Melo et al. Journal of Software Engineering Research and Development 2013, 1:3
www.jserd.com/content/1/1/3

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH Open Access

In-depth characterization of exception flows in software product lines: an empirical study

Hugo Melo, Roberta Coelho*, Uirá Kulesza and Demostenes Sena

* Correspondence:
souzacoelho@gmail.com
Informatics and Applied Mathematics Department (DIMAp), Federal University of Rio Grande do Norte, Natal, Brazil

Abstract

Software Product Lines (SPLs) play an essential role in contemporary software development, improving program quality and reducing the time to market. However, despite its importance, several questions concerning SPL dependability did not get enough attention yet, such as: how the exception handling code has been implemented in SPLs? The characteristics of the exception handling code may lead to faulty SPL products? The Exception Handling (EH) is a widely used mechanism for building robust systems and is embedded in most of mainstream programming languages. In SPL context we can find exception signalers and handlers spread over code assets associated to common and variable SPL features. If exception signalers and handlers are added to a SPL in an unplanned way, products can be generated on which variable features may signal exceptions that remain uncaught or are mistakenly caught by common features or other variable features. This paper describes an empirical study that categorizes the possible ways exceptions flow through SPL features and investigates whether some of their characteristics can lead to faulty exception handling behavior. The study outcomes presented in this paper are helpful in several ways, such as: (i) enhancing the general understanding of how exceptions flow through mandatory and variable features; (ii) providing information about the potential problems related to specific kinds of flows detected in this study; and (iii) presenting how a static analysis tool can be used to support the identification of potentially faulty exception handling flows.

Keywords: Software product line; Exception handling; Static analysis; Code inspection

1. Introduction

Software product line engineering advocates the development of software system families from a specific market segment (Clements and Northrop 2001). A system family is a set of programs that shares common functionalities and maintain specific functionalities that vary according to specific systems being considered (Parnas 1976). A software product line (SPL) is specified, designed and implemented in terms of common and variable features. A feature (Czarnecki and Eisenecker 2000) is a system property or functionality that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in SPLs. A SPL is developed through of a design of an extensible software architecture and subsequently implemented in terms of reusable code assets that address its common and variable features. The SPL development approach promotes benefits such as cost reduction, product quality, productivity

Springer

© 2013 Melo et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

and time to market (Clements and Northrop 2001). However it may bring new challenges to the software dependability.

The Exception Handling (EH) is a widely used mechanism for building robust systems (Goodenough 1975) (Garcia et al. 2001). As EH mechanisms are embedded in most of mainstream programming languages, it is also used in SPL engineering as a way of structuring fault detection and recovery solutions (Bertoncello et al. 2008). Exception signalers and handlers can then be found spread over code assets associated to common and variable features. Hence, intriguing questions arise when developing an exception-aware SPL, such as: How do exceptions flow through both variable and common features of SPLs? How do these exception flows can contribute or lead to a faulty exception handling behavior? A faulty exception behavior may happen when code assets implement common or variable features signals exceptions that are mistakenly caught inside the system. This is an exception handling bug very difficult to detect, known as Unintended Handler Action (Miller and Tripathi 1997).

In a previous study (Melo et al. 2012), we started seeking for answers to these questions. We performed an empirical study based on manual inspection and static code analysis, which presented a first categorization for the ways exceptions were signaled and handled by variable and common features of SPLs. The study was based on two well-known benchmark software product lines: MobileMedia (Young 2005) and Berkeley DB (Kästner et al. 2008). In this study, we have identified common ways in which exceptions are raised and handled inside the SPLs. We could observe that in many circumstances, exceptions raised by variable features were not adequately caught. Some of them were caught by generic handlers defined in the core, and other ones were caught by other variable features with no explicit relation between them.

This paper extends the previous study in the following ways: (i) a new medium-sized product line was analyzed, called Prevayler (Godil and Jacobsen 2005), which implements an open-source memory database configurable system that allows persisting serializable Java objects; (ii) an in-depth analysis of each flow was performed, which besides considering the exception signaler and handler to categorize each flow, also reports about the intermediate elements that compose the exception flows; (iii) a new version of the static code analysis tool was implemented in order to support the in-depth analysis of each flow; (iv) moreover, this work also presents an uncaught exception analysis for each SPL and it discusses about the fault-prone scenarios that may occur in SPL products in the exception handling context.

The contributions of this work allow the developers of dependable SPLs: (i) to consider the potential effects of variable features on the exception flow of SPL products; (ii) to define/use specific variability implementation techniques to deal with such effects; and (iii) to make more informed decisions when defining the possible SPL products. Moreover, it also allows for the designers of Exception Handling policies and strategies to consider improving existing EH solutions to make them more robust and resilient to flaws in the exception handling code.

The remainder of this paper is organized as follows. Section 2 presents the study settings. Sections 3 and 4 present the results of the two study phases. Section 5 provides further discussions and lessons learned. Section 6 describes related work. Finally, Section 7 presents our conclusions. Due to space limitations, throughout this paper we

assume that the reader is familiar with SPL techniques and terminology (Clements and Northrop 2001) (Czarnecki and Eisenecker 2000).

2. Study settings

This section describes the configuration of our empirical study in terms of its main goals and research questions, the investigated SPLs (Section 2.1), the study phases (Section 2.3), and the static analysis tool developed to support the investigation of the SPL exception flows (Section 2.4). Section 2.2 uses a code snippet extracted from one of the target SPLs to illustrate the exception handling concepts discussed in this study.

The main goals of our study were: (i) to analyze different Java-based SPL implementations for the purpose of characterizing the exception flows with respect to the code assets responsible for signaling and handling exceptions, and the common and variable features related to them; and (ii) to investigate the fault-proneness of each exception flow category identified in the study.

The research questions that guided this study were the following: (RQ1) How are exceptions signaled and handled through variable and common features of SPLs? (RQ2) Which kinds of exception flows contribute (or may lead) to a faulty (or inadequate) exception handling behavior?

2.1. The target software product lines

One major decision that had to be made for our investigation was the selection of the target SPLs. We have selected three medium-sized well-known benchmark SPLs implemented in Java: MobileMedia (Young 2005), Berkeley DB (Kästner et al. 2008), and Prevayler (Godil and Jacobsen 2005). Such SPLs implement variable behavior and associated features using CIDE (Kästner et al. 2008). CIDE (Colored IDE) is a tool that enables SPL developers to annotate code with feature information using background colors (similar to #ifdefs). All SPLs were used in several empirical studies (Figueiredo et al. 2008) (Brabrand et al. 2012) (Kästner et al. 2008) (Coelho et al. 2008)) (Godil and Jacobsen 2005), and each of them is a representative of different application domains, and heterogeneous realistic ways of incorporating exception handling.

The MobileMedia (MM) is a SPL of applications that manipulates media (e.g., photo, music and video) on mobile devices. There are subsequent Java releases available. All adopt the same architecture style (i.e., model-view-controller), varying in terms of the number of features available and design decisions taken in each version. Our study focused on the 8th release (http://sourceforge.net/projects/mobilemedia).

Berkeley DB (BkDB) is a SPL for embeddable databases of moderate size (42 features) that implements functionalities related to management of memory, logging, transactions, concurrency, and others databases functionalities.

Prevayler (Pvl) is a SPL for the context of in-memory database systems, based on object prevalence. Derived systems of this SPL support plain Java object persistence, snapshots, queries, transactions, and logging. The release analyzed in our study implements five variable features: Replication, GZip, Censor, Monitor, and Snapshot. This release is a subset of the original implementation that was defined in (Prevayler Project 2013).

Table 1 summarizes code characteristics of target SPLs: the number of lines of code (LOC); the number of lines of code dedicated to exception handling (EH LOC); the number

Table 1 Summary of characteristics of target SPLs

Metrics	Software product lines		
	MobileMedia	**BerkeleyDb**	**Prevayler**
LOC	3191	39233	5122
EH LOC	614	3028	458
#Throw Clauses	24	127	68
#Exception Flows	111	1522	164
#Classes and Interfaces	51	238	140
#Checked Exceptions	9	22	3
#Unchecked Exceptions	0	3	0
#Variable Features	11	42	5

of throw-clauses; number of exception flows; number of classes and interfaces; number of user-defined checked and unchecked exceptions; and the number of variable features.

2.2. SPL exception handling code example

In order to support the reasoning about the exception handling in SPL context, we illustrate the main concepts of an exception handling mechanism based on an example extracted from MobileMedia product line. Figure 1A presents a code snippet associated to a mandatory feature responsible for creating new albums of a given media (i.e. photo, music, or video) from the `MediaAccessor` class. Besides the normal execution flow, there are two scenarios on which the album cannot be created: either the album name is an empty string (lines 7–8), or if the album name is already in use because there is another album with the same name (lines 11–14). In such scenarios, the normal execution flow should be interrupted and an error message should be presented to the user. These scenarios are called exceptional scenarios, and they are implemented using the exception handling constructs that are briefly described in the next paragraphs.

In modern OO languages such as Java, C++ and C#, such abnormal situations are represented by *exceptions* which are represented as objects. An exception may be raised by a method whenever an abnormal computation state is detected. The *exception signaler* is the method that detects the abnormal state and raises the exception. In Figure 1A, the `createNewAlbum()` method from `MediaAccessor` class detects an abnormal condition and raises the exception `InvalidPhotoAlbumNameException` using the throw clause. Since this method is not annotated (colored in CIDE tool) with any variable feature, we say that such signaler belongs to the SPL core.

Some languages provide constructs to associate a method's signature with a list of exceptions that it may raise (see lines 3–4 in Figure 1A). Besides providing information for the method's callers, this information can be checked at compile time to verify whether handlers were defined for each specified exception. This list of exceptions represents the *exception specification* or *exception interface* of a method. Ideally, the exception interface should provide complete and precise information for the method user. However, some languages, such as Java, allow the developer to bypass this mechanism. In such languages exceptions can be of two kinds: (i) checked exception – that needs to be declared on the method's signature that throws it; and (ii) unchecked exception – that does not need to be declared on the signaler method's signature.

(A)

```
1.  public abstract class MediaAccessor
2.  {
3.      public void createNewAlbum(String albumName)
4.          throws PersistenceMechanismException, InvalidPhotoAlbumNameException {
5.
6.          ...
7.          if (albumName.equals("")){
8.              throw new InvalidPhotoAlbumNameException();
9.          }
10.
11.         String[] names  = getAlbumNames();
12.         for (int i = 0; i < names.length; i++) {
13.             if (names[i].equals(albumName))
14.                 throw new InvalidPhotoAlbumNameException();
15.         }
16.         ...
17.         // Create new album and persist it in the file system
18.         ...
19.     }
20.     ...
21. }
```


(B)

```
1.  public class AlbumController extends AbstractController
2.  {
3.      public boolean handleCommand(Command command) {
4.
5.          String label = command.getLabel()
6.          ...
7.
8.          if (label.equals("Save")) {
9.              ...
10.             try {
11.                 password = (PasswordScreen) getCurrentScreen();
12.                 getAlbumData().createNewAlbum(albumName.getLabelName());
13.                 getAlbumData().addPassword(albumName.getLabelName(), password.getPassword());
14.             } catch (PersistenceMechanismException e) {
15.                 ...
16.             } catch (InvalidPhotoAlbumNameException e) {
17.                 Alert alert = new Alert(..., "You have provided an invalid Photo Album name", ...);
18.                 ...
19.             }
20.         }
21.         ...
22.     }
23.     ...
24. }
```


Figure 1 Exception handling code example from MobileMedia SPL. Partial code of **(A)** MediaAccessor class; and **(B)** AbstractController class.

After a method signals an exception, the runtime system attempts to find the block of code that will be responsible for handling it. The exception handler in Java should be defined in the dynamic call chain of the method signaling the exception. In the scenario illustrated in Figure 1B the handler of the exception signaled by the method `createNewAlbum()` is caught by the method `handleCommand()` (lines 16–18) of the `AlbumController` class. In Java programs, a try-catch block represents the block of code responsible for handling exceptions. In this example the handler block is colored using the CIDE tool, meaning that it belongs to the Privacy feature. We call the *exception flow* a path in a program call graph that links the *signaler* and the *handler* of an exception.

