The Reduction Process for Change Type Cases to Consider

An Annex to "Maintaining Record-and-Replay Test Cases within Pragmatic Reuse Scenarios"

Soha Makady Robert J. Walker

Department of Computer Science
University of Calgary
Calgary, Canada

15 August 2014

Technical Report 2014-1061-12

Abstract

Pragmatic reuse tasks can be validated by a custom record-and-replay (R&R) technique that transforms automated test suites to focus on the reused code, reducing the need to develop test suites manually; this technique has previously been reified in the Skipper tool. In general, R&R test suites have been criticized as lacking maintainability when the source under test evolves, but the literature contains no empirical evidence on the merits or faults of R&R unit tests relative to alternatives. The associated paper "Maintaining Record-and-Replay Test Cases within Pragmatic Reuse Scenarios" reports on an empirical study with industrial developers that: (1) evaluates whether R&R unit tests within pragmatic reuse tasks are harder to maintain than ones written manually, and (2) investigates how developers would validate pragmatic reuse tasks in the absence of R&R tests. This technical report describes details of the process for reducing the number of cases to be considered to a manageable number.

1 Case Study Overview

While evaluating Skipper's prototype, the participants raised concerns on the longer term maintainability of Skipper's transformed test cases due to their reliance on serialized objects within. Participants suspected that massive changes to the reused source code can stop the serialized objects from being created, and hence render Skipper's tests useless. As such source code changes are inevitable [7], we wanted to analyze the effect of applying various source code changes to the reused source code, on its corresponding Skipper's test cases. Following such source code changes, we wanted to assess the developers' ability to maintain Skipper's test cases in comparison to their ability to maintain their equivalent manually-written tests. Such analysis can clarify whether our approach actually provides a real longer-term return on investment for validating pragmatic reuse tasks, versus alternative means to validate reuse tasks. From our perspective, alternative means to validating reuse tasks involved either creating new test cases for the reused code, or manually reusing the originating system's test cases. Both alternatives would result in test code that does not include serialized objects.

To undertake such an analysis, we needed to come up with a comprehensive list of source code changes that can severely affect manually-written test cases, and Skipper' test cases. Such list would then be used in our maintainability study, involving industrial developers, to compare the effect of those changes on both kinds of tests from developers' perspective. We refer to that list as the *candidate changes list* throughout the rest of this report.

To populate our candidate changes list, we investigated source code changes that (i) affect serialized object to break Skipper's tests (Section 2), and (ii) affect various source code elements to break both Skipper's tests, and manually-written JUnit tests (Section 3). Such investigation resulted in a total of 55 changes in our candidate changes list. Examining all such changes in a single study was infeasible, because we needed to analyze the individual changes

on the tests to know the problematic changes. Hence, we applied a three step process on our candidate changes list; filtering (Section 4), clustering (Section 5) and sampling (Section 6) to settle on a subset of 14 changes to be used in our study.

2 Serialization-related changes

We referred to the Java Object Serialization (JOS) specification [6] to identify serialization-related changes. The JOS listed 17 type (class-level) changes that could affect serialization. Those changes were described in terms of streaming an object (i.e., serializing an object) from one version of some class definition, then reading that stream back (i.e., deserializing an object) after applying some type change to that class definition. The JOS specification classified its 17 changes as *compatible* (8 changes) or *incompatible* (9 changes). Changes are incompatible if "the guarantee of interoperability cannot be maintained" [6]. Table 1 shows all the JOS changes.

	Change	JOS classification
JOS1	Add field	compatible
JOS2	Add class	compatible
JOS3	Remove class	compatible
JOS4	Add writeObject/readObject method	compatible
JOS5	Remove writeObject/readObject method	compatible
JOS6	Implement java.io.Serializable	compatible
JOS7	Change field access	compatible
JOS8	Change field from static to non-static, or from transient to non-transient	compatible
JOS9	Delete field	incompatible
JOS10	Move class up or down the hierarchy	incompatible
JOS11	Change field from non-static to static, or from non-transient to transient	incompatible
JOS12	Changing field declared type	incompatible
JOS13	Change writeObject/readObject method, so it does not write/read default data	incompatible
JOS14	Change a class from Serializable to Externalizable, or vice versa	incompatible
JOS15	Convert class to enum	incompatible
JOS16	Remove either Serializable, or Externalizable from a class definition	incompatible
JOS17	Add a writeReplace/readResolve method that would produce an incompatible object	incompatible

Table 1: The JOS specification changes classification.

