Noname manuscript No.
(will be inserted by the editor)

Fragile Base-class Problem, Problem?

Aminata Sabané · Yann-Gaël Guéhéneuc · Venera Arnaoudova · Giuliano Antoniol

Received: date / Accepted: date

Abstract The fragile base-class problem (FBCP) has been described in the literature as a consequence of "misusing" inheritance and composition in object-oriented programming when (re)using frameworks. Many research works have focused on preventing the FBCP by proposing alternative mechanisms for reuse, but, to the best of our knowledge, there is no previous research work studying the prevalence and impact of the FBCP in real-world software systems. The goal of our work is thus twofold: (1) assess, in different systems, the prevalence of micro-architectures, called FBCS, that could lead to two aspects of the FBCP, (2) investigate the relation between the detected occurrences and the quality of the systems in terms of change and fault proneness, and (3) assess whether there exist bugs in these systems that are related to the FBCP.

We therefore perform a quantitative and a qualitative study. Quantitatively, we analyse multiple versions of seven different open-source systems that use 58 different frameworks, resulting in 301 configurations. We detect in these systems 112,263 FBCS occurrences and we analyse whether classes playing the role of subclasses in FBCS occurrences are more change and–or fault prone than other classes. Results show that classes participating in the analysed FBCS are neither more likely to change nor more likely to have faults. Qualitatively, we conduct a survey to confirm/infirm that some bugs are related to the FBCP. The survey involves 41 participants that analyse a total of 104 bugs of three open-source systems. Results indicate that none of the analysed bugs is related to the FBCP.

Thus, despite large, rigorous quantitative and qualitative studies, we must conclude that the two aspects of the FBCP that we analyse may not be as problematic in terms of change and fault-proneness as previously thought in the literature. We propose reasons why the FBCP may not be so prevalent in the analysed systems and in other systems in general.

Keywords Inheritance · Overriding · Composition · Fragile base-class · Change proneness · Fault proneness · Empirical study

> From an API point of view defining that a method can be overridden is a stronger commitment that defining that a method can be called
> —Erich Gamma, June 6, 2005.

1 Introduction

Inheritance is one of the fundamental principles of object-oriented programming, with abstraction, encapsulation, polymorphism, and typing. It provides a powerful mechanism to extend and reuse classes. However, many authors pointed out some problems with the use of inheritance. One of the problems that has been raised is the so-called fragile base-class problem (FBCP). The problem was initially described in the context of component-based systems (IBM, 1994; Williams and Kinde, 1994) although its origins have been spotted earlier (Snyder,

A. Sabané · Y.-G. Guéhéneuc · G. Antoniol
DGIGL, Ecole Polytechnique de Montréal, Montreal, Canada
E-mail: aminata.sabane@polymtl.ca

Y.-G. Guéhéneuc
E-mail: yann-gael.gueheneuc@polymtl.ca

Giuliano Antoniol
E-mail: antoniol@ieee.org

V. Arnaoudova
Washington State University , USA
E-mail: Venera.Arnaoudova@wsu.edu

1986; Taenzer et al., 1989; Kiczales and Lamping, 1992; Hürsch, 1994) without being named "FBCP".

The FBCP occurs when three conditions are met: (1) a *framework-defined base-class* is inherited by a *user-defined sub-class*, *i.e.*, a class on which the developers of the base-class have not necessary the control, (2) the sub-class *overrides* some methods of the base-class, and (3) the base-class calls one or more of these overridden methods. Then, changes to either the sub-class or the base-class could cause the instances of either classes to behave unexpectedly. Only a study of the base-class and sub-class *together* could reveal the cause of the unexpected behaviour (Bloch, 2008), thus defeating the purpose of providing base-classes in frameworks to be extended by user-defined sub-classes. Given the size and complexity of today's systems, the user-defined sub-classes need not to be external to the organisation developing the base-classes, *i.e.*, lying on the *border* between a framework and a client system: in large systems and–or over time, developers may change base-classes or sub-classes within a same system *internally* and still break the behaviour of either one unwillingly.

Mikhajlov and Sekerinski (1998) performed an authoritative theoretical study of the FBCP, in which they expressed the problem in terms of a flexibility property and showed that restricting the use of inheritance could alleviate the FBCP. However, the proposed four restrictions constrain the changes that developers can apply to base-classes and their sub-classes and are, therefore, impossible to impose in practice. To the best of our knowledge, no current existing developers' IDE seeks to impose such restrictions. Other authors proposed different strategies to achieve the same results as inheritance—code reuse and typing—while avoiding the FBCP. These strategies include, for example, changing the semantics of the dispatch in object-oriented programming languages to distinguish between "open" and "non-open" methods (Aldrich, 2004). They also include designing inheritance hierarchies to prevent the FBCP altogether, as suggested for example by Gamma et al. (1995) and Bloch (2008). Typically, these strategies suggest to favour composition over inheritance by having classes implement interfaces (or pure virtual classes) for typing and delegate to instances of classes whose code they want to reuse. The classes would then delegate any appropriate calls to the composing instances to benefit from their behaviour without risking the FBCP.

These previous works described the FBCP and proposed solutions to avoid it theoretically and practically but did not provide any evidence of the prevalence of the FBCP and of its impact on systems. Therefore, **the question remains whether solutions to the FBCP should be applied extensively.** Consequently, we pursue two lines of study to answer this question. **Quantitatively**, we investigate the prevalence and impact of two aspects of the FBCP (mutual recursion and call to an overridden method) in real-world systems. We detect these two aspects using static analyses of the code to identify specific micro-architectures called in the following "fragile base-class structures" (FBCS). We formally define the FBCS while also providing a tool to detect their occurrences in systems. We observe the prevalence of FBCS occurrences in existing systems and the relation between these occurrences and the change and fault proneness of their participating classes. **Qualitatively**, we collect from three open-source systems bugs that could be due to the FBCP and ask participants to assess these bugs and confirm/infirm whether these bugs are indeed due to the FBCP. We thus answer these research questions:

- RQ1: Do FBCS occur in real-world configurations?
- RQ2: Do FBCS impact the change- and fault-proneness of their participating sub-classes?
- RQ3: Do bugs related to FBCP exist?

To answer these research questions, we performed two quantitative studies and one qualitative study. A preliminary quantitative study allows investigating the prevalence of FBCS in multiple versions of seven different open-source systems that use 58 different frameworks, resulting in 301 configurations. Using multiple versions allows evaluating whether observations made are version dependant or not. The following quantitative study aims at evaluating the change and fault proneness of sub-classes involved in the occurrences of the FBCS. The qualitative study, designed as a survey, intends to confirm/infirm the existence of bugs related to the FBCP. The survey involves 41 participants that analysed 104 bugs from three open-source systems.

Results suggest that there are few occurrences of mutual recursion FBCS but there is a significant proportion of call to an overridden method FBCS in most of the analysed systems. Our analyses reveal that these two aspects of the FBCP may not be as problematic as previously thought in the literature regarding software quality, defined in terms of change and fault proneness of the classes involved (or not) in the occurrences of the FBCS. Although, there is a significant proportion of classes involved in FBCS occurrences, there is no evidence that those classes, in particular classes playing the role of sub-classes, are more change or fault prone than other classes. Moreover, in the sample of bugs analysed by participants, none was confirmed as being caused by the FBCP.

We believe that many reasons to these results exist and include a savvy use of inheritance by developers,

the warnings available in IDEs to help developers avoid some faults, the popularity of dependency-injection and the use of interfaces over classes for typing, and thorough testing by developers.

Section 2 presents previous works and expands the motivation for our study. Section 3 provides the necessary definitions and tooling to our studies. Section 4 describes the set up of the empirical study. Sections 5–7 present the methodology and results of the studies. Section 8 discusses our findings. Section 9 reports and discusses the threats to the validity of our studies. Section 10 concludes the paper and suggests future work.

2 The Fragile Base-class Problem

We first recall the various contexts in which the FBCP can occur in Section 2.1, discuss related theoretical works in Section 2.2, and show in Section 2.3 that no extensive empirical works bring evidence of the prevalence and impact of the FBCP.

2.1 Contexts

Many authors have discussed the FBCP in the literature. While all agree that the problem is bound to inheritance and method overriding, the contexts in which the FBCP may occur vary among authors. Mikhajlov and Sekerinski (1998) and Ghezzi and Monga (2002) put the problem in the context of the maintenance of an existing framework without being able or willing to consider the clients of the framework:

- "Classes in the foundation of an object-oriented system can be fragile. The slightest attempt to modify these foundation classes may damage the whole system [...] In a closed system, all extensions are under control of system developers, who, in principle, can analyse the effect of certain base class revisions on the entire system. Although possible in principle, in practice this becomes infeasible." (Mikhajlov and Sekerinski, 1998);
- "In general, [framework] developers are unaware of extensions developed by the users and attempting to improve the functionality of the [framework] they may produce a seemingly acceptable revision of their classes which may conflict with user extensions." (Ghezzi and Monga, 2002).

Differently, Mens (2002) discusses the FBCP in the context of software merging: he argues that the FBCP does not occur only when a base-class and its sub-classes are developed by different developers but that simple evolutions of a base-class may introduce undesired behaviour in dependently-developed sub-classes, *i.e.*, within a same framework or client system.

These preceding works frame the FBCP in the context of maintenance and evolution and, generally, recommend to change the design of the frameworks to prevent any occurrence of the FBCP. We put our work in the same general context. We consider both contexts in which (1) the maintainers of some frameworks (respectively client systems) modify a base-class (respectively a sub-class) without being able or willing to consider client systems (respectively frameworks) and in which (2) the maintainers of an entire, large system modify some base-classes (respectively sub-classes) without considering immediately the impact of this modification on the rest of their system.

2.2 Theoretical Works

Many authors have discussed the FBCP from a theoretical point of view and have proposed solutions to prevent it. We divide previous works in three categories: (1) works that proposed to solve the FBCP through documentation; (2) works that proposed to prevent the FBCP by imposing restrictions on the use of inheritance; and, (3) works that proposed new inheritance-like mechanisms that do not lend themselves to the FBCP.

2.2.1 Prevention through Usage Documentation

Mens discussed two variants of the FBCP—syntactic and semantic. The syntactic variant refers to the need to recompile the sub-class if the base-class is changed whereas the semantic variant refers to semantic inconsistencies, *i.e.*, wrong behaviour. The syntactic variant is addressed by IBM System Object Model (SOM) and Microsoft Component Object Model (COM). In SOM, upward binary compatibility is achieved by a complete encapsulation of the implementation details of SOM classes (IBM, 1994)—method dispatch tables are computed at runtime—while Microsoft COM forbids inheritance and favours delegation. Steyaert et al. (1996) addressed the semantic variants—they advocated documenting the usages of inheritance using reuse contracts.

(Ruby and Leavens, 2000) used formal specifications to ensure the safe overriding and calling of methods defined in a base-class by methods of its sub-classes. Part of the specifications (pre- and post-conditions and invariants) are automatically generated and expressed and checked using JML.

Ghezzi and Monga (2002) proposed to deal with the FBCP by documenting class features using the .NET

support for attributes. They show how to use .NET framework component metadata to document dependencies between class features and thus provide information on methods that developers can overridden without undesired side effects. A design assistant or reflective mechanisms can then help to retrieve this information and guide developers.

Parkinson and Bierman (2008) proposed a proof system to formally specify and verify the behaviour of code that uses inheritance. The system is based on a logic separation that requires for each method two specifications: a static specification and a dynamic one. The static specification describes the implementation and direct calls to methods in base-classes while the dynamic specification indicates calls that are dynamically dispatched. This proof system supports all forms of inheritance with low proof-obligation overheads and thus can help developers to avoid inconsistencies when using inheritance.

2.2.2 Prevention through Usage Restriction

As part of their authoritative work on the FBCP, in which they expressed the problem in terms of a flexibility property, Mikhajlov and Sekerinski (1998) described the FBCP as a problem that occurs in open OO systems employing inheritance as an implementation-reuse mechanism. They argued that any open system using inheritance and self-recursion presents a vulnerability to this problem. They characterised the FBCP using five aspects of the usage of inheritance: (1) unanticipated mutual recursion between a base-class and some of its sub-classes; (2) unjustified assumptions in the new revision of a base-class; (3) unjustified assumptions in the methods of a sub-class extending a base-class (*i.e.*, its modifier (Wegner and Zdonik, 1988)); (4) direct access to the state of the base-class from a sub-class; and, (5) unjustified assumption of the invariants maintained by the base-class in the sub-classes. Based on these five aspects of the FBCP, the authors proposed to impose four restrictions on inheritance to prevent the FBCP by forbidding these five aspects: (1) a base-class and its sub-classes, taken together, should not introduce cycles; (2) the methods in the revision of a base-class should not make any additional assumption regarding other methods; (3) sub-classes should consider only the behaviour of the methods defined in the base-class, even if these methods can be overridden; and, (4) a sub-class must not access the state of a base-class directly. The authors showed that the proposed restrictions alleviate the FBCP using refinement calculus.

Gamma et al. (1995) and Bloch (2008) proposed design solutions to prevent the FBCP. Typically, these solutions favour composition over inheritance to distinguish clearly between typing on the one hand and reuse on the other. They suggested that developers type classes using interfaces (or pure virtual classes) in which methods are declared but not defined and build their classes as compositions of instances of classes whose behaviours they want to reuse. A typical example is that of a Java class that wants to behave generally like a set but wants to count the numbers of items added to its instances (Bloch, 2008, p.81–86). Rather than having class `CountingSet` inherit directly from `HashSet`, these works suggest to have `CountingSet` implement the interface `Set` and be composed of an instance of `HashSet` to which it would delegate any appropriate calls.

2.2.3 Prevention through New Mechanism Definitions

As a solution to evolution problems, including the FBCP, Mezini (1997) enhanced the semantics of sub-classing with a *smart composition*. A smart composition is a composition mechanism in which the contract implied by the base class is made explicit and available at the client site. This contract that includes design and implementation properties allows developers to monitor base classes and identify modifications that may invalidate existing sub-classes. As a proof of concept, the author showed a prototypical extension in Smalltalk-80.

Biberstein et al. (2002) introduced the concept of class sealing as a mean to control sub-classing and, thus, as a solution to the FBCP. To show that their new mechanism could be integrated in mainstream OO programming languages, the authors analysed the Java runtime library, combining it with 45 client systems, and showed that more than 80% of the classes are "sealed" even if only developers should decide whether a class must be sealed.

