**SQJ manuscript No.**
(will be inserted by the editor)

# Test Case Selection in Industry:

## An Analysis of Some Issues

**Vincent Blondeau**[1,2] · **Anne Etien**[2] · **Nicolas Anquetil**[2] · **Sylvain Cresson**[1] · **Pascal Croisy**[1] · **Stéphane Ducasse**[2]

Received: March 15, 2016/ Accepted: –

**Abstract** Automatic testing constitutes an important part of everyday development practice. Worldline, a major IT company, is creating more and more tests to ensure the good behaviour of its applications and gain in efficiency and quality. But running all these tests may take hours. This is especially true for large systems involving, for example, the deployment of a web server or the communication with a database. For this reason tests are not launched as often as they should and are mostly run at night. The company wishes to improve its development and testing process by giving to developers rapid feedback after a change. An interesting solution is to reduce the number of tests to run by identifying only those exercising the piece of code changed. Two main approaches are proposed in the literature: static and dynamic. The static approach creates a model of the source code and explores it to find links between changed methods and tests. The dynamic approach records invocations of methods during the execution of test scenarios. Before deploying a test case selection solution, Worldline puts together a partnership with us to investigate the situation in its projects and to evaluate these approaches on three industrial, closed source, cases to understand the strengths and weaknesses of each solution. We propose a classification of problems that may arise when trying to identify the tests that cover a method. We give concrete examples of these problems and list some possible solutions. We also evaluate other issues such as the impact on the results of the frequency of modification of methods or considering groups of methods instead of single ones. We found that solutions must be combined to obtain better results, problems have different impacts on projects. Considering commits instead of individual methods tends to worsen the results, perhaps due to their large size.

**Keywords** Test selection · Dynamic · Static · Industrial case

[1] Worldline
Z.I A Rue de la Pointe
59113 Seclin, France
E-mail: {firstname.lastname}@worldline.com ·
[2] Univ. Lille, CNRS, Inria, Centrale Lille,
UMR 9189 - CRIStAL,
F-59000 Lille, France
E-mail: {firstname.lastname}@inria.fr

## 1 Introduction

In industry, the need to test every piece of code becomes compulsory. Worldline, a major IT company, like many other companies (Ekelund and Engström, 2015), is facing a contradiction: to guarantee high quality level solutions, developers write tests; however, tests become so numerous that several hours are required to run them all. In an industrial environment where each line of code has to be written as fast as possible and where the code has to be flawless, this delay is not bearable. As a consequence, the developers often bypass the tests during the day and an automatic testing job is launched during the night to run all the tests. However, developers need feedback on the behaviour of their implementation as soon as possible to avoid spending time on potential future debugging. Wishing to improve its development process while improving the quality of the developed solutions, Worldline made a partnership with Inria RMod team for auditing its projects and provide solutions to reduce the feedback time of the tests and keep the test quality it provides to its client.

The envisioned solution consists in reducing the time needed to run the tests by reducing the number of tests. For a given change of the source code, some tests are not relevant because they do not cover the changed source. So, running only a subset of tests can be a solution to get faster feedback while actually exercising the new code.

Among the known approaches to select a subset of tests to execute to verify a change in the code, we found the dynamic or static ones. *Static approaches* consist in creating a model of the source code. This model can then be navigated, going up the chain of method calls, from a changed method back to the tests that exercise it. *Dynamic approaches* involve executing the tests and recording the methods invoked by each test. The test subset supplied by this approach is trivially composed of the tests executing (covering) a changed piece of code.

Both approaches have their strengths and weaknesses. But, they are seen as complementary by Ernst (2003). After evaluating 32 propositions, Engström *et al.* (2008) see evidences that differences between techniques are not very strong and sometimes contradictory. The authors found that each approach depends on its implementation and on the programming language used.

But these two previous analyses are essentially theoretical. In this paper, we propose a concrete study of different issues that occur on large industrial cases at Worldline. The issues encountered are not system dependent and can also happen in other organisations. We categorize these issues to help fully understand the problem. We also experimented possible solutions to get an idea of the impact of these issues on the results. We, finally, studied the impact of the frequency of change of the methods on the overall results but also the impact of considering commits (*i.e.,* groups of methods) instead of individual methods.

This paper makes four contributions. (i) We propose a classification of problems that may arise with the static approach when trying to identify tests that cover a method. (ii) We give concrete examples of these problems and list some possible solutions, evaluates other issues such as: (iii) the impact of the highly modified methods or (iv) considering groups of methods instead of individual ones. We found that solutions must be combined to obtain better results (*i.e.,* issues are intertwined); issues have different impacts on different projects; considering commits instead of individual methods tends to worsen the results, perhaps due to their large size.

In Section 2, we present the test case selection problem and define the existing approaches. In Section 3, we give a classification of the problems that may arise and we detail concrete occurrences of these problems. Then, in Section 4, we describe an experiment to evaluate the impact of each problem on the performances of an approach. Section 5 analyses

and discusses of the results of the experiment on the closed source projects and Section 6 presents the related works. Finally, we conclude in Section 7.

## 2 Problem Description

In large industrial projects, executing all tests after each change can turn into a costly operation requiring several hours. For example, we experimented with a project where executing all tests requires five hours of computation (see Section 4.2). To get feedback on the changed code more rapidly, it is important to reduce drastically this time. The solution generally adopted for this consists in trying to reduce the number of tests to run. A theoretical perfect approach would select only the tests demonstrating a flaw in the behaviour of the application after the change (called *modification-revealing tests* (Biswas *et al.*, 2011; Yoo and Harman, 2012)). But a real approach can only approximates this selection. For this, one usually concentrates on the tests covering the changed code (that is to say the tests that should lead to the execution of the changed code). The hope is that this real approach selects a suitable and small set of tests to detect a regression in the application behaviour.

### 2.1 The Problem of the Company

Worldline is the European leader in the payments and transactional-services industry. It is present in 17 countries across the globe with approximately 7500 employees. Worldline's end-to-end customized solutions help customers to optimize the performance of their digital transactions. Behind the scene, Worldline connects its clients with their customers through integrated and personalized digital services providing a seamless customer and citizen experience. This includes designing, implementing and managing services and solutions on behalf of its clients.

Tests are crucial for Worldline for different reasons. First, the company provides payment and transactional-services that are critical to its customers. Errors, bugs or denial of service are not allowed. Second, it provides solutions from design to deployment and maintenance. Maintainers can use information from test to help them understand, debug, and retest programs (Agrawal *et al.*, 1998). However, running all the tests on a project may take hours because they require installing and configuring database or another environment as well as testing abnormal running conditions such as timeout on server connection. In a daily development process, developers can not run the tests after a change to check the impacts of their modifications. Since they have no tool to detect tests impacted by a change, they very often skip tests during the day and these only run at night thanks to continuous integration servers.

We decided with Worldline to improve this situation. We work closely with a transversal team in this company that provides tools, expertise and support to the development teams. This team is aware of the issues met by the field teams and look for adapted solutions to simplify developers work while guaranteeing quality. The company provides real projects to analyse. The collaboration between academia and industry occurs mainly through the first author of this paper who is a PhD student paid by Worldline. To convince upper management of possibly imposing a change in work practices of thousands of developers, the transversal team needs convincing hard data on the pros and cons of the technique it will propose. We report in this paper some conclusions on our first studies. We proposed to first evaluate the advantages and drawbacks of several test case selection approaches. We report here our first

conclusions regarding static approaches to test case selection (see Section 2.3). Future work (not reported here) will include evaluating the impact of issues with the dynamic approach as well as a study of the way tests are currently used in the company. Based on our results, a solution will be proposed to improve test usage through test case selection mechanisms and its impact will be evaluated.

## 2.2 Test Case Selection

Test case selection techniques seek to reduce the number of test cases to execute after modifying the code. The selection is not only temporary (*i.e.,* specific to the current version of the program) but also focused on the identification of the modified parts of the program. Test cases are selected because they are relevant to the changed parts of the system under tests Yoo and Harman (2012). More formally, following Rothermel and Harrold, the selection problem is defined as follows:

*Definition Test case selection problem*
*Given*: The program, P, the modified version of P, P' and some test cases, T.
*Problem*: Find a subset of T , T', with which to test P'.

Biswas *et al.* (2011); Leung and White (1989); Yoo and Harman (2012) classify existing tests as *Reusable* or *Retestable*[1]. *Reusable* test cases exercise only unaffected parts of the program and therefore, can be omitted to test a change in the code base. *Retestable* tests exercise the modified parts of the program and therefore need to be re-run after a change.

Test case selection approaches are based on the notion of dependency graph. The general idea is that tests can be said to depend on the source code that they exercise. After a piece of code is changed, a test case selection technique just needs to go back from this piece to the tests that depend on it. Figure 1 illustrates this principle for two methods and two tests. `testMethod1` depends on `method1` and `method2` (for example `testMethod1` calls `method1` and `method2`), `testMethod2` depends on `method2`. If `method1` is changed, `testMethod2` is considered *reusable*, because it should not be impacted. On the other hand, `testMethod1` is *retestable*.