2.3. Study phases

In order to provide answers to the research questions – RQ1 and RQ2 – presented before, our study has been organized in two main phases, each of them focusing on one of the research questions stated previously. The study phases were the following: (i) an analysis and characterization of the exception flows in terms of variable and common features responsible for signaling and handling the exceptions; and (ii) a detailed analysis of the exception flows (i.e., method call chains, and handler actions).

There are 3 different approaches to verify a SPL (Apel et al. 2013): (i) the product-based strategy that analyzes every product individually; (ii) the sample-based strategy that focuses on a subset of existing products; and (iii) the family-based strategy that analyzes the design and implementation artifacts of the whole SPL in a single pass. In our exploratory study, we discarded the product-based strategy because it is costly and impractical to analyze the exception flows of every possible product of the target SPLs. The family-based strategy could not be adopted due to the non-existence of static analysis frameworks to support the interprocedural analysis of exception flows of the whole product line at once. Hence we have decided to use the sample-based strategy by focusing on the analysis of products of each investigated SPL that includes all variable features. In our study, it was possible to derive the product of each SPL that includes all the variable features – since the target SPLs do not define alternative features and do not define explicit exclusion constraints between their features. These three products are expressive and representative to be studied in order to characterize the behavior of exception flows in SPLs. We also noticed that during our analysis most of exception flows involved a few number of features, which means that these exception flows are present in all products that include those features.

Phase 1: Characterization of SPL exception flows

The main goal of this phase was to answer RQ1, in other words to explore the possible ways exceptions could flow inside a SPL. We have started this phase by manually inspecting the target product lines but soon such task became infeasible (i.e. the exception flows were too deep and there were too many flows to be inspected). The high cost of manual inspection led to the implementation of PLEA (Product Line Exception Analyzer) tool, which performs a feature-oriented exception flow analysis (see Section 2.4). In this phase, the exception flows of each SPL were calculated using PLEA tool. Every exception flow was analyzed and characterized in terms of the common and variable features of the code assets responsible to throw and handle the exceptions.

Phase 2: In-deph analysis of exception flows and SPL design issues

The main goal of this phase was to refine the answer given to RQ1 in the first phase while looking for answers to RQ2. To do so we manually inspected the exception flow in detail, evaluating not only the signaler and handler information, but also the intermediate elements that compose a flow. In addition, we also investigated the impact of SPL design issues on the types of flows found, and the fault proneness of each of them. Such phase contributes to understand the exception handling policy adopted by each investigated SPL, and which kind of flows can represent a risk for the robustness of the execution of the SPL products. The detailed analysis and results of this phase are presented in Section 4.

2.4. PLEA – a feature-oriented static analysis tool

The PLEA – Product Line Exception Analyzer – tool has been developed to help the analysis and characterization of the SPL exception flows. The main aim of the tool is to calculate the exception flows thrown from the SPL code assets, and to characterize the classes and methods that are part of these exception flows, including the ones responsible to signal and handle the exception. In addition, the tool also distinguishes the classes from each exception flow that are implementing the SPL commonalities or variabilities. Figure 2 shows a package diagram that illustrates the dependencies between the two modules – *Flow Analyzer* and *Feature Identifier* – of PLEA, and the two external tools – *Design Wizard* and *CIDE* – that PLEA is integrated.

2.4.1 PLEA overview

PLEA is structured as two mains modules, which are implemented as Eclipse plug-ins (Eclipse IDE 2012): (i) the Flow Analyzer and (ii) the Feature Identifier. The Flow Analyzer performs an inter-procedural analysis on the SPL bytecode. It implements one of the most used algorithms for call graph construction called class hierarchy analysis (CHA) (Grove and Chambers 2001), in order to build the program dependency graph (PDG) (Ferrante et al. 1987). It traverses the PDG, firstly looking for the checked exceptions, explicitly thrown by the SPL code assets, and then looking for handlers that may handle them. As a result the plug-in reports a set of exception flows. When building the PDG of a SPL, the Flow Analyzer uses the DesignWizard tool (Brunet et al.

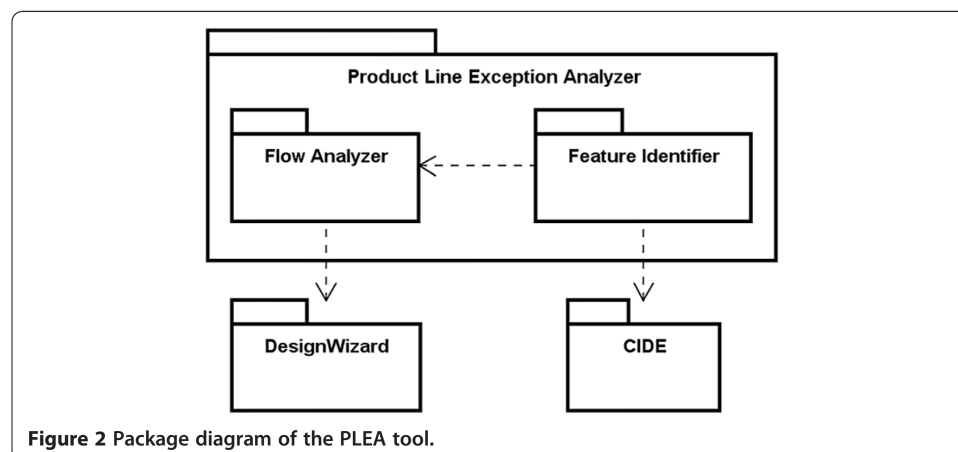

Figure 2 Package diagram of the PLEA tool.

2009), which provides an API that supports the automated analysis and inspection of Java programs.

The Feature Identifier plug-in is responsible for accessing the configuration knowledge (CK) of the SPL to obtain the information concerning the features associated to each exception flow: its signaler, handler and the intermediate methods between them. The CK (Czarnecki and Eisenecker 2000) defines how each code asset (class, interface, method, attribute) is mapped to the specific SPL common and variable features that they implement. The CK format depends on the tool used for the SPL implementation and variability management. The current version of PLEA manipulates the CK defined by CIDE (Kästner et al. 2008), the tool used for the variability management of the analyzed SPLs.

2.4.2 PLEA detailed design

Figure 3 shows a partial class diagram of the main packages and classes of the Flow Analyzer module. The `ExceptionFlowAnalyzer` class defines the `analyze()` method, which parses all the SPL code assets and creates a program dependency graph (PDG) in terms of instances of the `ClassNode` and `MethodNode` classes of the Design Wizard tool. After that, the `DesignWizardUtil` class is used to search the classes of the SPL in order to organize the relevant exceptions (`EAException`) and respective constructors (`EAConstructor`). Finally, the tool obtains the `MethodNode` that represent exception signalers and executes a depth-first search from each of these methods in order to find all the `MethodNode` instances that constitute their respective exception handlers. The depth-first search is executed over the call graph calculated for the Flow Analyzer module. All the information collected regarding exception signalers and handlers is organized in an `EAReport` instance that does not depend on the DesignWizard API.

The Feature Identifier module is responsible for identify the features associated to each signaler and handler methods listed in the `EAReport` instance. Figure 4 shows a class diagram with the main classes of this module. The `IdentifyFeatureAction` class is an action associated to a menu in the Eclipse IDE workspace that activates the execution of the module. It creates an instance of `IdentifyFeatureOperation`

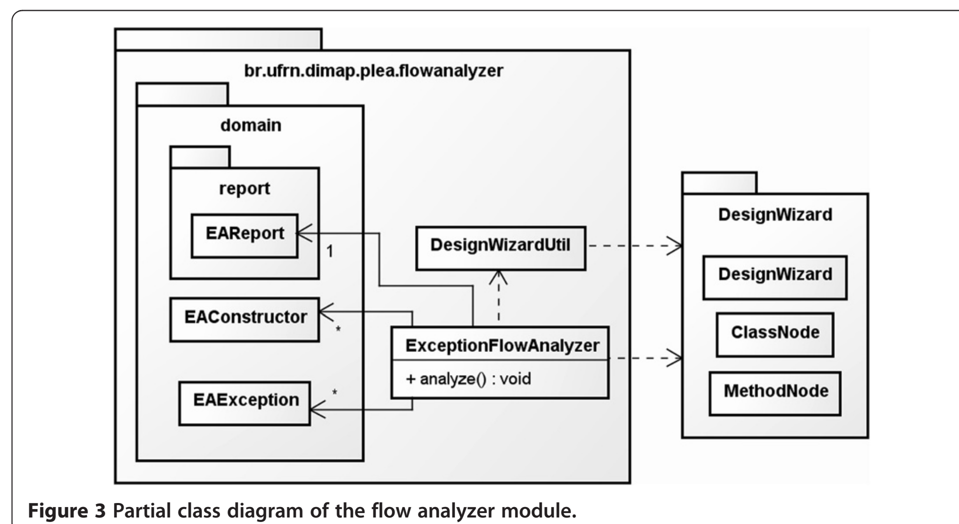

Figure 3 Partial class diagram of the flow analyzer module.

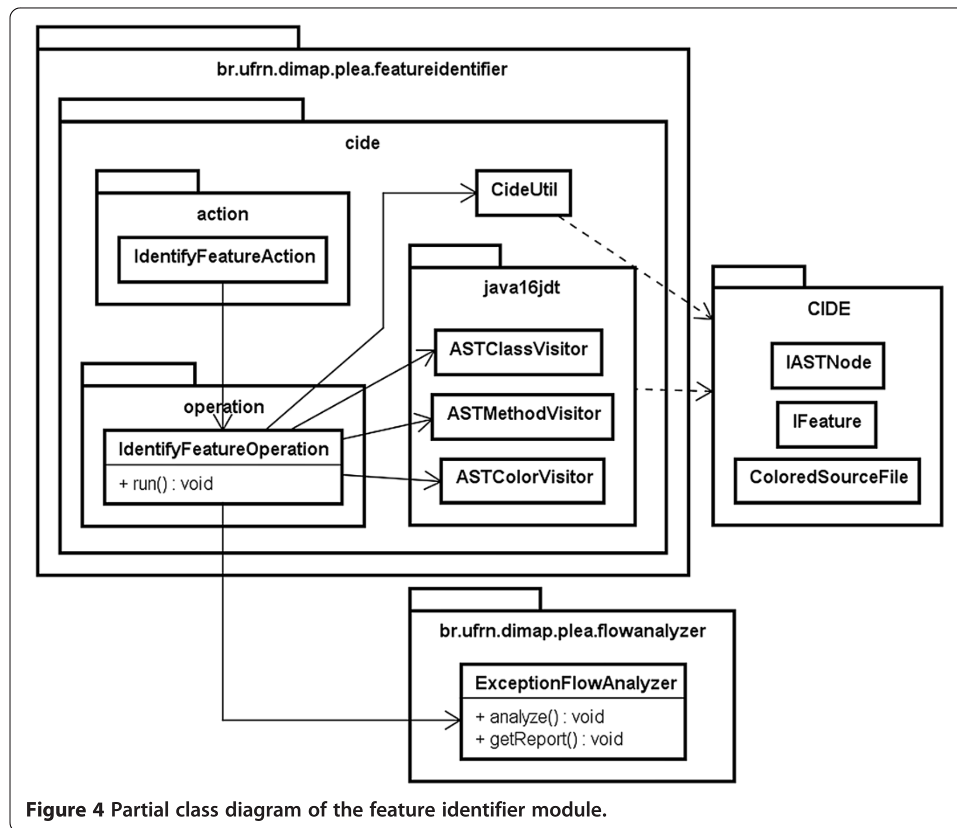

Figure 4 Partial class diagram of the feature identifier module.

and executes its `run()` method, which is responsible for identify the features of each signaler, handler and the intermediate methods of a flow. The current implementation of PLEA retrieves the configuration knowledge (CK) from the CIDE tool, which stores the features directly related to the code asset in the respective abstract syntax tree (AST) that represents each of them. The information regarding the features is obtained using the `CideUtil` class. In our implementation, we have extended a Visitor hierarchy (Gamma et al. 1994) that was implemented in the CIDE tool and helps identify the classes (`ASTClassVisitor`), methods (`ASTMethodVisitor`), methods call (`ASTCallerVisitor`), catch clauses (`ASTHandlerVisitor`), throw clauses (`ASTSignalerVisitor`) and features (`ASTColorVisitor`), when the ASTs associated to each signaler and handler class are traversing. The identified features for each method are attached to the same `EAReport` instance that was created in the Flow Analyzer module.