Ozaki et al. (2003) suggested to prohibit method redefinition and proposed to prevent the FBCP by decomposing inheritance into class addition and class refinement (corresponding to method overriding).

Aldrich (2004), following Biberstein et al. (2002) and Ozaki et al. (2003), proposed to distinguish between open methods (which can be overridden) and "non-open" methods that cannot be overridden and to change the dispatch mechanism to dispatch "non-open" methods statically if called on `this` to prevent the FBCP.

Ducasse et al. (2006) proposed *traits* to overcome the limitations of single and multiple inheritances and to promote the reuse of unrelated classes. A trait is a stateless programming construct that groups a set of methods that can be reused orthogonally from inheritance. This mechanism is an improvement over in-

heritance in code reuse and thus helps solve problems related to inheritance including the FBCP.

Although they do not explicitly introduced a new mechanism, Kegel and Steimann (2008) proposed to prevent the FBCP by systematically refactoring inheritance usages into delegations, thus effectively favouring composition over inheritance. They built a tool that performs the refactoring based on a set of pre- and post-conditions and applied on several open-source systems. They highlighted that the results are to be interpreted from a technical perspective, *i.e.*, applying the refactoring does not mean a better design but prevents the FBCP.

2.2.4 Discussion

Different from these works, in the context of software maintenance during which documentation and specifications may be missing or outdated, we want to observe the prevalence of occurrences of the FBCP to warn developers and possibly focus research work towards proposing "after the fact" solutions to the problem.

We argue that restricting the usage of inheritance or imposing the usage of new mechanisms cannot be justified for most developers or cannot be imposed on developers, who cannot or do not want to choose or change the programming languages that they use and who cannot impose such restrictions on their clients, unless empirical evidence show that the prevalence of the FBCP cannot be ignored by their managers.

Delegation and dependency injection are possible mechanisms to avoid the FBCP. Developers use these mechanisms, which may be one of the reasons that explain our results, as discussed in Section 8. However, for example, no programming language imposes the use of delegation instead of inheritance for typing and, thus, developers may not systematically use it. Showing evidence of the FBCP could encourage developers to change their development habits by using delegation systematically and researchers to propose new programming languages that enforce using delegation.

2.3 Empirical Works

To the best of our knowledge, there is no strong empirical evidence regarding the prevalence of the FBCP in real-world software systems. The most closely related empirical works, by Tempero et al. (2008), studied inheritance and method overriding. They studied inheritance in 93 open-source Java systems and found that a large proportion of user-defined classes mostly inherited from other user-defined classes. Focussing on three of the studied systems, they also showed that the number of inheritance relationships remained constant over time. However, they observed that, as systems evolved, inheritance relationships connected more user-defined classes, *i.e.*, the proportion of classes inheriting from user-defined classes increased while that of classes inheriting from framework decreased. Following this study, Tempero et al. (2010) empirically studied method overriding in 100 open-source systems. They showed that, in half of the systems, at least 53% of the sub-classes used method overriding internally. The authors did not consider method defined in third-party and standard frameworks and overridden in the studied systems. We share with these two previous works our interest of method overriding and will consider method overriding within systems but also across the borders between systems and the frameworks that they use.

Another related work is the empirical study performed by Robbes et al. (2015) on the prevalence of data and operation extensions in a large number of open-source Smalltalk systems. The authors also studied the change proneness of methods involved in these extensions. They found that inheritance is prevalent in systems during software evolution and that the two kinds of extensions are equally common. They also found that methods involved in these extensions tend to be less change prone than others. We share with this study our interest for the use of inheritance and its side effects on software quality but we focus on a specific use of inheritance. Moreover, our granularity to measure the change proneness is at class level.

Few other works studied the relation between inheritance and fault proneness. Briand et al. (2000) studied the relationship between different software metrics (11 of which are related to inheritance) and fault-proneness in eight C++ implementations of a medium-sized management information system software. They showed that the higher the number of overriding methods in a class, the higher the probability of the class being fault-prone. However, they did not distinguish between inheritance from system classes and inheritance from framework classes because they showed that there is few occurrences of the latter in the studied implementations. We get inspiration from this work in our study by comparing the change- and fault-proneness of classes participating to possible occurrence of the FBCP with that of classes elsewhere in the systems. However, contrary to Briand et al. (2000), who used student implementations, we consider real systems.

Daly et al. (1996) performed a study on the effect of inheritance depth on maintainability. They studied two C++ systems and asked students and recent graduates to perform maintenance tasks. They observed that maintaining a system with a depth of inheritance

equal to three is faster than maintaining a system with no inheritance but slower than maintaining a system with five levels of inheritance. However, Harrison et al. (2000) later conducted a similar study and concluded that systems with no inheritance relationships are easier to modify and understand. They explained the contradictory results by arguing that sizes and functionalities have a greater impact on program understanding than the depth of the inheritance trees. Different from these studies, we focus on possible occurrences of the FBCP and compare classes participating in such occurrences with other classes with inheritance relationships without focussing on their depths in their inheritance trees but rather on their location in the systems wrt. the frameworks (*i.e.*, border classes vs. internal classes).

3 Definition and Detection

Table 1 provides an overview of two aspects of the FBCP that are common to previous works in the literature and that are the focus of our empirical study: mutual recursion and call to an overridden method. We choose these two aspects because they can be defined and detected through static analyses. We do not study the other aspects because:

- Behavioural conflict, as defined by Mens (2002), would require studying the behaviour of the methods in the base-class and its sub-classes, which would require either formal specifications or dynamic analyses, the first being usually unavailable in most real-world systems and the second requiring accurate and complete test suites;
- Direct access to the base-class state, as presented by Mikhajlov and Sekerinski (1998), in general, is strongly discouraged and can be refactored semi-automatically by adding getters and setters and replacing any direct access by calls to these getters and setters.
- Conflict in the method interfaces as introduced by Steyaert et al. (1996), and specifically the case where the new or modified method of the base class is not invoked by other methods in the base class, does not cause a wrong behaviour but the developer's intention can be lost. Checking intention would require studying the behaviour of the methods in the base-class and its sub-classes.
- Unimplemented methods, as described by Steyaert et al. (1996), are usually catch by the compilers in most mainstream, OO, strongly-typed programming languages. The compiler raises an error when a concrete sub-class does not implement all of the abstract methods of its base-classes.

Table 1 also illustrates the two studied aspects with two typical structures that are the subjects of our empirical study: these two typical FBCS illustrate, using UML-like models, the aspects of a mutual recursion between the methods `m()` and `n()` and of a call to an overridden method `n()` by the base-class method `m()`.

3.1 Definitions

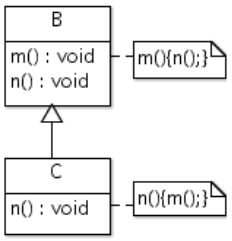
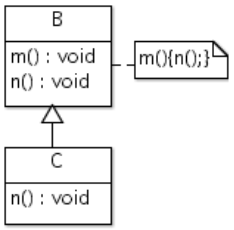
From the two aspects subject of our study, we extracted the characteristics of the concrete inheritance-related micro-architectures that can lead to the FBCP and which could be identified through static analyses of the source code. We call these structures Fragile Base Class Structures (FBCS). We define an FBCS as two classes in an inheritance relationship, not necessarily a direct relationship, and with specific method declarations and definitions. An FBCS is the location where the FBCP can occur if, for example, the sub-class overrides a method of the base-class and introduces a mutual recursion.

An FBCS is a quadruplet $\langle B, C, m, n\rangle$ where B is a base-class, C a sub-class of B, m a method or a constructor of B, and n an overriding method of C. m is the caller method and n the overriding/overridden method (n is overridden in the base-class B; it is overriding in the sub-class C (Tempero et al., 2010)). Thus, following our definition, two classes can be involved in more than one FBCS occurrences with different pairs of methods. An FBCS class is a class that appears at least in one occurrence of an FBCS, as a base-class or as a sub-class. An FBCS class can be the base-class in one FBCS occurrence and a sub-class in another. We call *direct FBCS* any FBCS occurrence in which the two classes defining the structure have a direct inheritance relation, else we say that it is *indirect*.

3.1.1 Contexts

To consider the different contexts discussed in the literature and summarised in Section 2.1, we define a *client* as a piece of code relying on some *framework* to function and a *configuration* as a pair of a client and a framework. We then can distinguish between *border FBCS* occurrences, in which the base-class is defined in the framework and the sub-class in the client, and *internal FBCS* occurrences, in which the base-class and sub-class are both defined in the same client or framework. Following this naming convention, when a sub-class inherits from a framework base-class, we refer to the former as *border sub-class* while we call *internal sub-class* any sub-class inheriting from a base-class defined in the same client or framework.

Table 1 FBCP: studied aspects, relationships with the aspects in the literature, and typical FBCS.

Aspects	Previous Works	Typical FBCS
Mutual Recursion	Ghezzi and Monga (2002): – Mutual recursion. Mikhajlov and Sekerinski (1998): – Unanticipated mutual recursion. Ruby and Leavens (2000): – Mutual recursion.	
Call to an Overridden Method	Mens (2002): – Inconsistent methods. Mikhajlov and Sekerinski (1998): – Unjustified assumption in revision class; – Unjustified assumption in modifier class; – Unjustified assumption of binding invariant in modifier. Ozaki et al. (2003): – Downcall. Steyaert et al. (1996): – Conflict in the method interfaces; – Method capture; – Inconsistent methods.	

We now present the 12 variants of the two typical FBCS shown in Table 1. We will focus on nine of them as concrete subjects of our study.

3.1.2 Mutual Recursion, Four Variants

An occurrence of typical *mutual recursion* FBCS, as shown in Figure 1(a), is a quadruplet $\langle B, C, m, n\rangle$ where B is a base-class and C is one of its sub-classes, m is declared and defined in B as invoking n, n is overridden in C but m is not, and m is invoked in n in C. We define three other variants of this FBCS, which correspond to the structures in which:

- The method n is not defined in B but inherited from one of its own super-class, as in Figure 1(b);
- The sub-class C is not a direct sub-class of B but an arbitrary number of intermediate sub-classes T exist between B and C, as illustrated in Figure 1(c);
- The class playing the role of sub-class is a child of C and does not itself override the method n but inherit the method from C, as in Figure 1(d).

The FBCS defined by $\langle B, C, m, n\rangle$ and shown in Figures 1(a) and 1(b) are direct FBCS while the one shown in Figures 1(c) and 1(d) are indirect.

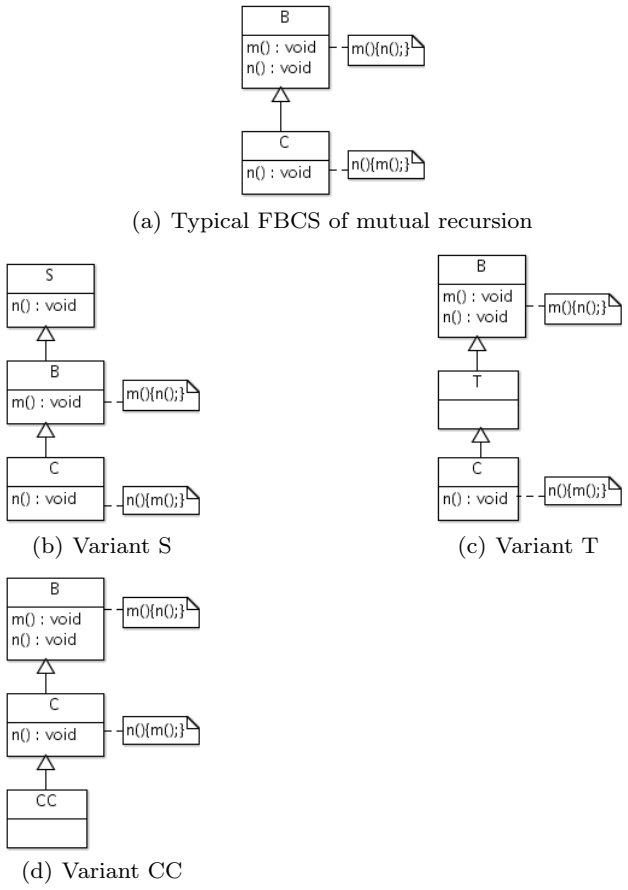
3.1.3 Call to an Overridden Method, Eight Variants

An occurrence of a typical *call to an overridden method* FBCS, as shown in Figure 2(a), is a quadruplet $\langle B, C, m, n\rangle$ where B is a base-class and C a sub-class of B, m is declared and defined in B and n is invoked from m, n is overridden in C but m is not, and m is not invoked from n in C. This FBCS has a "main" variant where the overridden method n is called from the constructor B, as depicted in Figure 2(b).

As with the *mutual recursion* FBCS, from both the typical FBCS (Figure 2(a)) and the variant with the constructor (Figure 2(b)), we derive three variants in which method n is declared, called, and overridden in different classes: Figures 2(c) and 2(f) in which method n is defined not in B but in one of the base-class of B; Figures 2(g) and 2(d) in which an arbitrary number of intermediate classes exist between B and C and Figures 2(e) and 2(h) in which a sub-class CC of C is considered in the occurrence of the FBCS.

3.1.4 Analysed FBCS Variants

The variants CC in the different FBCS aspects inherit the FBCS structure: the sub-class CC does not itself override the method n but inherits it. The FBCP can then arise when using the class CC but the problem



(a) Typical FBCS of mutual recursion

(b) Variant S

(c) Variant T

(d) Variant CC

Fig. 1 The four variants of mutual recursion.

cannot be fixed in class CC: the fix should be done in B or C. For this reason, we do not include the variant CC in our study. We will then limit our investigation to nine variants among the 12 described above.

3.2 Detection

We cannot use existing tools to detect the occurrences of FBCS that we study because none is publicly available and–or conforms strictly to our definitions. We could have adapted some existing tools but we estimated that the amount of effort was quite large because of the age of the tools and our lack of knowledge about their designs and implementations. For example, in 1999, Mezini et al. (1999) built a framework to detect incompatibly problems based on binaries, but their tool was not maintained since then.

Therefore, we built our own tool to detect the occurrences of FBCS. The tool is based on the PADL meta-model (Guéhéneuc and Antoniol, 2008), a representation of an OO system that includes all classes, methods, fields, and binary-class relationships Guéhéneuc and Albin-Amiot (2004) as well as method invocations. A PADL model contains all the information needed to perform the detection of the occurrences of the two typical FBCS and of their variants. Any meta-model that contains enough information about inheritance relations between classes and about method invocations in methods could be used for the detection. We only use PADL because it is known to us, (Guéhéneuc and Antoniol, 2008), and its models contain all the required information.

