Harmfulness of Code Duplication -

A Structured Review of the Evidence

Wiebe Hordijk, María Laura Ponisio[1], Roel Wieringa
University of Twente, The Netherlands
hordijkwtb|m.l.ponisio|roelw@ewi.utwente.nl

Duplication of code has long been thought to decrease changeability of systems, but recently doubts have been expressed whether this is true in general. This is a problem for researchers because it makes the value of research aimed against clones uncertain, and for practitioners as they cannot be sure whether their effort in reducing duplication is well-spent. In this paper we try to shed light on this issue by collecting empirical evidence in favor and against the negative effects of duplication on changeability. We go beyond the flat yes/no-question of harmfulness and present an explanatory model to show the mechanisms through which duplication is suspected to affect quality. We aggregate the evidence for each of the causal links in the model. This sheds light on the current state of duplication research and helps practitioners choose between the available mitigation strategies.

Keywords: Duplication, clones, changeability, maintainability, structured review

1. INTRODUCTION

Duplication of source code is an important factor that is suspected to affect the quality of systems in terms of changeability and the number of errors. We want to investigate how duplication affects quality. There is a vast body of research about code duplication, and in this review we aggregate the current knowledge about the effects of duplication on changeability and error levels.

1.1 Problems

There is a lot of literature about code duplication, but only a few studies have attempted to investigate if and how duplication actually has a negative effect on changeability and error levels. Therefore surprisingly little is known empirically about the harmfulness of duplication. This is a problem for researchers because many investigations are based upon the assumption that clones are harmful, and if this assumption is false, the value of the research would be called into doubt. Fortunately we have found positive though inconclusive evidence for this assumption.
For practitioners this lack of knowledge about harmfulness of duplication is a problem because they do not know if they should invest effort in avoiding or removing clones, and if so, how to prioritize those efforts between different kinds of clones. Based on practitioners' reports and our own experience, we see that little use is made of clone detectors in practice. We think that solid knowledge about the harmfulness of clones would make such tools more attractive to practitioners.

1.2 Contributions

This study is a structured review of the evidence in code clone literature for harmfulness of duplication. We have reviewed all the relevant clone duplication literature about the effects of duplication on system quality that we found

[1] Supported by the Netherlands Organisation for Scientific Research, project nr 638.004.609 (QuadRead).

using the systematic review method outlined in section 2. We integrate the results in a causal model, showing intermediate variables between duplication and quality attributes, and hypothesized causal links. We aggregate claims from the literature in section 3 and present the aggregated knowledge per hypothesis in section 4. We conclude that the main questions about the effect of duplication on quality are not yet answered, but that considerable progress has been made which positively supports sub-hypotheses of those questions, and that there is a reassuring amount of correlation between findings of different studies. For example, findings about how many clones are co-changed are quite consistent. There are however also contradictions among findings, which require further investigation, such as the question "what demands more change effort, cloned code or non-cloned code?". For researchers our review suggests topics for further empirical research.
For practitioners we include a table which lists the conclusions from the literature along with strategies to mitigate the negative effects of duplication on system quality. When there is more support for a conclusion, there will be more need for its corresponding mitigation strategy. We hope that this table will help practitioners select proper actions to reduce negative effects of duplication in their own projects.

2. METHODOLOGY

We gathered information only from primary research, not from empirical observations. We followed a method described by Kitchenham's general procedure for performing systematic reviews (Kitchenham, 2007). Though the entire investigation is not completely repeatable, as human judgment is involved in interpreting articles, Kitchenham's method makes steps of the process as repeatable as possible. The following sections summarize our steps.

2.1 Identification of research

We searched a number of literature sources with several search criteria, aimed at finding a set of articles with the most complete possible coverage of the field of code clones. We chose the criteria to reflect our main research question: "What effects does duplication of code have on the quality attributes of a software system?" We searched the following databases: DBLP, ACM Portal, CiteSeer and Scopus with the following search terms: "code clone", "clones", "code | duplication" ("duplication" yields too many false hits), on December 17, 2008. We discarded articles that were not about code clones; examples include compiler optimization, set theory and DNA research. After our extensive searches we have validated the completeness of the search actions by looking for references in the selected papers to other papers that were not present in our sample but that would pass our search criteria. We found only 2 such references, which were workshop papers. Altogether, this yielded 153 papers. A full list is in (Hordijk et al., 2009); to our knowledge we have thus exhausted all available evidence in the period under review.

2.2 Selection of primary studies

We applied the following criteria to the found sources for inclusion in this review.

