'08)*, pages 72–83, 2008.

[6] J. Cheesman and J. Daniels. *UML Components*. Addison-Wesley, 2000.

[7] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[8] F. Cristian. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*, pages 68–97. Blackwell Scientific Publications, 1989.

[9] S. Ferber, J. Haag, and J. Savolainen. Feature interaction and dependencies: Modeling features for reengineering a legacy product line. In *Proc. of the Second International Software Product Lines Conference (SPLC), LNCS 2379*, pages 37–60, 2002.

[10] E. Figueiredo et al. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE '08: Proc. of the 30th international conference on Software engineering*, pages 261–270, 2008.

[11] F. C. Filho, P. A. de C. Guerra, V. A. Pagano, and C. M. F. Rubira. A systematic approach for structuring exception handling in robust component-based software. *Journal of the Brazilian Computer Society*, 10(3):5–19, 2005.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software Systems*. Addison-Wesley, 1995.

[13] L. A. Gayard, C. M. F. Rubira, and P. A. de Castro Guerra. COSMOS*: a COmponent System MOdel for Software Architectures. Technical Report IC-08-04, Feb. 2008.

[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[15] P. Kruchten, J. H. Obbink, and J. A. Stafford. The past, present, and future for software architecture. *IEEE Software*, 23(2):22–30, 2006.

[16] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proc. of the 22nd international conference on Software engineering (ICSE'00)*, pages 418–427, 2000.

[17] D. L. Parnas and H. Würges. Response to undesired events in software systems. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 437–446, 1976.

[18] M. P. Robillard and G. C. Murphy. Designing robust java programs with exceptions. *SIGSOFT Softw. Eng. Notes*, 25(6):2–10, 2000.

[19] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, USA, second edition edition, November 2002.

[20] R. T. Tomita, F. Castor Filho, P. A. de C. Guerra, and C. M. F. Rubira. Bellatrix: An environment with arquitectural support for component-based development (in portuguese). In *Proc. of the IV Brazilian Workshop on Component-Based Development*, pages 43–48, 2004.