We first planned to include the incompatible changes only in our study, but on closer inspection, we ignored the JOS specification's classification. For instance, in Table 1, the Add field change is classified as compatible because the deserialized object would still be created. But that change can break a Skipper test case because the newly added field would be initialized to the default value for its type (e.g., null) in the changed-deserialized object. Thus, we initially included all the JOS changes in our candidate changes list, and then analyzed them individually through our changes' analysis process.

3 Various source code elements' changes

The JOS Specification mainly addressed the subset of type-level changes that affect serialization, but some type-level changes (e.g., class renaming) and all method-, and field-level changes were still missing from our candidate changes list. We examined several change type taxonomies to come up with a comprehensive list of changes that can happen in a source code evolution scenario.

3.1 Change types' taxonomy selection

As Skipper's current prototype works on Java source code, our target taxonomy needed to address source code changes within Object Oriented programming languages (OOPLs), and more preferably Java. But any such taxonomy would result in a large set of change types that would be impossible to cover within one study. One option to handle that issue was to divide the selected taxonomy's list of changes into smaller sets, and apply each smaller set to one piece of reused functionality within our study. Yet, that option would prohibit us from evaluating the individual effect of each change type on both kinds of tests. A second option was to pick, from the selected taxonomy, the changes types that happen in higher frequency within source code, as per existing systems' change histories. Nguyen et al. [5] attempted to study the frequency of code changes within projects' change histories, but they mainly identified the frequently modified code structures (e.g., control/loop statements), rather than the frequently applied change (e.g., adding/removing nesting from a control structure). Such results cannot generalize as well, for they were solely based on the SVN change histories of 2,481 projects. The third option was to look for a change type taxonomy that would prioritize its change types according to the severity of their impact on the source code. We could then rely on such prioritization to select highly impacting changes for our study.

Lehnert et al. [4] provided a graph-based taxonomy of change types that distinguished between atomic and composite changes, yet they provided no prioritization of their change types. Sun et al. [8] introduced a taxonomy of 39 change types for the Java programming language, and produced a set of impact rules for those change types, but they did not compare the sizes of the generated impact set across the different change types. Gupta et al. [3] compared the sizes of the generated impact sets across different change types, but had two main limitations. (1) Their changes were classified into four main categories: functional (within a function's body), logical (within a program's control flow), structural (within the program's code structure), or behavioural (changing the program's existing functionality). Besides not being specific for the Java programming language, such classification is too vague to be applied; deleting an code entity can be classified both as a structural and a behavioural change [4]. (2) Their calculated impact sets were based on the total number of affected statements; a measurement that was proven to be an insufficient indicator of changes' significance. [1]. The only taxonomy that targeted OOPLs (and more specifically Java) change types, and prioritized such changes according to their impact on surrounding code entities was the FGP taxonomy of Fluri et al. [1], and was thus consulted for our study.

3.2 The FGP taxonomy

The FGP taxonomy [1] lists 37 source code changes at type-, method-, and field-level that can happen in object-oriented programming languages, assigning a significance level to each change. That significance level reflects that changes impact on other software entities, taking four possible values: low, medium, high, or crucial. Table 2 shows all the taxonomy changes. Changes that do not affect a classs external interface (e.g., FGP34) are classified as low or medium. Changes that affect a classs interface (e.g., FGP33), are classified as high or crucial. The taxonomy's changes significance levels were distributed as 19 crucial, 5 high, 8 medium, and 5 low significance. We initially included the 37 FGP changes in our candidate changes list, and then analyzed them through our changes' 3 step analysis process.