3. Study results for phase 1: characterization of SPL exception flows

This section summarizes the study results of the first phase, which involves the general analysis and characterization of the exception flows of the target SPLs. These results were collected during the execution of PLEA over Mobile Media, Berkeley DB and Prevayler. Section 3.1 presents the criteria used for exception flow categorization and the number of exception flows found in this study characterized according to these criteria. Section 3.2 compares the exception flow information obtained from the investigated SPLs. Finally, Sections 3.3 and 3.4 discuss about each flow type, presenting real examples extracted from the analyzed SPLs.

3.1 Collected results for the SPL exception flow types

The main goal of this analysis and characterization phase was to explore the possible ways exceptions could flow inside a SPL. We took into account the information concerning which features were associated to the pieces of code responsible for signaling and handling exceptions. The exception flows were characterized according to three attributes:

(i) *The feature associated with the exception signaler.* The piece of code responsible for signaling the exception can be associated to the core (C) or to a variable feature (V);
(ii) *The feature associated with the exception handler.* The piece of code responsible for handling the exception can be associated to the core (C), to a variable feature (V), or the exception can escape from every handler and remain uncaught (E);
(iii) *The exception handling action.* The exception can be caught by a specialized handler (S), a handler whose type is the same of the exception being caught, or by a generic handler (G), a handler whose type is a supertype of the exception being caught.

Table 2 presents the number of exception flows found in our study, characterized according to these attributes. It also shows the percentage of each flow type in relation to all flows found per SPL release. The flow type, first column of Table 2, is an acronym based on the attributes values, for instance: a CC flow is an exception flow on which a

Table 2 Exception flows in MobileMedia, BerkeleyDb and Prevayler

				Flow occurrences (percentage per release)			
Flow	**Signaler**	**Handler**	**Handling**	**MM**	**BkDb**	**Pvl**	**All**
CC	Core	Core	Any	48 (43%)	413 (27%)	76 (46%)	537 (30%)
CCS	Core	Core	Specialized	22 (20%)	63 (4%)	5 (3%)	90 (5%)
CCG	Core	Core	Generic	26 (23%)	350 (23%)	71 (43%)	447 (25%)
CV	Core	Variable	Any	28 (25%)	210 (14%)	6 (4%)	244 (14%)
CVS	Core	Variable	Specialized	25 (22%)	62 (4%)	4 (3%)	91 (5%)
CVG	Core	Variable	Generic	3 (3%)	148 (10%)	2 (1%)	153 (9%)
CE	Core	Escaped	-	0 (0%)	435 (29%)	62 (38%)	497 (27%)
	Subtotal (CC + CV + CE)			76 (68%)	1058 (70%)	144 (88%)	1278 (71%)
VC	Variable	Core	Any	18 (17%)	183 (12%)	6 (4%)	207 (12%)
VCS	Variable	Core	Specialized	18 (17%)	17 (1%)	0 (0%)	35 (2%)
VCG	Variable	Core	Generic	0 (0%)	166 (11%)	6 (4%)	172 (10%)
VV	Variable	Variable	Any	8 (7%)	19 (1%)	0 (0%)	27 (2%)
VVS	Variable	Variable	Specialized	8 (7%)	19 (1%)	0 (0%)	27 (2%)
VVG	Variable	Variable	Generic	0 (0%)	0 (0%)	0 (0%)	0 (0%)
VaVb	Variable	Variable	Any	9 (8%)	100 (7%)	0 (0%)	109 (6%)
VaVbS	Variable	Variable	Specialized	9 (8%)	13 (1%)	0 (0%)	22 (1%)
VaVbG	Variable	Variable	Generic	0 (0%)	87 (6%)	0 (0%)	87 (5%)
VE	Variable	Escaped	-	0 (0%)	162 (10%)	14 (8%)	176 (9%)
	Subtotal (VC + VV + VaVb + VE)			35 (32%)	464 (30%)	20 (12%)	519 (29%)
Total				111	1522	164	1797

core asset signals an exception and another (or the same) core asset handles it. Depending on the way the exception is caught, this flow can be sub-characterized as: CCS - if the exception is caught by a specific handler; or CCG - if the exception is caught by a generic handler. When the exception signaled by a core element remains uncaught, it was classified as CE. It is worth mentioning the special flow type identified as VaVb in Table 2. It represents the flow on which a given variable feature throws an exception and a different variable feature handles it, and there are no inclusion constraints between them. The VV represents the flow on which the same feature signals and handles the exception. Additionally, Table 3 focuses on the number of flow types signaled only by variable features, and presents the percentage of such flows in relation to all flows originated by variable features.

3.2. Exception flow types across different SPLs

Figure 5 illustrates the proportion of exception flows found in each investigated SPL. We can observe that although proportion of flow types in every SPL differed, most of the flows in all SPLs represent exceptions signaled by core elements. Considering CC, CV and CE flows all together, they represent, 68%, 70% and 88% of all flows found in MobileMedia, BerkeleyDb and Prevayler, respectively. Moreover, we could find exceptions signaled by variable features in all analyzed SPLs. In Prevayler, in particular, most of exceptions signaled by variable features escaped (VE). In MobileMedia, most of such exceptions were handled in the core (VC) or by other features (VV or VaVb).

Considering the way the exceptions were handled (i.e., by generic or specialized handlers) – regardless of whether they represent exceptions signaled by variable or core features – we could observe that most of the handlers in MobileMedia are specialized handlers, while in BerkeleyDB and Prevayler, generic handlers caught most of the exceptions. Figure 6 shows the results for the exception handlers. The high number of specialized handlers found in MobileMedia may give the impression that the exceptions are being adequately handled inside the MM SPL, and they are inadequately caught in other SPLs. However, only a deeper analysis can really show what is happening in the real scenario.

Table 3 Exception flows signalized by variable features

				Flow occurrences (percentage per release)			
Flow	**Signaler**	**Handler**	**Handling**	**MM**	**BkDb**	**Pvl**	**All**
VC	Variable	Core	Any	18 (51%)	183 (39%)	6 (30%)	207 (40%)
VCS	Variable	Core	Specialized	18 (51%)	17 (4%)	0 (0%)	35 (7%)
VCG	Variable	Core	Generic	0 (0%)	166 (35%)	6 (30%)	172 (33%)
VV	Variable	Variable	Any	8 (23%)	19 (4%)	0 (0%)	27 (5%)
VVS	Variable	Variable	Specialized	8 (23%)	19 (4%)	0 (0%)	27 (5%)
VVG	Variable	Variable	Generic	0 (0%)	0 (0%)	0 (0%)	0 (0%)
VaVb	Variable	Variable	Any	9 (26%)	100 (22%)	0 (0%)	109 (21%)
VaVbS	Variable	Variable	Specialized	9 (26%)	13 (3%)	0 (0%)	22 (4%)
VaVbG	Variable	Variable	Generic	0 (0%)	87 (19%)	0 (0%)	87 (17%)
VE	Variable	Escaped	-	0 (0%)	162 (35%)	14 (70%)	176 (34%)
Subtotal (VC + VV + VaVb + VE)				35 (100%)	464 (100%)	20 (100%)	519 (100%)

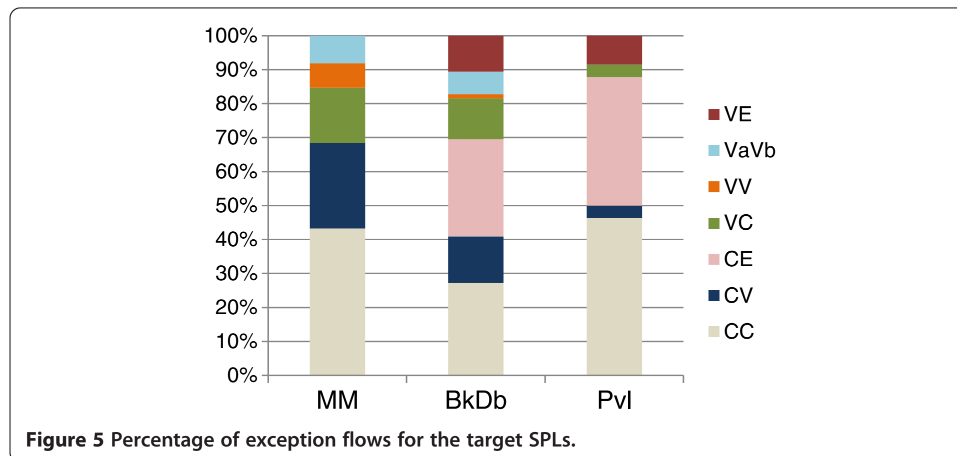

Figure 5 Percentage of exception flows for the target SPLs.

Next sections discuss about each flow type found in this study and presented above. Section 3.3 details the flow types signaled by core elements. Section 3.4 discusses and presents examples of the flow types signaled by variable features. The analysis of the exceptions that remained uncaught in every SPLs (i.e., CE and VE flows) is left to Section 4.

3.3 How the exceptions signaled by core elements are handled

We can observe that most of the exception flows from the target SPLs were signaled by core elements (see CC + CV + CE subtotal in Table 2, 68% in MM, 70% BkDB, and 88% Pvl). From this set, Figure 7 shows that a considerable amount was also caught by core elements (CCS and CCG flows) and a more reduced amount was handled by variable features (CVS and CVG).

The flows on which exceptions were signaled by core elements and handled by a variable feature were found in every SPL (see CV flows in Table 2, 25% in MM and 14% in BkDb and 4% in Pvl). One instance of the CV flows from Prevayler is depicted in Figure 8. The method `run()` that is defined in a variable feature calls a method from the same variable feature, called `CentralPublisher#subscribe()` (line 5). Such method calls other methods from core assets, and one of such methods signals an

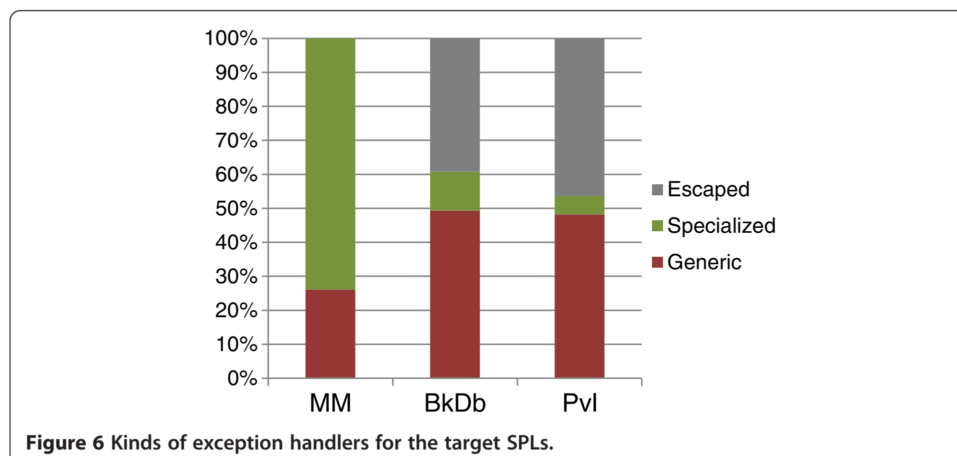

Figure 6 Kinds of exception handlers for the target SPLs.