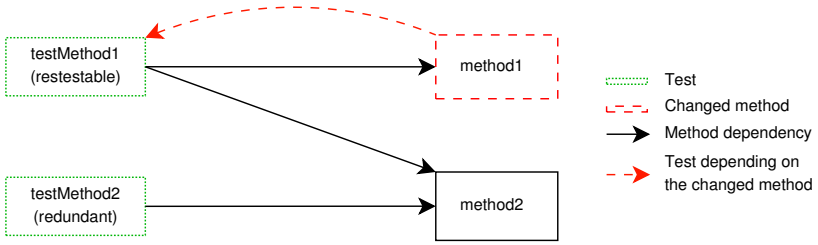


**Fig. 1** Test Selection Simple Case

This is, of course, a simplified example, in real cases, the dependency graph is much larger and deep, or, some other factors may make it very difficult for a given approach to

[1] We ignore a third category in this paper: "obsolete"

find out which tests depend on a piece of code. In this paper, we investigate what are these factors and what are their impact.

### 2.3 Dynamic and Static Approaches

Literature (*e.g.,* Engström *et al.* (2008, 2010); Ernst (2003)) recognizes several types of approaches for test case selection: some are based on source code introspection (*e.g.,* dynamic) or code analysis (*e.g.,* static), others are using specifications, meta-data or UML models. We will study two main approaches for test case selection: static and dynamic.

The *dynamic approach* consists in executing the tests and recording the code executed during each test. This is the execution trace of a test. A test depends on a piece of code if this piece of code is in its execution trace. For example, using this approach on the Figure 1, will result with a mapping for `method1` to `testMethod1`, and for `method2` to `testMethod1` and `testMethod2`. If `method1` is modified, the mapping will be used to relaunch only `testMethod1`.

The *static approach* does not require executing the tests. It relies on computing the dependency graph from the source code or some representation of it (*e.g.,* bytecode for Java). Several dependency graphs can be used (Biswas *et al.*, 2011; Engström *et al.*, 2008, 2010): Data dependency graph, Control dependency graph, Object relation diagram, etc. For example, by considering arrows in Figure 1 as calls between methods, a static approach will create a model of the source code where `method1` callers are `testMethod1`, and `method2` callers are `testMethod1` and `testMethod2`. If `method1` is modified, the `testMethod1` caller will be relaunched. Note that in real projects, the call graph is much deeper and recursion is used until a test is found.

Different kind of granularity can be considered (Engström *et al.*, 2010) from individual instructions (*e.g.,* Rothermel and Harrold (1993)) to modules (*e.g.,* White and Leung (1992)) or external components (*e.g.,* Willmor and Embury (2005)) passing through functions/methods (*e.g.,* Elbaum *et al.* (2003); Zheng *et al.* (2007)) and classes (*e.g.,* Hsia *et al.* (1997); White *et al.* (2005)). Using a smaller granularity should give better precision but is more costly (Engström *et al.*, 2010).

In this paper, we will mainly consider a granularity at the method level as it seemed to offer the best compromise between precision and cost for large industrial systems. We will see in Section 5 that this is not a clear choice. To get a base for comparison, we will also do some tests with a granularity at the class level. We could not find a tool implementing static test case selection at the level of line of code. We found two tools doing it with dynamic analysis (Ekstazi[2] and Clover[3]), but doing the actual experiment would have required too much manpower and computation time.

### 2.4 Evaluation Metrics

Biswas *et al.* (2011) use two characteristics of the test case selection approaches to evaluate if they are *safe* and *precise*. An approach is *safe* if it selects all the modification-revealing tests. An approach is *precise* if it selects only modification-revealing tests. In this paper, we will use the well known recall and precision metrics (see Section 4.4). A safe approach has a recall of 1 and a precise approach has a precision of 1.

[2] http://www.ekstazi.org

[3] https://confluence.atlassian.com/display/CLOVER

It should be noted that typically test case selection techniques concentrate on selecting *retestable* tests. Retestable is a super set of all *modification-revealing* tests. Therefore according to these definitions, all test case selection techniques will exhibit some level of *imprecision*.

Engström *et al.* (2010) also cite cost reduction criteria as important (76% of the 36 studies reported computes it). This is typically measured through the decrease in the number of tests. This metric represent the ratio of the number of not selected tests versus the total number of tests for the application. A value close to 1 means that the number of tests to relaunch is small compared to all the tests of the application. On the other hand, a value close to 0 means that the approach selects almost all the tests. However, the authors point that this may not be the best metric as reducing the number of test cases could be costlier than executing the redundant tests. As a result, 42% of the studies reported in their paper use the total time (test case selection + test execution) as a measure of cost reduction.

In this paper, we will consider only the ratio of the number of selected test, because the test cases selection time may depend on the implementation of a technique, and, we are not applying any specific technique but rather analysing the impact of different issues.

### 2.5 Known Pros and Cons

Ernst (2003) advances that a static approach guarantees generalization of the results for future executions. This approach usually results in a superset of all actual executions as some combination of cases might seem possible when looking at the code, but actually impossible in real cases. On the other hand, dynamic approach results in a subset of all actual executions as only some examples are actually run.

For Ernst, the dynamic approach is as fast as the program execution and does not require costly analyses. It must be noted that in our experiments with the Jacoco tool, we found a perceptible increase in time of up to one hour (from an initial five hours) needed to run only the tests. Static model uses an abstract representation of program state that looses information but is more compact and easier to manipulate than a more faithful dynamic model.

Beszedes *et al.* (2012) found that the dynamic approach is less reliable if the model is not updated frequently. Some code addition or modification can impact the accuracy of the approach if tests are not run again to recompute their new execution trace. Note that this is in complete opposition with the test case selection approach that tries to not re-execute all tests. Therefore, a compromise must be found between up to date model (to get good results) and actual tests selection. Finally, if the test setup (executed before all tests) or an instruction of a test fails, the execution is stopped and only a part of the code is exercised.

On the other hand, Ekelund and Engström (2015) consider that the creation of a static model can sometimes be a drawback. In case of large systems, representing the whole source code can be costly. Ekelund and Engström had to give up static approaches because of model creation time. This statement should be nuanced with our experiments, where running static approaches is 12 times faster than running the tests with Jacoco, *i.e.,* about 30 minutes (from an initial six hours).

After an analysis of 36 studies on test case selection, Engström *et al.* (2010) conclude that the empirical evidence for differences between the techniques is not very strong, and sometimes contradictory. As summary, there are no bases to select a technique as superior to the other. Techniques have to be tailored to specific situations.

## 3 Test Selection in Industrial Context

While experimenting with some existing test case selection tools, we were confronted with different issues. We are presenting here these issues that we classified in four categories. But before that, we wish to clarify our context to help understand the issues. The projects used in our analysis will be presented in more details in Section 4.

Most of the projects of the company are written in Java, we therefore limited ourselves to this language or at least to the Object-Oriented paradigm (note: Biswas *et al.* (2011) and Engström *et al.* (2010) also consider procedural applications).

The projects use diversified tools and frameworks, that can be off-the-shelf (EJB, Hibernate, Spring, Tomcat...) or proprietary (in-house development). Also these projects use a client/server architecture for the web. Finally, the projects often access databases (mainly Oracle) through Hibernate.

Part of the applications are generated through some kind of Model Driven Development (MDD) approach. Biswas *et al.* (2011) describe some approaches for test case selection in the presence of MDD. We will not work with them, instead we consider source code representations. This is justified because we found that often in industrial projects, the code has been modified manually after a first generation and is no longer synchronized with the model.

As already explained, we will mainly consider a granularity at the method level. This choice will be further discussed by Research Question 1 (see Section 4.1).

### 3.1 Proposed Classification of Issues

Problems in test case selection approaches arise when there is a break in the dependency graph representing the system. Such breaks may occur for several reasons. In our case, we identified four categories of reason. Note that this list might not be exhaustive. Some categories may be specific to the static or dynamic approaches.

Third-party breaks: The application uses external libraries or frameworks for which the source code is not available. In this case, a static analysis of the code cannot trace dependencies through the third-party code execution.

Multi-program breaks: The application consists in several co-operating programs (*e.g.,* client/server application). In this case, an analysis focused on one single program cannot trace dependencies into the other program.

Dynamic breaks: The application contains code treated as data (*e.g.,* lambda-expressions or reflexive API). Specific instructions allow to execute this code in another location than its definition. In this case, an analysis of the source code cannot yield the dependencies that will occur at execution.

Polymorphism breaks: The application uses polymorphism. In this case, a dependency analysis may reach a class on which nobody else depends because all dependencies point to a superclass of it.

We now describe the issues we were faced with while experimenting with some selection tools. In a following section, we present our experiment to quantify the impact of each issue on the test case selection results in terms of precision, recall, and cost reduction.

## 3.2 Third-Party Breaks

Third-party breaks are encountered when external source code is used in the application. Third-party can be frameworks or libraries. This category only impacts static approaches because dynamic approaches can still find in their execution traces the application methods called by the third-party.

This issue is actually protean. In some cases, we found that it could be bypassed (see below), in other cases, not so easily.

### *3.2.1 External Source Code*

*Context.* In a static approach context, using frameworks or libraries may lead to the impossibility to deduce the dependencies from this part of the application.