We added one category not identified by the FGP taxonomy or the JOS specification to our candidate changes list: deleting a class. Deleting a class was not mentioned in the FGP taxonomy because they care about changes happening within the class's body parts. JOS cares only about class deletion from a class's type hierarchy. We also wanted to examine the effect of deleting a class that is not part of a type hierarchy, but gets referenced within tests. Thus, we ended up with 55 possible changes that could be examined in our study, on which we applied a three step process; filtering, clustering, and sampling to settle on a subset of 13 changes to be used in our study. We explain our three step process in the following sections.

4 Changes filtering

After analyzing the 55 changes in our candidate changes list, we eliminated 10 changes from the JOS Specification's list, and 15 changes from the FGP taxonomy's list. Those 25 changes were eliminated for various reasons. We will explain those elimination reasons followed by an explanation on one of their corresponding eliminated changes.

- Four changes (JOS4, JOS5, JOS13, and JOS 17) demanded detailed knowledge about customized serialization techniques that we consider too specialized for normal developers to have.

  Explanation: JOS4 involves adding a user-defined writeObject or readObject method to the class definition of a serialized object. This change can affect how that class's fields are written to/read from a stream during a serialization process. Dealing with that effect of that change during an actual study demands deep understanding about (i) how those two methods work, and (i) when those two methods would be invoked during the serialization process. Such knowledge is too specialized for industrial developers to have, unless they have extensively used the Java serialization technology in their own work.

- Three changes (JOS6, JOS14, and JOS16) are guaranteed not to happen because of Skipper's current implementation.

  Explanation: JOS6 involves adding the java.io.Serializable interface to a parent class in the type hierarchy of the class definition of a serialized object, where that parent class did not implement the java.io.Serializable prior to this change. But, based on Skipper's current implementation, all reused classes are forced to implement the java.io.Serializable interface at all times. Thus, this change's scenario would not be allowed to happen while using Skipper to produce the reused tests.

- Six changes had crucial impact significance (FGP 4, FGP16, and FGP25) within the FGP taxonomy, or were classified as compatible/incompatible (JOS3, JOS7, and JOS11) within the JOS specification, but were discarded because they affect class's fields, hence such effect usually would not propagate to test code.

  Explanation: FGP16 involves decreasing a field's access modifier. This change effect depends on how the field's access modifier changes. For instance, if a field's access modifier changes from **public** to **private**, the code referencing that field would issue a compilation error that needs fixing. But, fields are mostly referenced from the source code methods, rather than from test cases. Hence, if such a change happens, it would affect the source code, rather than test code, and accordingly we eliminated it from our candidate changes list.

- Twelve changes had low/medium significance, and had no obvious, special effect on test code (FGP1, FGP5-FGP7, FGP10-FGP15, FGP28, and FGP34).

  Explanation: Even though we mainly picked the FGP taxonomy due to the presence of a ranking for the changes' impact, we thoroughly analyzed all its 13 low/medium impact changes during the filtering process. We found that most of them would not have a crucial impact on test code that would contradict their low/medium impact on source code, except for one change, FGP3: adding a field. Even though FGP3 is classified as a low impact change, it can break a Skipper test case because the newly added field would be initialized to the default value for its type (e.g., null) in the changed-deserialized object. The remaining 12 low/medium impact changes did not have a strikingly different effect on test code, and hence were eliminated from our candidate changes list.

Table 3 shows the updated candidate changes list after eliminating 25 changes from it.

5 Changes clustering

After the filtering step, we had 30 changes, which had repetition and similarity. The repetition came from identical changes that were mentioned in both the JOS list and the FGP taxonomy (e.g., changing a field's declared type: JOS12 in Table 1, and FGP 20 in Table 2). The similarity came from different changes that would have the same effect on the test code. We clustered the identical/similar changes to represent each cluster by only one change. For instance, JOS10 in Table 1 refers to moving a class up or down the type hierarchy. From a serialization perspective, moving a class *down* the type hierarchy would imply adding new fields to the *changed*-deserialized class. Such change is identical

in effect to the Add field change (JOS1), so both changes were put in one cluster. The clustering process resulted in two clusters of nine similar/identical changes (in total) (see Table 4), plus another 21 clusters each containing a unique change. Each cluster was represented by a single change, thus reducing our 30 changes to 23 changes that our study had to cover.