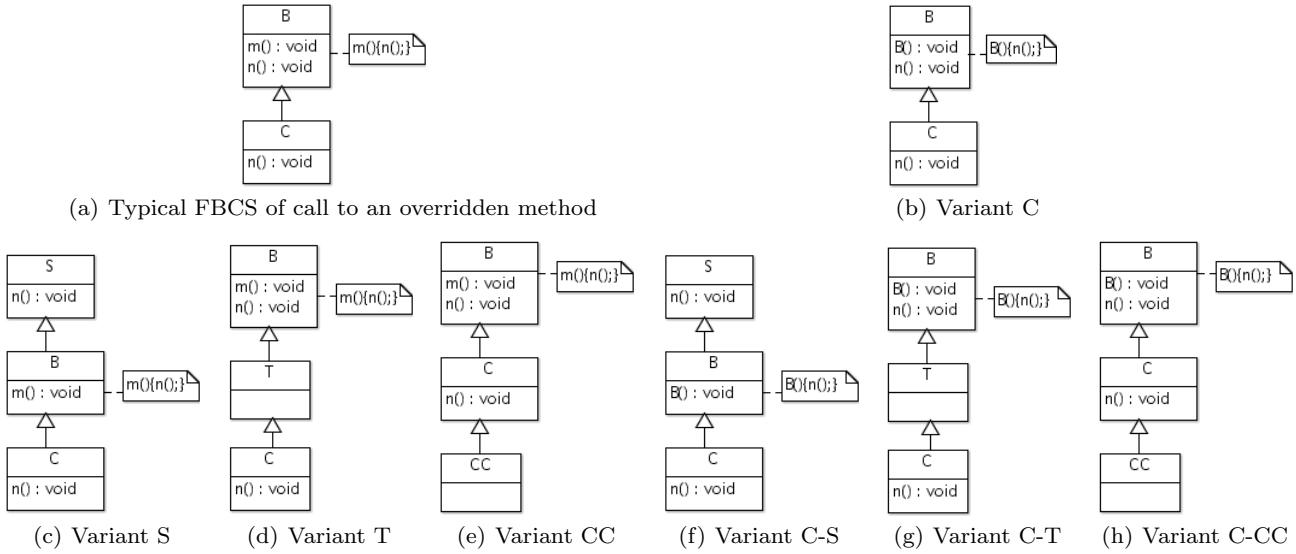
We obtain PADL models of configurations using the Ptidej tool-suite. The tool is available on-line [1]. The detection process is as follows:

- Build the PADL model of a configuration;
- Identify all super-classes;
- Identify all their sub-classes;
- For each couple of super-class and sub-class, check whether a method m of the super-class invokes another method n (declared or inherited) of the super-class and n is overridden in the sub-class;
  - If the previous condition is true, check whether the quadruplet $\langle B, C, m, n\rangle$ fulfils one of the FBCS variants.
  - When n in C invokes m, the occurrence is a mutual recursion; Otherwise, it is a call to an overridden method. In both cases, m should not be also overridden in the sub-class C.

We thus extract all classes involved in FBCS occurrences as well as all classes involved in inheritance relationships and all classes overriding at least one method of another class. We obtain the following sets:

- $S_{classes}$, the set of all classes in a configuration;
- S_{FBCS}, the set of all occurrences of the detected FBCS;
- $S_{FBCS_{border}}$, the set of all FBCS occurrences in which the base-class belongs to a framework and the sub-class to a client;
- $S_{FBCS_{border,direct}}$, the set of all FBCS occurrences with a direct inheritance relationship and included in $S_{FBCS_{border}}$;
- $S_{FBCS_{border,indirect}}$, the set of all FBCS occurrences in $S_{FBCS_{border}}$ that have an indirect inheritance relationship;
- $S_{FBCS_{internal}}$, the set of all FBCS occurrences in which both base-class and sub-class belong to the same framework or the same client;
- $S_{FBCS_{internal,direct}}$, the set of all FBCS occurrences in $S_{FBCS_{internal}}$ and having a direct inheritance relationship;
- $S_{FBCS_{internal,indirect}}$, the set of all FBCS occurrences in $S_{FBCS_{internal}}$ and having an indirect inheritance relationship;

[1] http://www.ptidej.net/download/experiments/emse15a/



(a) Typical FBCS of call to an overridden method (b) Variant C

(c) Variant S (d) Variant T (e) Variant CC (f) Variant C-S (g) Variant C-T (h) Variant C-CC

Fig. 2 The eight variants of overridden method.

- $S_{inheritance}$, the set of all pairs of classes $I = \langle C1, C2 \rangle$ in which $C2$ inherits directly from $C1$;
- $S_{overriding}$, the set of all pairs of classes $O = \langle C1, C2 \rangle$ in which at least one method of $C2$ overrides a method of $C1$.

For each set defined above but $S_{classes}$, we define a corresponding set of classes playing the role of sub-classes. Thus, the notation $SSub_X$ will be used to name the sub-classes of occurrences of the set X. For example, the set $SSub_{S_{FBCS}}$ represents the set of sub-classes in all FBCS occurrences while the set $SSub_{S_{overriding}}$ represents the set of sub-classes in $S_{overriding}$. We summarise in Table 2 the relations existing between the different sets.

4 Studies Definitions

The *context* of our quantitative and qualitative studies is the maintenance of OO systems, which combine frameworks and clients code. Our *main goal* is to infer the prevalence and impact of the FBCP through some related FBCS, (1) by detecting the occurrences of the FBCS defined in the previous section, (2) by comparing the change- and fault-proneness of the classes participating in these occurrences with respect to that of other classes, and (3) by assessing whether there exist bugs whose cause is the FBCP. The *quality focus* is the changeability and faultiness of classes as well as the prevalence of the FBCP in bugs, which both have a concrete effect on developers' effort and on the cost and time of maintenance. The *perspective* of our study is that of researchers, interested in the relationship between the FBCP, FBCS, and characteristics of classes impacting maintenance and bugs. Also, our study is of interest to developers, who perform development and maintenance activities and must target and forecast their effort, for example by focussing part of their activities on replacing inheritance by composition to remove FBCS and thus prevent FBCP. They can also be of interest to managers or quality assurance personnel, who could use the results of our study to assess the likelihood of changes and faults when extending frameworks or maintaining systems using frameworks and, thus, to better assess their quality.

4.1 Studies Sub-goals

We perform three studies with the following sub-goals that compose our main goal defined above:

- A preliminary quantitative study to investigate the prevalence of occurrences of the FBCS in real-world systems classified into three groups: clients, frameworks, and configurations that are combination of clients and frameworks. We show that occurrences of the nine variants do exist, which motivates the two following studies;
- A quantitative study to compare the change and fault proneness of classes, particularly sub-classes, in FBCS and that of other classes. The results reveals any evidence about the fault proneness of FBCS sub-classes but shows a correlation between change proneness and FBCS sub-classes in two systems out of the four analysed. However, the analysis of possible confounding factors suggests that the change proneness of FBCS sub-classes observed in these

Table 2 Relations between the Defined Sets.

The different kinds of occurrences compose the overall set of occurrences $S_{FBCS_{border,direct}} \subset S_{FBCS_{border}}$ $S_{FBCS_{border,indirect}} \subset S_{FBCS_{border}}$ $S_{FBCS_{internal,direct}} \subset S_{FBCS_{internal}}$ $S_{FBCS_{internal,indirect}} \subset S_{FBCS_{internal}}$ $S_{FBCS_{border}} \subset S_{FBCS}$, $S_{FBCS_{internal}} \subset S_{FBCS}$
The different kinds of occurrences compose the overall set of occurrences $S_{FBCS_{border,direct}} \cup S_{FBCS_{border,indirect}} = S_{FBCS_{border}}$ $S_{FBCS_{internal,direct}} \cup S_{FBCS_{internal,indirect}} = S_{FBCS_{internal}}$ $S_{FBCS_{border}} \cup S_{FBCS_{internal}} = S_{FBCS}$
An occurrence is either direct or indirect $S_{FBCS_{border,direct}} \cap S_{FBCS_{border,indirect}} = \oslash$ $S_{FBCS_{internal,direct}} \cap S_{FBCS_{internal,indirect}} = \oslash$
An occurrence is either border or internal $S_{FBCS_{border,direct}} \cap S_{FBCS_{internal,direct}} = \oslash$ $S_{FBCS_{border,indirect}} \cap S_{FBCS_{internal,indirect}} = \oslash$
The set of classes playing the role of sub-classes in FBCS occurrences is a subset of classes in which a method overrides another, which in turn is a subset of classes that extend other classes $SSub_{S_{FBCS}} \subset SSub_{S_{overriding}} \subset SSub_{S_{inheritance}}$

systems is due to the fact that these classes are also overriding sub-classes;
– A qualitative study to identify whether some bugs are due to the FBCP. The results show that none of the bugs in the sample of bugs analysed by participants is caused by the FBCP.

To perform the two first studies, we detect the nine variants of the two types of FBCS defined in the previous section in real-world Java systems. We use the change and fault data provided by Khomh et al. (2011) to assess the change- and fault-proneness of classes in these configurations. For the qualitative study, we conduct a survey in which participants must analyse bugs and assess whether these bugs are caused by the FBCP.

The follow subsections describe the systems, frameworks, and configurations of systems that we use in these three studies. The next three Sections 5–7 describe each study independently: first introducing the research questions, then the analysis procedures and methods, then the results, and finally some conclusions and discussions on each sub-goal. Section 8 discusses our main goal generally wrt. the results of the three studies.

4.2 Pool of Systems

We must choose frameworks and clients for internal FBCS and associate frameworks with clients into configurations border FBCS. We choose clients and frameworks for which bug tracking systems are available so that we can mine bugs and ask developers to assess whether some of these bugs are caused by the FBCP.

4.2.1 Clients

We choose seven clients that represent a wide range of application domains, maturities, and sizes. Many of those chosen systems have been used in previous studies regarding inheritance or overriding (Tempero et al., 2008, 2010). These clients are all open-source. They form a convenient sample because we did not choose them randomly from the (nonexisting) set of all possible systems but, rather, based on previous works and our own use of some of these systems (Wohlin et al., 2000, p.52).

ArgoUML is an open-source software for modeling UML diagrams. Barcode4J is a flexible generator for barcodes in Java. CheckStyle Eclipse Plugin is an implementation of the CheckStyle tool to enforce coding

Table 3 Clients.

Clients	Versions	#classes (min–max)
ArgoUML	0.10.1, 0.12, 0.14, 0.16, 0.18.1, 0.20	908–1,444
Barcode4J	1.0, 2.0, 2.1.0	104–178
CheckStyle Eclipse Plugin	4.1.0, 4.3.2, 4.4.2, 5.0.0, 5.3.0	300–368
Eclipse^{-Core}	1.0, 2.0, 2.1.1, 2.1.2, 2.1.3	7,193–14,236
JBoss Server	3.2.7, 3.2.8, 4.0.5, 4.2.2	2,366–4,708
Mylyn	2.0.0, 2.1, 2.2.0, 2.3.0, 2.3.1, 2.3.2, 3.0.0, 3.0.1, 3.0.2, 3.0.3, 3.0.4, 3.0.5, 3.1.0	1,403–2,366
Rhino	1.4R3, 1.5R1, 1.5R2, 1.5R3, 1.5R4, 1.5R5, 1.6R1, 1.6R2, 1.6R3, 1.6R4, 1.6R5, 1.6R6	100–318
Total: 7	**Ranges #versions: 3–13, #classes: 100–14,236**	

standards. Eclipse is an open-source OSGi implementation on which several plugins are built. We considered the "core" of Eclipse as a framework and all of its other plugins as clients of this framework. It is used both in open-source communities and in industry. We distinguish Eclipse core packages from other packages of the system by the prefix "org.eclipse.core" in their names. JBoss Server is an application server implementing the EJB API from J2EE. Mylyn is an Eclipse plugin that records developers' tasks. Rhino is an open-source Java implementation of a JavaScript interpreter.

4.2.2 Frameworks

The seven clients use 58 different third-party frameworks. We consider the Java runtime libraries (`rt.jar`) as a framework because Tempero et al. (2008) showed that developers treat standard libraries and third-party libraries alike from the point of view of inheritance. Our choice of the frameworks as well as their versions is guided by the dependencies with the seven chosen clients. The chosen frameworks thus also form a convenient sample and some of them are quite old but still required by clients, like antlr 2.7.6.

4.2.3 Configurations

In total, we analyse 48 clients (seven different clients in multiple versions), 158 frameworks (58 different frameworks in multiple versions), and 301 client–framework configurations. We choose to analyse different versions of the clients and frameworks to evaluate whether our observations are version-dependent or could be generalised to multiple versions.

We treat each configuration independently and as unique regardless whether two configurations include the same client or frameworks in different versions. Thus, we do not seek to study the evolution of occurrences of FBCS across versions.

Table 4 Frameworks.