*Example.* Figure 2 illustrates this problem. One supposes that `methodC` has changed. The dependency chain cannot be traced through the library. On the opposite side of the dependency chain, `testMethod` depends on the library.

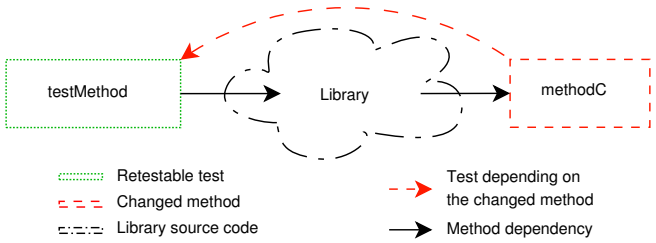


**Fig. 2** Libraries Case

*Possible solution.* In Java, the dependency graph can be analysed and built from the compiled code. Fortunately, Java bytecode is a somehow high-level language. We experimented a tool implementing such a mechanism. In other languages, or for tools working on the source code, this might be a more serious issue.

### *3.2.2 Anonymous Classes*

*Context.* It is accepted behaviour in Java development to implement callback mechanism through anonymous classes. For example, in GUI frameworks (Swing, SWT, Android), clicking on a button results in a call, by the framework, to a specific method of the application (callback). Very often, this method is implemented by an anonymous class, defined in another method of the application.

*Example.* Figure 3 exposes this issue. Test method `testMethod` depends on `methodA` which defines and instantiates `AnonymousClassB`. The method `anonymousMethodB` on the other hand depends on `methodC`, but this last one is never explicitly called from `methodA`. The dependency between `testMethod` and `methodC` can only be deduced if the containment link (link between `methodA` and `AnonymousClassB`) is included in the dependency graph. For test case selection, it is not important that `methodA` never actually calls `methodC`, all that is important is that `testMethod` depends on `methodC`, a dependency that the solution can discover accurately.

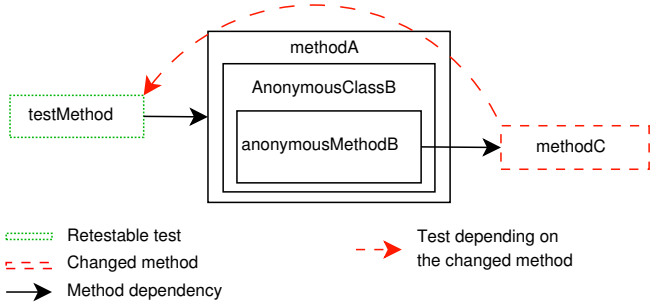


**Fig. 3** Anonymous Classes Case

*Possible Solution.* It would be easy in this case to modify the static analysis tool to include containment links in the dependency graph, whether for all classes or only in the case of anonymous classes.

### *3.2.3 Delayed Execution*

*Context.* This case is very similar to the previous one in its description, but the solution is different. In this issue, a class implements the `Callable` interface and, as such, implements the `call` method. This method can itself be called asynchronously according to different mechanisms (Future, Thread...).

*Example.* Figure 4 illustrates this issue. `testMethod` depends on `methodA` which depends on the `CallableImpl` constructor. `CallableImpl` implements `Callable` by implementing the `call` method. Finally, `call` depends on `methodB`. `methodA` uses an engine to perform the asynchronous task by adding the callable class and fetching the result. In this case, there is no static dependency between the constructor and `call`, which breaks the dependency graph.

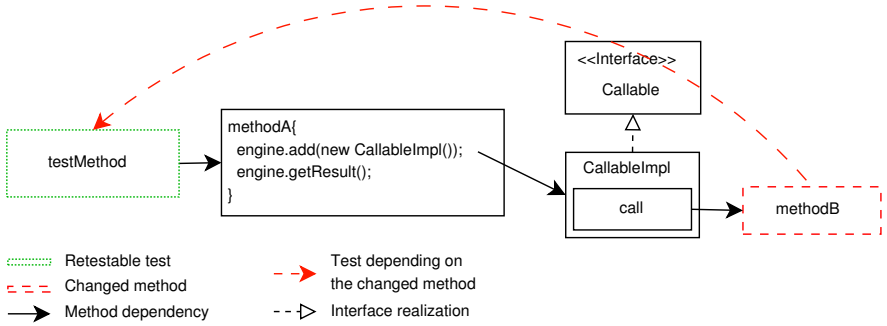


**Fig. 4** Delayed Execution Case

*Possible Solution.* It is possible to identify a chain of dependency going back to a `call` method in a class implementing the `Callable` interface. The static analysis tool may be modified to prolong this dependency chain from the class implementing the `Callable`

interface to the method that instantiates it. The reminder of the chain does not raise issue and can be traced back to the tests that depend on this method.

## 3.3 Multi-program Breaks

Multi-program breaks arise when two applications, from two distinct execution environments, interact. This category impacts both dynamic and static approaches, however, solutions for static approaches seem more practical.

### *3.3.1 Dynamic Calls Through Annotations*

*Context.* Some methods are called through source code annotations. It happens, for example, when the application is composed of a client and a server side. The server side usually exposes some methods callable by the client. For example, Java J2EE defines, on the server side, objects representing the database which can be used by a client application, through a lookup mechanism.

*Example.* Figure 5 describes the client/server problem. `testMethod` depends on `methodClient` which uses `methodServer` through a remote call. This call is possible thanks to the annotation `@Remote` defined on `ClassServer`.

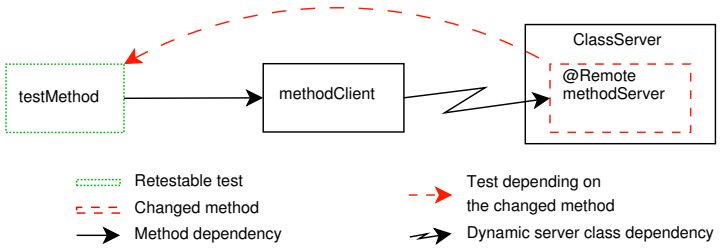


**Fig. 5** Annotation Case

*Proposed Solution.* The mapping between the client and the server classes in a dynamic approach requires to synchronize and join the execution traces of the client and the server for each individual test. With a static approach, there are usually markers in both applications (*e.g.,* `@Remote` on the server side) that allow to deduce the remote calls. Another possibility would be to consider a hybrid approach using both static and dynamic analyses.

### *3.3.2 External Tests*

*Context.* In some cases, an external framework provides its own automated tests that call the developed application (*e.g.,* SoapUI[4]). In this case, it can be difficult to instrument the code to know which test is running and to compute its execution trace. For a static analysis approach, the problem is similar to a third-party break.

*Example.* Figure 6 illustrates the problem. An external test `externalMethod` exercises `methodB`, contained in the program under test.

[4] http://www.soapui.org

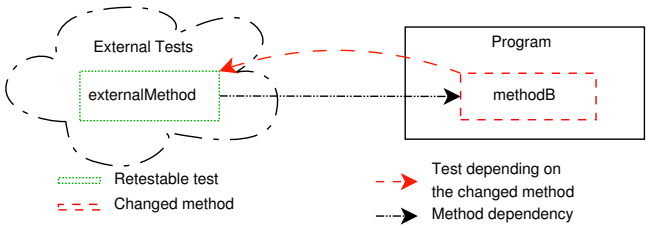


**Fig. 6** External Test Case

*Proposed Solution.* In this case, a pure dynamic approach may not be able to solve this issue because it might be difficult to separate the execution traces of each test. Again, a hybrid approach could potentially solve this issue.

## 3.4 Dynamic Breaks

Dynamic breaks arise when pieces of code are treated as data with specific instructions to execute it when needed. A break might occur because the execution can be located in a entirely different location than the definition of the code. This category of issue only impacts static approaches.

### *3.4.1 Dynamic Execution*

*Context.* Some programming languages, for example with a reflexive API like Java, allow developers to invoke code dynamically from a string.

*Example.* Figure 7 illustrates the problem. `testMethod` depends on `methodA` which invokes dynamically `methodB`. Despite the invocation, there is no link in the dependency graph between `methodA` and `methodB`.

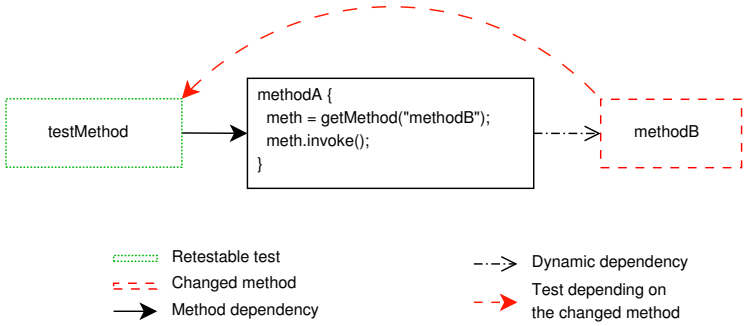


**Fig. 7** Dynamic Execution Case

*Proposed Solution.* In this case, a pure static approach would be difficult in a generic case. However, uses of the reflexive API are very rare in real cases.

### *3.4.2 Lambda-Expression*

Lambda-expressions are anonymous methods. They can be passed as method parameters and therefore executed anywhere. In Java, they can be used since version 8, but many other languages also support them.