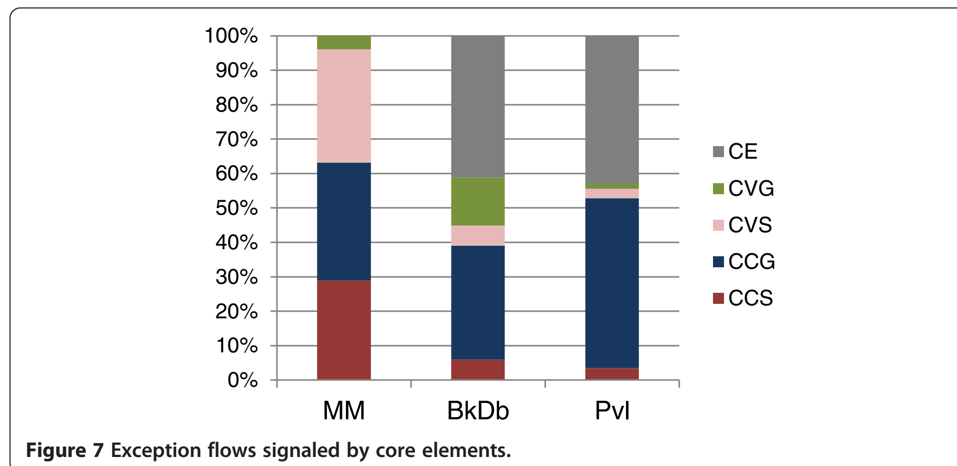

Figure 7 Exception flows signaled by core elements.

exception. Hence, we can observe that such flow happens when a variable feature reuses existing methods from core assets, Section 4 discusses in more details the causes and consequences of such flows.

3.4 How the exceptions signaled by variable features are handled

Table 2 also shows that a considerable amount of exception flows originated from variable features (32% in MM, 30% in BkDB, and 12% in Pvl). Figure 9 focuses on such exception flows, originated from variable features. From this set, most of them were handled by core elements (51% in MM, 39% in BkDB, and 30% in Pvl). Only part of such flows was indeed handled by the same variable feature that had signaled it (23% in MM, 4% in BkDb, and 0% in Pvl), as illustrated by the VV flows in Table 2 and Figure 10. We also observed flows on which a different variable feature caught the exception (26% in MM, 22% in BkDB and 0% in Pvl, see the VaVb flows in Figure 11). Next subsections present examples of these flow types.

Exceptions signaled by a variable feature and handled in the core (VC)

In BerkeleyDB and Prevayler, most of the exceptions signaled by variable features were captured by generic handlers defined in a core element, while in MobileMedia most of the exceptions signaled by variable features were caught by specialized handlers in the core (see VCG flows in Figure 9). Figure 12 illustrates a scenario on which a method from the core of BerkeleyDB calls methods from different variable features.

The method illustrated in Figure 12 pertains to a core asset and has around 100 lines of code. It contains method calls to 8 distinct variable features (through method calls), which can signal specific exceptions. The code snippet only presents three of such variable features that can signal instances of `DatabaseException` (see colored tags in the code). Moreover, such method also accesses other core methods that can also signal specific exceptions. Although distinct exceptions may flow in such method a single and generic treatment is given for all of them (lines 30–34).

3.4.1 Exceptions signaled and handled by the same variable feature (VV)

In MobileMedia and BerkeleyDB, we found exceptions signaled and handled by the same variable feature (7% in MM, 1% in BkDB). Figure 10 illustrates one of such flows

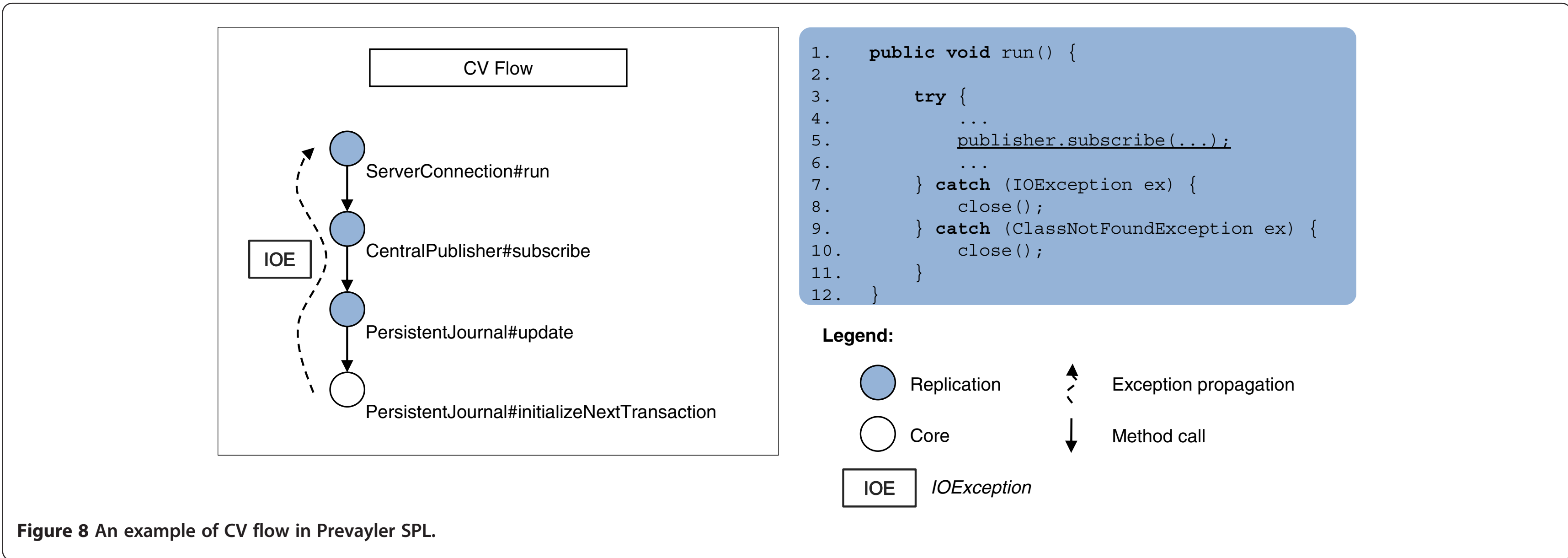

Figure 8 An example of CV flow in Prevayler SPL.

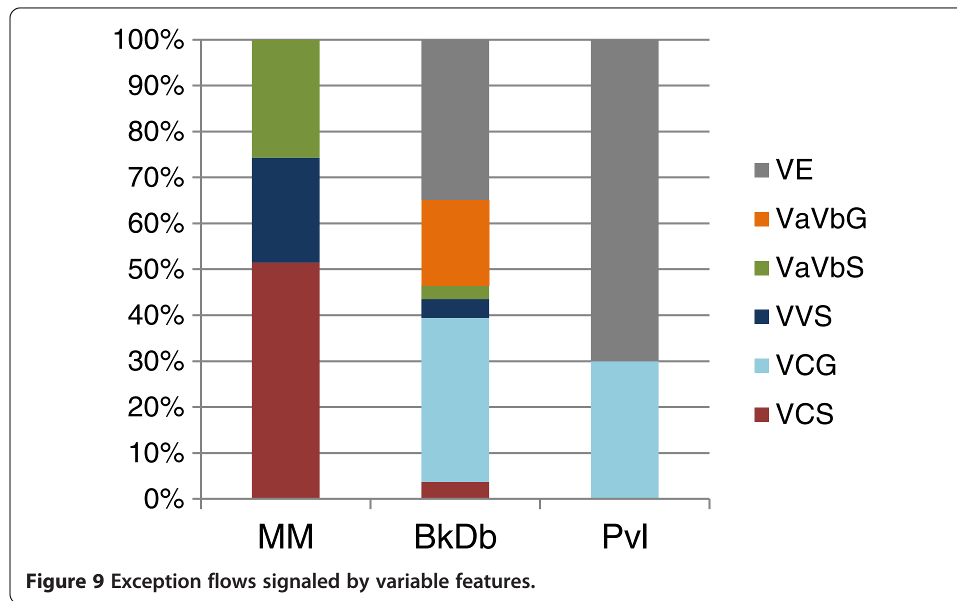

Figure 9 Exception flows signaled by variable features.

found in MobileMedia SPL. Inspecting such flow, we could observe that the intermediate elements (the methods called between the signaler and the handler) were also associated to a single feature, which means that, in such flows, no core functionality were reused.

In addition, considering the handler action associated to such flows, we observed that all flows signaled and caught inside the feature context (the VV flows) were handled using specialized handlers – which can lead to an adequate handling as a single feature can have enough information to handle the exception. An interesting finding was that even in BerkeleyDB, where most of the exceptions were handled by generic handlers (see Figure 6), the exceptions signaled and handled in the context of a single feature were caught by specific handlers.

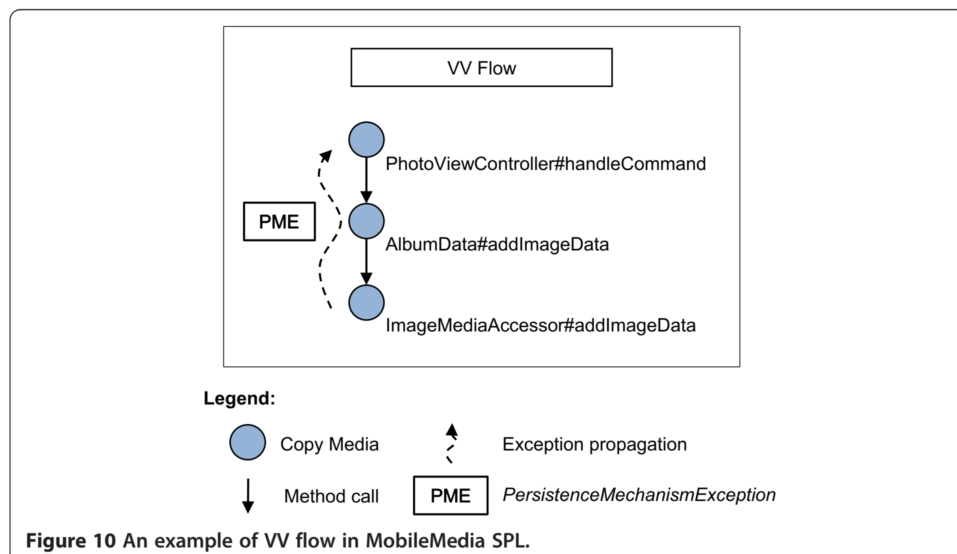

Figure 10 An example of VV flow in MobileMedia SPL.

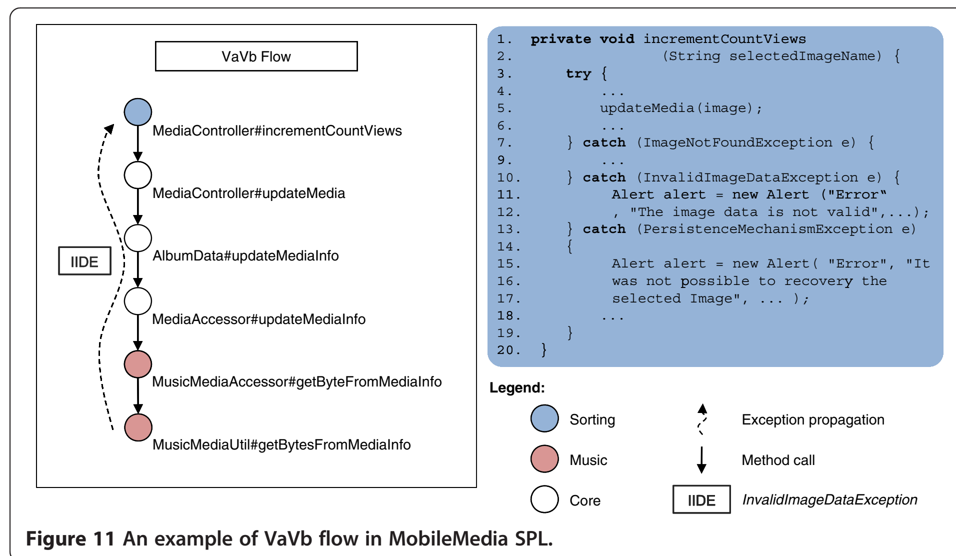

Figure 11 An example of VaVb flow in MobileMedia SPL.