- The article must be published in a journal or conference proceedings. This excludes drafts of articles and technical reports found on web sites of research groups.
- The article should present evidence for a relation between duplication and a quality attribute of the system, or between intermediate variables, e.g. between duplication and co-change. We judged this by reading the entire papers, not just the titles and abstracts, because sometimes evidence is stated in a case study which is used as an illustration of, for example, a clone detector, on which the paper focuses.
- The article should not be published before 1990. This boundary is chosen arbitrarily to limit the search for sources.

We have not applied quality criteria to the primary sources, because so few papers passed the selection criteria that no additional selection was needed. The resulting set contains 18 papers, which are discussed in section 3.

2.3 Aggregation of evidence

We analyzed the evidence in the included papers. When a claim was made, we analyzed the external validity, that is, for which situations the claim would hold. For example, if a paper draws conclusions from an experiment with one system, then those conclusions may not be valid in another system because of differences between those systems. However if conclusions are based on five different open source Java systems, and another paper draws the same conclusions from two other open source Java systems, we may generalize the conclusions to the class of open source Java systems. An overview of the conclusions is in section 4. Since we are interested in the circumstances under which duplication is harmful, we also list what is known about the context factors under which these conclusions hold.

3. CAUSAL MODEL

Figure 1 contains our hypothesized causal model of the effects of duplication on system quality. The model has been constructed after analysis of the reviewed literature; for space reasons in this paper we only summarize the

evidence for and against each of the hypotheses in the model. The names in the figure are variables and the arrows are causal effects. Each arrow represents a separate hypothesis such as, “duplication increases co-change”. This should be read thus: in general, all else being equal, a system with more duplication will exhibit more co-change than an equivalent system with less duplication.

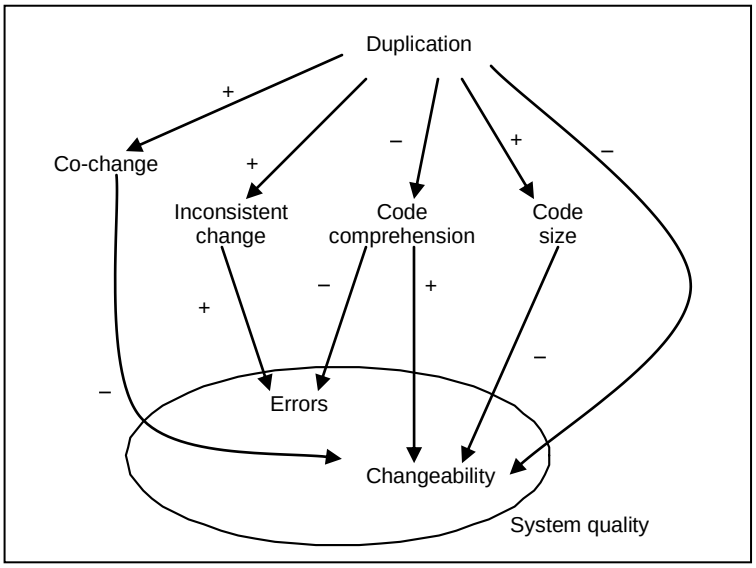

FIGURE 1: Causal Model

We now discuss the concept of change and then review the evidence concerning the role of each of the four intermediate variables in our model.

3.1 General Model of Change

To be able to speak about effects of duplication on changeability we introduce a general model of change processes. A change is a set of activities that starts when a change request is presented by the customer to the maintenance team and that ends when the adapted software product is accepted by the customer. To see the possibly negative influence of duplication, we need to compare the situation where the part of the software that needs to be changed has some form of duplication with the situation in which this particular duplication is not present, all else being equal. It is possible that the only way of avoiding the duplication would be to make the software more complex, which would have its own negative effect on changeability. Without disregard for the differences in software development and maintenance methodologies we generalize the change process as consisting of three phases.

- Analysis: the maintainers analyze the change request and the code, determining which parts of the code need to be changed.
- Coding: the maintainers change the code.
- Test: the maintainers test whether the changed system meets the requirements of the change request, and whether no new bugs have been introduced. When errors are found the process loops back to coding or analysis, depending on the type of error.

This distinction makes it easier to explain the mechanisms through which duplication is thought to affect changeability. Suppose a change request reaches the maintenance team. After analysis the team concludes which parts of the code need to be changed. Now several conditions can hold.