Frameworks	Versions	#classes (min–max)
activation	1.0.1, 1.0.2, 1.1, 1.1.0	29–38
ant	1.7.0	1,151
antlr	1.2.2, 1.4.1 (2003), 2004, 2005, 2.7.2, 2.7.6, 2.7.6brew	147–1,515
argouml-model	2005	41
avalon	4.1.5brew, 4.1.5, 4.2.0	69–71
bcel	5.1, 5.1brew	373
bsf	2.3.0, 2.3.0brew	140–145
bsh	1.3.0, 1.3.0brew	133–135
cglib	2.1.3brew, 2.1.3nodep	258
checkstyle-core	4.2, 4.3, 4.4, 5.0, 5.3	326–360
com-sun-syndication	0.9.0	120
commons-cli	1.0	20
commons-codec	1.3	25
commons-collections	3.1-brew	446
commons-discovery	0.2.0	58
commons-httpclient	3.0.1	148
commons-io	1.2	44
commons-lang	2.1, 2.3	110–124
commons-logging	1.0.2, 1.0.3 (2004), 2005, 1.0.4, 1.0.4.1, 1.0.5, 5.0.28	16–21
dom4j	1.6.1	190
el-servlet-api	2.0.1	67
gef	0.9.5, 0.9.6 (2002), 2003, 0.10.14, 0.10.4, 0.11.2	264–321
hibernate-ejb3	3.2.1	162
hibernate3	3.2.0, 3.2.4	1,303–1,343
hsqldb	1.8.0.8brew, 1.8.0RC3, 1.8.0.2, 1.8.0.2jdk13	286–305
i18n	1.4.1.02b06, 1.4.2.02b03 (2004), 2005	32–36
itext	1.3, 2.0.1	567–683
jacorb	2.2, 2.2.2, 2.3.0	4,533–5,026
javassist	3.3.0GA, 3.6.0GA	254–305
javax-xml-rpc	1.1.0	54
javax-xml-soap	1.2.0	27
jaxen	v1.1beta9	221
jcert-jnet-jsse	1.0.3	400
jcommon	1.0.0, 1.0.9, 1.0.16	203–209
jdom	1.0, 1.0.0, 1.0b8	53–75
jfreechart-swt	1.0.5	34
jfreechart	1.0.1, 1.0.13, 1.0.5	482–611
jgroups	2.2.7SP1	835
jmi	1.0 (2003), 2004, 2005	25
joesnmp	0.3.4, 0.3.4brew	56
jpl-pattern-util	1.0	26
jsp-api	2.0.1GA	60
log4j	1.1.3, 1.2.8, 1.2.6 (2002), 2003, 2004, 2005	101–244
mail	1.3, 1.3.1, 1.4, 1.4.0 (2008)	209–250
nsuml	0.4.19 (2001), 2003, 0.4.20 (2004), 200501, 200504	249–250
ocl-argo	1.1, 2001, 2003, 2004, 2005	518
org.eclipse.core	1.0, 2.0, 2.1.1, 2.1.2, 2.1.3	388–449
org-apache-axis	1.4.0 (200806), 200807	785
quartz	1.5.2brew	149
rt	1.2.1, 1.3, 1.3.1, 1.4.0, 1.4.2, 1.4.2u4, 1.5, 1.5u4, 1.5u6, 1.5u8, 1.5u12, 1.6, 1.6u4, 1.6u7, 1.6u14, 1.6u22	4,250–17,058
serializer	2.7.0	89
servlet	2.2, 2.4.0	42
swidgets	0.1.1 (20050422), 20050523	42
toolbar	2004, 1.1.1 (2005), 2005	19–36
wsdl4j	1.5.1, 1.6.2	119–143
xalan	2.5.2, 2.6.0, 2.6.0j, 2.7.0	687–1,495
xerces	2001, 1.2.3 (2003), 2004, 2.6.0, 2.6.2, 2.9.0	513–887
xml-apis	1.3.04, 1.3.1	207–303
Total: 58	**Ranges #versions: 1–16, #classes: 16–17,058**	

Studying the evolution of FBCS is future work because, in this study, we want to assess whether classes participating in FBCS are more change- or fault-prone than others. If we answer positively, then it would become interesting to study the evolution of FBCS.

4.3 Object Systems of the Quantitative Studies

For the preliminary study, we analyse all the 48 clients, the 158 frameworks, and the 301 configurations combining each client and one of its frameworks.

For the quantitative study, Khomh et al. (2011) provide data for change and fault proneness for multiple versions of ArgoUML, Eclipse JDT, Mylyn, and Rhino. Thus, we analyse only the subset of clients (or part of a client in the case of Eclipse JDT included in Eclipse^{-Core}) for which this data is available: 36 versions and 166 configurations.

4.4 Object Systems of the Qualitative Study

In the qualitative study, we conduct a survey to identify bugs related to the FBCP. For that purpose, we mine bugs repositories of three out of the the four analysed in the quantitative study namely Eclipse JDT, Mylyn, and Rhino. The survey involved 41 participants who analysed 104 bugs to assess if some are due to the FBCP.

5 Preliminary Quantitative Study

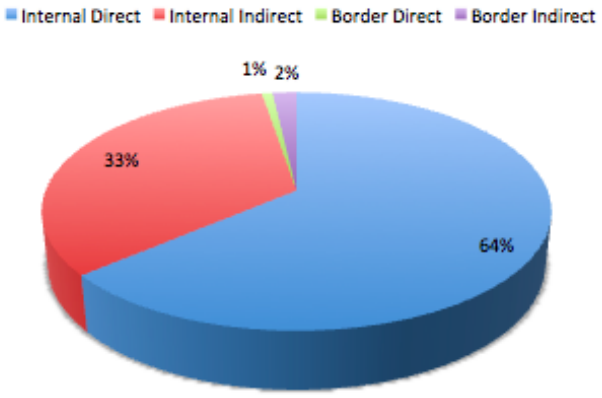
Table 5 Number of FBCS Occurrences

		Direct	Indirect	Total
Internal	Client	17,328	10,902	28,230
	Framework	54,191	26,923	81,114
Total Internal		71,519	37,825	109,344
Total Border		899	2,020	2,919
Total Internal and Border		72,418	39,845	112,263

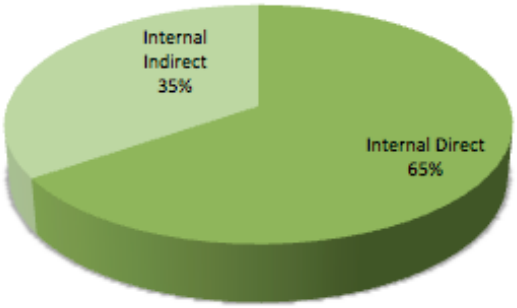
The goal of this preliminary study is to investigate the prevalence of the occurrences of the nine FBCS variants defined in Section 3 in the real-world systems presented in Section 4.

5.1 Research Question

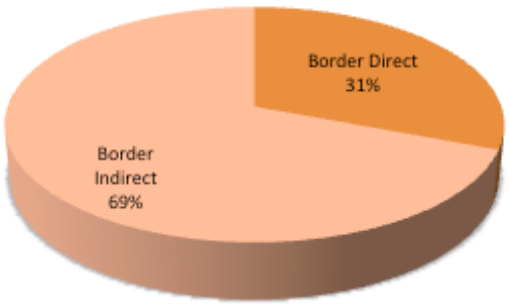
We want to answer the following research question: **RQ1: Do FBCS occur in real-world configurations?** FBCS must occur for two reasons: first, the main characteristics of FBCS are the use of inheritance



(a) FBCS



(b) Internal FBCS



(c) BoderFBCS

Fig. 3 Repartition of Kind of FBCS.

Table 6 FBCS Classes Proportions to Classes

		FBCS Classes			Direct FBCS Classes			Indirect FBCS Classes		
		min	avg	max	min	avg	max	min	avg	max
Internal	Client	0.00	13.25	32.09	0.00	11.19	27.88	0.00	4.12	12.50
	Framework	0.00	12.17	69.93	0.00	9.02	30.07	0.00	5.21	66.43
Internal		0.00	12.42	69.93	0.00	9.53	30.07	0.00	4.95	66.43
Border		0.00	0.15	7.07	0.00	0.06	1.75	0.00	0.10	6.27

Table 7 FBCS Sub-classes Proportions to Classes

		FBCS Classes			Direct FBCS Classes			Indirect FBCS Classes		
		min	avg	max	min	avg	max	min	avg	max
Internal	Client	0.00	11.04	29.33	0.00	8.99	25.36	0.00	3.28	11.09
	Framework	0.00	9.74	69.23	0.00	6.55	25.17	0.00	4.52	65.73
Internal		0.00	10.04	69.23	0.00	7.12	25.36	0.00	4.23	65.73
Border		0.00	0.11	6.00	0.00	0.03	0.88	0.00	0.09	5.47

Table 8 FBCS Sub-classes Proportions to Inheritance Sub-classes

		FBCS Classes			Direct FBCS Classes			Indirect FBCS Classes		
		min	avg	max	min	avg	max	min	avg	max
Internal	Client	0.00	22.01	45.86	0.00	18.12	39.66	0.00	6.04	16.90
	Framework	0	16.53	81.15	0	11.90	57.14	0	6.60	77.05
Internal		0.00	17.81	81.15	0	13.35	57.14	0.00	6.47	77.05
Border		0.00	0.17	7.69	0.00	0.06	1.75	0.00	0.12	7.01

Table 9 FBCS Sub-classes Proportions to Overriding Sub-classes

		FBCS Classes			Direct FBCS Classes			Indirect FBCS Classes		
		min	avg	max	min	avg	max	min	avg	max
Internal	Client	0.00	44.05	73.70	0.00	36.18	64.71	0.00	11.81	28.37
	Framework	0	35.17	100.00	0	27.13	100.00	0	11.47	87.04
Internal		0	37.24	100.00	0	29.24	100.00	0	11.55	87.04
Border		0.00	0.35	16.61	0.00	0.13	4.76	0.00	0.23	15.13

and overriding, two mechanisms that are at the core of OO programming and, second, a significant number of related works investigating the FBCP suggested the existence of FBCS. With this research question, we want to infer how prevalent is the FBCS and therefore opportunities to have the FBCP.

Therefore, to answer **RQ1**, we detect the nine FBCS variants in the seven clients, their 58 frameworks, and the 301 configurations resulting from their association. We then observe the detected occurrences to study their numbers and the proportions of classes involved in these occurrences. We distinguish between border and internal as well as direct and indirect FBCS.

Consequently, we define the following four research sub-questions.

- **RQ1.1**: What is the cardinality of $S_{FBCS_{internal+direct}}$ and its proportion to S_{FBCS}?
- **RQ1.2**: What is the cardinality of $S_{FBCS_{internal+indirect}}$ and its proportion to S_{FBCS}?
- **RQ1.3**: What is the cardinality of $S_{FBCS_{border+direct}}$ and its proportion to S_{FBCS}?
- **RQ1.4**: What is the cardinality of $S_{FBCS_{border+indirect}}$ and its proportion to S_{FBCS}?

5.2 Procedure and Method

To answer RQ1 and have a complete picture, we compute the number of the variants of FBCS and their proportion wrt. to all the detected FBCS (S_{FBCS}). We also analyse the proportion of classes involved in FBCS occurrences wrt. to all analysed classes. Finally, we analyse the proportion of sub-classes in FBCS occurrences wrt. to sub-classes in $S_{inheritance}$ and in $S_{overriding}$.

5.3 Results

5.3.1 FBCS Occurrences

We summarise the results of the FBCS detection in Figure 3 and in Table 5. Table 5 presents the numbers of different kinds of FBCS while Figure 3(a) represents their proportions to the total number of detected FBCS. Thus, we found 17,328 and 54,191 direct FBCS respectively in the clients and frameworks, resulting in a total of 71,519 internal direct FBCS. Regarding the internal indirect FBCS, we found a total of 37,825 occurrences with 10,902 in clients and 26,923 in frame-

works. Except Barcode4j 1.0, each client contains direct FBCS. The number of occurrences goes from three to 2,695. Out of the 47 clients, six clients do not contain any indirect FBCS; they are Barcode4j 1.0 and all versions of Checkstyle-Eclipse. In Checkstyle Eclipse versions, this fact can be explained by the low depth of inheritance (DIT): all classes have 1 or 2 as DIT value but three classes with a DIT of 3. Barcode4j 1.0 does not contain any FBCS occurrence. After long investigations, we could not find any reason why Barcode4j 1.0 does not contain any FBCS occurrence. Barcode4j 1.0, although small (only 88 classes), is not the smallest, the smallest being rhino that contains 83 classes and 12 FBCS occurrences. Moreover, about 40% and 21% of classes are respectively sub-classes or overriding classes. Future work will investigate this issue. The number of occurrences of indirect FBCS in clients is between one and 2,049. Direct FBCS occur in 124 frameworks out of 158 and the indirect ones in 93. Most of the frameworks that do not contain FBCS occurrences are small in size (they contain less than 50 classes) and have no or very limited use of inheritance and–or overriding mechanisms. When FBCS exist, the number of occurrences in frameworks goes from one to 1,827 for direct FBCS while it is between two and 3,614 for the indirect ones.

We detected a total of 2,919 border FBCS with 899 border direct FBCS and 2,020 border indirect FBCS. The border direct FBCS occur in 63 and the indirect ones in 31 out of 301 configurations. The number of occurrences varies from one to 68 for border direct FBCS and from one to 382 for the indirect ones. The internal and the border FBCS represent respectively 97% and 3% of the total number of FBCS occurrences as shown in Figure 3(a). The direct FBCS count for 65% of the internal FBCS while they represent only 31% of the border FBCS (Figures 3(b) and 3(c)).

5.3.2 FBCS Classes

Table 5 shows the proportions of classes involved in FBCS occurrences (FBCS classes) either as base-classes (FBCS base-classes) or sub-classes (FBCS sub-classes) in comparison to all classes. It is worthwhile to remind that a class can be part of many occurrences of direct and–or indirect FBCS. When considering only clients in which direct FBCS exist, the proportion of participating classes is, in average, close to 11% with a maximum about 28%. When indirect FBCS exist, the proportion of participating classes is about 4% and the maximum is close to 13%.

When considering only frameworks, the average proportion of classes in direct FBCS is consistent with the observations in the clients: 9% of classes, in average, are involved in direct FBCS with a maximum close to 30%. Regarding the indirect FBCS, the proportion of participating classes, in average, is lower, about 5% while the maximum is higher, about 60%. The high value of the maximum proportion is due to the versions of nsuml that act as outlier: 95 classes out of 143 participate in indirect FBCS. Indeed, when removing nsuml versions, the maximum proportion drops to 25%. nsuml stands for Novosoft UML Library for Java. It is a code generator library from UML to Java. We analysed five versions of nsuml and all have 143 classes. In those versions, there is an intensive use of inheritance and overriding: only 20 classes inherit from java.lang.Object, the DIT of 91 classes is at least equal to 4, and there are 1098 overriding methods. 108 classes play the role of sub-classes in the indirect FBCS. When looking the indirect FBCS, we observe that all are based on only two base-classes: `ru.novosoft.uml.foundation.core.MModelElementImpl` and `ru.novosoft.uml.MBaseImpl`. `ru.novosoft.uml.foundation.core.MModelElementImpl` overrides `toString()` of `java.lang.Object` and in its implementation, the method `toString()` calls the method `getUMLClassName()` that is overridden in 67 classes leading to 67 indirect FBCS. A similar scenario happens with the class `ru.novosoft.uml.MBaseImpl` that has a method `remove()` that invokes `cleanup(java.util.Collection)`, which, in turn, is overridden by 95 classes. The intensive use of inheritance and overriding mechanisms increases the occurrences of FBCS. Moreover, implementation of some "util methods" that are systematically overridden in the sub-classes of the inheritance hierarchy can explode the number of occurrences with the same pairs of methods.