6 Changes Sampling

To study the effect of a change type on tests' maintainability, each test case should break due to one change kind only. To cover all 23 changes, developers in a typical study setup would have to understand and to fix 23 tests. We felt that this would take far too long. Therefore, we sampled the 23 changes, but not at random. The sampling process involved grouping the changes on the basis of the source code concepts involved and then selecting a single change from each group; for example, adding, updating, deleting, and renaming formal parameters all involve formal parameters so these were grouped. Such selection criterion was chosen because different source code concepts demand maintaining the broken test in a different way, hence leading to exploring more variations of fixing the test cases. For instance, changing a method's return type (FGP37) might demand maintain a test's assertions, whereas changing a method's parameter type (FGP33) demands changing the test's used arguments. Some changes could not be sampled due to represented a unique source code concept, thus were used-as-is in the study. Table 5 shows the complete sampling process. The sampling arrived at the 14 change kinds that were actually applied (see Table 6).

References

[1] B. Fluri, H. C. Gall, and M. Pinzger. Fine-grained analysis of change couplings. In *Proc. IEEE Int. Wkshp. Source Code Analysis and Manipulation*, pages 66–74, 2005.

[2] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Engineering*, 33(11):725–743, Nov. 2007. doi: 10.1109/TSE.2007.70731.

[3] C. Gupta, Y. Singh, and D. S. Chauhan. A dynamic approach to estimate change impact using type of change propagation. *JIPS*, 6(4):597–608, 2010.

[4] S. Lehnert, Q. Farooq, and M. Riebisch. A taxonomy of change types and its application in software evolution. In *Engineering of Computer Based Systems (ECBS), 2012 IEEE 19th International Conference and Workshops on*, pages 98–107. IEEE, 2012.

[5] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 180–190. IEEE, 2013.

[6] Oracle. Java object serialization specification. URL `http://docs.oracle.com/javase/7/docs/platform/serialization/spec/version.html#6678`.

[7] D. L. Parnas. Software aging. In *Proc. ACM/IEEE Int. Conf. Software Engineering*, pages 279–287, 1994. doi: 10.1109/ICSE.1994.296790.

[8] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang. Change impact analysis based on a taxonomy of change types. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pages 373–382. IEEE, 2010.

	Change	FGP sign.
FGP1	Class body: Add method	low
FGP2	Class body: Delete method	crucial
FGP3	Class body: Add field	low
FGP4	Class body: Delete field	crucial
FGP5	Method body: Structure statement: Condition expression change	medium
FGP6	Method body: Structure statement: else part insert	medium
FGP7	Method body: Structure statement: else part delete	medium
FGP8	Method body: Structure statement: insert and move changes to increase nested method depth	high
FGP9	Method body: Structure statement: decrease and move changes to decrease nested method depth	high
FGP10	Method body: Statement delete	medium
FGP11	Method body: Statement insert	medium
FGP12	Method body: Statement ordering change	low
FGP13	Method body: Statement parent change	medium
FGP14	Method body: Statement update	low
FGP15	Class/method/field: Increasing access modifier	medium
FGP16	Field declaration: decreasing access modifier	crucial
FGP17	Method declaration: decreasing access modifier	crucial
FGP18	Class declaration: decreasing access modifier	crucial
FGP19	Class declaration: Class renaming	high
FGP20	Class declaration: Attribute type change	crucial
FGP21	Class declaration: Attribute renaming	high
FGP22	Class declaration: Parent class insert	crucial
FGP23	Class declaration: Parent class delete	crucial
FGP24	Class declaration: Parent class update	crucial
FGP25	Field declaration: Final modifier insert	crucial
FGP26	Method declaration: Final modifier insert	crucial
FGP27	Class declaration: Final modifier insert	crucial
FGP28	Class/method/field: Final modifier delete	low
FGP29	Method declaration: Method renaming	high
FGP30	Method declaration: Parameter delete	crucial
FGP31	Method declaration: Parameter insert	crucial
FGP32	Method declaration: Parameter ordering change	crucial
FGP33	Method declaration: Parameter type change	crucial
FGP34	Method declaration: Parameter renaming	medium
FGP35	Method declaration: Return type delete	crucial
FGP36	Method declaration: Return type insert	crucial
FGP37	Method declaration: Return type update	crucial

Table 2: The significance levels of Fluri et al. (FGP) for their taxonomy of changes.