3.4.2 Exceptions signaled and handled by distinct variable features (VaVb)

The exception flows, whose signaler was defined on a variable feature and the handler on a different variable feature, were found in two of the analyzed SPLs (8% in MM, and 7% in BkDb). Such flows bring out an implicit relation between features that may lead to EH faulty behavior as illustrated in Figure 11.

Figure 11 illustrates the code snippet of a method from a variable feature of MM called *Sorting*, which is responsible for sorting lists of different kinds of media. The

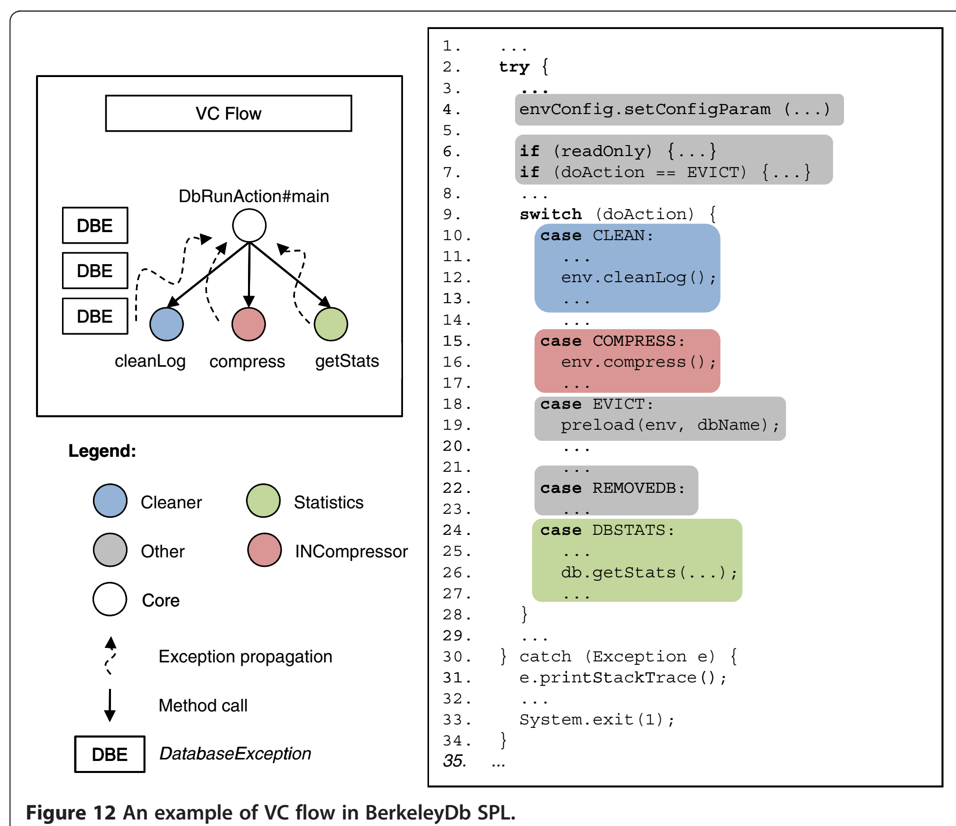

Figure 12 An example of VC flow in BerkeleyDb SPL.

`updateMedia()` method (line 5) transitively calls other methods as illustrated in the call chain illustrated on Figure 11. One of such methods in the call chain belongs to the *Music* feature (i.e., `getBytesFromMediaInfo()`), which can signal `InvalidImageDataException` (represented as IIDE in Figure 11). When such exception is signaled by *Music* feature, it is caught by a specific handler (lines 11–12), and a message related to another feature (i.e., *Photo* feature that may also interact with the *Sorting* feature) is presented to the user. Such exception handling confusion problem is a consequence of the implicit feature interaction in exception handling scenarios.

In the scenarios where VaVb flows were detected, there was no constraint associating the feature that signaled the exception and the feature that handled it. The handler feature does not have enough contextual information to adequately handle the exception. Such scenarios usually happen by mistake. In other words, the exception signaled by the feature would remain uncaught but was mistakenly caught by a handler defined by another feature. This is an instance of the Unintended Handler Action (Miller and Tripathi 1997) problem, a faulty exception handling behavior very difficult to diagnose at runtime. Such kind of interaction, which only happens in exception handling scenarios, was not documented in any of the SPLs analyzed in this study.

4. Study results for phase2: an in-depth analysis of exception flows

In the previous section the exception flows from the investigated SPLs were classified according to their signalers, handlers, and the features associated to each of them. This characterization enabled us to identify patterns related to exceptions signaling and handling while dealing with a high number of exception flows. In order to obtain a more fine-grained view of how exceptions are signaled and handled inside SPLs, we performed an in-depth analysis of each flow on which: (i) we investigated the intermediate methods that composed the exception flows call chains – this analysis was based on manual inspection and guided by results provided by PLEA static analysis tool; and (ii) we analyzed how the flow types that are related could lead to faulty or inadequate exception handling behavior.

Table 4 illustrates the number of flows inspected in this second study phase. All exceptions flows of MobileMedia and Prevayler were inspected, they contain 111 and 164 exceptions flows, respectively. The mean size of EH flows in MM was approximately 5, while in PvL it was 7. Table 4 also presents the mean, minimum and maximum size of exception flows of target SPLs. Since BerkeleyDb product line contains 1522 exception flows, whose mean size is approximately 12, only a few of its flows were evaluated during this study phase, which was strongly based on manual inspection. As a result, most of the findings of our in-depth analysis were related to scenarios found in MM and Pvl.

Table 4 Exception flows information

	MM	BkDb	Pvl
Size of shorter flow (# number of methods)	2	1	2
Size of longer flow (# number of methods)	7	31	12
Mean	4,76	11,91	7,01
Median	5	11	7
Total Number of flows	111	1522	164

4.1. Inspecting the intermediate elements of a flow

During such in-depth analysis, the exception flows were fully inspected and were sub classified according to the following criteria: the presence of variable or core features affecting the intermediate elements on the flow. Table 5 presents the kinds of flows found according to such criteria.

Such in-depth analysis is worth doing because it may reveal new ways of interaction between features which can lead to faulty exception handling behavior, as follows: although the intermediate element of a flow does not catch the exception, during a maintenance task a general handler (e.g. catch `Throwable` clause) can be added to it, and it can mistakenly catch the exception that was flowing through it – leading to the *Unintended Handler Action* (Miller and Tripathi 1997). Next we present examples for some of such sub-categories of flows, and Section 4.2 discusses about their causes and consequences.

CC flows and their intermediate elements

In the first phase of our study (Section 3), we observed that the CC flows was the kind of exception flow with the highest frequency in the analyzed product lines (see CC flows in Table 2, 43% in MM and 46% in Prevayler). However, the in-depth analysis of CC flows revealed that most of such flows was actually affected by a variable feature. Such flows were

Table 5 Amount of flow subtypes in SPLs

Flow Type	Flow Subtype	Description	Occurrences MM	(Percentage) Pvl
CC	Pure CC	All the intermediate elements pertain to the core.	6 (5%)	22 (25%)
	C[V]C	At least one intermediate element pertains to a variable feature.	42 (38%)	54 (61%)
VV	Pure VV	All the intermediate elements pertain to the same variable feature.	6 (5%)	0 (0%)
	V[C]V	At least one intermediate element pertains to the core.	2 (2%)	0 (0%)
CV	Pure CV	All intermediate elements pertain to the variable feature that handles the exception, or to the core that signals it.	20 (18%)	6 (7%)
	C[Va]V	At least one of the intermediate elements pertains to a variable feature different from the one that handles it.	8 (7%)	0 (0%)
VC	Pure VC	All intermediate elements pertain to the variable feature that signals the exception, or to the core that handles	8 (7%)	6 (7%)
	V[Va]C	At least one of the intermediate elements pertains to a variable feature different from the one that signals it.	10 (9%)	0 (0%)
VaVb	Pure VaVb	All intermediate elements pertain to the variable feature that signals the exception, or to the one that handles it.	2 (2%)	0 (0%)
	Va[C]Vb	At least one of the intermediate elements pertains to the core.	2 (2%)	0 (0%)
	Va[Vc]Vb	At least one of the intermediate elements pertains to a variable feature different from the ones that signals or handles it.	3 (3%)	0 (0%)
	Va[VcC]Vb	At least one of the intermediate elements pertains to the core and other one pertains to a variable feature different from the ones that signals or handles it.	2 (2%)	0 (0%)
Total			111 (100%)	88 (100%)

classified as C[V]C. In this kind of flow, a core element signals an exception that propagates through one or more methods of a variable feature, and it is caught by a core element. We can observe in Table 5 that C[V]C flows represent 87% of the CC flows in MM and 71% in Pvl. Therefore, only a few flows were indeed unaffected by a variable feature, specifically 5% and 25% of all exception flows in MM and Pvl, respectively.

Figure 13 shows an example of a kind of C[V]C flow in Prevayler SPL. We can observe that the initial and final method in exception propagation is related to the SPL core, although there is an intermediate method related to variable feature (i.e. Snapshot).

VV flows and their intermediate elements

Another flow type found in the first phase of our study was the VV flow, 7% in MM and 0% in Prevayler. It gives us the impression that the whole exception flow – from signaler to handler – is composed by methods of a single variable feature. However, the in-depth analysis of the intermediate elements of VV flows revealed that in 25% of them in MM, one or more of the intermediate elements pertained to the core. Such flows were classified as V[C]V, and they represent a scenario on which a variable feature element signals an exception that propagates through a method from the core, and it was finally caught by a method from the same variable feature that had signaled it.

CV flows and their intermediate elements

The first phase of our study also revealed that the CV flows corresponded to 25% in MM and 4% in Prevayler. In MM and Pvl SPLs, our in-depth analysis of CV flows revealed that in 24% of them, one of the intermediate elements related to a variable feature is different from the one that handled it. Figure 14 shows an example of CVaV