- Two or more parts that are clones of each other must be changed in the same way, so that after the change they are still clones of each other.
    - The programmers change them in the same way. This is co-change and costs extra effort because the same editing operations are performed multiple times.
    - The programmers change them in different ways, or change only one. This is inconsistent change. This leads to errors during test which must be fixed, incurring extra effort during the change process, or when the errors are not found it can lead to errors later on.
- There are no such same changes in clones. Duplication in other parts of the system may still affect the effort required for analysis. Simply put, the fact that the total size of the code is larger makes it harder to find the part one is looking for.

3.2 Co-change

We define co-change as the situation where two or more clone occurrences are changed in the same way because of one change request, so that after the change they are still clone occurrences of each other.
Co-change can be observed after the fact by detecting clones and changes in multiple consecutive versions of a system. When in two clone occurrences in version n some change is performed, and in version n+1 they are still clone occurrences of each other, they have been co-changed. This operationalisation leaves open the possibility that the occurrences have by coincidence been changed in the same way due to completely different change requests. This seems unlikely enough to disregard the quantity of such 'accidental co-changes'.

Code clone genealogies
(Kim et al., 2005) examined clone genealogies in 2 open source Java programs of 20 to 25 KLoC each. A genealogy is a clone set (a set of code fragments which are all clones of each other) traced over multiple versions of the system. In this study CCFinder was used to detect clones of at least 30 tokens. They found that 36% to 38% of code clone genealogies contained consistently changing patterns. Even though clone genealogies are not the same as clone pairs, we can conclude that co-change was observed in a significant portion of clone pairs. This is strong evidence that at least some clones negatively affect changeability through co-change. A smaller portion of genealogies – 26% to 34% of all genealogies which were abandoned in later versions – dissolved because occurrences were modified in different ways. The researchers have investigated these genealogies by hand, so we can assume that these were not changed inconsistently while they should have been co-changed. This is thus a group of clones which did not harm changeability. It's not clear what happened to the rest of the abandoned genealogies. Presumably, they were refactored. That must have cost effort, which is another way in which duplication increases change effort. The researchers also manually examined the genealogies that lived in more than half of the history of their systems to see if the clones could be refactored. The researchers judged that about two thirds of these genealogies contain unrefactorable clones, and most genealogies containing unrefactorable clones exhibited co-change. It has not been investigated if there are any clone properties that predict whether a clone will be co-changed or evolve differently. Conclusions (with supporting evidence):

- A significant portion of clones are co-changed (2 small open source Java programs).
- Of the clones that are removed in later versions, a minority is removed because of different changes, while the rest is presumably removed through refactoring (2 small open source Java programs).
- Clones that are not refactorable tend to live longer than refactorable clones and are very likely to exhibit co-change (2 small open source Java programs).

How clones are maintained
(Aversano et al., 2007) have investigated how clones are maintained in two Java systems of 25 and 200 KLoC respectively. They used SimScan, an AST-based clone detector, with a minimum clone size of 6 LOC. In one system they found that, from one version to the next, 45% of clones were co-changed; 32% were changed in different ways; 2% were refactored; 5% were removed otherwise; and 16% were changed inconsistently (see paragraph 3.3 about inconsistent change). The latter group can be identified by late propagation. Late propagation is the situation where clone occurrences are changed in different ways, and are later changed so they are the same again; this can be detected automatically. Not every inconsistent change leads to late propagation, because not all inconsistencies lead to errors. Not every late propagation is preceded by inconsistent change: it may be caused by similar change requests to different modules at different times. The researchers have manually identified many causes of late propagation, and argue that the majority of cases of late propagation are not actually instances of inconsistent change, but rather they are cases where similar change requests are performed for different subsystems at different times. This seems more like co-change than inconsistent change. In another system they found 74% co-change, 13% independent change and 13% inconsistent change, though in a very small total number of clones. We can conclude that these numbers are roughly consistent with those of Kim et al. The authors have observed an interesting relation between the *granularity* (block, method or class) of clones and how they are maintained: most class-level clones were co-changed, while method-level clones most often evolve independently. This indicates that clones with larger granularity have more chance of being harmful (inducing co-change) than smaller ones. Conclusions (with support):

- A significant portion of clones are co-changed (1 small and 1 large open source Java program).
- Clones with class-level granularity have more co-change than clones with smaller granularities (1 small and 1 large open source Java program).

Visualizing maintenance patterns of clones
(Balint et al., 2006) present a tool, integrated in the Moose reengineering suite that visualizes maintenance patterns of clones. This tool can show lines of cloned code that are maintained consistently or inconsistently and made consistent later. Without giving numbers they report that many clones are changed consistently, and a few are

changed inconsistently but made consistent later. This agrees with the findings of Kim et al. and more specifically with those of Aversano et al.