On first sight, it seems to be an issue for a static approach because the lambda-expression can be executed anywhere in the program. However, to be used, a lambda-expression has to be declared. It means that a test depending on it must first invoke the method declaring it. Therefore, we are back to normal static analysis position considering that a possible call is equivalent to an actual call.

Another view on the problem is to consider that lambda-expressions are recent in Java, and as such not present in legacy code. They are also more advanced programming artefacts that would not be found in typical information systems.

### *3.4.3 Attribute Automatic Initialization*

*Context.* The declaration of an attribute may include its initialization through a method call. This call is performed directly in the class and not in a method scope when a new instance is created. As such, the dependency might be to the class itself and not the method called during the initialization.

*Example.* Figure 8 illustrates this problem. `testMethod` has a dependency to `methodA` which creates an instance of `ClassB`. This class defines a class attribute, named `attribute`, which is initialized with the return of `methodB`.

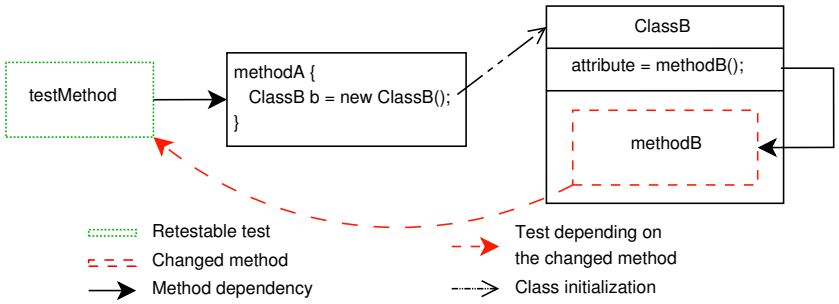


**Fig. 8** Attribute Direct Access

*Proposed Solution.* This kind of dynamic execution can be solved in a static way. For the classes that exhibit this problem (attribute initialisation by calling a method), the creation of instances (use of `new`) should be treated as a dependence to the method.

### *3.4.4 Static Attribute Initialization*

This issue is a variation of the previous problem where `attribute` is static. The solution seems to be the same as the static attribute is likely initialized only when the class is actually used (through a call to `new`).

### 3.5 Polymorphism Breaks

Polymorphism breaks are specific to object-oriented applications. This category of issue only impacts static approaches.

*Context.* In order to specify the public methods of a class, developers often use interfaces declaring these methods. The methods of the class are not called as such but through a receiver with the interface as its type. This problem is also encountered in case of inheritance where a subclass can override a superclass method.

*Example.* Figure 9 presents the problem. `testMethod` depends on `ClassInterface.method`. And `ClassImpl.method` depends on `methodA`. `ClassImpl` implements `ClassInterface`. This implementation link is absent in the dependency graph.

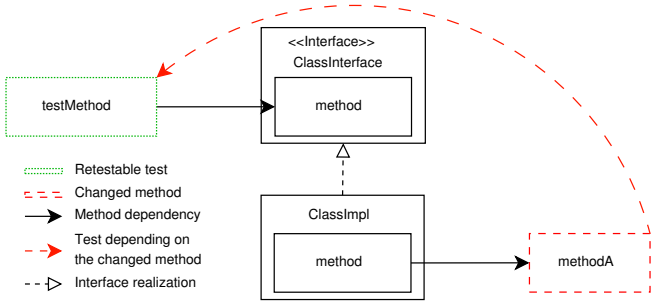


**Fig. 9** Tests Selection Approach Through Interfaces

*Proposed Solution.* The static analysis tool can be modified to include the implementation link in the dependency graph to trace back the dependency chain.

## 4 Experimental Setup

As Worldline is considering putting in practice test case selection, we experimented with large representatives projects of this company, written in Java, with the idea of evaluating the impact of the various problems identified in Section 3. For this purpose, existing open-source tools have been used. This section presents the tools and software projects used to carry out these experiments.

### 4.1 Experimental Protocol

We made our experiments at method level. The idea was to strike a balance between accuracy of the selection on one hand and processing time and data size on the other hand. This choice is evaluated in Research Question 1 (see below).

We will be looking for the answers to the following Research Questions:

**RQ1:** What is the impact on the results of a chosen granularity (Section 2.3)?
**RQ2:** What is the impact on the results of the third-party breaks issue (Section 3.2)?
**RQ3:** What is the impact on the results of the dynamic breaks issue (Section 3.4)?

**RQ4:** What is the impact on the results of the polymorphism breaks issue (Section 3.5)?
**RQ5:** What is the impact on the results of combining the solutions to different problems?
**RQ6:** What is the impact on the results of changing the same methods repeated times (as occurs in real life)?
**RQ7:** What is the impact on the results of considering real commits (that change several methods jointly)?

We will not investigate the impact of the multi-program breaks issue (Section 3.3) because we could not compute the required baseline on which to compare to. As we saw, this category of problems impacts the dynamic approach which is used to create the baseline. This is a study we plan to do in a future experiment on dynamic approach.

All experiment follow the same pattern that we will illustrate with RQ1. To assess the impact of the chosen granularity level we will compare the results of two similar experiments at different granularity level. In this case, it will be one experiment at the class granularity level and another at the method granularity level. Results at method granularity level involve:

i. We fixed one version of the source code on which we work. This version never changes, all changes are virtual. More or less we will ask the question "If Java method `m()` was changed, would we be able to recover the tests that cover it?" This decision was necessary because fetching the source code and the dependencies, recompiling, and running the tests for one version is resource intensive and could not be computed in reasonable time and space for such a large experiment.
   Test coverage is given by the dynamic approach (Jacoco tool) that is our baseline. The coverage from Jacoco for the baseline is perfect because our version of the source code never actually changes. The baseline is only computed once.
ii. We consider as "changed" each and all Java method of the application that is covered by at least one test. Considering one static approach, we try to recover the test cases covering this Java method.
iii. From the test cases recovered, we compute different metrics (see Section 4.4)
iv. The metrics values are averaged over all Java methods (covered by at least one test) to produce a result for the static approach considered.
v. The same process is repeated for another static approach and we compare their respective results to answer the research question. The difference in the results is considered as the impact of the problem that one of the two static approaches solves.

To answer RQ1, we apply the same static approach twice, once on all Java methods (Steps ii. and iv.), and once on all Java classes (Steps ii. and iv., replacing "Java methods" by "Java classes").

To answer RQ2, we apply one static tool (Infinitest) that can overcome the third party break issue and another one (Moose) that cannot.

To answer RQ3, RQ4, and RQ5, we apply the same static tool (Moose) including or not the solutions to the different problems considered. For RQ5 (combining all solution), we will not be able to include the solution to the third party break issue, because it cannot be easily done with Moose.

To answer RQ6, we apply both static tools (Infinitest and Moose) on all Java methods. The difference in the two static approaches is in the way the metrics results are averaged (Step iv.). In one case, we use a weighted mean where each Java method has a weight corresponding to the number of commits (in the history of the system) where it appears in. A Java method must appear in at least in one commit (at its creation) but may be modified frequently (more than one hundred times for some cases). The weighted mean is considered more realistic as in any system, all methods are not changed with the same frequency.

Finally for RQ7, we apply the same static tools (Infinitest and Moose) on all Java methods in one case and all system commits in the other case. As for RQ6, we use all commits in the history of the system that touched at least one method appearing in the version of the code we use. Commits differ from individual methods in that they may change many Java methods (up to 125 in one case). Commits can give better results because some Java methods with good results can "cover" for some other Java methods in the same commit with bad results. For example, one Java method suffering from a polymorphism break would still see its own tests recovered because another Java method not suffering from this problem was changed in the same commit. Again commits are considered more realistic than individual Java methods. We believe this "projection" of past commits on one code version is acceptable because they still indicate that several Java methods were changed together and we would like to know if we would be able to recover all the tests if that were the case again.

## 4.2 Projects

To perform our experiments, we selected three projects (P1, P2 and P3). P1 and P2 are financial applications with more than 400 KLOC. P1 is a service (in term of Service Oriented Architecture (SOA)) dealing with card management. P2 is an issuing banking system based on SOA and reusing the card management system developed in P1 (P2 uses P1 as a third party). P3 has no relation with the two other projects, and is an e-commerce application. P2 and P3 test suites are mainly composed of integration tests, that ensure the good behaviour of the application with its dependencies and the data base. P1 test suite includes mainly unit tests that guarantee the results of the algorithms. In these projects, each test is a Java method using JUnit[5]. In project P1, the external code is mainly composed of libraries. Project P2 and P3 use frameworks, including P1, making frequent inversion of control[6] and sub-classing. Table 1 gives some detailed metrics on the size of the projects and their test suites:

**KLOC Core:** thousands of lines of code implementing the features of the application, excluding the tests;
**KLOC Test:** thousands of lines of code to test the behaviour of these features;
**KLOC Covered Core:** thousands of lines of code covered by at least one test;
**#Green Tests:** number of tests that are green on the version of the application considered for our experiments;
**#Method:** total number of methods in the application;
**#Covered Methods:** number of methods with at least one line of code covered;
**Avg Methods/Test:** average number of covered methods by test;
**Avg Tests/Methods:** average number of tests covering a method which is covered by at least one test;
**#Commits:** number of commits for each project;
**Avg Methods/Commit:** average number of core methods changed by commit;
**Repository Creation:** year of creation of the repository (remember however that we only consider commits touching a method that still exist in the fixed code version).