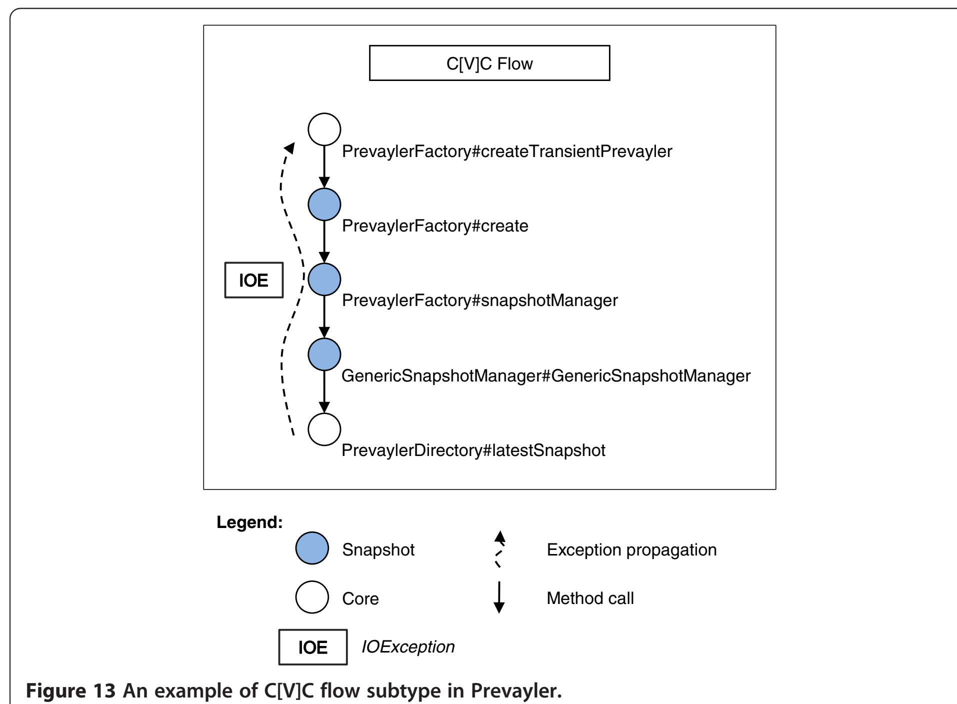

Figure 13 **An example of C[V]C flow subtype in Prevayler.**

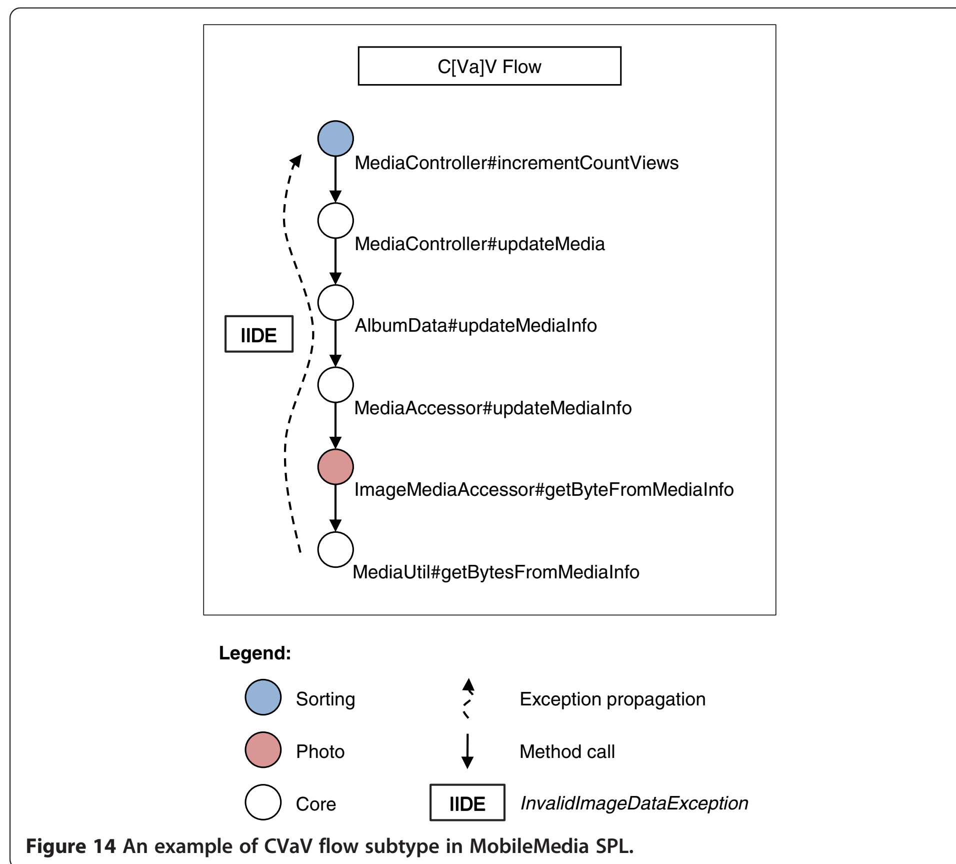

Figure 14 An example of CVaV flow subtype in MobileMedia SPL.

flow in MobileMedia. *Sorting* feature is responsible for sort media using amount views criteria, and for this functionality this code piece requires core information about loaded media. After that, a method of other variable feature is called (i.e. Photo), and, finally, this reuses a core method. Therefore, a signalized core exception propagates by code pieces of different variable features.

VC flows and their intermediate elements

The results of the first study phase also showed that the VC flows corresponded to 17% in MM and 4% in Prevayler. In MM and Pvl, our in-depth analysis revealed that in 42% of these VC flows, similarly to the previous scenario, one of the intermediate elements related to a variable feature is different from the one that signaled the exception. Such flows were classified as V[Va]C.

VaVb flows and its intermediate elements

The VaVb flows represented only a few ones of the analyzed as we observed in our first study phase, specifically 8% in MM and 0% in Prevayler. Our in-depth analysis of VaVb flows revealed others interesting scenarios, which have the following patterns: (1) Va[C] Vb; (2) Va[Vc]Vb; and (3) Va[VcC]Vb. Figure 15 shows an example of call graph in MobileMedia. In this example, three variable features interact by exception flow. Neither initial, final or intermediate methods are related to SPL core, because of that circumstances this flow subtype was classified like Va[Vc]Vb.

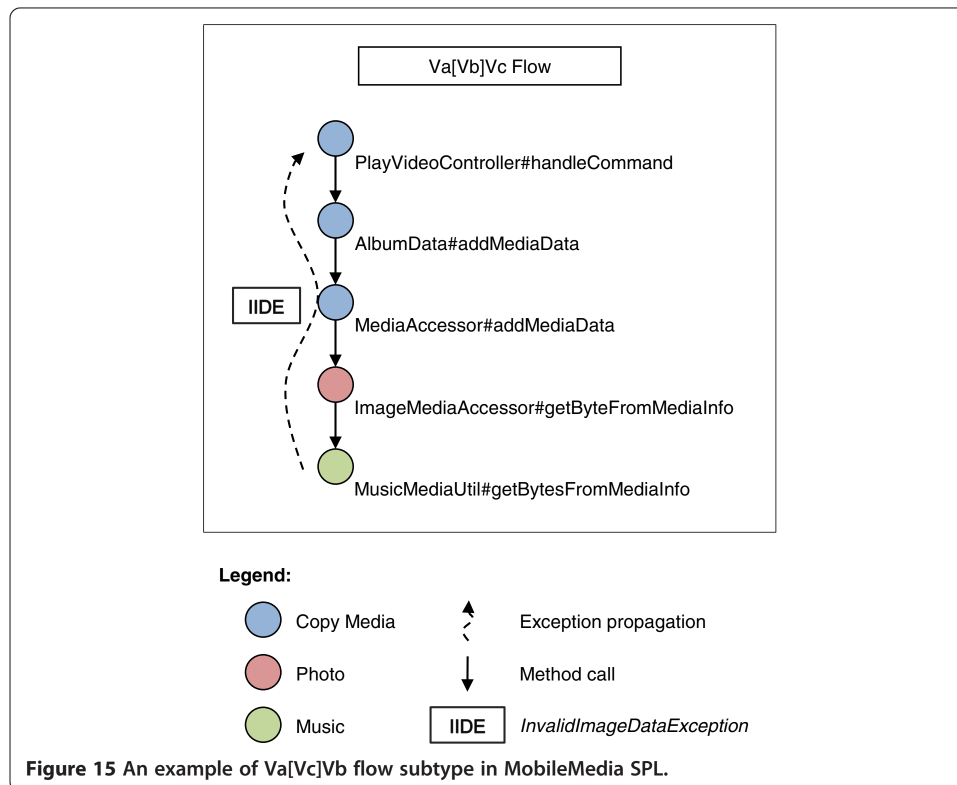

Figure 15 An example of Va[Vc]Vb flow subtype in MobileMedia SPL.

4.2 Causes and consequences of exception handling flow types

When manually inspecting the code, we observed that flows such as C[V]C, V[C]V and C[Va]V were caused by common design strategies adopted by both SPLs: (i) variabilities were added to the SPLs by extending their core classes that implement variation points; and (ii) methods belonging to the core were reused by the classes that implementing some of their variable features.

The first strategy is a well-known design technique to implement framework-based SPLs. In this technique, the core elements are responsible to refer to abstract classes or interfaces that must be implemented by the variable features. Hence, when a concrete class that implements a variable feature extends or implements a framework extension point, all handlers defined to the exceptions signaled by the parent method can be reused. On the other hand using an annotative approach such as CIDE, the code of variable features can be introduced in the middle of methods belonging to the core features. In both cases the exceptions signaled by the new piece of code that are related to an implementation of the variable feature should obey the *Exception Conformance* principle (Miller and Tripathi 1997).

According to the *Exception Conformance* principle when a method inherits or redefines another method this cannot signal a checked exception that is not a subtype of the exceptions defined in the parent method. However, sometimes a new functionality (added due to an inclusion of a new variable feature) may need to throw new exceptions, which are not subtypes of exceptions of the extended method. In order to manage these exceptions, the methods defined in parent class usually use sufficiently generalized exception types, so that any possible new exception that may be signaled by an extension method would be a subtype. However, the generalized exception types

may be so general that they have limited value in defining a clear and consistent *exception interface*. The intended benefit for the *Exception Conformance* is that the handlers defined to the parent method can be reused by the methods that will redefine it.

In our study, we observed that although some core methods were overriding by variable features (or modified through the use of annotative mechanism), the errors (exception information) were too specific, thereby the reuse of the same handler of the parent method cannot always be the adequately strategy. We should say that overriding of functionality is acceptable and often desirable, but the overriding of exception information is usually not [a]. We also noticed that the exception handlers defined in the core classes were reused by many flows that were thrown by the code that implements the variable features. Flows such as VC and C[V]C are examples of such reuse, where such problem may happen. This problem could be minimized if the SPL developers adapt the handler code as soon as the code of the variable feature was added. However, the study results have shown that developers usually ignore the way the exceptions signaled by variable features are or not adequately caught (see Section 4.3).

4.3 Uncaught exception analysis

Our study also investigated the number of exceptions that remained uncaught on each one of the investigated SPLs. This analysis is important because uncaught exceptions abort the program's execution, which is one of the main causes of software crashes. Moreover, the number of uncaught exceptions is one way of checking whether the exception handling policy is adequately implemented. The exception handling policy states that when an exception is signaled inside the system, a handler should be defined to deal with it. When this is not the case, the exception flows through the system and may remain uncaught or may be mistakenly handled by any method in the call chain of the exception flows. Another indicative that the exception might be inadequately caught is the number of exceptions that are handled on a general catch clause. This general clause is usually located on the entry point of the system (i.e. main method), which only exists to avoid the exception to remain uncaught. Usually such exceptions handled on main are only logged and abort the program's execution.

Table 6 illustrates the number of uncaught exceptions and exceptions that are only caught on the system entry point. We can observe that considering MM, no exceptions remained uncaught or were caught by a general handler on the entry point. On the other hand, a significant number of flows in the other SPLs represent exceptions that remained uncaught – 46% of all flows in Prevayler (see Figure 16). The number of exceptions that were caught on the main method was also high in this SPL – 24% in Prevayler.

When considering both types of flows as indicators that the exception handling policy might not have been obeyed, we can see that in BerkeleyDb and Prevayler, 68% and

Table 6 Number of unhandled exceptions and exceptions handled on entry point

	Occurrences		
	MM	**BkDb**	**Pvl**
Unhandled	0	597	76
Handled on entry point	0	440	40
Total flows	111	1522	164

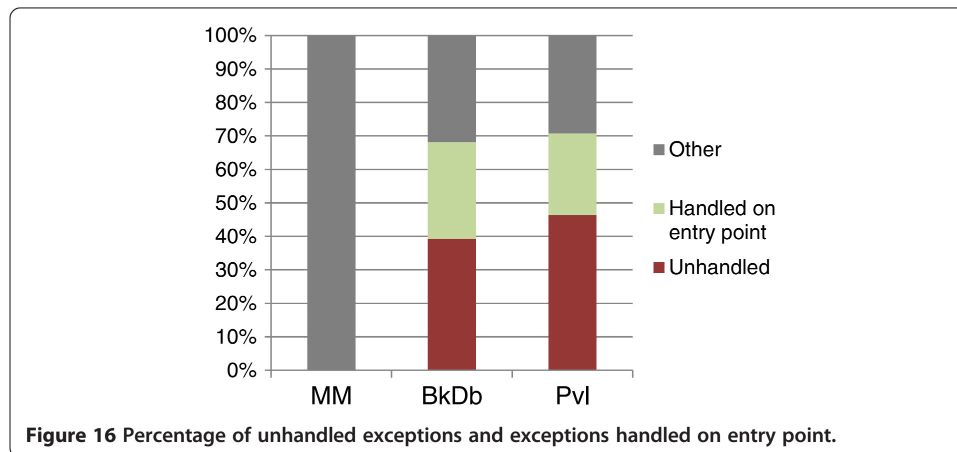

Figure 16 Percentage of unhandled exceptions and exceptions handled on entry point.