ID	Change
JOS1	Add field
JOS2	Add class
JOS8	Change field from static to non-static, or from transient to non-transient
JOS9	Delete field
JOS10	Move class up or down the hierarchy
JOS12	Changing field declared type
JOS15	Convert class to enum
FGP2	Class body: Delete method
FGP3	Class body: Add field
FGP8	Method body: Structure statement: insert and move changes to increase nested method depth
FGP9	Method body: Structure statement: decrease and move changes to decrease nested method depth
FGP17	Method declaration: decreasing access modifier
FGP18	Class declaration: decreasing access modifier
FGP19	Class declaration: Class renaming
FGP20	Class declaration: Attribute type change
FGP21	Class declaration: Attribute renaming
FGP22	Class declaration: Parent class insert
FGP23	Class declaration: Parent class delete
FGP24	Class declaration: Parent class update
FGP26	Method declaration: Final modifier insert
FGP27	Class declaration: Final modifier insert
FGP29	Method declaration: Method renaming
FGP30	Method declaration: Parameter delete
FGP31	Method declaration: Parameter insert
FGP32	Method declaration: Parameter ordering change
FGP33	Method declaration: Parameter type change
FGP35	Method declaration: Return type delete
FGP36	Method declaration: Return type insert
FGP37	Method declaration: Return type update
Study	Delete class

Table 3: The filtered candidate changes list.

Cluster ID	Change ID	Change
1	JOS1	Add field
	JOS2	Add class
	JOS8	Change field from static to non-static, or from transient to non-transient
	JOS9	Delete field
	JOS10	Move class up the hierarchy
	FGP3	Class body: Add field
	FGP21	Class declaration: Attribute renaming
2	JOS12	Changing field declared type
	FGP20	Class declaration: Attribute type change

Table 4: Clusters of similar/identical effect

Table 5: Changes sampling process.

Change ID	Change	Sample change
JOS1	Add field	JOS1
JOS12	Changing field declared type	JOS12
JOS15	Convert class to enum	JOS15
FGP2	Class body: Delete method	FGP2
FGP8	Method body: Structure statement: insert and move changes to increase nested method depth	FGP8
FGP9	Method body: Structure statement: decrease and move changes to decrease nested method depth	FGP9
FGP17	Method declaration: decreasing access modifier	FGP17
FGP18	Class declaration: decreasing access modifier	
FGP19	Class declaration: Class renaming	FGP19
FGP22	Class declaration: Parent class insert	FGP22
FGP23	Class declaration: Parent class delete	
FGP24	Class declaration: Parent class update	
FGP26	Method declaration: Final modifier insert	FGP26
FGP27	Class declaration: Final modifier insert	
FGP29	Method declaration: Method renaming	FGP29
FGP30	Method declaration: Parameter delete	FGP33
FGP31	Method declaration: Parameter insert	
FGP32	Method declaration: Parameter ordering change	
FGP33	Method declaration: Parameter type change	
FGP35	Method declaration: Return type delete	FGP37
FGP36	Method declaration: Return type insert	
FGP37	Method declaration: Return type update	
Study	Delete class	Study

Table 6: The sets of applied changes

ID	Origin of case(s)	Applied change(s)	aTunes tasks
CS1	{ JOS/FGP, FGP }	{ Add field, Increase structure statement }	derived
CS2	JOS/FGP	Change field type	derived
CS3	JOS	Convert class to enum	synthesized
CS4	{ JOS/FGP, novel }	{ Delete method, Delete class }	synthesized
CS5	FGP	Delete structure statement	actual
CS6	FGP	Rename class	synthesized
CS7	FGP	Parent class insert	synthesized
CS8	FGP	Insert method final modifier	synthesized
CS9	FGP	Rename method	synthesized
CS10	FGP	Parameter type change	actual
CS11	FGP	Return type change	actual
CS12	FGP	Decrease method visibility	synthesized