P1, P2 and P3 are big applications (hundred of KLOC). P1 includes 5,323 valid tests; P2, 168; and P3, 3,035. In P1, the tests cover 4,720 methods (48%), in P2, only 3,261 methods (6%), and in P3, 8,143 methods (18%). One could think that these are rather low coverage

[5] http://junit.org/
[6] The client does not call the framework but the framework calls the client.

**Table 1** Global metrics of projects P1, P2 and P3

| Metric | P1 | P2 | P3 |
|---|---|---|---|
| KLOC Core | 447 | 716 | 302 |
| KLOC Tests | 184 | 48 | 74 |
| KLOC Covered Core | 97 | 49 | 74 |
| #Green Tests | 5,323 | 168 | 3,035 |
| #Method | 9,808 | 56,661 | 45,671 |
| #Methods Covered | 4,720 | 3,261 | 8,143 |
| Avg Methods/Test | 83 | 134 | 152 |
| Avg Tests/Method | 54 | 2 | 35 |
| #Commits | 2,217 | 467 | 2,115 |
| Avg Methods/Commit | 24 | 129 | 37 |
| Avg Files/Commit | 7 | 18 | 17 |
| Repository Creation | 2009 | 2015 | 2009 |

values (particularly P2). Two facts may explain this. First, the projects are old and date back a period when there was a less stronger emphasis on automated testing. Second, we could not use all the tests, either because they were based on specialized tooling that we could not instrument (see Section 3.3.2), or because they failed. For these projects, only the tests handled by the dynamic approach, are considered for the experiment, *i.e.,* those that run without crashing on the project. We were surprised to find that some tests in each project were failing outright. This impedes to use the dynamic approach (no execution trace) and thus to use them in our experiments. The values of the KLOC Covered Core and #Green Tests metrics, in Table 1, only take these tests into account. P1 and P3 are 6 years older than P2, with respectively 2,217, 467, and 2,115 commits that we considered. P2 is actually a rework on some legacy code dating back from 2010. However, we were not able to recover the commits from the early version of the project. These projects are still alive and evolving regularly.

Test execution (compilation and test execution included) requires 3 hours for P1; 2 hours for P2; and 30 minutes for P3. This only includes the tests that we are considering in our experiments, not all tests of the projects. This time is mainly due to the setup of each test (database population, server startup and configuration); test data volume; and the fact that there are abnormal conditions tests (timeout).

Commits seem rather big: 100+ methods, 18 files. One of the possible and positive outcome of our whole research project would be to see the developers doing smaller commits that would be easier to test.

### 4.3 Dynamic and Static Approaches Tooling

For the dynamic approach, we used a coverage tool named Jacoco[7] (Lingampally *et al.*, 2007). This tool aims to compute the test coverage of an application. For this purpose, the Java Virtual Machine (JVM) is instrumented by adding an agent to add behaviour to the source code and to record method dependencies during the tests execution. No recompilation nor modification of the source code is needed. However, a synthesis of the results is needed after the execution. It can have an impact on the execution time. For the studied projects, about one hour is needed for this tests coverage synthesis for each project.

[7] http://eclemma.org/jacoco/

However the data provided by Jacoco is not directly usable for our experiments. Information concerning the different tests are mixed up. We modified the tool in order to separate information relative to each test and thus know what test cover what Java methods. An aspect has been implemented to start the recording at the beginning of each test and serialize the coverage results at the end. The method dependencies associated to each test are thus available. As already explained (Section 4.1) this information is always up-to-date because we never actually change the source code.

For the static approaches, two different tools were used: the first one, Infinitest[8], analyses the bytecode; the second one, Moose[9] (Ducasse *et al.*, 2000), analyses the source code.

Infinitest is a tool integrated in a Java IDE (Eclipse or Intellij). It is precisely intended to do test case selection, but, the algorithm is slow on complex projects. For the purpose of the experiment, Infinitest has been modified to be used in a standalone mode rather than through an IDE.

Infinitest works at class granularity level (see Section 2.3 and RQ1, Section 4.1). For a given class, it gathers all references to other classes (*i.e.,* the classes of the methods invoked, the types used, the annotation types...) and thus provides a graph of class dependencies. It also selects tests at the class level, that is to say not individual test methods, but their classes.

Moose relies on the FAMIX meta-model (Ducasse *et al.*, 2011) and proposes to represent source code entities in a model. This model gathers entities such as packages, classes, methods, and the links between them (invocations, references, inheritances, and accesses); statements are omitted. A method dependency graph, linking a changed method to the tests, is thus available.

Moose is a tool dedicated to pure static analysis, and so does not need any compilation of the source code. However, the source code has to be parsed to create the model. This parsing can take up to several minutes for large applications. In our experiment, the result overhead was smaller than for the other approaches.

## 4.4 Metrics

An optimal approach selects the smallest set of all test cases covering a change. According to the experiment protocol (Section 4.1), the methods are not changed but considered as such each in turn. As a consequence, the dynamic approach is both precise and safe: all the tests selected by the dynamic approach cover the changes; no other test covers a given change.

Due to the issues we identified, static approaches may miss some tests covering a change and select others that do not cover it. To compare the approaches, we use four metrics that can be computed from the traditional quantities:

- **True Positives** ($TP$) is the number of tests selected by the static approach and the dynamic approach;
- **False Positives** ($FP$) is the number of tests selected only by the static approach;
- **False Negatives** ($FN$) is the number of tests selected only by the dynamic approach;
- **True Negatives** ($TN$) is the number of tests selected neither by the dynamic nor by the static approaches.

From these quantities we compute the following metrics: *Selected tests*, *Precision*, *Recall*, and *F-Measure*.

[8] http://infinitest.github.io/

[9] http://www.moosetechnology.org/

*Selected tests* represents the number of tests selected by the approach as a ratio. It compares the number of selected tests to the total number of tests. This metric corresponds to $1 - cost\ reduction\ criteria$ as defined in Engström *et al.* (2010) (see Section 2.4).

$$Selected\ tests = \frac{TP + FP}{TP + FN + TN + FP}$$

*Precision* is the fraction of retrieved tests that are retestable (see Section 2.2). A high precision means that the static approach selects essentially tests that cover the changed method.

$$Precision = \frac{TP}{TP + FP}$$

*Recall* is the fraction of retestable tests that are retrieved. A high recall means that the approach is safe and that the tests covering a given changed method are selected by the static approach.

$$Recall = \frac{TP}{TP + FN}$$

*F-Measure* is the harmonic mean of *Precison* and *Recall* to show the overall performance of an algorithm.

$$F - Measure = \frac{2 * Recall * Precision}{Recall + Precision}$$

*Precision* and *Recall* metrics are strongly linked together. Enhancing the precision of an approach often decreases its recall, and vice-versa. Priority will be given to a higher *Recall* to make sure one does not overlook a test that could discover a bug in the changed code. However, achieving good recall is easy by selecting many tests. This would show up in the *Precision* and *Selected tests* metrics. We must remember that our goal is ultimately to be able to give rapid feedback to the developer right after he commits a change.

## 5 Results and Discussion

This section presents our experimental results to answer the seven research questions. Table 2 gives the metrics for each static approach.

**Table 2** Comparison of the static approaches to the dynamic one for test case selection considering all Java methods individually

| | | Selected Tests | | | Precision | | | Recall | | | F-Measure | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | P1 | P2 | P3 | P1 | P2 | P3 | P1 | P2 | P3 | P1 | P2 | P3 |
| Jacoco (dynamic) | | 0.8% | 1% | 0.4% | - | - | - | - | - | - | - | - | - |
| Infinitest | RQ2 | 23% | 5% | 3% | 9% | 39% | 15% | 72% | 66% | 44% | 12% | 43% | 18% |
| Moose for Infinitest | | 19% | 2,2% | 2% | 10% | 27% | 18% | 70% | 63% | 41% | 13% | 44% | 20% |
| Moose (classes) | RQ1 | 8% | 1% | 0.7% | 15% | 23% | 13% | 50% | 37% | 19% | 18% | 26% | 12% |
| Moose (methods) | | 0.4% | 0.1% | 0.1% | 43% | 11% | 24% | 36% | 11% | 13% | 35% | 11% | 15% |
| Moose w/ delayed exec. | | 0.4% | 0.1% | 0.1% | 43% | 11% | 24% | 36% | 11% | 13% | 35% | 11% | 15% |
| Moose w/ anonym. classes | RQ3 | 0.4% | 0.1% | 0.1% | 43% | 11% | 24% | 36% | 11% | 13% | 35% | 11% | 15% |
| Moose w/ attributes | | 0.4% | 0.2% | 0.1% | 43% | 17% | 24% | 36% | 17% | 13% | 35% | 17% | 15% |
| Moose w/ polymorphism | RQ4 | 3% | 0.4% | 2% | 43% | 25% | 34% | 91% | 26% | 41% | 50% | 25% | 29% |
| Moose w/ att. & anon. & polym. & delayed exec. | RQ5 | 3% | 0.8% | 2% | 43% | 61% | 34% | 91% | 64% | 41% | 50% | 62% | 29% |

### 5.1 RQ1 – Class vs Method Granularity

To answer this Research Question, we consider Moose approaches at method and class granularity.