Quantification of co-change
(Krinke, 2007) investigated co-change of clone groups in five systems (of 10, 90, 118, 193 and 228 KLoC respectively) written in Java, C and C++. They used Simian, a text-based clone detector, with a minimum clone size of 11 lines. They found that 45% to 55% of the clone sets are co-changed all along the investigation period of 200 weeks. This is higher than the range reported by Kim et al. (36% - 38%) and narrower than that reported by Aversano et al. (45% - 75%). A likely explanation for these differences is the normalization of source code which is done before detecting changes: comments are removed and the code is reformatted so that changes to whitespace are disregarded. The paper reports that without this normalization, the ratio of co-changed clones is much lower, more towards that reported by Kim et al. Neither Kim nor Aversano report that they use normalization of the code, so this explanation is plausible. Also, Krinke uses more and larger systems. We conclude therefore that the ratio of co-changed clone sets is likely to lie in the 45%-55% range. Krinke et al. have also investigated how the ratio of co-change depends on minimum clone size: this has only small and inconsistent impact. Conclusions:

- 45% to 55% of clone sets are co-changed during a long part of the life cycle of systems (5 systems: Java, C, C++; various sizes).
- Clone size has little impact on the chance of co-change (5 systems: Java, C, C++; various sizes).

Correlation between cloning ratio and change coupling
(Geiger et al., 2006) have investigated the correlation between cloning and co-change on file level. As a measure for cloning between files they use the ratio of lines that are duplicated between those files over the total number of lines, and as a measure for change coupling they use the ratio of the number of times the two files are changed at the same time (same check-in) over the total number of changes for the files. Clones are detected using CCFinder on a number of releases of Mozilla, in the C programming language, using a minimum clone size of 30 tokens. They show a dot plot which indicates that, on average, files with more cloning between them have more change coupling. The correlation is not statistically significant, but for files with very high cloning ratios the change coupling is also very high. It has not been investigated how often the changes are in cloned code, or whether the same changes are made in different files. Therefore this amounts to rather weak evidence for duplication's negative effect on changeability through co-change. Conclusions:

- Files between which there is a larger ratio of cloning seem to be changed in the same release more often than other files (weak correlation; 1 large C system).
- Files between which there is a very high ratio of cloning are very often changed in the same release (1 large C system).

Comparing co-change on method level
(Lozano et al., 2007) have investigated methods that have contained clones at some point in their life. For those methods they have compared the periods during which the methods contained clones with the periods during which they did not. They measured the number of changes to the methods themselves and the number of co-changes between pairs of methods between which there is a cloning relation. The definition of co-change here is that the methods were both changed within a very short time interval; the changes need not be the same. The first finding is that in periods with clones, the methods were changed more often than in periods without clones. This result does not have a clear explanation. It is seconded by Monden but contradicted by (Krinke, 2008) (see section 3.6) in which cloned code was found to be changed less often than non-cloned code. The second finding is that in periods in which a method pair has a cloning relation, the pair is co-changed *less* often than in periods without a cloning relation. This second result is very surprising, and the authors offer no explanation, nor can we think of one. The second result contradicts the conclusion by (Geiger et al., 2006). Conclusions:

- Methods are changed more often during periods in which they contain clones than during periods in which they do not (1 small Java system).
- Method pairs have less co-change during periods in which they have a cloning relation than during periods in which they have not (1 small Java system).

Discussion
If we want to investigate the harmfulness of duplication, it is important to compare co-change between duplicated code with co-change between non-duplicated code fragments. Of these studies, only (Geiger et al., 2006) and (Lozano et al., 2007) have done so. The results are contradictory, so more research is needed in which cloned code is compared to non-cloned code with respect to co-change. Also we need investigations with more systems and with more attention to which circumstances may explain the different findings. Perhaps the differences can be explained from the fact that they investigated different systems; perhaps because they measured on different levels of granularity; perhaps other clone properties or context factors play a role.

3.3 Inconsistent change

Inconsistent change is the situation where two or more occurrences of the same clone set should be changed in the same way because of a change request, but at least one of them is changed differently from the others. This may lead to an error in the system when the inconsistency leads to different behavior, decreasing the reliability of the system. In that case the bug will have to be fixed, increasing change effort. It is also possible that the inconsistency does not lead to incorrect behavior, but then when the clone occurrences need to be co-changed in a later version the inconsistency will make the change more difficult. In that case too, inconsistencies increase change effort. It is hard to detect inconsistent change automatically, because it may be quite legitimate that clone occurrences evolve independently. Some researchers have investigated changes to clones manually to see whether different changes were inconsistencies or not. Others have used late propagation (see §3.2, 'How clones are maintained') as an indicator.