When border direct FBCS exist, the proportion of participating classes is less than 2% of the total number of classes with an average less than 0.1%. When border indirect FBCS occur, the maximum proportion of FBCS classes is less than 7% with an average of 0.1%.

5.3.3 FBCS Sub-classes

As explained in Section 3, the manifestation of the FBCP is visible in sub-classes. Thus, classes playing the role of sub-classes in FBCS are of great of concern. We analysed those classes relatively to all classes (Table 5), to inheritance classes (Table 5), and to overriding classes (Table 5).

In general, the proportions of FBCS sub-classes to classes are not so far from that of FBCS classes to classes because as observe in the indirect FBCS occurrences of "nsuml", a same FBCS base-class can appear in multiple FBCS occurrences with different subclasses. With respect to all classes, the average propor-

tion of FBCS sub-classes is about 9% and the maximum about 26% for direct FBCS in clients. When considering indirect FBCS in clients, FBCS sub-classes count, in average, for 3% of classes with a maximum at 11%. Regarding frameworks, the proportion of FBCS sub-classes to all classes is, in average, about 7% in direct FBCS with a maximum at 25%. For indirect FBCS, the proportion of FBCS sub-classes to all classes is about 5% and the maximum is about 66% because of the outlier nsuml. Without the versions of nsuml, the maximum proportion is about 20%. For border FBCS, the proportion of FBCS sub-classes to all classes varies from 0 to less than 1% (with an average of 0.03%) for direct FBCS and from 0 to 6.27% (with an average of 0.1%) for indirect FBCS.

When taking as reference all sub-classes, in the clients, the average proportion of FBCS sub-classes is 18% with a maximum of 40% in direct FBCS. For indirect FBCS, the proportion of FBCS sub-classes is maximum 17% with an average of 6%. In frameworks, the proportion of FBCS sub-classes to all sub-classes varies between 0% and 58% with an average of 12% for direct FBCS. For indirect FBCS, the average proportion is about 7% and the maximum is 38% when excluding nsuml versions. The proportion of FBCS sub-classes to all sub-classes for border direct FBCS is in average less than 0.1% with a maximum at about 2%. For border indirect FBCS, the average proportion is slightly higher than 0.1% while the maximum proportion is about 7%.

When compared to overriding sub-classes, the average proportion of direct FBCS sub-classes in clients is about 36% and the maximum about 65%. For indirect FBCS, the proportion is about 12% in average with a maximum about 29%. Regarding frameworks, the proportion of FBCS sub-classes to overriding sub-classes for direct FBCS is 27% in average with a maximum at 100%. In one or more versions of three frameworks namely el-servlet-api, joesnmp, and servlet, all overriding classes participate to a direct FBCS. The versions of these frameworks contain between zero and six occurrences of direct FBCS. For indirect FBCS, the average proportion is about 12% with a maximum of about 60% when removing nsuml versions. In border direct FBCS, the average proportion of sub-classes to overriding sub-classes is 0.13% and the maximum about 5%. For border indirect FBCS, the proportion is about 0.20% in average with a maximum about 15%.

5.4 Conclusions and Discussions

We thus answer our research question as follows:

RQ1: FBCS occurrences exist in most of the systems using inheritance. Internal FBCS occur much more often than border FBCS.

We detected very few occurrences of mutual recursion be it in clients and frameworks (internal FBCS). In configurations (border FBCS), we did not detect any occurrence of mutual recursion in all configurations. In clients, three versions of Eclipse without core contain one occurrence (the same in the three versions) of indirect mutual recursion out of 2,049 indirect occurrences. 20 frameworks consisting in multiple versions of rt, Hibernate, and Checkstyle^{-Core} and one version of itext contain between one and 12 direct occurrences of mutual recursion. These occurrences represent a proportion to direct FBCS occurrences of less than 0.5%. Concerning the indirect mutual recursion occurrences, 15 frameworks including 13 versions of rt and two of hibernate contain between one and six occurrences accounting for a proportion to indirect FBCS between 0.1 and 1.16%. The rare occurrences of mutual recursion is expected and can be explained by the fact that mutual recursion occurrences lead to an infinite loop that are easily detectable when running the code. When looking to the detected occurrences, we observe that in the base class and-or in the sub-class, if statements prevent the execution to enter in an infinite loop. This shows that developers are aware when using this kind of structure. Most of occurrences of all the detected FBCS occurrences are then of type call to an overridden method from a method or a constructor.

The proportion of border FBCS to all FBCS is negligible. We observe that classes do not often inherit from third-party libraries classes, confirming the findings in (Tempero et al., 2008). When they inherit from them, they rarely override their methods, thus reducing the opportunity to give rise to a border FBCS. The low proportion of border FBCS and consequently of classes playing the role of sub-class in border FBCS reduce the opportunity for the FBCP, as defined in the literature, to occur. However, today's systems are so large that even in the same system, the conditions that make inheritance from third part libraries less-safe can be encountered. Moreover, sub-classes involved in internal FBCS represent a significant proportion to all classes: in average about 10%. The next study will then investigate to what extent those classes can impact the quality of the system in terms of change and fault proneness.

We performed a correlation test between number of occurrences and size (number of classes or LOC) and, as expected, we observed that the larger the systems,

the greater the numbers of occurrences of the FBCS. We expected this correlation because, intuitively, the larger a system, the more chance to have pairs of classes with an inheritance relationship and consequently the more chance to have FBCS to occur.

6 Quantitative Study

The previous study showed that FBCS occur and that their numbers and proportions cannot be ignored. The goal of this quantitative study is to investigate the impact of the FBCS on the class change and fault proneness. We perform this study from the researchers' point of view. We found in the literature many papers about the use of inheritance, its potential drawbacks, and the fragile base-class problem. Yet, none provided empirical evidence on the impact of the FBCP. This study is also useful to developers who should be aware whether FBCS have negative impact on their classes and, thus, decide whether to refactor them or not.

6.1 Research Question

Following the literature on the FBCP and our definition of an FBCS, we focus on the classes playing the role of sub-classes because these are the classes whose behaviour is, by definition, under the control of clients (be them external clients or internal clients from another group in a same organisation) and whose behaviour may be modified by a change to either their base-classes or themselves. Thus, we formulate our research question as follows: **RQ2: Do FBCS impact the change- and fault-proneness of their participating sub-classes?** We want to observe whether sub-classes involved in FBCS occurrences are more change-prone and–or fault-prone than other classes. We divide this question as follows:

- **RQ2.0**: Are FBCS sub-classes, *i.e.*, in $SSub_{S_{FBCS}}$, more change- or fault-prone than other classes, *i.e.*, classes in $S_{classes} \setminus SSub_{S_{FBCS}}$?
- **RQ2.1**: Are classes in $SSub_{S_{FBCS_{border}}}$ more change- or fault-prone than classes belonging to $S_{classes} \setminus SSub_{S_{FBCS}}$?
- **RQ2.2**: Are classes in $SSub_{S_{FBCS_{internal}}}$ more change- or fault-prone than sub-classes in $S_{classes} \setminus SSub_{S_{FBCS}}$?
- **RQ2.3**: Are sub-classes in internal FBCS occurrences ($SSub_{S_{FBCS_{internal}}}$) more change- or fault-prone than classes in border FBCS occurrences ($SSub_{S_{FBCS_{border}}}$)?

6.2 Procedure and Method

For all our research questions, we compute the different sets and test whether the proportion of classes exhibiting (or not) at least one change (respectively, one fault), significantly varies between classes with a specific role and another set of classes. For example, in RQ2.1, the two sets of classes are $SSub_{S_{FBCS_{border}}}$ and $S_{classes} \setminus SSub_{S_{FBCS}}$.

We use Fishers exact test (Sheskin, 2007b) to check whether the proportion varies between the two populations. The Fisher's exact test is a non-parametric statistical test designed to determine if there are non-random associations between two categorical variables. This test works by testing the independence of rows and columns in a 2×2 contingency table based on the exact sampling distribution of the observed frequencies. We also compute the odds ratio (OR) (Sheskin, 2007b) that indicates the likelihood for an event to occur. The odds ratio is defined as the ratio of the odds p of an event occurring in one sample, *i.e.*, the odds that classes participating in one population underwent a change (experimental group, *e.g.*, in **RQ2.1**, classes belonging to $SSub_{S_{FBCS_{border}}}$), to the odds q of the same event occurring in the other population, *i.e.*, the odds that classes that do not belong to the experimental group change (control group, in **RQ2.1**, classes in $S_{classes} \setminus SSub_{S_{FBCS}}$): $OR = (p/(1-p))/(q/(1-q))$. An odds ratio of 1 indicates that the event is equally likely in both samples. An OR greater than 1 indicates that the event is more likely in the first sample (*e.g.*, classes belonging to $SSub_{S_{FBCS_{border}}}$), while an OR less than 1 indicates that the event is more likely in the second sample (*e.g.*, classes in $S_{classes} \setminus SSub_{S_{FBCS}}$).

Thus, we consider the following independent and dependent variables.

6.2.1 Independent Variables

Each sub-research question **RQ2** has different independent variables, consisting of two sets:

- **RQ2.0**: $SSub_{S_{FBCS}}$ and $S_{classes} \setminus SSub_{S_{FBCS}}$;
- **RQ2.1**: $SSub_{S_{FBCS_{border}}}$ and $S_{classes} \setminus SSub_{S_{FBCS}}$;
- **RQ2.2**: $SSub_{S_{FBCS_{internal}}}$ and $S_{classes} \setminus SSub_{S_{FBCS}}$;
- **RQ2.3**: $SSub_{S_{FBCS_{internal}}}$ and $SSub_{S_{FBCS_{border}}}$;

When investigating the subset direct and indirect in $RQ2.1$ to $RQ2.3$, we consider the corresponding subsets as independent variable. For example, the independent variable for sub-classes in internal direct FBCS (cf. **RQ2.2**) is $SSub_{S_{FBCS_{internal,direct}}}$.

6.2.2 Dependant Variables

For change proneness, our dependant variable is a boolean variable indicating if a particular class has changed at least once between two successive versions. To determine whether a class has changed, we rely on the commits in their version control systems (CVS or SVN). For fault proneness, our dependant variable is a boolean variable indicating if there exists a bug involving a particular class between two successive considered versions. For further details on the computations of class change- and fault-proneness, we refer to the work by Khomh et al. (2011).

6.2.3 Objects

We analysed several versions of ArgoUML, Eclipse JDT, Mylyn, and Rhino, *i.e.*, four different clients that correspond to 36 versions. Those clients are associated with 26 frameworks resulting in 166 configurations. We choose these clients because we have independently-computed change and fault data for their classes.

6.2.4 Populations

Figure 4 describes the proportions of the different analysed populations in the present study per version and per system. In particular, the graphs report the proportions relatively to classes of FBCS sub-classes, of classes that override a method without being FBCS sub-classes, and finally of sub-classes that do not override any method. Thus, we can infer from those graphs the proportion of sub-classes as well as the proportion of overriding classes in each version. In general, the different proportions are similar across the versions within a system but different from one system to another.

ArgoUML exhibits the highest proportions of classes in each category. At least three out of four classes inherit from a class that is not java.lang.Object. The proportions of overriding classes are between 40 and 50% while the proportions of FBCS sub-classes between 24 to 32%.

In the versions of Eclipse JDT, almost half of the classes are sub-classes. The proportions of overriding classes is around 33% and that of FBCS sub-classes 19% in all versions except in Eclipse JDT 1.0 where it is about 24%.

Mylyn presents the smallest proportions of each kind. The proportion of sub-classes is about 28% except in Mylyn 3.1.0 where it is about 35%. The proportions of overriding classes are between 10% and 16% and the proportions of FBCS sub-classes do not exceed 5%.

Rhino has similar proportions than Eclipse JDT. In most of the versions, sub-classes count for about 50% of the classes. The proportions of overriding classes vary from 25% to 40%. FBCS sub-classes account for about 20% of the classes.

At least 18% of classes play the role of FBCS sub-classes in all analysed versions except in Mylyn versions where the proportions of FBCS sub-classes do not exceed 5%. However, ArgoUML versions present higher proportions of FBCS sub-classes, between 24% and 32%. Thus, in general, FBCS sub-classes represent a significant proportion of classes.

6.3 Results

We now report the results of our analyses and answers to the research questions.

Table 10 summarises the results of Fisher's exact test for the different **RQ2** and gives for each of them the number of versions where the test is significant out of the total number of versions per system. The detailed results with the p-values and the odd ratio are reported on-line[1].