70% of the flows, respectively, escaped or was handled inside the main method (see Figure 16). Such impressive numbers raise an alert to the reader that the exception handling policies of such SPLs might not have received enough attention during development.

Although such numbers indicate that the exception handling policies have not been appropriately defined, we cannot ensure this information because the developers of such SPLs have not made them explicit on the SPL artifacts. On the other hand, the MobileMedia implementation has explicitly defined the exception handling as one of the primary concerns (Figueiredo et al. 2008). The quality of the exception handling policy is also reflected by the considerable number of user-defined exceptions in MM (see Table 1) and specialized handler actions (see Table 2), in contrast with BkDb and PvL on which most the exception handler actions were generic (i.e., catch `Throwable` or catch `Exception` clauses) – see Table 2.

After discovering the number of uncaught exceptions and the exceptions handled on the main method, we extended our analysis to include the elements responsible for signaling such exceptions. Table 7 shows the number of uncaught exceptions and exceptions caught in main, that are signaled by core and variable features. We observed that 41% of the BerkeleyDB exceptions signaled by core elements escaped, and 21% of such flows were handled on the entry point. In Prevayler, 43% of the exceptions signaled by core elements escaped and 25% were handled on the entry point (see Figure 17). Focusing on the exceptions signaled by variable features, our analysis showed that in BerkeleyDB 35% of such exceptions escaped and 27% were handled on the entry point. In Prevayler 70% escaped and 20% were caught on the entry point (see Figure 17). As in Prevayler most of the exceptions escaped we could find only few VC flows and neither VV nor VaVb flows.

Table 7 Signaler (core or variable) of unhandled exceptions and exception handled on entry point

	Occurrences					
	MM		BkDb		Pvl	
	Core	Variable	Core	Variable	Core	Variable
Unhandled	0	0	435	162	62	14
Handled on entry point	0	0	314	126	36	4
Total flows	76	35	1058	464	144	20

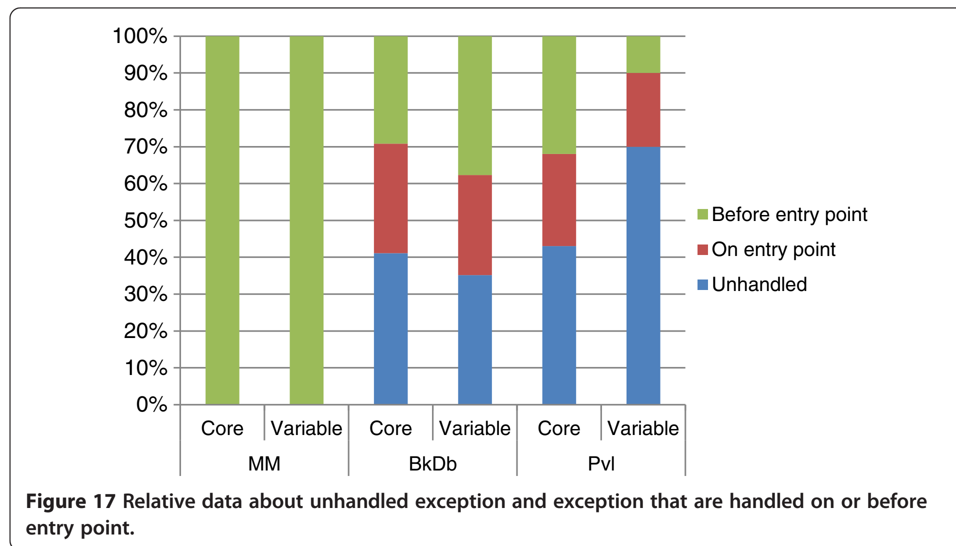

Figure 17 **Relative data about unhandled exception and exception that are handled on or before entry point.**

5. Discussions and lessons learned

This section provides further discussion of issues and lessons learned while performing this study.

Collateral effects of specific flow types

In our study we classified all exception flows found on three SPLs according to their signalers and handlers. Such characterization enabled us to identify the most common flow types and analyze how the characteristics of such flows could lead to faulty exception handling behaviors. It is known that the exception handling policy of a system or product lines depends on others factors than the intrinsic ones (i.e., software architecture). Design decisions, coding patterns or company-specific policies, and developer's experience (Shah et al. 2010) may also affect the way exceptions are signaled and handled inside the system. Hence, what is an inadequate handling for a system may be a design decision for another. However, it is also known that specific exception handling patterns may lead to faulty exception handling behavior affecting the system robustness (e.g., Unintended Handler Action, and Generic handling (Miller and Tripathi 1997)). In this study, by performing a deeper analysis of specific flow types, we could consistently detect such faulty scenarios in the analyzed SPLs, such as the ones associated to VC, VaVb, and V[C]V flows presented in Sections 3 and 4.

Dealing with feature overlapping

In our study we could find pieces of code associated to more than one feature, which is known as *feature overlapping* (Kästner et al. 2008). There are two main ways of feature overlapping: (i) *AND overlapping* – when a piece of code is annotated with feature A AND feature B, in this case both features should be selected in order to include a given piece of code; and (ii) *OR overlapping* – a piece of code is annotated with feature A OR feature B, and hence at least one of the features should be selected for the piece of code to be included in a product. In CIDE, feature interactions become apparent when colors denote different feature overlap (in this cases the colors are blended in the overlapping region). Such blended colors represent *AND overlappings*. Since all target SPLs were implemented using

the CIDE tool, only AND feature overlapping could be found. Considering the feature overlapping on code related to exception signaling or handling, we found the following: (i) in MobileMedia 10 out of 111 (about 10%) flows presented features overlapping on signaling or handling code; (ii) in Berkeley DB 12 out of 1522 (less than 1%) flows presented features overlapping on signaling or handling code; and (iii) none of the flows in Prevayler presented feature overlapping on signaling or handling code. We adopted the following strategy for classifying these specific scenarios: (i) if the signaler was annotated with feature A, and the handler was annotated with feature A AND feature B, we classified such flow as VV; (ii) if the signaler was annotated with feature A and feature B, and the handler was annotated with feature A and feature B, we classified such flow as VV; (iii) if the signaler was annotated with feature A and feature B, and the handler was annotated with feature B, we classified such flow as VaVb; and finally (iv) if the signaler was annotated with feature A, and the handler annotated with feature B and feature C, we also classified such flow as VaVb. In doing so, we could prevent false positives and false negatives on the feature overlapping scenarios found in our study.

Exception handling guidelines for software product lines

The outcomes of our study also emphasizes the need for the definition of EH guidelines for SPLs. Such guidelines could motivate, for instance, application engineers to avoid throwing exceptions from their variable feature implementations to the core assets. In practice, however, there are several technical and organizational factors that impairs the full adoption of such kind of guidelines, such as: (i) the runtime exceptions thrown from by third-party libraries used by the variable code assets; (ii) the difficulty in coordinating the work of product line and application engineers; and (iii) the natural complexity of the dependencies between SPL common and variable code assets. Given such restrictions to the full adoption of an EH guideline, feature-oriented exception flow analysis tools, such as PLEA, would be strongly useful to detect violations of the practices defined on EH guidelines. In addition, the variability implementation techniques must also be considered when defining exception handling guidelines for a SPL.

The implicit feature interaction and its consequences

In our work, we found an implicit relation that arises between features in the exceptional flow (when a feature handles the exception signaled by other feature), and we observed that it could lead to the exception handling confusion problem already mentioned in the context of aspect-oriented development (Figueroa 2011) – the exception intended to be handled by a given component is mistakenly handled somewhere else; because using the current exception handling mechanism embedded in languages such as Java, we cannot prevent one exception from being caught by a general handler on a method in the call chain between the signaling and handling points. The in-depth analysis of the exception flows (phase 2 of the study – Section 3) also showed us that feature interaction can be even more complex and involve many different common and variable features.

Study limitations

One may argue that performing the characterization in a sample of three different SPLs is a limiting factor. Even under such restriction, the study analyzed 47 KLOC of Java

source code of which around 4,1 KLOC are dedicated to EH handling. From these base code, 1797 exception flows were found, categorized and analyzed. Another limitation of this study is the fact that it only considered the exceptions explicitly thrown by SPL code assets - excluding exceptions signaled from libraries and Java runtime environment. There are also the limitations inherent to the use of a static analysis tool (i.e., inheritance, polymorphism and virtual calls) (Robillard and Murphy 2003), however the limitations of this study are similar to the ones imposed on the other empirical studies with similar goals (Figueiredo et al. 2008) (Coelho et al. 2008, 2011) (Ferrari et al. (2010). Moreover, one may also argue that during the second phase of our study the exception flows of BerkeleyDB were not analyzed, bringing another limitation to the study. However, the different kinds of flows found represent interesting scenarios that may happen in the exception handling of real SPLs.

6. Related work

This section presents related work organized in three categories: (i) empirical studies investigating the exception handling code of SPLs; (ii) studies on implicit feature interactions; and (iii) exception flow analysis tools and methods.

Empirical studies investigating the exception handling code of SPLs

Figueiredo et al. (2008) present an empirical study that aims to compare AO and OO Java implementations of the MobileMedia SPL. In their study, they have analyzed the stability of the EH feature across the SPL evolution in terms of modularization metrics (on the EH source code). In our study we discovered the exception flows originated from the EH code (manually and automatically through PLEA) and evaluated how such flows differ across different MM releases. Coelho et al. (2008) performed an empirical study considering the fault-proneness of aspect-oriented implementations for handling exceptions. Two releases of both Java and AspectJ implementation of MobileMedia product line were assessed as part of that study. Although the study has analyzed the EH code of MM product line, it neither performed a feature-oriented analysis of the EH exception flows, nor discussed the fault-proneness of specific flow types related to variable features, as we have investigated in our work. Bertoncello et al. (2008) propose a method for refactoring OO product line architecture in order to separate their normal and exceptional behavior into different software components. The proposed method motivates the introduction of variations points in the SPL core architecture to address different choices of exception handlers during product derivation. Our approach can be seen as complementary to the refactoring method proposed. First, the static analysis tool proposed in our work can be used to detect violations in the EH strategies established when evolving a SPL implementation with the introduction of new features or modification of existing ones. Second, our exploratory study also emphasizes the need to establish effective EH strategies to address the design and implementation of the core and variable features.

Studies on implicit feature interactions

Recent research work shows the importance and difficulty to analyze features dependencies in the context of SPL implementations using conditional compilation

techniques, or similar approaches like CIDE (Kästner et al. 2008). Ribeiro et al. (2010) propose the concept of emergent interface in order to address the analysis of feature dependencies when evolving a software product line. An emergent interface is used to capture the dependencies between code assets previously annotated and associated to specific features. The Emergo tool (Ribeiro et al. 2012) is used to automatically compute the emergent interfaces on demand based on intraprocedural or interprocedural dataflow analysis. Brabrand et al. (2012) propose and compare three different intraprocedural data analysis to detect the feature dependencies, in terms of undeclared variables, unused variables and null pointer. One of the great benefits of their approach is the capacity to analyze the feature dependencies for the complete SPL implementation instead of analyzing the code assets for each individual product separately. Similar to these works, our work improves the code assets dependency analysis using information about the kind of features (common or variable) they implement. However, such works address neither the analysis of exception flows in the context of SPL implementations, nor the implicit feature relation that comes about in the exceptional control flow.