Late propagation

Aversano et al. have found that between 13% and 16% of clones were involved in late propagation in two systems, of which only a minority however could be attributed to inconsistent change upon manual inspection. This indicates that inconsistent change does not occur very often. Conclusions:

- A small portion of clones are changed inconsistently which is repaired in later versions. However only a part of the cases in which a change is applied to other occurrences later on (late propagation) stem from inconsistent change (1 small and 1 large open source Java program).

Copy-paste related bugs

(Li et al., 2004, Li et al., 2006) have built a tool to detect copy-paste related bugs in code. They search for clones which differ only in the names of identifiers used between occurrences. When in one occurrence of a clone, identifier A is used exactly in the places where in another occurrence of the clone identifier B is used, there is no problem; but when one instance of A is changed into C, that may indicate a bug. Such bugs can easily be introduced by inconsistent changes. Li et al. have found a number of bugs in Linux, FreeBSD, Apache and PostgreSQL using their tool, CPMiner. To give an idea of numbers: the total number of clone occurrences in Linux is 122,282 (without allowing gaps in the clones); CPMiner reports 421 potential errors due to inconsistencies between identifiers; after manual inspection, 28 of these are real bugs and 21 are 'careless programming', meaning they should be fixed in the interest of maintainability although they will not lead to erroneous behavior. These findings indicate that duplication can indeed lead to errors in software, but only in very small numbers, which agrees with the findings of Aversano et al. Conclusions:

- In a small number of cases it was observed that the act of copying code led to errors (4 large open source C programs).

(Jürgens et al., 2008) are working on an approach to detect bugs caused by inconsistent change. Their work is ongoing. So far their results seem to be in line with the other sources mentioned in this paragraph. They found a relatively small number of bugs which stem from inconsistent change. Conclusions:

- In a small number of cases it was observed that the act of copying code led to errors (1 large industrial C# program).

Context differences

(Jiang et al., 2007) take a different approach to identifying clone-related bugs. Their intuition is that similar code should be used in the same way, so they look for differences in the context of clone occurrences. This will yield different errors than the approaches by Li and Jürgens, but quantitatively the results are similar: a minority of clones with context differences were confirmed to contain bugs. Conclusions:

- In a small number of cases it was observed that copying code led to errors (1 large C and 1 large Java program).

Discussion

When clones are changed inconsistently, that is clearly a source of errors. However, had the code not been duplicated, it might have been more complex, and other errors might have been made when changing it. To know whether duplication causes an increase of errors through inconsistent change, we need to compare the amount of errors introduced into duplicated code with that in non-duplicated code. If the extra errors introduced through inconsistent changes do not lift the amount of errors in duplicated code above the normal amount of errors in a comparable body of code in the same system, then we cannot hold the conclusion that duplication causes more errors.

3.4 Code comprehension

Code comprehension is inherently hard to study. Many papers assume duplicated code is harder to understand.

Maintaining mental models

(LaToza et al., 2006) have done a survey in which a majority of programmers interviewed for a case study reported that finding duplicated code was a problem for them, especially when clones were created inadvertently. The study

shows that programmers put much effort in creating rich mental models of the system they are working on, which are rarely documented, and that when trying to understand new code the biggest problem is to understand the rationale of its design. 59% of the programmers indicated that 'Finding all the places code has been duplicated' is a problem for them. When interviewed about why duplication is a problem, programmers referred to 'making the same change in different places', which seems to be the same as co-change; however the interviewees used this notion in a broad way, including cases where similar functionality was implemented in different programming languages, where similar functionality was implemented using different algorithms, or different branches of the same code base. Latoza et al. make a case for including such cases in the definition of co-change and in code duplication research. Conclusions:

- Programmers experience that duplication hinders code comprehension.
- Researchers should be aware of types of duplication that go beyond syntactical clones.
- Clones created by different programmers are harder to understand and harder to refactor than clones created by one programmer.
- Issues with code ownership may introduce duplication.
- Changes that involve aspects (in the AOP sense) are especially hard to analyze.