At class granularity, we get more *Selected tests* for the three projects (resp. 8%, 1%, and 0.7%). Lower is better for this metric, the "right value" being the results for Jacoco (dynamic approach, our baseline). For example, 8% for P1 at class granularity means that the approach selects 10 times more tests that what is required (0.8% for Jacoco). A more complete experiment on total time reduction (see Section 2.4) would need to be conducted to decide whether the ratio of selection the approaches could achieve are enough to give rapid feedback to the developers. The results for the method granularity (<0.5%) do seem very interesting.

*Precision* is higher at method granularity for P1 and P3 (43% for P1; 24% for P3) and lower for P2 (11% at method granularity and 23% at class granularity). For P1 and P3, the method granularity has a clear advantage with higher *Precision* and much less *Selected tests*. P2 behaves as expected: more *Selected tests* would usually result in lower *Precision*.

The three projects have better *Recall* at class granularity (resp. 50%, 37%; and 19%). Again this is conform to expectations as more *Selected tests* would usually result in higher *Recall*.

We said we would prefer *Recall* over *Precision* which would point toward selecting the class granularity level. But the higher rates of Selected tests could be a show stopper.

Results of the *F-Measure* are intended to help striking a balance between good *Recall* or good *Precision*, in our experiments they are not clear: for P2, *F-Measure* is better at class granularity, for P1 and P3, it is better at method granularity.

Note that the *Precision* and *Recall* results do not seem very good overall. This could be due to the different issues identified previously and that are not treated in this experiment.

Since we prefer recall over precision, there does seem to exist some arguments in favour of class granularity, but our decision to work at Method granularity cannot be ruled out either.

### 5.2 RQ2 – Third-Party Breaks Impact

To answer this Research Question, we consider Infinitest (that overcomes this issue) and Moose (that does not). Because Infinitest works at the class level, we do the same for Moose. Infinitest considers some dependencies (*e.g.* references to the classes) that we did not consider in the previous experiment (line "Moose (classes)" in Table 2). This is why a different line was created here (line "Moose for Infinitest") that does consider such dependencies and will give better ground for comparison. Infinitest has higher *Selected tests* for the three projects, but the difference is less important than in the previous experiment for example. Again, for P1 and P3, the *Precision* is better for the experiment with less tests selected and P2 behaves as would be expected (higher *Precision* when there are less tests selected). There might be some special condition on P1 and P3 that makes them behave this way.

*Recall* is better for Infinitest for the three projects (resp. 72%, 66%, and 44%) which is normal since it selects more tests. The difference however is small as will be seen when looking at the next metric.

*F-Measure* is more consistent and gives better results for the three projects to Moose (not solving the third party breaks).

Based on the *F-Measure* results, one could conclude that there is no urgent need to solve the third party issue on our three projects. The *Recall* results are slightly lower (from 72% to 70% for P1; from 66% to 63% for P2; and from 44% to 41% for P3), but this could be acceptable since the time to run the selected tests will be shorter.

### 5.3 RQ3 – Dynamic Breaks Impact

For this Research Question, three Moose approaches (at method granularity level) bypassing different dynamic break issues are experimented. They are to be compared with the Moose approach at method granularity level (line "Moose (methods)" in Table 2).

For the three projects, we see almost no change between Moose solving one of the specific Dynamic Break issues and Moose not solving any issue. The only exception is slightly more *Selected tests* for P2 when solving the Attribute Automatic Initialization (Section 3.4.3) issue, followed by significantly better *Precision*, *Recall* and consequently *F-Measure*.

The first conclusion would be that it is mostly useless to try to solve these issues. We will see however in Section 5.5 that issues may be intertwined and that solving one alone might not be enough.

### 5.4 RQ4 – Polymorphism Breaks Impact

For this Research Question, Moose approach at method granularity is compared to the Moose approach (at the same granularity) bypassing the polymorphism issue.

The three projects have more *Selected tests*. *Precision* improves significantly for P2 and P3 and remains equal for P1. Again an improvement here is unexpected since we selected more tests. *Recall* improves dramatically for P1, from 36% to 91% and significantly for P2 and P3. And of course *F-Measure* improves also for the three projects.

The conclusion is that it was very important to solve this specific issue in our cases. P1 particularly shows excellent results, with $>$90% *Recall*, a still good *Precision* (43%, about half of the selected tests do cover the changed method) and a similarly good rate of *Selected tests*.

### 5.5 RQ5 – Impact of Combining Solutions

For this research question, we look at Moose approaches combining solutions for anonymous classes, delayed execution, polymorphism, and dynamic break issues. This will be compared to the bare Moose approach at method granularity with no solution implemented, as well as other Moose approaches with any individual solution included.

The combination of all implemented solutions gives very good results. Overall, the results for P1 and P3 are similar to the ones of the previous experiment (RQ4, Section 5.4) and P2 is showing more *Selected tests*, much better *Precision*, *Recall*, and *F-Measure*.

Results for P2 show that combining solutions to several issues at the same time improves the situation in a way that would not be expected if one looks at the individual results of each solution. The conclusion we draw from this is that issues are intertwined and must be solved jointly.

Another conclusion is that by answering all the issues (minus the third party breaks that Moose cannot solve easily), we end up with very good results, *Precision* ranges from 34%

(P3) to 61% (P2), and *Recall* ranges from 41% (P3) to 91% (P1). These results position the static approach as a viable solution to the test case selection problem.

### 5.6 RQ6 – Weighting of Results with the Number of Commits

For this experiment, we summarized the results in a new table (Table 3) that should be compared to the first one. Although we give the results of all experiments for the sake of completeness, we will be focusing on the last line in our discussion. In these new experiments, the results of each methods are weighted according to the number of commits the method appears in (Section 4.1). The idea is that the most committed Java methods could have consistently good (or bad) results.

**Table 3** Comparison of the static approaches to the dynamic one with a weighting of Java methods with the number of commits they appear in

| | | Selected Tests | | | Precision | | | Recall | | | F-Measure | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | P1 | P2 | P3 | P1 | P2 | P3 | P1 | P2 | P3 | P1 | P2 | P3 |
| Jacoco (dynamic) | | 1% | 2% | 1% | - | - | - | - | - | - | - | - | - |
| Infinitest | RQ2 | 18% | 5% | 4% | 7% | 35% | 14% | 59% | 61% | 39% | 9% | 40% | 16% |
| Moose for Infinitest | | 15% | 2% | 2% | 9% | 28% | 17% | 57% | 59% | 38% | 11% | 40% | 19% |
| Moose (classes) | RQ1 | 5% | 1% | 1% | 14% | 21% | 13% | 40% | 35% | 17% | 15% | 23% | 11% |
| Moose (methods) | | 0,3% | 0,2% | 0,2% | 36% | 13% | 23% | 27% | 12% | 12% | 27% | 12% | 13% |
| Moose w/ delayed exec. | | 0,3% | 0,2% | 0,2% | 36% | 13% | 23% | 27% | 12% | 12% | 27% | 12% | 13% |
| Moose w/ anonym. classes | RQ3 | 0,3% | 0,2% | 0,2% | 36% | 13% | 23% | 27% | 12% | 12% | 27% | 12% | 13% |
| Moose w/ attributes | | 0,3% | 0,2% | 0,2% | 36% | 16% | 23% | 28% | 17% | 12% | 27% | 16% | 13% |
| Moose w/ polymorphism | RQ4 | 3% | 0,4% | 2% | 42% | 27% | 33% | 92% | 28% | 39% | 50% | 27% | 28% |
| Moose w/ att. & anon. & polym. & delayed exec. | RQ5 | 3% | 1% | 2% | 42% | 59% | 33% | 92% | 62% | 39% | 50% | 60% | 28% |

In summary, the results (last lines of tables 2 and 3) are not very different. This new experiment consistently brings marginal decrease in *Precision* and *Recall* and small increase in *Selected tests*. All these results points towards a worsening of the results.

This would suggest that the methods where static approaches are not able to find the tests are more frequently committed. This is not good news, but the differences are small (typically one percentage point) and would need to be more formally tested in a specific experiment.

### 5.7 RQ7 – Aggregation of the Results by Commit

To answer this last Research Question, we again replicate all experiments, but working with commits instead of individual methods. All the results are summarised in Table 4 but we will concentrate on the last line and compare it to the one in Table 2.