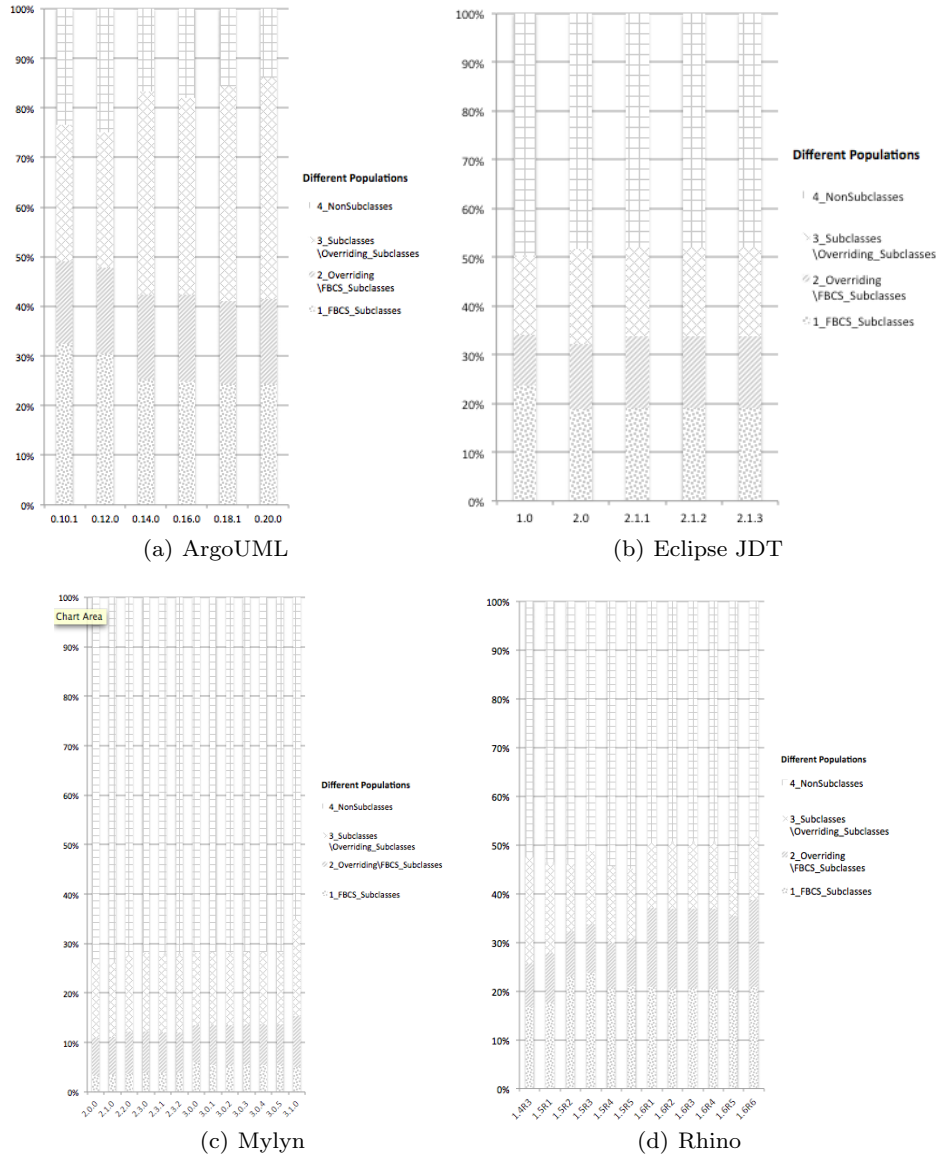
*6.3.1 **RQ2.0**: Are FBCS sub-classes, i.e., classes in $SSub_{S_{FBCS}}$, more change- or fault-prone than other classes, i.e., classes in $S_{classes} \setminus SSub_{S_{FBCS}}$?*

Regarding the change proneness of FBCS sub-classes, the results show that in 15 versions out of 36, FBCS sub-classes are more change prone than other classes. In most of the versions of ArgoUML (five out of six) and in all Eclipse JDT versions, the test is significant with an odds ratio varying from 1.93 to 5.38 but in one case, where the odds ratio is infinite. Thus, in those versions, FBCS sub-classes are at least 2 times more change prone than other classes. In contrast, in only few versions (no more than a third) of Mylyn and Rhino the test is significant: FBCS sub-classes are more change prone than the other sub-classes in three versions of Mylyn out of 13 and four versions of Rhino out of 12.

With respect to fault proneness, the test is significant in only seven out of 36 versions, *i.e.*, about 20% of the analysed versions. FBCS sub-classes are more fault prone than the other classes in three versions of ArgoUML, one version of Eclipse JDT, and three versions of Rhino. In those versions, the odd ratio varies between 0.38 and and 7.85.

*6.3.2 **RQ2.1**: Are classes in $SSub_{FBCS_{border}}$ more change- or fault-prone than classes belonging to $S_{classes} \setminus SSub_{FBCS}$?*

As shown in Table 10, compared to non FBCS sub-classes, Border FBCS sub-classes are more change prone



(a) ArgoUML (b) Eclipse JDT

(c) Mylyn (d) Rhino

Fig. 4 Repartion of Kind of FBCS.

Table 10 Results RQ2: FBCS Sub-classes vs Non FBCS Sub-classes

	ArgoUML		Eclipse JDT		Mylyn		Rhino	
	Change	Bug	Change	Bug	Change	Bug	Change	Bug
$SSub_{S_{FBCS}}$ vs $S_{classes} \setminus SSub_{S_{FBCS}}$	4/6	3/6	4/5	1/5	3/13	0/13	4/12	3/12
$SSub_{S_{FBCS_{border}}}$ vs $S_{classes} \setminus SSub_{S_{FBCS}}$	3/6	2/6	1/5	0/5	3/13	0/13	0/12	1/12
$SSub_{S_{FBCS_{internal}}}$ vs $S_{classes} \setminus SSub_{S_{FBCS}}$	4/6	3/6	4/5	1/5	4/13	0/13	7/12	3/12
$SSub_{S_{FBCS_{internal}}}$ vs $SSub_{S_{FBCS_{border}}}$	4/6	3/6	1/5	0/5	3/13	0/13	5/12	4/12

in only seven versions (three versions of ArgoUML, one version of Eclipse JDT, and three versions of Mylyn) and more fault prone in only three versions (two versions of ArgoUML and one version of Rhino).

6.3.3 ***RQ2.2****: Are classes in $SSub_{FBCS_{internal}}$ more change- or fault-prone than sub-classes in $S_{classes} \setminus SSub_{FBCS}$?*

Internal FBCS sub-classes are more change prone than non FBCS sub-classes in 19 versions representing about 53% of the analysed versions: seven versions of Rhino

and four versions in each of the three other systems. Regarding the fault proneness, the test is significant in only seven versions: three versions of ArgoUML, one version of Eclipse JDT, and three versions of Mylyn.

6.3.4 ***RQ2.3****: Are sub-classes in internal FBCS occurrences ($SSub_{FBCS_{internal}}$) more change- or fault-prone than classes in border FBCS occurrences ($SSub_{FBCS_{border}}$)?*

According with the results reported in Table 10, internal FBCS sub-classes are more change prone than border FBCS sub-classes in 13 versions out of 36: four versions of ArgoUML, one version of Eclipse JDT, three versions of Mylyn, and five versions of Rhino. Regarding the fault proneness, the test is significant in only seven versions: three versions of ArgoUML and four versions of Rhino.

In almost half of the versions, FBCS sub-classes are more change prone than other classes, particularly the internal FBCS sub-classes as suggested by the results of **RQ2.1** and **RQ2.2**. We can distinguish two groups based on the trends regarding the change proneness. In most versions of ArgoUML and Eclipse JDT, FBCS sub-classes are more change prone while in Mylyn and Rhino, there is rarely more than one third of the versions with a significant test.

Regarding fault proneness, the results show that in most versions, there is no evidence that FBCS sub-classes are more fault prone than other classes. Except in ArgoUML where the test is significant in half of the versions, in the other systems, in general, there is no more than 25% of versions with a significant test. Moreover, results suggest that internal direct FBCS sub-classes are the ones that are more fault prone than the non FBCS sub-classes.

6.4 Conclusions and Discussions

We thus answer our research question as follows:

RQ2: There is no evidence that the sub-classes in the two typical FBCS considered in this study—mutual recursion and call to an overridden method—are more fault prone than other classes. Our results suggest that sub-classes in these FBCS are more change prone in two systems out of the four analysed but the confounding factors discussed in Section 8 infirm these results: the change proneness of FBCS sub-classes in these systems may due to them being also overriding sub-classes, which are generally more change prone in these systems.

The results presented above and the ones in Section 8 suggest that FBCS sub-classes are neither change nor fault prone than other classes. The analysis of possible confounding factors in Section 8 points out that the change proneness of FBCS sub-classes observed in ArgoUML and Eclipse JDT may be due to the overriding feature. The results suggest that being an overriding sub-class can increase the likelihood of a class to change. However, this trend is not consistent through the analysed systems. In ArgoUML and Eclipse JDT, overriding classes and therefore FBCS sub-classes are more change prone but in Mylyn and Rhino, they are not. Future work should investigate the impact of overriding on change proneness with more systems to study this difference.

Regarding sub-classes in general, we can see that they are not more change-or fault prone than other classes. The previous studies that show that using inheritance can be problematic use a specific characteristic of that use. For example, inheritance depth in (Daly et al., 1996) and (Harrison et al., 2000).

For each research question, from **RQ2.1** to **RQ2.3**, we also investigate separately the change or fault proneness of the subsets of direct and indirect FBCS sub-classes compared to that of non FBCS sub-classes. The results again do not show any evidence of a correlation between those subsets and change or fault proneness.

7 Qualitative Study

The quantitative study that we presented in the previous section does not show that FBCS sub-classes are more change or fault prone than other classes. Yet, FBCS could still impact, positively or negatively, the design of systems: FBCP could impact systems but be solved immediately by developers and, thus, hinder development without increasing the change and fault proneness of involved classes. We contacted some developers of the systems under study in the previous section but received little feedback. Consequently, we decided to study the bugs issued against some of the systems in the previous quantitative study to perform a qualitative study.

We choose to do a survey because a quantitative, automatic study would not fully answer our research question below: we could automatically analyse whether classes involved in faults are part of some FBCS but even if classes in the fault-fixes are part of some FBCS, it would not prove that the faults are caused by the FBCP and, conversely, there could exist faults in which fixed classes do not belong to known occurrences of the FBCS but still relate to the FBCP. Thus, we perform a survey hoping that some participants will identify faults due to the FBCP (possibly without base and–or sub-

classes in some FBCS being involved directly in the description of the faults).

7.1 Research Question

The aim of the qualitative study is to gather subjective evidence that supports or not previous results on the lack of negative impact of FBCS on change or fault proneness. We conducted an on-line survey asking participants to analyse a representative sample of real bugs and their fixes to answer the research question:

RQ1: Do bugs related to FBCP exist?

The existence of such bugs would reveal that the FBCP actually exists, although our quantitative study fails to provide evidence of the negative impact of FBCS.

7.2 Procedure and Method

The design of our study is a survey of a random sample of bugs within the studied systems. The sample is representative in terms of size with a confidence level of 95% and a confidence interval of 10%.

7.2.1 Objects

The systems that we use for this qualitative study are: Eclipse JDT Core, Mylyn, and Rhino. We mine bug repositories of the three systems to extract bugs that could relate to the FBCP problem. Because FBCP is a problem related to certain uses of inheritance and overriding, we use keywords that refer to these features to identify bugs of interest. Table 11 summarises these keywords and the corresponding regular expressions used during bugs repositories mining. We selected only resolved bugs to ensure that accurate information about the causes and the resolutions of the bugs are available and also to allow participants to analyse the classes involved in the fix. Using the technique described in (An et al., 2014), we retrieved the fixing commits of each bug and provided to participants the classes in those commits as source of analysis.

Table 12 describes the results of the mining of bugs repositories giving the total number of resolved bugs per systems and the number of bugs retrieved based on the keywords. We randomly sampled bugs to analyse among all bugs related to inheritance and overriding. The size of each sample is statistically significant with a confidence level of 95% and a confidence interval of 10% (Sheskin, 2007a). Information about the sample of each system is given in Table 12.

7.2.2 Questionnaires

The survey was designed as an on-line questionnaire. Each questionnaire contains six pages. The first page describes the FBCP and the goal of the study. The remaining pages are analysis forms, one per bug. Each form contains seven questions listed in Table 13. All questions require an answer except the last one. The first question ensures that the participant read and understood the bug. It also serves to check the validity of an answer. Questions 2 to 5 evaluate the link between the bug and inheritance, polymorphism, and overriding. These questions serve to guide the participant towards the answer to the sixth question: "Do you think that this bug is related to the FBCP?". Questions 2 to 6 are multiple choice questions whose possible answers are "Yes", "No", or "I do not know". However, participants are encouraged to support their answers by comments. The last question gives the opportunity to the participants to leave any other comment regarding the analysed bugs. We randomly group bugs to analyse in packages, each package containing five bugs. To increase the chance to have more than one analysis per bug, each bug appears in two packages. Thus, we created 43 packages; each randomly assigned to at least two participants.

7.2.3 Participants

We invited a total of 121 participants via e-mail to analyse the selected bugs using the questionnaires presented in the previous section. The participants include students (Ph.D. and M.Sc.) and professionals (developers and researchers) with enough knowledge of OO programming and Java. None of the participants belongs to the developers' team of the analysed systems. We chose participants from authors contacts list, old and current student contacts lists of our research labs, and also contacts in industry. Although participants do not know the systems under analysis neither the bugs to analyse, inspecting many of the bugs to analyse shows that they generally have enough information from the description of the bug and its solution to answer the questions. They can also analyse the classes that have been committed to fix the bug if needed. We remind participants at least three times before closing the survey that run for two months.

7.3 Results

Out of the 121 invited participants, 41 participants answered a total of 33 packages. The response rate is about 34%, higher than what is usually reported in

Table 11 Keywords Used to Extract Identify Reports of Interest.

Feature	Keywords	Regular Expression
Inheritance	child, children	child*
	base class, super-class	base[|−]class, super[|−]class
	sub-class	sub[|−]class
	inheritance, extension, inherits, extends	inherit*, exten*
Overriding	overrid, overriding, overriden	overrid*

Table 12 Bugs Repositories Mining Results.

System	# resolved bugs	# Bugs of interest	Sample Size
Eclipse JDT	2530	177	62
Mylyn	868	11	10
Rhino	520	55	35

the literature (R. M. et al., 2009). All bugs have been analysed at least once. A preliminary check helps us to discard irrelevant or incomplete responses. For example, we discarded bug analyses without an answer to the first question. Because, the answer to this question is mandatory, we use the following heuristic to identify those cases: if the length of the answer is smaller than 10 then the bug analysis is discarded. We also discarded bug analyses whose answers to questions 2 to 6 are all "I do not know". We discarded 22 bugs analysis. The remaining 183 bug analysis concern 104 bugs; 34 of them have been analysed by one participant, 61 by two participants, and nine by three participants. Table 15 gives the details of this distribution per system. As explained before, to increase the chance to have more than one analysis per bug, each bug appears in two packages and each package has been assigned to at least two participants. Because not all participants have answered to the survey and also not all packages have been analysed, we end up with differences in the number of times each bug has been analysed. Moreover the fact that we discarded some bugs analysis also reduces the total number of analysed bugs and–or the number of times they have been analysed. Tables 15 and 16 respectively summarise the responses of bugs analysed by one participant and bugs analysed by at least two participants.

As shown in Table 15, participants identified nine bugs possibly related to the FBCP: seven in Eclipse, one in Rhino, and one in Mylyn. We summarise the participants' answers to the analyses of these bugs in Table 17. Regarding bugs analysed more than once, participants identified 38 of them as potentially related to the FBCP. Among them, participants agreed on five bugs whose analyses are summarised in Table 18.