Complex refactorings

(Balazinska et al., 1999) present a technique to automatically refactor certain kinds of clones in Java, replacing the cloned code with an application of the Strategy design pattern. Though this may help for large clones with many occurrences, the paper contains a necessarily smaller example. Consequently, the paper example contains more classes and more lines of code after refactoring than before, which arguably increases its complexity. This serves as a counter-example to the claim that cloned code is harder to comprehend than its non-cloned equivalent. For the same reason (Fowler, 1999) argues that when the same code appears only twice, that is not yet enough reason to refactor it; the duplicated code may be more understandable. Conclusions:

- Duplicated code is not necessarily more complex or harder to understand than the equivalent non-duplicated code (1 Java example).

Discussion

We cannot draw conclusions about the relation between duplication and code comprehension. We can conclude that co-change is a problem for programmers, and that co-change goes farther than the syntactic clones that can be detected with clone detectors. Co-change occurs between different systems, different programming languages and different branches of the same system. This calls for more research.

3.5 Code size

(Baker, 1992) assumes that when a clone set has n occurrences, n-1 of them could have been avoided, so the increase is n-1 times the size of the cloned fragment. However, we should compare the increase in code size due to duplication to code size required by the alternative. (Kamiya et al., 2002) takes into account that the occurrences have to be replaced with a function call. (Balazinska et al., 1999) show that refactoring clones can be more difficult because of dependences and differences in the occurrences, so in their example, after refactoring the resulting code is actually bigger than the cloned code was before refactoring. Conclusions:

- It is simple to measure the total size of code involved in a clone set.
- It is difficult to estimate the size of code needed in an alternative solution, for example after refactoring. Replacing clones by an alternative does not always result in reducing the code size.

Discussion

It is easy to compute how much code is involved in duplication once the clones have been detected, but it is hard to measure how much code can be saved by replacing clones by an alternative. When clone occurrences have no differences between them and no dependencies on their context, they can easily be replaced by functions or methods. For more complex situations, Balazinska et al. have presented a method to automatically replace clones by a Strategy pattern, which works for certain well-defined clones; other clones can be refactored with other mechanisms or by hand. It cannot be stated in general that duplication increases code size compared to alternatives.

3.6 Direct relationships

Some studies have attempted to measure the impact of duplication on changeability directly.

Revision number

(Monden et al., 2002) tried to show a direct effect of duplication on changeability. They correlated the amount of duplication in modules with the modules' revision numbers: a higher revision number would indicate lower changeability. They found a clear positive correlation between the maximum size of clones in a module and its revision number. The number of revisions of a module however is not a good indicator of changeability. When more change

requests have impacted a module, that only says something about the customers needs, or perhaps about the design of a system when a module is changed for very dissimilar change requests. Also the causality may be reversed: when a module is changed more often, its design may degrade and more clones may be introduced. As the authors remark, the nature of this very interesting correlation has yet to be investigated.

Number of errors
Monden et al. also have investigated the number of errors found in modules, and compared modules that contain clones with modules that do not. They found that modules without clones have 1.7 times as many errors per line of code than modules with clones. Their explanation is that when one copies code that works, one will not generally introduce bugs; this could imply that the practice of duplication reduces the number of errors in a system. However, with clones, more lines of code are needed to implement the same functionality, and those extra lines dilute the number of errors per line. The interesting question is whether the system as a whole would contain more or less errors if the clones were removed; the authors do not discuss this question.

Stability
(Krinke, 2008) has investigated the stability of cloned code versus that of non-cloned code. It turns out that overall, cloned code is changed less often than non-cloned code. The reason for this phenomenon has not been investigated. A possible explanation is that programmers try to avoid changing cloned code because it is more difficult to change, which would strengthen the hypothesis that duplication decreases changeability. However, it could also be that code contains fewer clones because it is changed more often, when those changes include refactorings to remove clones. Also the programmers may have been careful enough to only clone code which was not likely to be changed often. Yet another explanation is that most code clones are templates, company standards etc. that are never changed. So Krinke's observation is very interesting, but more research is needed before we can explain the observation.

Discussion
A direct link between duplication and changeability has not been proven. It is very hard to do so, because as yet no good indicators for changeability are known, and it is very cumbersome to measure change effort. Also, there are many factors that affect changeability. It seems more likely that we will advance by further investigating the mechanisms through which duplication affects changeability than by trying to measure the effect directly.

4. CONCLUSIONS AND FUTURE RESEARCH

The goal of our research was to find out under which circumstances code duplication harms system quality. Is it easier to change a system if we have removed the clones first? Our literature survey suggests that a direct link between duplication and changeability has not been proven yet, but not rejected either. We have refined the existing theory by bringing together the various mechanisms through which duplication is thought to affect quality, and we have aggregated the evidence for and against each of these hypothesized mechanisms. In Table 1 we list the hypotheses from the cause/effect graph and the evidence for and against each of them. For the first two hypotheses, we give two related statements which have been proven to some extent; those are indented. The status of the most interesting hypotheses is that 'more research is needed'. Though there is a large volume of research in code duplication, there are but few studies aimed at the question whether clones really are harmful, which clones are more harmful than others, and under which circumstances.

TABLE 1: Hypotheses and their evidence

Hypothesis	**Evidence for**	**Evidence against**	**Status**
There is more co-change between clone pairs than between other code fragments	Geiger2006 (weak)	Lozano2007	Proven nor rejected: open for investigation (see §3.2)
Co-change occurs in 30% to 50% of clone sets	Kim2005, Aversano2007, Krinke2007		Proven for various open source systems in C, C++, Java of various sizes
Changes to duplicated code result in more errors than changes to non-duplicated code			Proven nor rejected: open for investigation (see §3.3)
Inconsistent change, resulting in errors, happens to a minority of clones	Aversano2007, Li2004, Jiang2007, Juergens2008		Proven for various open source and industrial systems in C, C#, Java of various sizes
Duplicated code is generally harder to understand than its non-duplicated equivalent	LaToza2006	Balazinska1999	Likely, considering the consensus in LaToza's study, but not generally valid for each clone (see §3.4).
Duplicated code is generally longer than its non-duplicated equivalent		Balazinska1999	Can be proven analytically for many clones, but there are counter-examples. It depends on the alternative (see §3.5).

In Table 2 we list all the conclusions that we have distilled from the literature, along with their supporters. We also give strategies, based on our own experience and reasoning, for practitioners to mitigate the harmful effects that clones have on system quality. We hope to make it easier for practitioners to use the knowledge present in the vast body of literature on this topic. When there is more support for a conclusion, there will be more need for its corresponding mitigation strategy. The effectiveness of the mitigation strategies themselves has not been investigated, as there is too little evaluation research available about mitigation strategies.

TABLE 2: Conclusions, support and mitigation strategies

Conclusion	Supported by	Mitigation
Co-change		
A significant portion of clones are co-changed.	Kim2005, Aversano2007, Balint2006, Krinke2007	For co-changing clones that cannot be refactored, consider using change propagating tools, such as Linked Editing (Toomim et al., 2004).
Of the clones that are removed in later versions, a minority is removed because of different changes, while the rest is presumably removed through refactoring.	Kim2005	
Clones that are not refactorable tend to live longer and are very likely to exhibit co-change. 45% to 55% of clone sets are co-changed during a long part of the life cycle of systems.	Kim2005, Krinke2007	We cannot mitigate the negative effect of these clones by refactoring, so simultaneous editing seems like the only solution.
Clones with class-level granularity have more co-change than clones with smaller granularities.	Aversano2007	These also cause more increase in code size. These should be the first to refactor.
Files between which there is a very high ratio of cloning are very often changed in the same release.	Geiger2006	When removing clones, give priority to files between which a lot of clones exist.
Clone size has little impact on the chance of co-change.	Krinke2007	
Inconsistent change		
A small portion of clones are changed inconsistently which is repaired in later versions. However only a part of the cases in which a change is applied to other occurrences later on (late propagation) stem from inconsistent change.	Aversano2007	
In a small number of cases it was observed that the act of copying code led to errors.	Li2004, Jiang2007, Juergens2008	Consider using the approaches mentioned as an additional means to search for potential errors in the code, especially when a lot of copy-paste has happened.
Code comprehension		
Programmers experience that duplication hinders code comprehension.	LaToza2006	Consider adding clone detection results to system documentation to aid comprehension.
Researchers should be aware of types of duplication that go beyond syntactical clones.	LaToza2006	
Clones created by different programmers are harder to understand and harder to refactor than clones created by one programmer.	LaToza2006	
Issues with code ownership may introduce duplication.	LaToza2006	
Changes that involve aspects (in the AOP sense) are especially hard to analyze.	LaToza2006	When aspect-related code is cloned and changes to those aspects are expected in the future, use AOP techniques to avoid co-change.
Duplicated code is not necessarily more complex or harder to understand than the equivalent non-duplicated code.	Balazinska1999	Be careful when refactoring clones to not increase the complexity.
Code size		
It is simple to measure the total size of code involved in a clone set. However it is difficult to estimate the size of code needed in an alternative solution, for example after refactoring. Removal of clones does not always result in reducing the code size.	Baker1992, Kamiya2002, Balazinska1999	Be careful when refactoring clones to not increase the code size.

Below we list the open questions that we have encountered in the discussions above.

- Does duplicated code get involved in co-change more often than non-duplicated code? Which kinds of clones get involved in co-change and which do not? What context factors influence co-change? For example, can the different find-

ings of Geiger and Lozano be explained by differences in the granularity of their measurements? Comparisons between duplicated and non-duplicated code have been performed (see 3.2) but with contradicting results.
- Are more errors introduced in cloned code than in non-cloned code? Under which circumstances? It has been shown that the presence of clones causes errors during maintenance (see 3.3), but this effect has not been compared to the errors that could have been introduced if the code had not contained these clones.
- Under which circumstances is cloned code more difficult to understand than its non-cloned equivalent?

We thank the anonymous reviewers for their helpful suggestions for improving this paper.

REFERENCES.

AVERSANO, L., CERULO, L. & DI PENTA, M. (2007) How clones are maintained: An empirical study. *European Conference on Software Maintenance and Reengineering.* Amsterdam.

BAKER, B. S. (1992) A Program for Identifying Duplicated Code. *Computing Science and Statistics,* 24**,** 49-57.

BALAZINSKA, M., MERLO, E., DAGENAIS, M., LAGUE, B. & KONTOGIANNIS, K. (1999) Partial redesign of Java software systems based on clone analysis. *6th Working Conference on Reverse Engineering.* Atlanta, GA, USA, IEEE.

BALINT, M., GÎRBA, T. & MARINESCU, R. (2006) How developers copy. *14th IEEE International Conference on Program Comprehension.* Athens.

FOWLER, M. (1999) *Refactoring - Improving the Design of Existing Code*, Addison-Wesley.

GEIGER, R., FLURI, B., GALL, H. & PINZGER, M. (2006) Relation of Code Clones and Change Couplings. *Fundamental Approaches to Software Engineering.*

HORDIJK, W., PONISIO, M. L. & WIERINGA, R. (2009) Structured Review of the Evidence for Effects of Code Duplication on Software Quality. University of Twente, The Netherlands.

JIANG, L., SU, Z. & CHIU, E. (2007) Context-based detection of clone-related bugs. *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering***,** 55-64.

JÜRGENS, E., HUMMEL, B., DEISSENBOECK, F. & FEILKAS, M. (2008) Static Bug Detection Through Analysis of Inconsistent Clones. *Software Engineering (Workshops)***,** 443-446.

KAMIYA, T., KUSUMOTO, S. & INOUE, K. (2002) CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering,* 28**,** 654-670.

KIM, M., SAZAWAL, V., NOTKIN, D. & MURPHY, G. C. (2005) An empirical study of code clone genealogies. *10th European Software Engineering Conference.*

KITCHENHAM, B. (2007) Procedures for Performing Systematic Reviews. University of Durham, UK.

KRINKE, J. (2007) A Study of Consistent and Inconsistent Changes to Code Clones. *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering***,** 170-178.

KRINKE, J. (2008) Is Cloned Code More Stable than Non-cloned Code? *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on***,** 57-66.

LATOZA, T. D., VENOLIA, G. & DELINE, R. (2006) Maintaining mental models: A study of developer work habits. *International Conference on Software Engineering.* Shanghai.

LI, Z., LU, S., MYAGMAR, S. & ZHOU, Y. (2004) CP-Miner: a tool for finding copy-paste and related bugs in operating system code. *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation***,** 20-20.

LI, Z., LU, S., MYAGMAR, S. & ZHOU, Y. (2006) CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering,* 32**,** 176-192.

LOZANO, A., WERMELINGER, M. & NUSEIBEH, B. (2007) Evaluating the Harmfulness of Cloning: A Change Based Experiment. *Fourth International Workshop on Mining Software Repositories.* IEEE Computer Society.

MONDEN, A., NAKAE, D., KAMIYA, T., SATO, S. A. & MATSUMOTO, K. A. (2002) Software quality analysis by code clones in industrial legacy software. IN NAKAE, D. (Ed.) *Eighth IEEE Symposium on Software Metrics.*

TOOMIM, M., BEGEL, A. & GRAHAM, S. L. (2004) Managing Duplicated Code with Linked Editing. *Symposium on Visual Languages - Human Centric Computing, VLHCC.* IEEE Computer Society.