The first observation regards the number of *Selected tests*. Since commits comprise many Java methods (average for P1 is 24, for P2 it is 129, see Table 1), it is expected that more tests would be selected. This is the case with our baseline (Jacoco) with a larger percentage of all tests selected (P1, from 0.8% to 8%; P2, from 1% to 21%; P3, from 0.4% to 14%). This is an increase in the range of an order of magnitude. However, we see that the

**Table 4** Comparison of the static approaches to the dynamic one to test case selection, considering Java methods grouped in commits

| | | Selected Tests | | | Precision | | | Recall | | | F-Measure | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | P1 | P2 | P3 | P1 | P2 | P3 | P1 | P2 | P3 | P1 | P2 | P3 |
| Jacoco (dynamic) | | 8% | 21% | 14% | - | - | - | - | - | - | - | - | - |
| Infinitest | RQ2 | 13% | 8% | 7% | 25% | 26% | 33% | 62% | 51% | 62% | 28% | 30% | 35% |
| Moose for Infinitest | | 9% | 3% | 5% | 10% | 36% | 28% | 30% | 42% | 44% | 11% | 28% | 28% |
| Moose (classes) | RQ1 | 6% | 2% | 3% | 19% | 25% | 34% | 32% | 36% | 38% | 19% | 25% | 29% |
| Moose (methods) | | 0,8% | 0,7% | 0,9% | 54% | 30% | 63% | 36% | 21% | 33% | 39% | 23% | 38% |
| Moose w/ delayed exec. | | 0,8% | 0,7% | 1% | 54% | 30% | 63% | 36% | 21% | 33% | 39% | 23% | 38% |
| Moose w/ anonym. classes | RQ3 | 0,8% | 0,7% | 1% | 54% | 30% | 63% | 36% | 21% | 33% | 39% | 23% | 38% |
| Moose w/ attributes | | 0,9% | 0,7% | 1% | 54% | 30% | 63% | 36% | 21% | 34% | 39% | 23% | 38% |
| Moose w/ polymorphism | RQ4 | 4% | 2% | 6% | 55% | 34% | 49% | 81% | 31% | 56% | 56% | 29% | 42% |
| Moose w/ att. & anon. & polym. & delayed exec. | RQ5 | 4% | 3% | 6% | 55% | 45% | 49% | 81% | 45% | 56% | 56% | 40% | 42% |

static approaches (mainly at the method granularity level) tend to exhibit a smaller increase in the number of selected tests (P1, from 3% to 4%; P2, from 0.8% to 3%; P3, from 2% to 6%). So even-though the static approaches selected more tests, one could conclude that they are actually more selective than necessary here.

P1 and P3 improve their *Precision* (resp. from 43% to 55%; and from 34% to 49%), but P2 decreased (from 61% to 45%). So good news for P1 and P3, that are more selective but also more precise.

Being more selective, the *Recall* results were bound to worsen: the approaches select less tests than they should according to our baseline. This is what happens with P1 and P2 (resp. from 91% to 81%; and from 64% to 45%), but P3, which previously had the lower *Recall*, improved it (from 41% to 56%). This is coherent with P3 also improving its *Precision*.

One conclusion is that it does not seem to be the case that one Java method in a commit "covers" for another one. That might indicate that the commits touch various concerns for which different subsets of tests are necessary. That would be coherent with the large size of the commits that we already mentioned.

Considering commits instead of individual Java methods also means that Java methods are weighted, just like in the preceding experiment. We saw a very small worsening of the results when weighting the Java methods and this could account for a part of the bad results here: it is possible that Java methods with bad results are committed more often than the other ones.

### 5.8 Overall Conclusions

We draw three overall conclusions from these experiments.

First, problems might be intertwined and one needs to combine several solutions together to fully resolve any of the issues.

Second, problems do not have the same impact on the projects. This might be the consequence of different coding conventions or rules. For example the attribute initialization issue (Section 3.4.3) is not present in P1. Such issues might be helped by establishing better coding conventions.

Third, considering commits instead of individual Java methods tend to worsen the results with approaches that are too selective to keep the same level of good results. The large size

of the commits might be an important factor in this behaviour. As already stated, a positive consequence of the entire experiment (still in progress) would be to see developers make smaller commits that would help in test case selection and would also give them better and faster feedback on their changes.

Another conclusion from Section 3.1 would be that even in one category, issues can unfortunately be very different and require each a specific solution.

For our experiments, we used the dynamic approach as oracle because it is safe and accurate. However, this approach has two major drawbacks: First, this approach is not generic; it depends strongly from the data used for the tests. Second, if a test is failing, no execution trace is recorded and it cannot be selected by this approach.

## 6 Related Works

Two other studies (Ekelund and Engström, 2015; Soetens *et al.*, 2013) implement tests selection approaches and provide the same metrics as for our experiment (ratio of selected tests, precision, and recall). Table 5 gathers the results of these studies.

**Table 5** Comparison of the static approaches to the dynamic one to select the tests after a method change

| | Project/Approach | Ratio of Selected Tests | Precision | Recall |
|---|---|---|---|---|
| Soetens *et al.* | PMD | 1% | 83% | 58% |
| | Cruisecontrol | 1% | 87% | 77% |
| Ekelund and Engström | Wide approach | 37% | 1.5% | 95% |
| Ekelund and Engström | Narrow approach | 4% | 7.4% | 79% |

Soetens *et al.* (2013) propose a static approach at method granularity based on the FAMIX meta-model. Their approach relies on real change sets gathered in commits. For each of these change sets, their static approach is compared to a dynamic one used as reference. The dependency graph they use only contains links between methods. Two open-source applications (PMD and Cruisecontrol) are used as input data for their experiment. 1% of the test cases is selected for both applications. They obtain respectively for each application a recall of 77% and 58% and a precision of 84% and 83%. These results are better than our Moose/method approach. First, P1 and P2 are wider projects than PMD and Cruisecontrol that counts around 20 less times lines of code. Second, P1 and P2 use a lot of frameworks and libraries what seems not the case for PMD and Cruisecontrol. However, by solving the identified issues, we obtain close results to Soetens *et al.*.

Ekelund and Engström (2015) select tests based on test result history. This history archives changes at package granularity and corresponding test results for each build of the application (*i.e.,* the execution of all the test cases by a continuous integration server). Such an approach has been defined since no other existing approach based on source code, bytecode, or dynamic analysis was possible due to the huge size of the studied application that counts several million of lines of code. When a package changes, thanks to history data mining, the authors know the potentially affected tests and select them. These tests have at least once failed when, in the past, this package changed (narrow approach) or whatever the package changed (wide approach). The accuracy of the selection algorithm is related to the number of builds used. However, considering a too large history may introduce noise in the selection mechanism since the source code may have evolved a lot. The authors found that the algorithm is optimal for a history containing 100 builds. This approach is language

independent and uses few resources but relies on a history of the build results. Such a history does not often exist in companies and requires time and effort to be built. In the case of the wide approach, the ratio of selected tests reaches 37%, the precision and the recall are respectively 1.5% and 95%. In the case of the narrow approach, only 4% of the tests are selected with a precision of 7.4% and a recall of 79%. The experiments of Ekelund and Engström lead to recall with the same order of magnitude than ours. However, precision results are very low, because they work at package level.

Parsai *et al.* (2014) extend Soetens *et al.* (2013) approach to integrate the resolution of polymorphism issue. As a simple workaround to this issue, they consider all methods that a call can possibly be referring to. Our approach is very similar, except that we only add possible methods of the class hierarchy. As a consequence, the ratio of selected tests strongly increases to reach 13% for PMD and 37% for Cruisecontrol. The increase is not so high in our cases but even so exists. The other results are not comparable since they do not use the same metrics as us. In Soetens *et al.*, the authors deal with groups of changes gathered in commits in a first experiment presented in this paper and also use mutation testing. Parsai *et al.* approach deals only with mutation testing whereas we consider that each individual method is at its turn modified.

Gupta *et al.* (1992) use static slicing to select test cases. The changes are studied at instruction level on procedural programs. The authors use data-flow graph to select test cases from a set of changes. More precisely, the identification of tests to relaunch relies on forward and backward slices. Thanks to the data-flow graph (representing the program variables read and written), the backward walk algorithm identifies statements containing definitions of variables that reach a point in the program. The forward walk algorithm identifies uses of variables that are directly or indirectly affected by a change in a value of a variable at a point in the program. After a change, these algorithms are used to complete the existing data flow graph and so to retrieve the tests. Static slicing, as all data-flow techniques, is imprecise, but neither data flow history nor re-computation of the data-flow for the entire program is required: only changes re-computation has to be made. This static approach seems too heavy to use on industrial applications due to the use of a instruction grained algorithm.

Zheng *et al.* (2007) improved their *I-BACCI* tool, based on binary analysis, to focus on the external source code issue. They aim to check the successful integration of a new version of the external source code only available in DLLs (Dynamic Link Libraries). An analysis of the binary representation enables them to identify the changes that occurred in it and then to select the impacted tests. This work deals with an issue that we identified but is not tackled in this paper. Static approaches based on source code can not manage it, what would have led to not comparable results between comparable approaches.

Huang *et al.* (2011) focus on what we call multi-program break category. More specifically, they provide an approach to enhance the method dependency graph in case of J2EE applications using JSP (JavaServer Pages) to implement web pages. For this purpose, they merge two dynamic approaches, each instrumenting a part of the application: one deals with Java, and the other with JSP. This issue is not tackled in this paper since we considered the dynamic approach an oracle whereas this category of issue only impacts dynamic approaches.