A bug is related to the FBCP if it is related to inheritance and overriding. Moreover, answers to questions 3 and 4 should not be "No". Answering "No" to one of the questions from 2 to 5 dismisses one of the necessary conditions to have a FBCP. However, as we can see in Tables 17 and 18, in most cases, participants answered "No" to one or more of the preliminary questions and positively to the question relating the bugs to the FBCP. Surprisingly, one of the participants answer negatively to all the preliminary questions but conclude that the bug is caused by the FBCP. Because we are aware that the answers to these questions may not be obvious by analysing the bugs and fixes, we chose to manually check each of these 14 bugs. One of the authors manually checked the bugs. In most of the cases, from the description of the problem, it is clear that the bug is not related to the FBCP but to one or both of its characteristics without satisfying the other conditions. For instance, the fix to the bug 123514 in Eclipse JDT is described as follows:" CompletionParser must override consumeTypeParameter1() and not consumeTypeParameters1()". It is then clear from this comment that the bug is caused by the fact that the sub-class overrides the wrong method. The bug is indeed related to inheritance and overriding but not to the FBCP. In other cases which we identified as false positive, participants answer positively to all the preliminary questions. For instance, the bug 80063 in Eclipse JDT is such case. The participant answers "Yes" to all the preliminary questions but the description of the bug says: "Code assist allows overriding super class private method. Superclass' private members should be omitted instead.". It is clear from this description and the following comments that this bug concerns the code assist that should not propose to override private classes. This bug is not caused by the use of inheritance or overriding and thus far from being due to the FBCP. However, some cases were more tricky. For those cases, at least two authors check the bugs to decide whether or not they are related to the FBCP. An example of such case is the bug 28064 in Eclipse JDT. The summary of this bug is the following: "the compilation unit CU had definite compilation errors, and it had an anonymous subclass declaration in a field initialiser that probably caused an infinite loop.". Looking to the classes and the fix, we saw that the error is not due to the anonymous subclass and therefore not to the use of inheritance. The results of the analysis finally revealed that none of these 14 bugs are actually due to the FBCP.

Regarding the remaining 32 bugs on which participants disagreed whether they are or not related to the

Table 13 Bug Analysis Questions.

Id	Question
1	Please, give in your own words a summary of the cause of this bug?
2	Is the bug related to inheritance?
3	Is the bug related to polymorphism?
4	Is the bug related to a method being overridden in a sub-class?
5	Is the bug related to a method being overriden in a sub-class and calling (or not) a method of the super-class?
6	Do you think that this bug is related to the FBCP?
7	Please, feel free to share with us any other comments you judge interesting about this bug.

Table 14 Distribution of responses.

System	# Total bugs analysed	# Bugs Analysed Once	# Bugs Analysed twice	# Bugs analysed three times
Eclipse JDT	59	29	30	0
Mylyn	8	1	4	3
Rhino	35	4	25	6

Table 15 Responses of Bugs Analysed by one Participant.

System	# bugs analysed	# Bugs related to FBCP
Eclipse JDT	29	7
Mylyn	1	1
Rhino	4	1

Table 16 Responses of Bugs Analysed by at least Two Participants.

System	# bugs analysed at least twice	# Bugs related to the FBCP with Agreement	# Bugs related to the FBCP with Disagreement
Eclipse JDT	30	2	18
Mylyn	7	0	4
Rhino	31	3	11

FBCP, following the process described above, we performed a manual check that indicated that none of them is caused by the FBCP.

7.4 Conclusions and Discussions

We thus answer our research question as follows:

RQ3: none of the bugs that participants analysed is due to the FBCP although some of them have been wrongly identified as such.

Participants identified some of the analysed bugs as related to the FBCP although they are actually not. These participants made their decisions based on one or two of the conditions and not all required ones. Indeed, when a bug is related to inheritance or overriding, they identified such bugs as related to the FBCP. Other participants answer "Yes" to all questions but the manual check or other participants contradict them. These false positive could be also caused by an experiment learning bias. Because, participants were asked to analyse bugs to identify bugs related to the FBCP, they could think that at least one bug in the set they analyse should be related to the FBCP.

Besides the survey, we also manually checked bugs involving the border FBCS sub-classes of all versions of Barcode4J and CheckStyle Eclipse plugin in configurations with Checkstyle-core. We could not analyse all border FBCS because of their sheer numbers. We choose Barcode4J and CheckStyle because they have lows numbers of classes: Barcode4j (104–178) and CheckStyle (300–368), which allow manual analyses. The results of this manual check go along with the results of the survey and the quantitative study: In none of the 66 bugs analysed, the issue is related to the FBCP.

8 General Discussions

Our *main goal* was to infer the prevalence and impact of the FBCP by considering two particular FBCS—mutual recursion and call to an overridden method and: (1) by detecting FBCS occurrences, (2) by comparing the change and fault-proneness of the classes participating in these occurrences as sub-classes with respect to that of other classes, and (3) by assessing whether there exist bugs whose cause is the FBCP.

The results of the preliminary study show that there exists a significant proportion of call to an overridden method FBCS occurrences in most of the analysed systems but few mutual recursion FBCS occurrences. However, the study of sub-classes involved in these FBCS does not indicate that those classes are more fault prone than other classes. However, the analysis performed to answer **RQ2** in Section 6 shows that, in most versions of two systems out of the four analysed, they are more change prone than the other classes.

In this section, we analyse these results in the light of possible confounding factors that could affect the outcome of our studies.

8.1 Confounding Factors for the Quantitative Study

Inheritance and overriding are two characteristics of FBCS classes. Thus, they can be confounding factors when answering **RQ2** in Section 6. Therefore, to avoid a false inference about the relation between FBCS sub-classes and change and fault proneness, we also consider

Table 17 Summary of Responses of Bugs Identified as Related to the FBCP by One Participant.

#	Bug Id	System	Q1	Q2	Q3	Q4	Q5	Q6
1	80063	Eclipse JDT	The code Assist suggests overriding a private method. Although the implementation leads to have a new method that is unrelated to the one in the base class.	Yes	Yes	Yes	Yes	Yes
2	93396	Eclipse JDT	Abstract method not implemented	Yes	Yes	I don't know	Yes	Yes
3	160652	Eclipse JDT	The class can not be sub-classed by clients but it is	Yes	No	No	No	Yes
4	162026	Eclipse JDT	calling of the method fetch is not explicit	Yes	Yes	No	No	Yes
5	191247	Eclipse JD	API breakage	Yes	No	Yes	No	Yes
6	210681	Eclipse JDT	inheritance issue	Yes	No	No	No	Yes
7	426048	Eclipse JDT	My understanding is that a class had its variables not initialised, which ended up causing NPE once they were used indirectly by a subclass.	Yes	I don't know	I don't know	I don't know	Yes
8	352933	Mylyn	this error is based on interaction events and coupled to the context framework. To improve modularity and reuse of tasks task activity related classes should be extracted to a separate bundle and injected through an extension point.	No	No	Yes	No	Yes
9	328924	Rhino	classes are package protected and can't be reused. need to duplicate all of the classes	Yes	No	Yes	Yes	Yes

the following two research questions: **MF1**: What is the relation between sub-classes involved in any inheritance relations, belonging to $SSub_{S_{inheritance}}$, and change and fault proneness? We want to observe whether sub-classes are more change- or fault-prone than other classes in general. Similarly, we ask: **MF2**: What is the relation between sub-classes involved in any overriding relations, belonging to $SSub_{S_{overriding}}$, and change and fault proneness? We want to observe whether overriding classes are more change- or fault-prone than other classes in general. For both confounding factors, we compare the change- and fault-proneness of classes belonging to either $SSub_{S_{inheritance}}$ or $SSub_{S_{overridding}}$ with that of classes not belonging to these sets. Thus, we can confirm whether any differences among sub-classes and other classes observed while answering **RQ2** are due to their role in FBCS occurrences or to them being sub-classes or sub-classes overriding some methods.

We thus ask the following questions:

- **MF1**: $SSub_{S_{inheritance}}$ and $S_{classes} \backslash SSub_{S_{inheritance}}$;
- **MF2**: $SSub_{S_{overridding}}$ and $S_{classes} \backslash SSub_{S_{overridding}}$.

8.1.1 ***MF1****: Are sub-classes ($SSub_{S_{Inheritance}}$) more change- or fault-prone than other classes$S_c lasses \setminus SSub_{S_{inheritance}}$?*

Table 19 shows that in 10 versions out of 36, sub-classes are more change prone than other classes. These versions include two versions of ArgoUML and Mylyn and three versions of Eclipse JDT and Rhino. The odd ratio in these cases vary from 0.54 to 5.1. These results suggest that the change-proneness of FBCS sub-classes may not due to the fact that they are sub-classes.

Regarding the fault proneness, only five versions out of 36 show a significant difference between sub-classes and other classes with a odd ratio going from 0.53 to 3.08. The concerned versions comprise three versions of Eclipse JDT and two versions of Rhino. FBCS sub-classes are, like sub-classes in general, not more fault-prone than other classes.

8.1.2 ***MF2****: Are sub-classes ($SSub_{S_{Overriding}}$) more change- or fault-prone than other classes$S_c lasses \setminus SSub_{S_{Overriding}}$?*

As shown in Table 19, overriding sub-classes are more likely to change than other classes in 17 versions: three versions of ArgoUML, all versions of Eclipse JDT, five versions of Mylyn, and four versions of Rhino. These results are similar to that of FBCS sub-classes. To identify the predominant factor that trigger these results, between being an overriding sub-class or being an FBCS sub-class, we compare the change proneness of FBCS sub-classes to that of other overriding sub-classes. Results show that FBCS sub-classes are more change-prone than other overriding sub-classes in only eight systems out of 36. Moreover, the odd ratio in these cases is less than 1. Thus, these results suggest that the change-proneness of FBCS sub-classes may be due to the fact that they are overriding sub-classes.

Overriding sub-classes are likely to be faulty more than other classes in only eight systems out of 36 with a

Table 18 Summary of Responses of Bugs Identified as Related to the FBCP by at least Two Participants.

#	Bug Id	System	Q1	Q2	Q3	Q4	Q5	Q6
1	83600	Eclipse JDT	Signature construct not well respected	Yes	Yes	Yes	I don't know	Yes
2	83600	Eclipse JDT	The List types verificator does not perform well when is has to accept subtypes with the ? extends Number notation	Yes	No	Yes	Yes	Yes
3	123514	Eclipse JDT	The autocompletion feature crash because the implementation for this specific case override the wrong methods	Yes	Yes	Yes	Yes	Yes
4	123514	Eclipse JDT	Wrongly override the method of parent class.	Yes	No	Yes	No	Yes
5	271401	Rhino	Classes that extend ScriptableObject cannot use polymorphism to define.JavaScript-accessible properties and functions because Class.getDeclaredMethods() is used instead of Class.getMethods()	No	No	No	No	Yes
6	271401	Rhino	Probably a problem with casting objects during its lifecycle	Yes	Yes	I don't know	Yes	Yes
7	42097	Rhino	a global property overwrite is occuring instead of a local prop being.created in function now	Yes	Yes	Yes	No	Yes
8	42097	Rhino	Rédéfinition involontaire de la superclasse.	Yes	Yes	Yes	Yes	Yes
9	462827	Rhino	The main is to allow JavaAdapter class to extend Scriptable objects without additional wrapping.	Yes	No	Yes	Yes	Yes
10	462827	Rhino	Baseclass is responsible for wrapping the instance of Scriptable wrapper in the subclass..	Yes	No	Yes	No	Yes
11	462827	Rhino	Tis is an extension to allow JavaAdapter to to extend Scriptable objects directly, without additional wrapping as NativeJavaObject..	Yes	No	No	No	Yes

Table 19 Results RQ2: Confounding Factors

	Argouml		Eclipse JDT		Mylyn		Rhino	
	Change	Bug	Change	Bug	Change	Bug	Change	Bug
$SSub_{S_{inheritance}}$ vs $S_{classes} \setminus SSub_{S_{inheritance}}$	2/6	0/6	3/5	3/5	2/13	0/13	3/12	2/12
$SSub_{S_{overridding}}$ vs $S_{classes} \setminus SSub_{S_{overridding}}$	3/6	4/6	5/5	2/5	5/13	0/13	4/12	2/12
$SSub_{S_{FBCS}}$ vs $SSub_{S_{overridding}} \setminus SSub_{S_{FBCS}}$	4/6	2/6	4/5	0/5	0/13	0/13	0/12	0/12

odd ratio varying between 0.22 and 2.64. These results are similar to that of FBCS sub-classes. When looking to the fault proneness of FBCS sub-classes compared to that of other overriding classes, we can see that FBCS sub-classes are more fault prone than other overriding sub-classes in only two versions of ArgoUML with a odd ratio of 0.5 and 2.8. These results suggest again that the few cases where FBCS sub-classes are more fault prone than other sub-classes may be due to the fact they are overriding sub-classes.

The analysis of the possible confounding factors that are inheritance and overriding shows that the results observed could be due to overriding. Thus, FBCS sub-classes are more change prone than other classes in ArgoUML and Eclipse JDT because they are also overriding sub-classes. Finally, we conclude that FBCS sub-classes are not more change or fault prone than other classes. The results of the qualitative study supports also these findings: none of the analysed bugs is caused by the FBCP. The present section presents the analysis of the confounding factors and discusses the reasons that could explain our findings.

8.2 Confounding Factors for the Qualitative Study

Some factors related to both the bugs and the participants could mitigate our results and conclusions. The choice of the bugs and the mean to administer the survey could impact the participants' answers. We tried to have as many participants as possible and also carefully studied manually the bugs in which more than one participants indicated the presence of the FBCP.

Participants' demographics and characteristics could also impact the results and conclusions of our survey. For example, their knowledge (or lack thereof) of Java and–or of the systems and–or of the FBCP would impact their answers. We believe that these factors exist but do not negatively impact our results and conclusions because we contacted 121 participants, out of which 41 provided answers, without choosing and–or disciminating against participants. Hence, our participants include a variety of knowledge.

8.3 Reasons for Lack of Impact and Causes

Although the significant proportion of the considered FBCS in the analysed systems, the quantitative study failed to establish a correlation between FBCS and change or fault proneness. The results of the qualitative study also support the results of the quantitative one. None of the 104 bugs that participants analysed is related to the FBCP. We explain in the following the main reasons which we believe can explain this lack of impact and causes. All the reasons could explain the results of our studies and are all means that can help to prevent the occurrence of the FBCP.