Exception flow analysis tools and methods

Some research works propose solutions based on static analysis to calculate the exception flows of a system (Fu and Ryder 2007) (Robillard and Murphy 2003) (Coelho et al. 2011) (Garcia and Cacho 2011). None of these tools however perform a feature-sensitive analysis as the one proposed in this study. The tool presented in this study performs a feature-oriented analysis of the exception flows, which allows a more accurate and detailed analysis of how exceptions flow through the code assets implementing the mandatory and variable features of a SPL.

7. Conclusions

This paper reported an empirical study that characterized and quantitatively assessed the ways exceptions flow through SPL features. Moreover, it also investigated fault-proneness of specific exception flow types in SPL implementations. The analysis was conducted in three existing SPLs – Mobile Media, Berkeley DB, and Prevayler – using manual inspection and static code analysis. As part of our study, we also developed a tool that performs an automated feature-oriented exception flow analysis. The tool can be useful to support the developer when implementing the exception handling behavior of variable and common features.

Overall this new study corresponds to 47KLOC of Java source code of which around 4,1KLOC are dedicated to exception handling. From such code base, 1797 exception flows were found and characterized. Some outcomes consistently detected through this study refined the findings of our previous work and also pin-pointed new interesting information about the ways exceptions flow on SPLs, as follows:

- Only few exception flows of each product line were unaffected by their variable features. We call such flows *pure CC* flows. In relation to the other flows they corresponded to 5% of EH flows in MobileMedia, and 25% in Prevayler;
- Most of the flows found on this study were somehow affected by a variable feature: 95% in MobileMedia and 75% in Prevayler. They represent: (i) flows on which exceptions are signaled or handled by a variable feature; or (ii) flows on which the

variable feature did not signal nor handle the exception but affected the method call chain on which the exception flowed.

- Considering the flows that were signaled by variable features, most of them were handled by core elements (51% in MobileMedia, 39% in Berkeley DB, and 30% in Prevayler), and some by the same variable feature that signaled it (7% in MobileMedia, 1% in Berkeley DB, and 0% in Prevayler). The study also identified flows on which a different variable feature caught the exception thrown by another variable feature (26% in MobileMedia, 22% in Berkeley DB and 0% in Prevayler).
- Moreover, a significant number of uncaught exceptions were found in two of the investigated SPLs (39% in Berkeley DB, 46% Prevayler). In general, we observed a high prevalence of uncaught exceptions and exceptions caught by the program entry point (i.e. main method). Specially in Prevayler, most of the exceptions signaled by variable features remained uncaught.
- Finally, we could also detect that some of the flows originated from variable features were caught by a different variable feature (8% in MobileMedia, and 7% in Berkeley DB).

We believe that these and the other study outcomes presented in the paper are helpful in several ways, such as: (i) enhancing the general understanding of how exceptions flow through mandatory and variable features; (ii) providing information about the potential problems related to specific kinds of flows detected in this study (for instance the VaVb flows); and (iii) presenting how a feature-oriented static analysis tool can be used to support the identification of potentially faulty exception handling flows in the context of software product lines.

Endnote

[a] The unchecked exceptions may bypass this principle, but in our study we only focused on checked exceptions.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
HFM implemented PLEA tool and carried out most of the manual inspections conducted during the empirical study, and contributed with several findings found during manual inspections and during PLEA execution. RC worked on the study design, defined the research questions and hypothesis for conducting the study, as well as analyzed the data generated by PLEA tool and devised some of its discussions. UK also worked on the study design and analysis of data; he was also responsible for comparing the present work with related works. DT worked on the manual inspection step of Prevayler product line, and also contributed on the paper discussions. All authors were responsible for writing the paper and all of them read and approved the submitted manuscript.

Authors' information
Hugo Faria Melo holds a MSc from the Department of Informatics and Applied Mathematics (DIMAp) of the Federal University of Rio Grande do Norte, Brazil (2010–2012). He conducted empirical studies in the context of reliability of the product lines. His research interests include static analysis, model driven development and exception handling.
Roberta Coelho is an Associate Professor at the Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte (UFRN), Brazil. She holds a PhD (2008) from the Informatics Department of the Pontifical Catholic University of Rio (PUC-Rio) and worked as a researcher at Lancaster University, where she conducted empirical studies in the context of reliability of OO and AO applications. Her research interests include static analysis, exception handling, dependability and empirical software engineering.
Uirá Kulesza is an Associate Professor at the Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte (UFRN), Brazil. He obtained his PhD in Computer Science at PUC-Rio – Brazil (2007), in cooperation with University of Waterloo (Canada) and Lancaster University (UK). His main research interests include: software product lines, generative development and software architecture. He has co-authored over 120 referred papers in journals, conferences, and books. He worked as a post-doc researcher member of the AMPLE project

(2007-2009) – Aspect-Oriented Model-Driven Product Line Engineering (www.ample-project.net) at New University of Lisbon, Portugal. He is currently a CNPq (Brazilian Research Council) research fellow level 2.
Demostenes Sena is an Associate Professor at Federal Institute of Education, Science and Technology of Rio Grande do Norte. He is also a PhD candidate at Federal University of Rio Grande do Norte. His main research interests are software product lines and empirical software engineering.

Acknowledgements
This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES) - CNPq under grants 573964/2008-4 and CNPQ 560256/2010-8. Roberta Coelho is also supported by CNPq under grant 484037/2010-2.

Received: 8 March 2013 Accepted: 6 August 2013
Published: 29 October 2013

References

Apel S, Von Rhein A, Wendler P, Größlinger A, Beyer D (2013) Strategies for product-line verification: case studies and experiments, Proceedings of the 35th International Conference on Software Engineering (ICSE 2013), San Francisco. IEEE Press Piscataway, NJ, USA, pp 482–491
Bertoncello I, Dias M, Brito P, Rubira C (2008) Explicit Exception Handling Variability in Component-based Product Line Architectures, Proceedings of the 4th International Workshop on Exception Handling, Atlanta. ACM, New York, NY, USA, pp 47–54, doi:10.1145/1454268.1454275
Brabrand C, Ribeiro M, Tolêdo T, Borba P (2012) Intraprocedural Dataflow Analysis for Software Product Lines, Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD 2012), Potsdam. ACM, New York, NY, USA, pp 13–24, doi:10.1145/2162049.2162052
Brunet J, Guerrero D, Figueiredo J (2009) Design Tests: An Approach to Programmatically Check your Code Against Design Rules, Proceedings of 31st International Conference on Software Engineering (ICSE 2009). New Ideas and Emerging Results, Vancouver, pp 255–258, doi:10.1109/ICSE-COMPANION.2009.5070995
Clements P, Northrop L (2001) Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA
Coelho R, Rashid A, Garcia A, Ferrari F, Cacho N, Kulesza U, Staa A, Lucena C (2008) Assessing the Impact of Aspects on Exception Flows: An Exploratory Study, Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008). Cypress, Paphos, pp 207–234, doi:10.1007/978-3-504-70592-5 10
Coelho R, Staa A, Kulesza U, Rashid A, Lucena C (2011) Unveiling and taming liabilities of aspects in the presence of exceptions: a static analysis based approach. Information Sciences 181:2700–2720
Czarnecki K, Eisenecker U (2000) Generative Programming: Methods, Tools, and Applications. ACM Press/ Addison-Wesley Publishing Co, New York, NY, USA
Eclipse IDE (2012) Eclipse., http://www.eclipse.org/. Accessed 03 April 2012
Ferrante J, Ottenstein K, Warren J (1987) The program dependence graph and its Use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS) 9:319–349
Ferrari F, Burrows R, Lemos O, Garcia A, Figueiredo E, Cacho N, Lopes F, Temudo N, Silva L, Soares S, Rashid A, Masiero P, Batista T, Maldonado J (2010) An Exploratory Study of Fault-proneness in Evolving Aspect-oriented Programs, Proceedings of the 32nd International Conference on Software Engineering, Cape Town. ACM New York, NY, USA, pp 65–74, doi:10.1145/1806799.1806813
Figueiredo E, Cacho N, Sant'Anna C, Monteiro M, Kulesza U, Garcia A, Soares S, Ferrari F, Khan S, Castor Filho C, Dantas F (2008) Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability, Proceedings of the 30th International Conference on Software Engineering. ACM Press, Leipzig, pp 261–270, doi:10.1145/1368088.1368124
Figueroa I (2011) Avoiding Confusion with Exception Handling in Aspect-oriented Programming, Proceedings of the 10th International Conference on Aspect-oriented Software Development. Porto de Galinhas, ACM New York, NY, USA, pp 81–82, doi:10.1145/1960314.1960345
Fu C, Ryder B (2007) Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications, Proceedings of the 29th International Conference on Software Engineering. ACM Press, Minneapolis, pp 230–239, doi:10.1109/ICSE.2007.35
Gamma E, Helm R, Johnson R, Vlissides J (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional
Garcia A, Rubira C, Romanovsky A, Xu J (2001) A comparative study of exception handling mechanisms for building dependable object-oriented software. Journal of Systems and Software 59:197–222
Garcia I, Cacho N (2011) eFlowMining: An Exception-Flow Analysis Tool for .NET Applications, Proceedings of the 1st Workshop on Exception Handling in Contemporary Software Systems, São José dos Campos., pp 1–8, doi:10.1109/LADCW.2011.18
Godil I, Jacobsen H-A (2005) Horizontal Decomposition of Prevayler, Proceedings of the Centre for Advanced Studies on Collaborative research (CASCON'05). IBM Press, Toronto, Canada, pp 83–100
Goodenough J (1975) Exception handling: issues and a proposed notation. Communications of the ACM 18:683–696
Grove D, Chambers C (2001) A framework for call graph construction algorithms. ACM Transactions on Programmming Languages and Systems (TOPLAS) 23:685–746
Kästner C, Apel S, Kuhlemann M (2008) Granularity in Software Product Lines, Proceedings of the 30th International Conference on Software Engineering, Leipzig., ACM New York, NY, USA, pp 311–320, doi:10.1145/1368088.1368131
Melo H, Coelho R, Kulesza U (2012) On a Feature-Oriented Characterization of Exception Flows in Software Product Lines, Proceedings of the 26th Brazilian Symposium on Software Engineering (SBES)., Natal, Natal, pp 121–130, doi:10.1109/SBES.2012.15

Miller R, Tripathi A (1997) Issues with Exception Handling in Object-Oriented Systems, Proceedings of the 21st European Conference on Object Oriented Programming (ECOOP 97). Springer-Verlag, Berlin, pp 85–103

Parnas D (1976) On the design and development of program families. IEEE Transactions on Software Engineering 2:1–9

Prevayler (2013) Prevayler., http://prevayler.org/. Accessed 07 June 2013

Ribeiro M, Pacheco H, Teixeira L, Borba P (2010) Emergent Feature Modularization, Proceedings of ACM Conference on Systems, Programming, Languages and Applications (OOPSLA 2010). Software for Humanity Onward! Reno, pp 17–21

Ribeiro M, Tolêdo T, Winther J, Brabrand C, Borba P (2012) Emergo: A Tool for Improving Maintainability of Preprocessor-based Product Lines, Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD 2012). Hasso-Plattner-Institut, Potsdam, Hasso-Plattner-Institut, Potsdam, March 2012., ACM New York, NY, USA, pp 23–26, doi:10.1145/2162110.2162128

Robillard M, Murphy G (2003) Static analysis to support the evolution of exception structure in object-oriented systems. ACM Transactions on Software Engineering and Methodology 12:191–221

Shah H, Gorg C, Harrold M (2010) Understanding exception handling: viewpoints of novices and experts. IEEE Transactions on Software Engineering 36:150–161

Young T (2005) Using AspectJ to Build a Software Product Line for Mobile Devices. University of British Columbia, Canada, MSc Thesis

doi:10.1186/2195-1721-1-3
Cite this article as: Melo *et al.*: In-depth characterization of exception flows in software product lines: an empirical study. *Journal of Software Engineering Research and Development* 2013 **1**:3.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com