Nanda *et al.* (2011) present a dynamic approach tackling the external source code and the multi-program breaks issues. The first category of issue is exercised by analysis the impact on the integration tests of changes occurring in the configuration file. The second category of issue is addressed through the analyse of database changes. Ratio of selected tests are respectively 57% and 36%. By this experiment, the authors highlight that these issues are

relevant and deserve to be solved. They also illustrate that the existing regression-testing techniques are not adapted to analyse modern software systems that are characterized by heterogeneity and environment dependence. Analysing such systems with their complexity and their reality is part of our future work.

Gligoric *et al.* (2015) describe *EKSTAZI*, a tool to select regression tests. This tool is based on a dynamic approach. Contrary to Jacoco that, thanks to our aspect, provides the test methods covering a given application method, *EKSTAZI* is able to select regression test methods from a changed method. It is thus our aspect and the way we used Jacoco (considering that each application method is arbitrarily at its turn changed), that make these two tools similar. An experiment on 20 revisions per projects on around 20 Java projects is performed at class and method granularity. The ratio of selected tests is lower at method granularity (8%) than at class granularity (11%). However, time is a lot more reduced in case of class granularity. These results cannot be directly compared with the ones we provide for Jacoco since it is the oracle. They have the same order of magnitude than the other provided results in this paper.

Beszedes *et al.* (2012) experiment a dynamic test case selection approach based on code change coverage very similar to Jacoco. Principles are the same; but the analysed language differs (Java for Jacoco and C++ for their tool). After a first analysis of the whole application coverage, the authors limit the study of the coverage to the parts that have changed between two commits. The studied application is an open-source web browser engine named WebKit, composed of 1.9 million of lines of C++ and of around 24 000 tests cases, over 126 commits. Contrary to our experiment, the authors do not compare their approach with another one being an oracle. The recall metric can not be computed, because they do not consider another experiment as an oracle. Instead they use the notion of inclusiveness (how many failing tests are included in the selection). This metric emphasises the modification-revealing tests. The average value of this metric is 95% what can be explained, according to the authors, by a combination of factors: non deterministic tests, determination of the changes and their correlation to C++ methods... This experiment shows that such a dynamic approach is not perfect; some tests are ignored during its analysis.

Other static approaches can be used to select the test cases. For example, Gupta *et al.* (1992) approaches is static slicing. The changes are impacted at instruction level on procedural programs. The authors use data-flow graph to select the test case from a set of changes. The technique uses the concepts of forward and backward slices to determine the tests that should be relaunched. At level of the data-flow graph (representing the program variables read and write flow), the backward walk algorithm identifies statements containing definitions of variables that reach a point in the program. The forward walk algorithm identifies uses of variables that are directly or indirectly affected by a change in a value of a variable at a point in the program. Depending of the change made, algorithms are used to retrieve the tests. Static slicing, as all data-flow techniques, is imprecise, but neither data flow history nor re-computation of the data-flow for the entire program is required: only changes re-computation has to be made. This static approach seems to heavy to use on industrial applications due to the use of a instruction grained algorithm.

## 7 Conclusion

Testing an industrial application can take several hours whereas quality would require to test often. Reducing the testing time is thus essential for companies. One solution is to reduce the number of tests to execute by selecting only tests that may possibly fail after a change.

Called by a major IT company to look how developers could get faster feedback when modifying some code, we experimented different approaches on two of their Java projects counting several thousand of lines of code. During this experiment, we met several issues that we generalized and categorized. Solutions to these issues were also provided and implemented.

From the experiment we carried out, we draw three conclusions: First, the issues we discovered might be intertwined and several solutions need to be combined together to fully resolve any of the issues. Second, problems do not have the same impact on the projects. And third, even in one category, issues can unfortunately be very different and each requires a specific solution.

Among these approaches, the dynamic one has been considered as an accurate baseline. However, this approach has two major drawbacks: First, it is not generic and so depends strongly on the data used for the tests. Second, if a test is failing, it cannot be selected by this approach.

As future work, we identified some issues that cannot be solved with a static or dynamic approach alone. We plan to solve the pending issues by experimenting a hybrid solution. Moreover, we foresee to conduct such experiments on real changes.

## References

Agrawal H, Alberi JL, Horgan JR, Li JJ, London S, Wong WE, Ghosh S, Wilde N (1998) Mining system tests to aid software maintenance. IEEE Computer 31(7):64–73, DOI 10.1109/2.689678, URL http://dx.doi.org/10.1109/2.689678

Beszedes A, Gergely T, Schrettner L, Jasz J, Lango L, Gyimothy T (2012) Code coverage-based regression test selection and prioritization in webkit. In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pp 46–55, DOI 10.1109/ICSM.2012.6405252

Biswas S, Mall R, Satpathy M, Sukumaran S (2011) Regression test selection techniques: A survey. Informatica (03505596) 35(3)

Ducasse S, Lanza M, Tichelaar S (2000) Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In: Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools, CoSET '00, URL http://scg.unibe.ch/archive/papers/Duca00bMooseCoset.pdf

Ducasse S, Anquetil N, Bhatti U, Cavalcante Hora A, Laval J, Girba T (2011) MSE and FAMIX 3.0: an interexchange format and source code model family. Tech. rep., RMod – INRIA Lille-Nord Europe, URL http://rmod.lille.inria.fr/archives/reports/Duca11c-Cutter-deliverable22-MSE-FAMIX30.pdf

Ekelund ED, Engström E (2015) Efficient regression testing based on test history: An industrial evaluation. In: International Conference on Software Maintenance and Evolution, IEEE Computer Society

Elbaum S, Kallakuri P, Malishevsky AG, Rothermel G, Kanduri S (2003) Understanding the effects of changes on the cost-effectiveness of regression testing techniques. Journal of Software Testing, Verification, and Reliability

Engström E, Skoglund M, Runeson P (2008) Empirical evaluations of regression test selection techniques: a systematic review. In: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, ACM, pp 22–31

Engström E, Runeson P, Skoglund M (2010) A systematic review on regression test selection techniques. Information and Software Technology 52(1):14–30

Ernst MD (2003) Static and dynamic analysis: Synergy and duality. In: WODA 2003: ICSE Workshop on Dynamic Analysis, Citeseer, pp 24–27

Gligoric M, Eloussi L, Marinov D (2015) Practical regression test selection with dynamic file dependencies. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2015, pp 211–222, DOI 10.1145/2771783.2771784, URL http://doi.acm.org/10.1145/2771783.2771784

Gupta R, Harrold MJ, Soffa ML (1992) An approach to regression testing using slicing. In: Software Maintenance, 1992. Proceedings., Conference on, IEEE, pp 299–308

Hsia P, Li X, Chenho Kung D, Hsu CT, Li L, Toyoshima Y, Chen C (1997) A technique for the selective revalidation of oo software. Journal of Software Maintenance: Research and Practice 9(4):217–233

Huang S, Li ZJ, Zhu J, Xiao Y, Wang W (2011) A novel approach to regression test selection for j2ee applications. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on, pp 13–22, DOI 10.1109/ICSM.2011.6080768

Kent S (2002) Model driven engineering. In: Integrated formal methods, Springer, pp 286–298

Leung HK, White L (1989) Insights into regression testing. In: Software Maintenance, 1989., Proceedings., Conference on, IEEE, pp 60–69, URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=65194

Lingampally R, Gupta A, Jalote P (2007) A multipurpose code coverage tool for java. In: System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on, pp 261b–261b, DOI 10.1109/HICSS.2007.24

Nanda A, Mani S, Sinha S, Harrold M, Orso A (2011) Regression testing in the presence of non-code changes. In: Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, pp 21–30, DOI 10.1109/ICST.2011.60

Parsai A, Soetens QD, Murgia A, Demeyer S (2014) Considering polymorphism in change-based test suite reduction. In: Dingsoyr T, Moe N, Tonelli R, Counsell S, Gencel C, Petersen K (eds) Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation, Lecture Notes in Business Information Processing, vol 199, Springer International Publishing, pp 166–181, DOI 10.1007/978-3-319-14358-3_14, URL http://dx.doi.org/10.1007/978-3-319-14358-3_14

Rothermel G, Harrold MJ (1993) A safe, efficient algorithm for regression test selection. In: Proceedings of the International Conference on Software Maintenance (ICSM '93), IEEE, pp 358–367

Soetens QD, Demeyer S, Zaidman A (2013) Change-based test selection in the presence of developer tests. In: Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, IEEE, pp 101–110

White L, Leung H (1992) A firewall concept for both control-flow and data-flow in regression integration testing. In: Software Maintenance, 1992. Proceedings., Conference on, pp 262–271, DOI 10.1109/ICSM.1992.242535

White L, Jaber K, Robinson B (2005) Utilization of extended firewall for object-oriented regression testing. In: Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, pp 695–698, DOI 10.1109/ICSM.2005.101

Willmor D, Embury S (2005) A safe regression test selection technique for database-driven applications. In: Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, pp 421–430, DOI 10.1109/ICSM.2005.15

Yoo S, Harman M (2012) Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2):67–120, DOI 10.1002/stvr.430, URL http://dx.doi.org/10.1002/stvr.430

Zheng J, Williams L, Robinson B, Smiley K (2007) Regression test selection for black-box dynamic link library components. In: Incorporating COTS Software into Software Systems: Tools and Techniques, 2007. IWICSS '07. Second International Workshop on, pp 9–9, DOI 10.1109/IWICSS.2007.8