8.3.1 Savvy use of inheritance and overriding

The observation that FBCS occurrences are not more change or fault prone than other classes may suggest that developers are careful when using inheritance and overriding even from third-party libraries. The fact that a significant proportion of classes participate in FBCS occurrences also indicates that FBCS are usual and that developers use them frequently in their code.

Moreover, many participants of the survey, which included professional and students with a good knowledge and practice of Java and OO programming, identified some bugs as related to the FBCP albeit they were not. This observation shows that participants may not be aware of the FBCP and do not encounter this kind of problem in their work.

8.3.2 IDEs Assistance

Current IDEs help developers by detecting some inconsistencies when coding. This assistance may also explain why FBCS occurrences seem not to be harmful and are adequately used. Although there is no specific warning detecting a FBCS, many warnings exist regarding overriding. For example, the option `Missing '@Override'` annotation in Eclipse IDE will require to explicitly annotate an overriding method. Thus, accidental naming, which is one of the possible cause of the FBCP, is prevented by current IDEs.

8.3.3 Testing

Testing is one of the main means for developer to find bugs in their systems and ensure software quality. The development of xUnit frameworks (such as JUnit) and their integration into most IDEs increase testing activities and allow developers to catch "coarse" errors, which could explain our results and conclusions. Indeed, all the analysed systems are accompanied by test cases that could help (and have helped) developers to detect bugs related to the FBCP early during development.

8.3.4 Dependency Injection

One of the proposed solutions to prevent the FBCP consists in favouring composition over inheritance (Gamma et al., 1995; Bloch, 2008). This solution is nowadays easy to use thanks to the introduction of the concept of interfaces for typing as well as to the prevalence of frameworks using "dependency-injection". Dependency injection is a programming idiom (or a design pattern depending on the researcher) that implements inversion of control. Instead of having an inflexible dependency between classes, this idiom allows injecting the dependencies when needed into the framework and thus facilitate the use of composition by clients and is a means to prevent the FBCP.

9 Threats to validity

This section discusses the threats to validity that can affect our studies.

Construct validity threats concern the relationship between theory and observation. In our quantitative study, these threats are due to the detection of FBCS occurrences. Our detection tool is based on Ptidej (Gueheneuc, 2007), a tool suite for the analysis of code and design models, which is actively maintained and which has been used in many previous studies. Ptidej forms a proven framework on which to build our analysis tool. Yet, it is possible that errors in our analysis tool would bias the detected occurrences. We thoroughly reviewed our code and tested it and, also, we perform several manual validations. Thus, we accept this threat. We also provide our analysis tool on-line[1] for other researchers to use and–or review.

Another threat concerns the identification of faulty and changed classes. We relied on change and fault data independently computed and published in a previous work (Khomh et al., 2011). Therefore, we believe that this threat is minimal even though different faults and changes would impact our results and conclusions.

Another threat is the use of FBCS as proxy to measure the existence of the FBCP in the quantitative studies. FBCS occurrences are defined as opportunities for the FBCP, but notwithstanding the results and conclusions of our studies, not all FBCS would lead to the FBCP. We did not find any significant correlation between FBCS occurrences and change and fault proneness, and discussed whether this lack of correlation is due to the FBCP or other confounding factors. Yet, as mentioned in Section 3, there are other

aspects of the FBCP problem that are not captured by the FBCS in general and by the FBCS considered in our studies—mutual recursion and call to an overridden method, such as conflicts in the method interfaces and behavioural conflicts. Those other aspects of the FBCP are out of the scope of this paper and will be subject to a future work.

Finally, the FBCP is thought in the literature to impact software development and reduce software quality if and only if changes are made to the base- or sub-classes participating to related FBCS. In our studies, we assumed that we could measure this impact a posteriori, by measuring the change and fault-proneness of classes that were committed in the control-version systems of the analysed systems. Our assumption does not and cannot simply consider other impact, such as developers introducing faults due to the FBCP but identifying these faults through testing and fixing them before committing their changes or such as developers reverting their changes before committing them because the presence of the FBCP make them too difficult to perform successfully. Such threats exist and should be the subject of future work.

Internal validity threats concern confounding factors that can affect our dependant variables. Regarding the quantitative study on change and fault proneness of FBCS sub-classes, we dealt with these threats by analysing two possible confounding factors: inheritance and overriding.

Another such threat is related to the particular choice of the systems and the versions that we analysed. We tried to mitigate this threat in the preliminary study by using multiple versions of seven systems from different application domains, with different sizes, and which have been used in previous studies (Tempero et al., 2008, 2010). Moreover, these seven systems led to the analyses of a wide variety of frameworks. Regarding the second quantitative study, we used a subset of this pool of systems guided by the availability of change and fault data. The four chosen systems are also from different application domains and have different sizes.

Conclusion validity threats deal with the relation between the treatment and the outcome. We used proper statistical tests to answer our research questions. We used the non-parametric statistical Fisher test to assess the differences in terms of change and fault proneness of the analysed populations. We also computed the odd-ratios to evaluate the magnitudes of the differences.

Another such concern is related to the representativeness of the sample of bug that we used in the survey. To mitigate this threat, we performed a random sampling across bugs containing terms related to inheritance and–or overriding. To have a representative sample in terms of size, we considered a confidence level of 95% and a confidence interval of 10%. Although we cannot guarantee to have a representative sample in terms of relevance of bugs, the random sampling makes it unlikely that we accidentally excluded all relevant bugs.

Another threat concern the manual checking that we performed to evaluate the responses of the participants. This manual checking could be affected by subjectiveness or human error. To mitigate this threat, we took in consideration more than one opinion when the bugs were not easy to understand.

External validity threats concern the generalisability of our results. The preliminary study involved multiple versions of seven clients and 58 frameworks from different application domains and different sizes. We thus could expect that our results and conclusions about the existence of FBCS occurrences could be generalised to other Java open-source systems.

The second quantitative study is limited to multiple versions of four different systems. The trends in the change and fault-proneness of the FBCS sub-classes are not the same in all the systems. Therefore, we would need to perform larger studies with more systems to generalise our conclusions. Such larger studies are part of future work.

Regarding the qualitative study, the survey involved samples of bugs from three systems. Future work should target more systems to generalise the results and conclusions. Yet, this survey is the first of such survey and it involved 41 participants (out of 121 invited participants), which makes it the largest survey about the FBCP.

10 Conclusion and Future Work

The fragile-base class problem (FBCP) has been defined and studied in the literature for many years (IBM, 1994; Mikhajlov and Sekerinski, 1998; Aldrich, 2004). Previous works proposed various solutions to prevent or alleviate the FBCP when designing and developing object-oriented systems. Yet, **the question remains whether solutions to the FBCP should be applied extensively.** To answer this question, we performed qualitative and quantitative studies.

First, we performed a preliminary quantitative study that confirms the existence of occurrences of some fragile-base class structures (FBCS) in open-source systems. These FBCS are some of the opportunities in systems architectures that can lead to two aspects of the FBCP: the mutual recursion and the call to an overridden method. Second, we performed another quantitative study on

the relation between FBCS occurrences and change and fault proneness to assess the impact of these occurrences on system and, indirectly, the impact of these tow aspects of the FBCP. Using our detection tool, we analysed multiple versions of seven different open-source Java systems (clients), 58 different frameworks, and 301 configurations resulting from the combination of clients and frameworks. Our results showed that there exists a significant proportion of call to an overridden method FBCS occurrences in most of the analysed frameworks, clients, and configurations but few mutual recursion FBCS occurrences. Our results also showed that there is no statistical evidence that classes participating in these FBCS as sub-classes are more change-prone or fault-prone than classes not participating in any FBCS. Third, we performed a qualitative analysis by the means of a survey. We select 104 bugs from three systems, namely Eclipse JDT, Mylyn, and Rhino, and asked participants to analyse these bugs to assess whether they are caused by the FBCP. We received answers from 41 participants and, although some participants identified some of the bugs as related to the FBCP, a further manual validation revealed that none of the bugs was due to the FBCP.

Thus, to the extent of the threats to the validity of our studies, we concluded that the two aspects of the FBCP that we analysed—mutual recursion and call to an overridden method—may not be as problematic as thought in previous works, in terms of change and fault proneness. Generally, our results suggested that **it may not be worth the effort to implement the solutions proposed in the literature to prevent extensively and systematically the FBCP related to mutual recursion and call to an overridden method**. Yet, we also discussed that many reasons could justify the lack of negative impact of the FBCP on systems. In particular, we argued that a savvy use of inheritance by developers, the warnings available in IDEs to help developers avoid some faults, the popularity of dependency-injection and the use of interfaces over classes for typing, and thorough testing by developers all help prevent the FBCP. Among these reasons, dependency injection, which is very popular in frameworks, was introduced by Gamma et al. (1995) and Bloch (2008) as a means to avoid the FBCP. Thus, we believe that it is not so surprising that FBCS do not have a negative impact on systems.

Future work should investigate in more details mutual recursion and call to an overridden method FBCS as well as other FBCS and should also use more systems to generalise our findings. Investigating other aspects of the FBCP may lead to other results regarding the change and fault proneness of their participating classes for which we currently cannot say anything. A survey with developers of the analysed systems could help to understand whether the FBCP could impact other aspects of software development that cannot be measured by a post-hoc analyses of releases. Future work also include another survey to observe, organise, and study the measures that developers take when writing code containing FBCS, especially when third party libraries are involved. Other aspects of the FBCP, such as conflicts in the method interfaces and behavioural conflicts, should also be studied.

Finally, as several of the clients and frameworks considered in our studies come with tests, we could use these available tests to assess whether classes and methods participating into FBCS have been tested and, in particular, if some tests have been designed particularly to avoid a possible fault related to the FBCP.

Acknowledgements This work has been partly funded by the NSERC Research Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software. The authors are grateful to all the anonymous participants.

References

Aldrich, J. (2004). Selective open recursion: A solution to the fragile base class problem. School of Computer Science Carnegie Mellon University.

An, L., Khomh, F., and Adams, B. (2014). Supplementary bug fixes vs. re-opened bugs. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*.

Biberstein, M., Sreedhar, V. C., and Zaks, A. (2002). A case for sealing classes in Java. In *Israeli Workshop on Programming Languages & Development Environments*.

Bloch, J. (2008). *Effective Java*. Addison-Wesley, 2 edition.

Briand, L. C., Wüst, J., Daly, J. W., and Porter, D. V. (2000). Exploring the relationship between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51:245–273.

Daly, J., Brooks, A., Miller, J., Roper, M., and Wood, M. (1996). Evaluating inheritance depth on the maintainability of object-oriented software. *Journal of Empirical Software Engineering*, 1:109–132.

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388.

Gamma, E., Helm, R., R.Johnson, and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object*

Oriented Software. Addison-Wesley, Boston, MA, USA.

Ghezzi, C. and Monga, M. (2002). Fostering component evolution with C# attributes. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 22–28. ACM.

Gueheneuc, Y. (2007). Ptidej: A flexible reverse engineering tool suite. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 529–530.

Guéhéneuc, Y.-G. and Albin-Amiot, H. (2004). Recovering binary class relationships: Putting icing on the uml cake. In Schmidt, D. C., editor, *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press.

Guéhéneuc, Y.-G. and Antoniol, G. (2008). DeMIMA: A multi-layered framework for design pattern identification. *IEEE Transactions on Software Engineering*, 34.

Harrison, R., Counsell, S., and Nithi, R. (2000). Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, 52:173–179.

Hürsch, W. (1994). Should superclasses be abstract? In Tokoro, M. and Pareschi, R., editors, *Proceedings of the European Conference on Object-Oriented Programming*, volume 821 of *Lecture Notes in Computer Science*, pages 12–31. Springer.

IBM (1994). IBM's System Object Model (SOM): Making reuse a reality. White paper, IBM Corporation, Object Technology Products Group.

Kegel, H. and Steimann, F. (2008). Systematically refactoring inheritance to delegation in Java. In *Proceedings of the International Conference on Software Engineering*, pages 431–440. ACM.

Khomh, F., Penta, M. D., Guéhéneuc, Y.-G., and Antoniol, G. (2011). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering (EMSE)*.

Kiczales, G. and Lamping, J. (1992). Issues in the design and speci
cation of class libraries. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 435–451.

Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28:449–462.

Mezini, M. (1997). Maintaining the consistency of class libraries during their evolution. *SIGPLAN Notices*, 32:1–21.

Mezini, M., Pipka, J. U., Dittmar, T., and Boot, W. (1999). Detecting evolution incompatibilities by analyzing java binaries. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 126–135. IEEE CS Press.

Mikhajlov, L. and Sekerinski, E. (1998). A study of the fragile base class problem. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 355–382.

Ozaki, H., Gondow, K., and Katayama, T. (2003). Class refinement for software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 51–56. IEEE CS Press.

Parkinson, M. J. and Bierman, G. M. (2008). Separation logic, abstraction and inheritance. *SIGPLAN Notices*, 43:75–86.

R. M., G., F. J., Fowler, J., M. P., C., J. M., L., E., S., and R., T. (2009). *Survey Methodology*. Wiley, 2 edition.

Robbes, R., Rthlisberger, D., and Tanter, r. (2015). *Empirical Software Engineering*, 20(3):745–782.

Ruby, C. and Leavens, G. T. (2000). Safely creating correct subclasses without seeing superclass code. *SIGPLAN Notices*, 35:208–228.

Sheskin, D. J. (2007a). *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition.

Sheskin, D. J. (2007b). *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All.

Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 38–45.

Steyaert, P., Lucas, C., Mens, K., and D'Hondt, T. (1996). Reuse contracts: Managing the evolution of reusable assets. *SIGPLAN Notices*, 31:268–285.

Taenzer, D., Gandi, M., and Podar, S. (1989). Problems in object-oriented software reuse. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 25–38. Cambridge University Press.

Tempero, E., Counsell, S., and Noble, J. (2010). An empirical study of overriding in open source java. In *Proceedings of the Australasian Computer Science Conference*, pages 3–12. Australian Computer Society, Inc.

Tempero, E., Noble, J., and Melton, H. (2008). How do java programs use inheritance? An empirical study of inheritance in java software. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 667–691. Springer.

Wegner, P. and Zdonik, S. (1988). Inheritance as an incremental modification mechanism or what like is and isnt like. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77. Springer Berlin Heidelberg.

Williams, S. and Kinde, C. (1994). The Component Object Model: Technical overview. *Dr. Dobbs Journal*.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers.