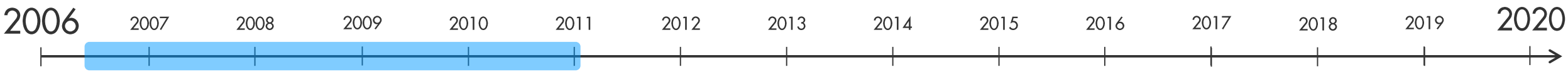
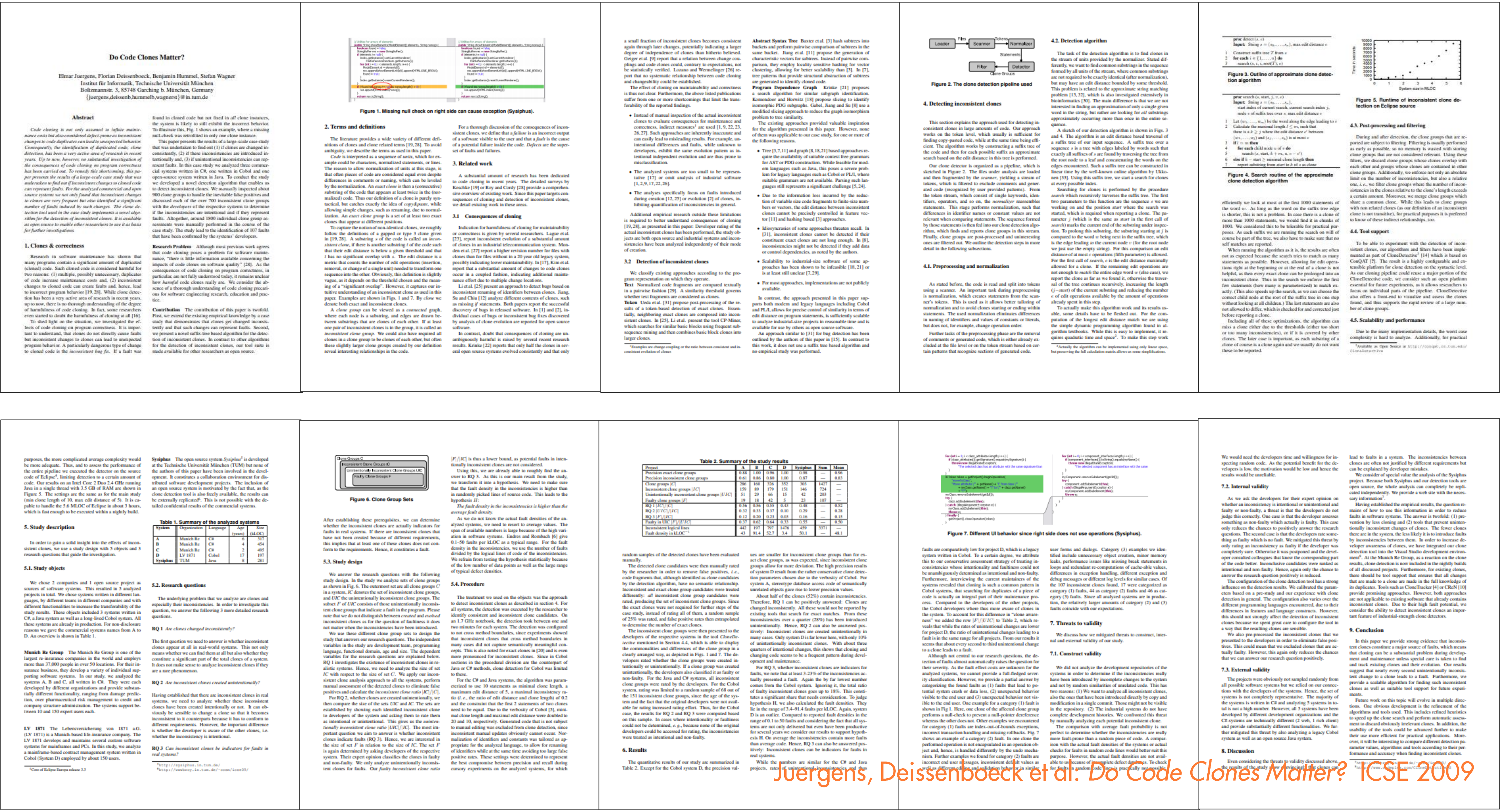
Kosten-Nutzen-Berechnung von Qualitätsanalysen
Erfahrungen bei der Munich Re

Uwe Proft (Munich Re)
Elmar Jürgens (CQSE)

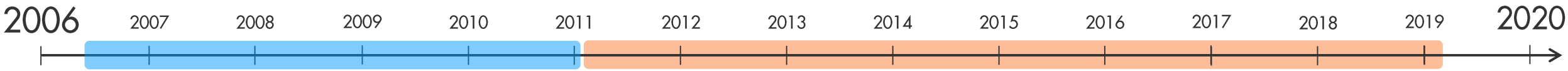

Requirement
Engineers
Entwicklung
& Test
User


2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020



Juergens, Deissenboeck et al. *Do Code Clones Matter?* ICSE 2009


Requirement
Engineers
Entwicklung
& Test
User


2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020

Dashboard
Activity
Findings
Metrics
Tests
Issues
Tasks
Architecture
Delta
Projects
System
Admin

Collapse

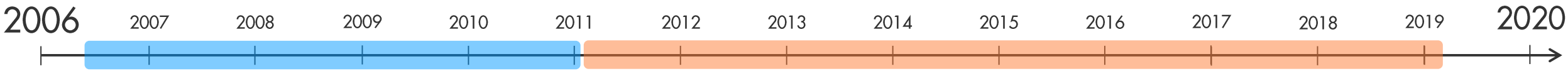
jenkins/test/src/main/java/org/jvnet/hudson/test/HudsonTestCase.java (revision 12d96a56...)

```
 if (lhs==null && rhs==null)     return;
 if (lhs==null)      fail("lhs is null while rhs="+rhs);
 if (rhs==null)      fail("rhs is null while lhs="+lhs);

 Constructor<?> lc = findDataBoundConstructor(lhs.getClass());
 Constructor<?> rc = findDataBoundConstructor(rhs.getClass());
 assertEquals("Data bound constructor mismatch. Different type?",lc,rc);

 List<String> primitiveProperties = new ArrayList<String>();

 String[] names = ClassDescriptor.loadParameterNames(lc);
 Class<?>[] types = lc.getParameterTypes();
 assertEquals(names.length,types.length);
 for (int i=0; i<types.length; i++) {
     Object lv = ReflectionUtils.getPublicProperty(lhs, names[i]);
     Object rv = ReflectionUtils.getPublicProperty(rhs, names[i]);

     if (Iterable.class.isAssignableFrom(types[i])) {
         Iterable lcol = (Iterable) lv;
         Iterable rcol = (Iterable) rv;
         Iterator ltr,rtr;
         for (ltr=lcol.iterator(), rtr=rcol.iterator(); ltr.hasNext() && rtr.hasNext();)
             Object litem = ltr.next();
             Object ritem = rtr.next();

             if (findDataBoundConstructor(litem.getClass())!=null) {
                 assertEqualDataBoundBeans(litem,ritem);
             } else {
                 assertEquals(litem,ritem);
             }
         }
         assertFalse("collection size mismatch between "+lhs+" and "+rhs, ltr.hasNext() ^
     } else
     if (findDataBoundConstructor(types[i])!=null || (lv!=null && findDataBoundConstructo
         // recurse into nested databound objects
         assertEqualDataBoundBeans(lv,rv);
     } else {
         primitiveProperties.add(names[i]);
     }
 }

 // compare shallow primitive properties
 if (!primitiveProperties.isEmpty())
     assertEqualBeans(lhs,rhs,Util.join(primitiveProperties,","));


*
 Makes sure that two collections are identical via {@link #assertEqualDataBoundBeans(Objec
/
blic void assertEqualDataBoundBeans(List<?> lhs, List<?> rhs) throws Exception {
 assertEquals(lhs.size(), rhs.size());
```

jenkins/test/src/main/java/org/jvnet/hudson/test/JenkinsRule.java (revision 3909f5ac...)

```
 if (lhs==null && rhs==null)     return;
 if (lhs==null)      fail("lhs is null while rhs="+rhs);
 if (rhs==null)      fail("rhs is null while lhs="+lhs);

 Constructor<?> lc = findDataBoundConstructor(lhs.getClass());
 Constructor<?> rc = findDataBoundConstructor(rhs.getClass());
 assertThat("Data bound constructor mismatch. Different type?", (Constructor)rc, is((Cons

 List<String> primitiveProperties = new ArrayList<String>();

 String[] names = ClassDescriptor.loadParameterNames(lc);
 Class<?>[] types = lc.getParameterTypes();
 assertThat(types.length, is(names.length));
 for (int i=0; i<types.length; i++) {
     Object lv = ReflectionUtils.getPublicProperty(lhs, names[i]);
     Object rv = ReflectionUtils.getPublicProperty(rhs, names[i]);

     if (lv != null && rv != null && Iterable.class.isAssignableFrom(types[i])) {
         Iterable lcol = (Iterable) lv;
         Iterable rcol = (Iterable) rv;
         Iterator ltr,rtr;
         for (ltr=lcol.iterator(), rtr=rcol.iterator(); ltr.hasNext() && rtr.hasNext();)
             Object litem = ltr.next();
             Object ritem = rtr.next();

             if (findDataBoundConstructor(litem.getClass())!=null) {
                 assertEqualDataBoundBeans(litem,ritem);
             } else {
                 assertThat(ritem, is(litem));
             }
         }
         assertThat("collection size mismatch between " + lhs + " and " + rhs, ltr.hasNex
                 is(false));
     } else
     if (findDataBoundConstructor(types[i])!=null || (lv!=null && findDataBoundConstructo
         // recurse into nested databound objects
         assertEqualDataBoundBeans(lv,rv);
     } else {
         primitiveProperties.add(names[i]);
     }
 }

 // compare shallow primitive properties
 if (!primitiveProperties.isEmpty())
     assertEqualBeans(lhs,rhs,Util.join(primitiveProperties,","));


*
 Makes sure that two collections are identical via {@link #assertEqualDataBoundBeans(Objec
/
blic void assertEqualDataBoundBeans(List<?> lhs, List<?> rhs) throws Exception {
```


Requirement
Engineers
Entwicklung
& Test
User


2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020

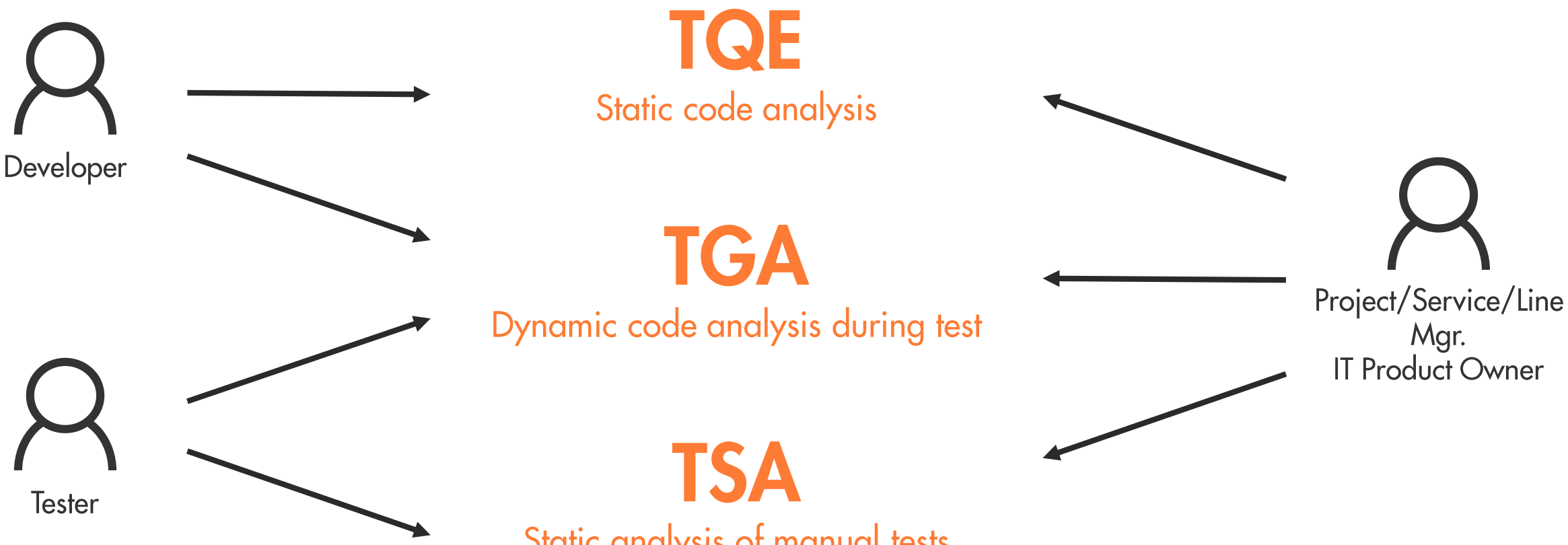
Wie sehen die Werkzeuge & Prozesse bei der Munich Re aus?

Uwe Proft

- Background im Software Engineering und Provider Management

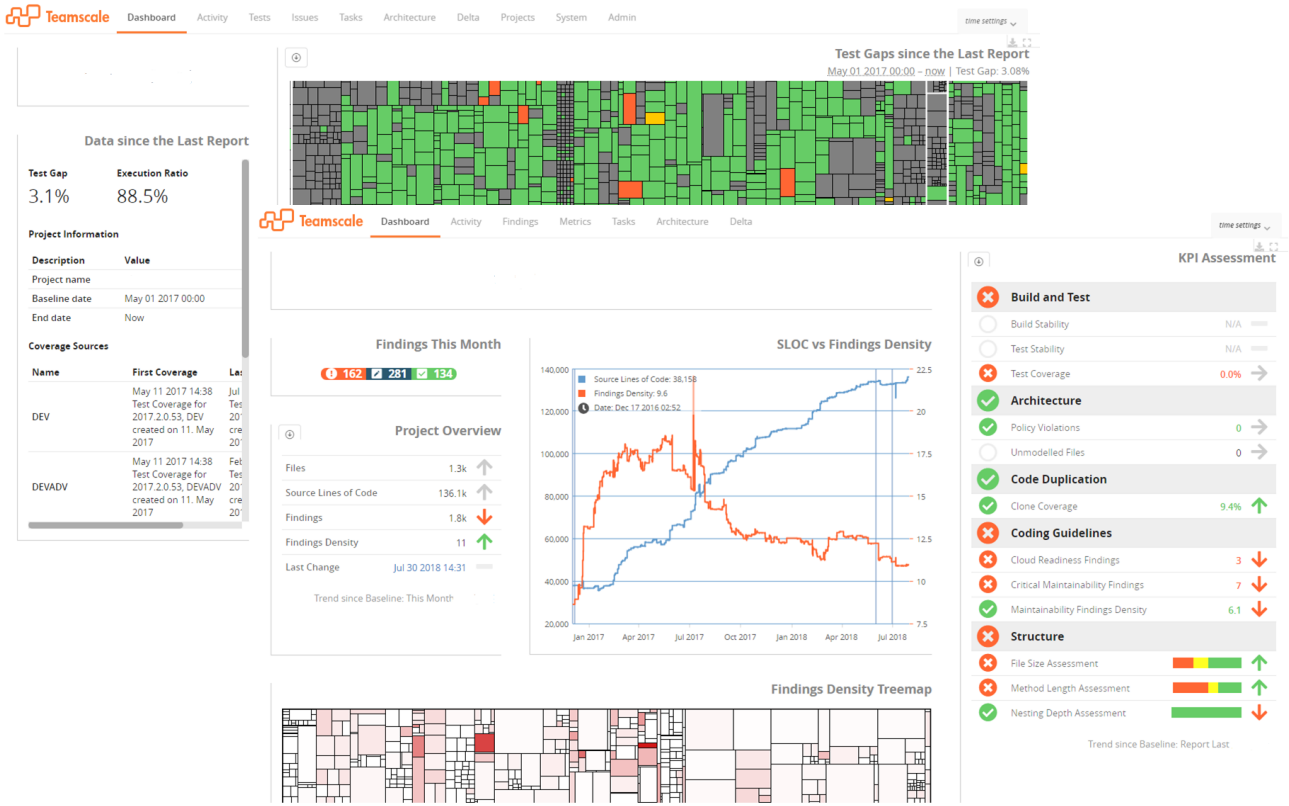
- Seit 7 Jahren bei der MR in Rollen zum Qualitätsmanagement
  - Erläutern Nutzen und Aufwand intern
  - Ausrollen, auch international an unterschiedlichen Standorten
  - Change-Management
  - Vermittlung der Messergebnisse für Beurteilung der Qualität von Zulieferern und Projekten
  - Steuerung des Teams der Quality Engineers (CQSE) bei der Munich Re

Quality Tools

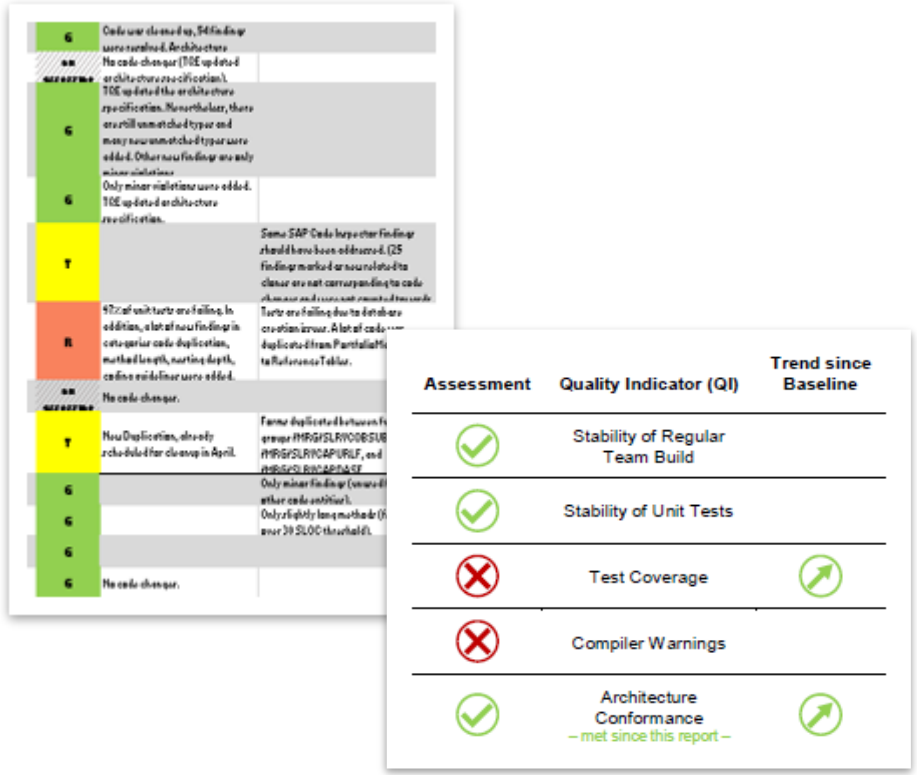


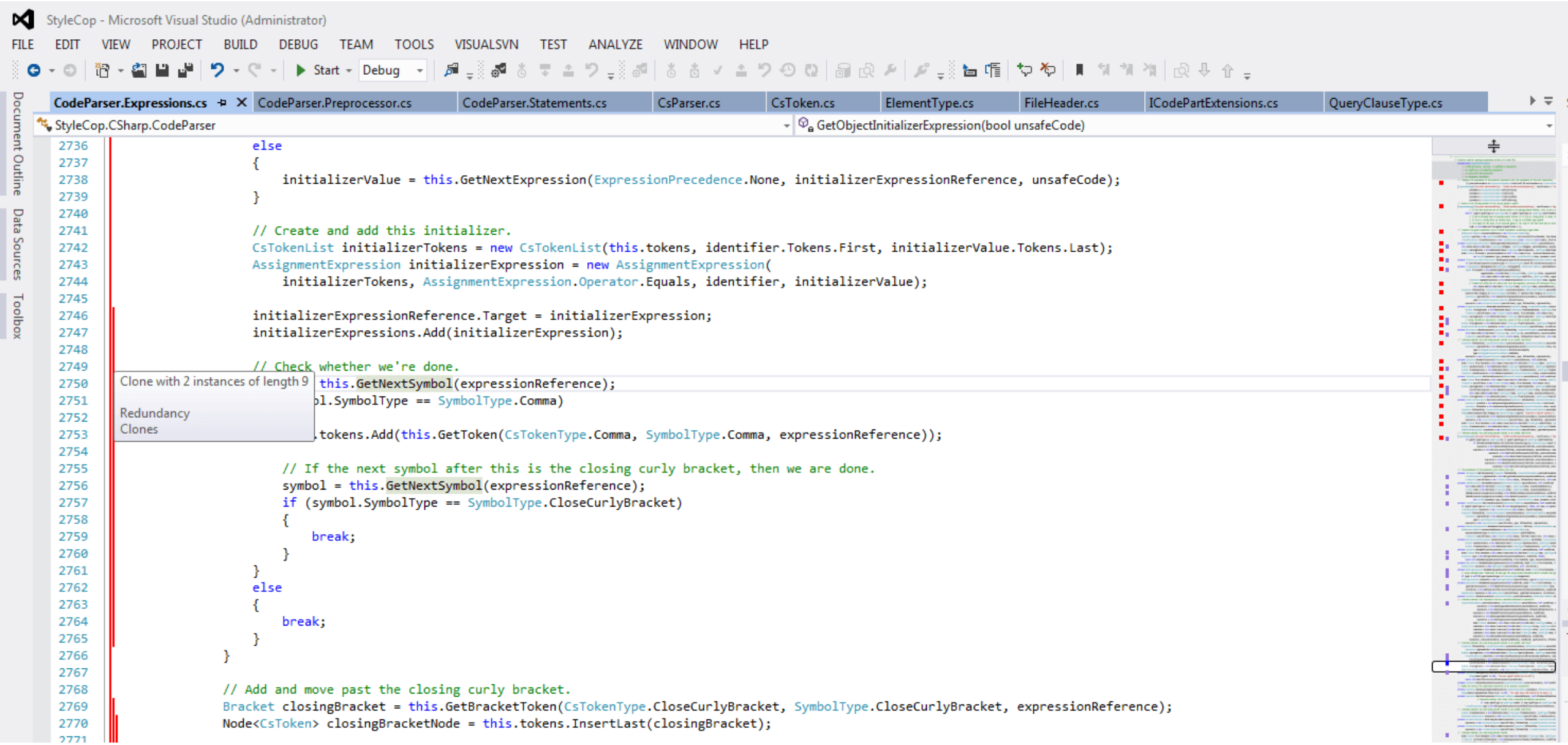
Munich Re Internal Services

Dashboards, IDE Plugin, Azure DevOps

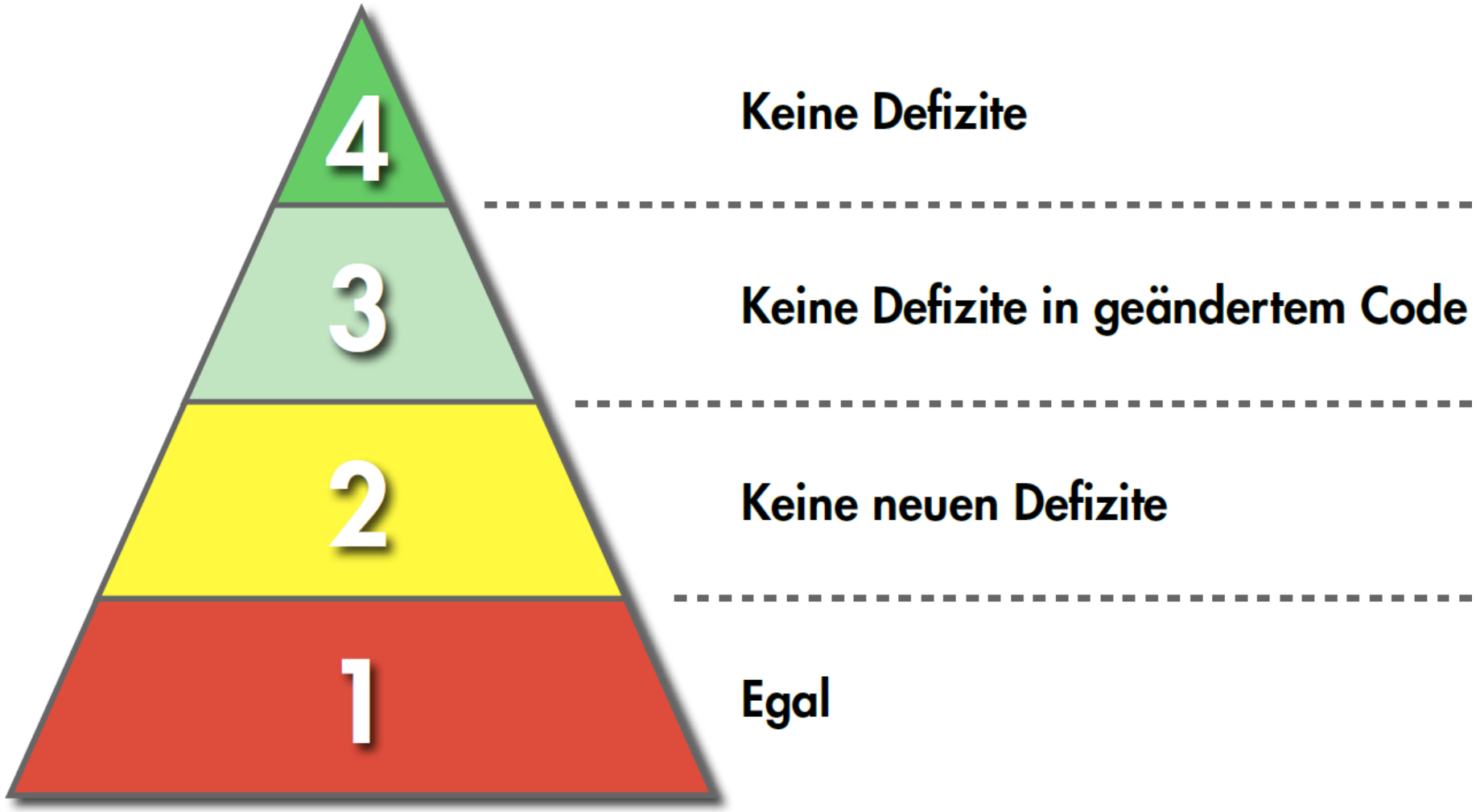


Monthly Assessments, Reports

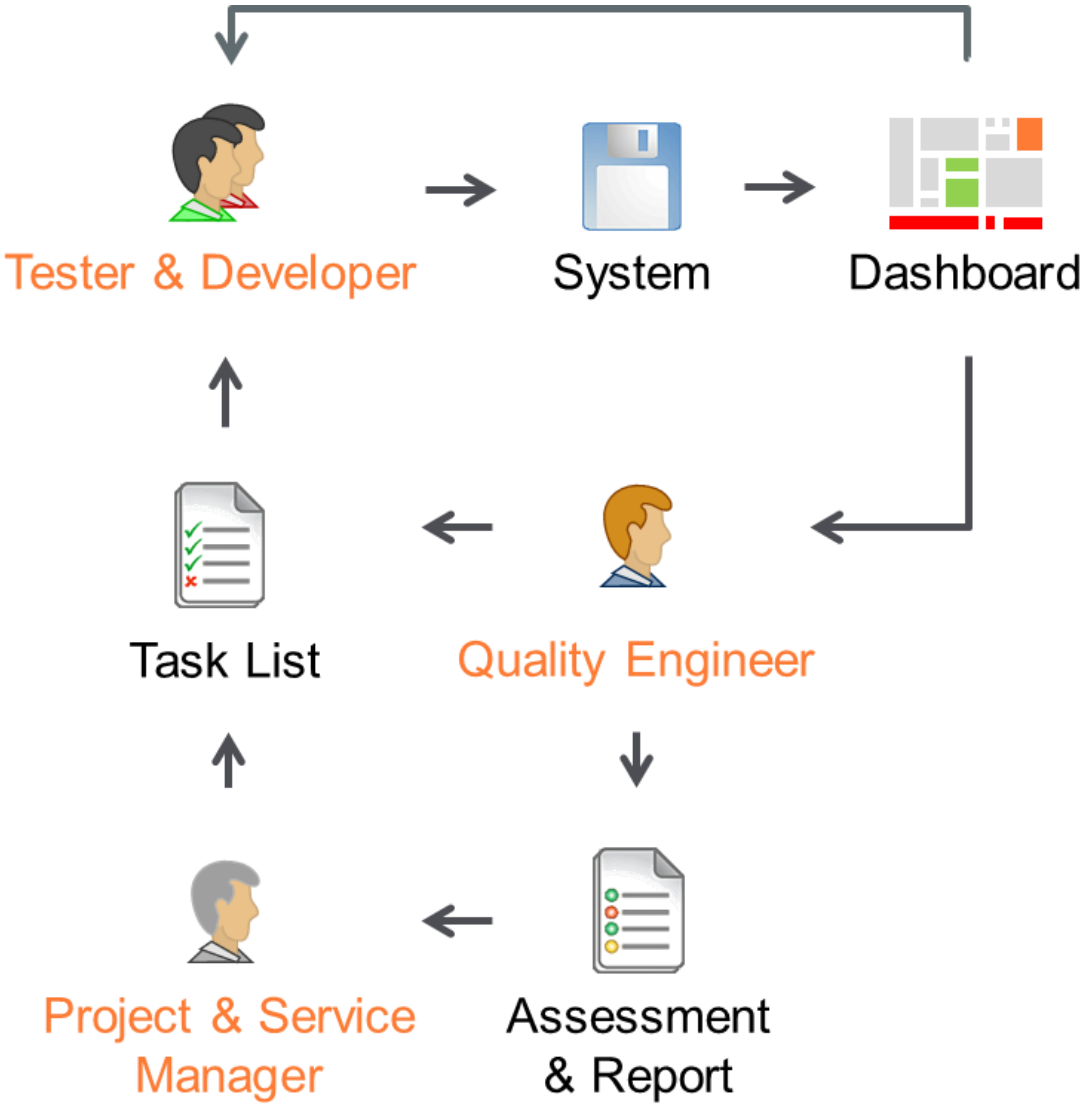



StyleCop - Microsoft Visual Studio (Administrator)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS VISUALSVN TEST ANALYZE WINDOW HELP
Start Debug
CodeParser.Expressions.cs
CodeParser.Preprocessor.cs
CodeParser.Statements.cs
CsParser.cs
CsToken.cs
ElementType.cs
FileHeader.cs
ICodePartExtensions.cs
QueryClauseType.cs
StyleCop.CSharp.CodeParser
GetObjectInitializerExpression(bool unsafeCode)
Document Outline
Data Sources
Toolbox
Clone with 2 instances of length 9
Redundancy
Clones
2736 else
2737 {
2738 initializerValue = this.GetNextExpression(ExpressionPrecedence.None, initializerExpressionReference, unsafeCode);
2739 }
2740
2741 // Create and add this initializer.
2742 CsTokenList initializerTokens = new CsTokenList(this.tokens, identifier.Tokens.First, initializerValue.Tokens.Last);
2743 AssignmentExpression initializerExpression = new AssignmentExpression(
2744 initializerTokens, AssignmentExpression.Operator.Equals, identifier, initializerValue);
2745
2746 initializerExpressionReference.Target = initializerExpression;
2747 initializerExpressions.Add(initializerExpression);
2748
2749 // Check whether we're done.
2750 this.GetNextSymbol(expressionReference);
2751 ol.SymbolType == SymbolType.Comma)
2752
2753 .tokens.Add(this.GetToken(CsTokenType.Comma, SymbolType.Comma, expressionReference));
2754
2755 // If the next symbol after this is the closing curly bracket, then we are done.
2756 symbol = this.GetNextSymbol(expressionReference);
2757 if (symbol.SymbolType == SymbolType.CloseCurlyBracket)
2758 {
2759 break;
2760 }
2761 }
2762 else
2763 {
2764 break;
2765 }
2766 }
2767
2768 // Add and move past the closing curly bracket.
2769 Bracket closingBracket = this.GetBracketToken(CsTokenType.CloseCurlyBracket, SymbolType.CloseCurlyBracket, expressionReference);
2770 Node<CsToken> closingBracketNode = this.tokens.InsertLast(closingBracket);

Qualitätsziele



Prozess


Tester & Developer
System
Dashboard
Task List
Quality Engineer
Project & Service Manager
Assessment & Report

Portfolio Overview – Links to Dashboards & Monthly Assessments

Application	Dev	Test	TQE	TGA	TSA
[redacted]	[redacted] IT1.5 [redacted]	[redacted]	2019-12		
[redacted]	[redacted] IT1.2 [redacted]	[redacted]			2019-12
[redacted]	[redacted] IT1.5 [redacted]	[redacted]	2019-12	2019-12	2019-12
[redacted]	[redacted] IT1.5 [redacted]	[redacted]			
[redacted]	[redacted] IT1.8 [redacted]	[redacted]	2019-12		2019-12

Portfolio Overview – Trends

Application	Dev	TGA	TSA
[redacted]	[redacted] IT		
[redacted]	[redacted] IT1.2 [redacted]		2019-12
[redacted]	[redacted] IT1.5 [redacted]	2019-12	2019-12
[redacted]	[redacted] IT1.5 [redacted]		
[redacted]	[redacted] IT1.8 [redacted]		2019-12

TQE assessment trend for

Assessment	Comment	QG relevant findings	Details
2019-09	Only one small finding in changed code	1	Show Details
2019-08	No code changes.	0	
2019-07	Only minor new findings	6	Show Details
2019-06	Only a small change with no findings churn.	0	
2019-05	Only 2 small findings in modified code.	2	
2019-04	Mostly minor violations.	70	Show Details
	Notable findings: • Naming convention violations in `TmOneParam` • Method threshold violation in method `DeleteProcessYear` of class `ProcessYearsController` • Method threshold violation in a lambda in class `LossChartService` • Cloning between `ReverseTriangleGrid` and `CommissionTriangleGrid` (c.f. here)		
2019-03	Minor violations only.	3	
2019-02	Only 5 new findings. Remaining findings are located in code that was changed during a migration to Angular 7 and thus can be ignored for this assessment.	39	Show Details

Portfolio Overview – Trends

Application	Dev	TSA
[redacted]	[redacted] IT	
[redacted]	[redacted] IT1.2 [redacted]	
[redacted]	[redacted] IT1.5 [redacted]	
[redacted]	[redacted] IT1.5 [redacted]	
[redacted]	[redacted] IT1.8 [redacted]	

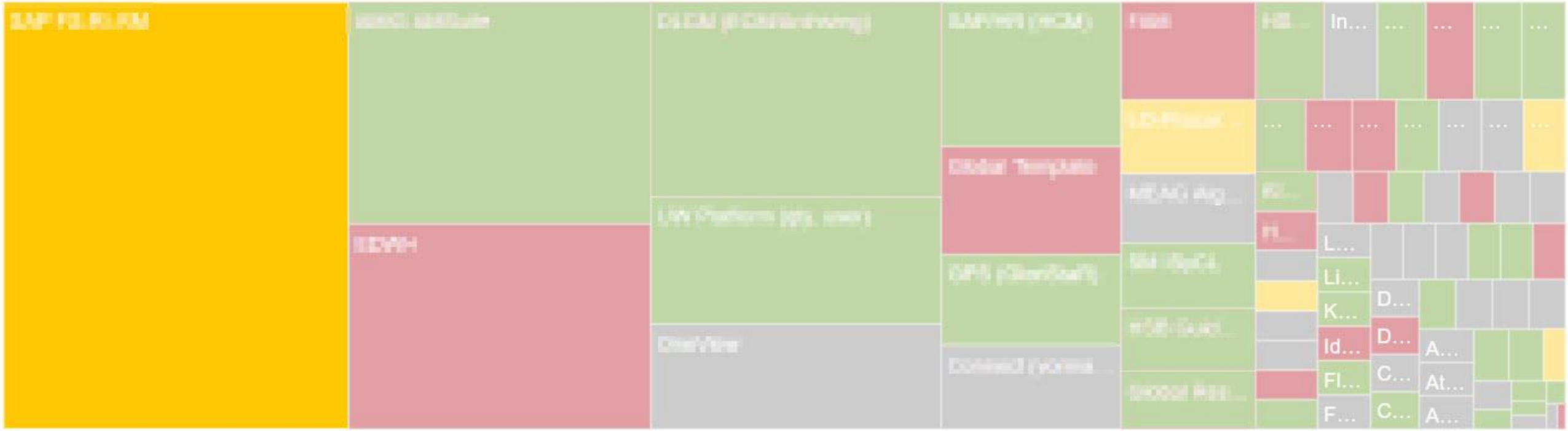
TQE assessment trend for

Assessment	Comment	QG relevant findings	Details
2019-09	Only one small finding in changed code	1	Show Details
2019-08	No code changes.	0	
2019-07	Only minor new findings	6	Show Details
2019-06	Only a small change with no findings churn.	0	
2019-05	Only 2 small findings in modified code.	2	
2019-04	Mostly minor violations.	70	Show Details
2019-03	Minor violations only.	3	
2019-02	Only 5 new findings. Remaining findings are located in code that was changed during a migration to Angular 7 and thus can be ignored for this assessment.	39	Show Details

Notable findings:

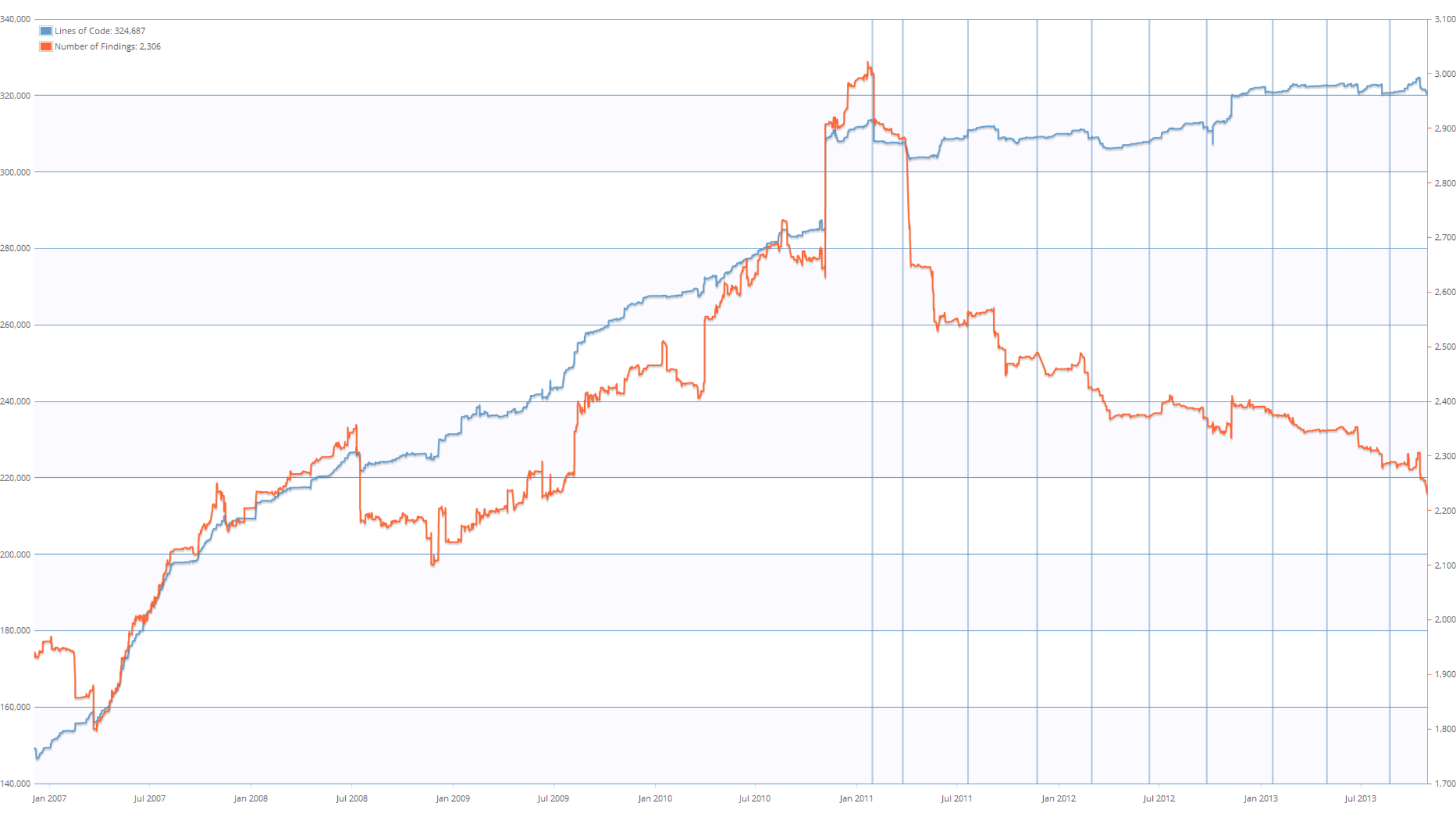
- Naming convention violations in `TmOneParam`
- Method threshold violation in method `DeleteProcessYear` of class `ProcessYearsController`
- Method threshold violation in a lambda in class `LossChartService`
- Cloning between `ReverseTriangleGrid` and `CommissionTriangleGrid` (c.f. here)

Monthly Assessment Results Portfolio Aggregation



Name	Tool	Assessment	Comments
[redacted]	TQE	GREEN	Only few violations
[redacted]	TGA	YELLOW	Some relevant test gaps
[redacted]	TSA	GREEN	The team added 47 new test case, changed 11 test case, moved 5 test case and removed 15 test cases, which introduced 0 new findings.

Was bringt's?

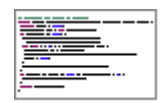

Lines of Code: 324,687
Number of Findings: 2,306
340,000
320,000
300,000
280,000
260,000
240,000
220,000
200,000
180,000
160,000
140,000
3,100
3,000
2,900
2,800
2,700
2,600
2,500
2,400
2,300
2,200
2,100
2,000
1,900
1,800
1,700
Jan 2007
Jul 2007
Jan 2008
Jul 2008
Jan 2009
Jul 2009
Jan 2010
Jul 2010
Jan 2011
Jul 2011
Jan 2012
Jul 2012
Jan 2013
Jul 2013

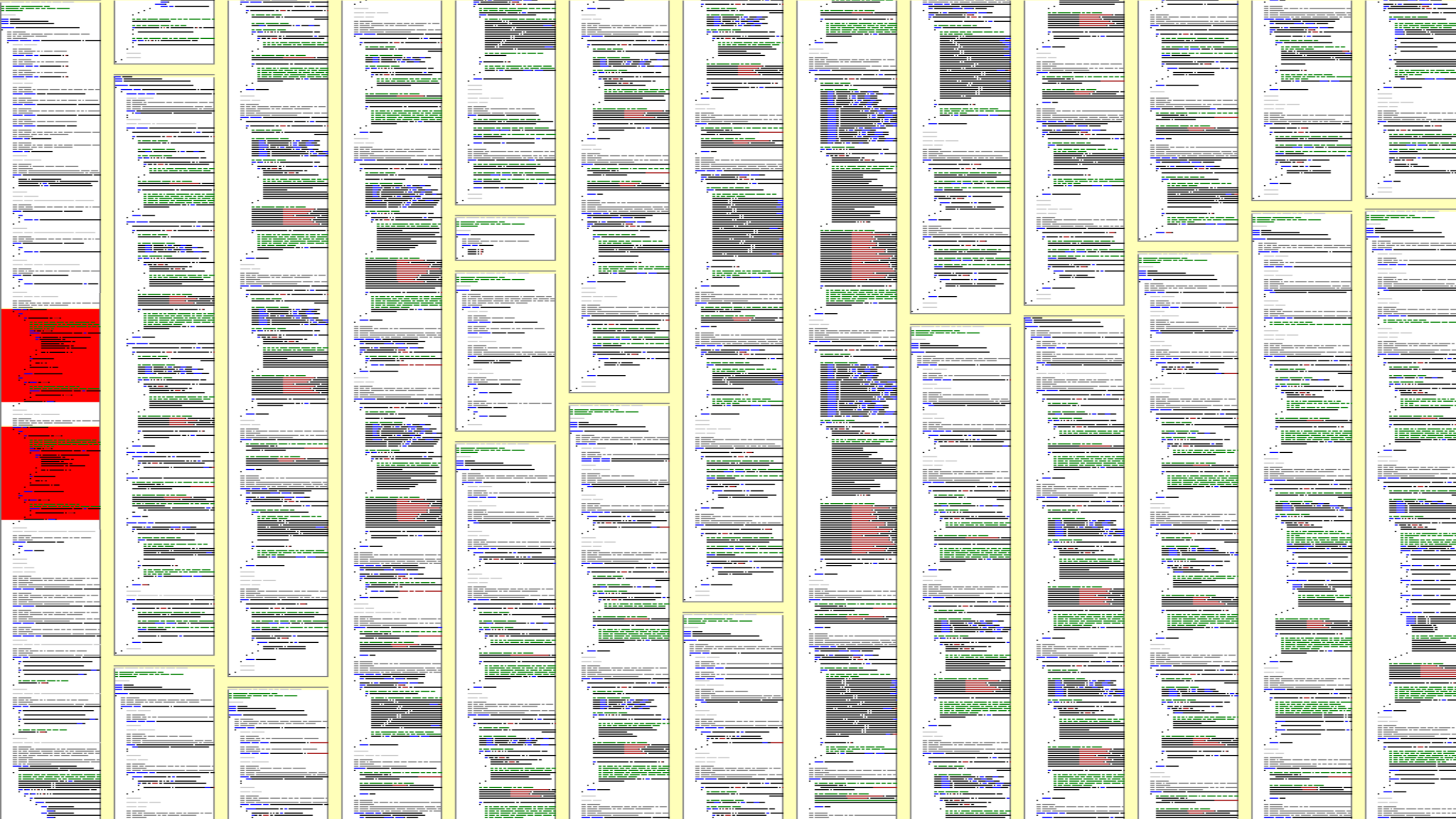
Wie können wir den Nutzen quantifizieren?

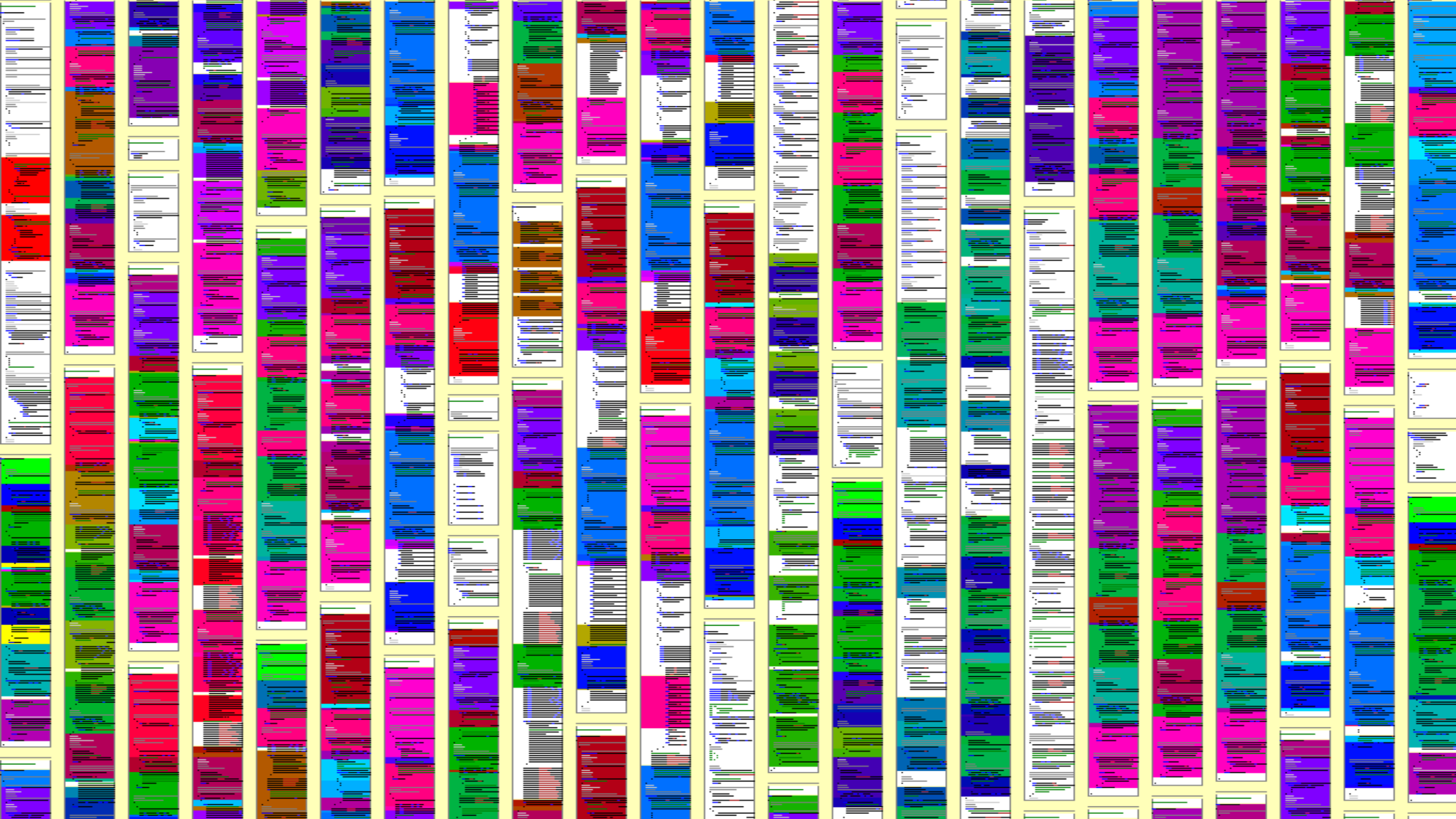
```java
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
   boolean found = false;
   StringBuffer res = new StringBuffer();
   if (elements != null) {
      Index.getInstance().setCurrentRenderer(
            FlatReferenceRenderer.getInstance());
      for (int i = 0; i < elements.length; i++) {
         ModelElement el = elements[i];
         res.append(showElementLink(el)).append(HTML.LINE_BREAK);
         found = true;
      }
      Index.getInstance().resetCurrentRenderer();
   }
   if (!found && nomsg != null && nomsg.length() > 0) {
      res.append(HTML.italics(nomsg));
   }
   return res.toString();
}
```

```java
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
   boolean found = false;
   StringBuffer res = new StringBuffer();
   if (elements != null) {
      Index.getInstance().setCurrentRenderer(
            FlatReferenceRenderer.getInstance());
      for (int i = 0; i < elements.length; i++) {
         ModelElement el = elements[i];
         res.append(showElementLink(el)).append(HTML.LINE_BREAK);
         found = true;
      }
      Index.getInstance().resetCurrentRenderer();
   }
   if (!found && nomsg.length() > 0) {
      res.append(HTML.italics(nomsg));
   }
   return res.toString();
}
```

```java
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
    boolean found = false;
    StringBuffer res = new StringBuffer();
    if (elements != null) {
        Index.getInstance().setCurrentRenderer(
                FlatReferenceRenderer.getInstance());
        for (int i = 0; i < elements.length; i++) {
            ModelElement el = elements[i];
            res.append(showElementLink(el)).append(HTML.LINE_BREAK);
            found = true;
        }
        Index.getInstance().resetCurrentRenderer();
    }
    if (!found && nomsg != null && nomsg.length() > 0) {
        res.append(HTML.italics(nomsg));
    }
    return res.toString();
}
```







```java
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
   boolean found = false;
   StringBuffer res = new StringBuffer();
   if (elements != null) {
      Index.getInstance().setCurrentRenderer(
            FlatReferenceRenderer.getInstance());
      for (int i = 0; i < elements.length; i++) {
         ModelElement el = elements[i];
         res.append(showElementLink(el)).append(HTML.LINE_BREAK);
         found = true;
      }
      Index.getInstance().resetCurrentRenderer();
   }
   if (!found && nomsg != null && nomsg.length() > 0) {
      res.append(HTML.italics(nomsg));
   }
   return res.toString();
}
```

```java
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
   boolean found = false;
   StringBuffer res = new StringBuffer();
   if (elements != null) {
      Index.getInstance().setCurrentRenderer(
            FlatReferenceRenderer.getInstance());
      for (int i = 0; i < elements.length; i++) {
         ModelElement el = elements[i];
         res.append(showElementLink(el)).append(HTML.LINE_BREAK);
         found = true;
      }
      Index.getInstance().resetCurrentRenderer();
   }
   if (!found && nomsg.length() > 0) {
      res.append(HTML.italics(nomsg));
   }
   return res.toString();
}
```

$$\text{Anzahl } \frac{Fehler}{Jahr} \times \text{Fehlerfolgekosten } \frac{PT}{Fehler}$$

$$\text{Anzahl } \frac{Fehler}{Jahr} \times \text{Fehlerfolgekosten } \frac{PT}{Fehler}$$

Studie

Munich Re LV 1871

- Über 100 Fehler in produktiver Software

17	44	46
Kritisch	Nutzersichtbar	Nicht nutzersichtbar

- 52% aller ungewollten Unterschiede fehlerhaft

Juergens, Deissenboeck et al. *Do Code Clones Matter?* ICSE 2009

#Fehler durch inkonsistente Klone

Daten aus Studie

- 3 Systeme von Munich Re analysiert
- 79 Fehler gefunden (Impact auf Funktionalität, nicht nur Wartbarkeit o.ä.)
- Systeme waren produktiv, einzelne Fehler schon durch Anwender als Tickets reportet
- 1 Produktionsfehler durch inkonsistente Klone / 17k SLOC

Bedeutung heute

- Betrachtetes Portfolio der Munich Re umfasst ca. 8,25 Millionen SLOC
- Konservative Annahme: Clone Management spart 1 Produktionsfehler pro 50k SLOC pro Jahr
- 8,25 Millionen SLOC / 50k = 165

$$\text{Anzahl } \frac{Fehler}{Jahr} \times \text{Fehlerfolgekosten } \frac{PT}{Fehler}$$

$$165\ \frac{Fehler}{Jahr} \times \text{Fehlerfolgekosten}\ \frac{PT}{Fehler}$$

$$165\ \frac{Fehler}{Jahr} \times \textbf{Fehlerfolgekosten}\ \frac{PT}{Fehler}$$

ØFehlerfolgekosten von Fehlern in Produktion

Mögliche Auswirkungen fehlerhafter Software

- Nutzer bekommen falsche Ergebnisse
- Anwendung stürzt ab
- Daten gehen verloren
- Frustration bei Nutzern (Kunden und Mitarbeiter)

? PT

Aufwand für Reparatur

- Nutzer schreibt Ticket für Fehler
- Debugging (Nachstellen, Diagnose, …)
- Fixing
- Test
- Ggf. Deployment

? PT

ØFehlerfolgekosten von Fehlern in Produktion

Mögliche Auswirkungen fehlerhafter Software

- Nutzer bekommen falsche Ergebnisse
- Anwendung stürzt ab
- Daten gehen verloren
- Frustration bei Nutzern (Kunden und Mitarbeiter)

0 PT: bewusste Unterschätzung

Aufwand für Reparatur

- Nutzer schreibt Ticket für Fehler
- Debugging (Nachstellen, Diagnose, …)
- Fixing
- Test
- Ggf. Deployment

3 PT

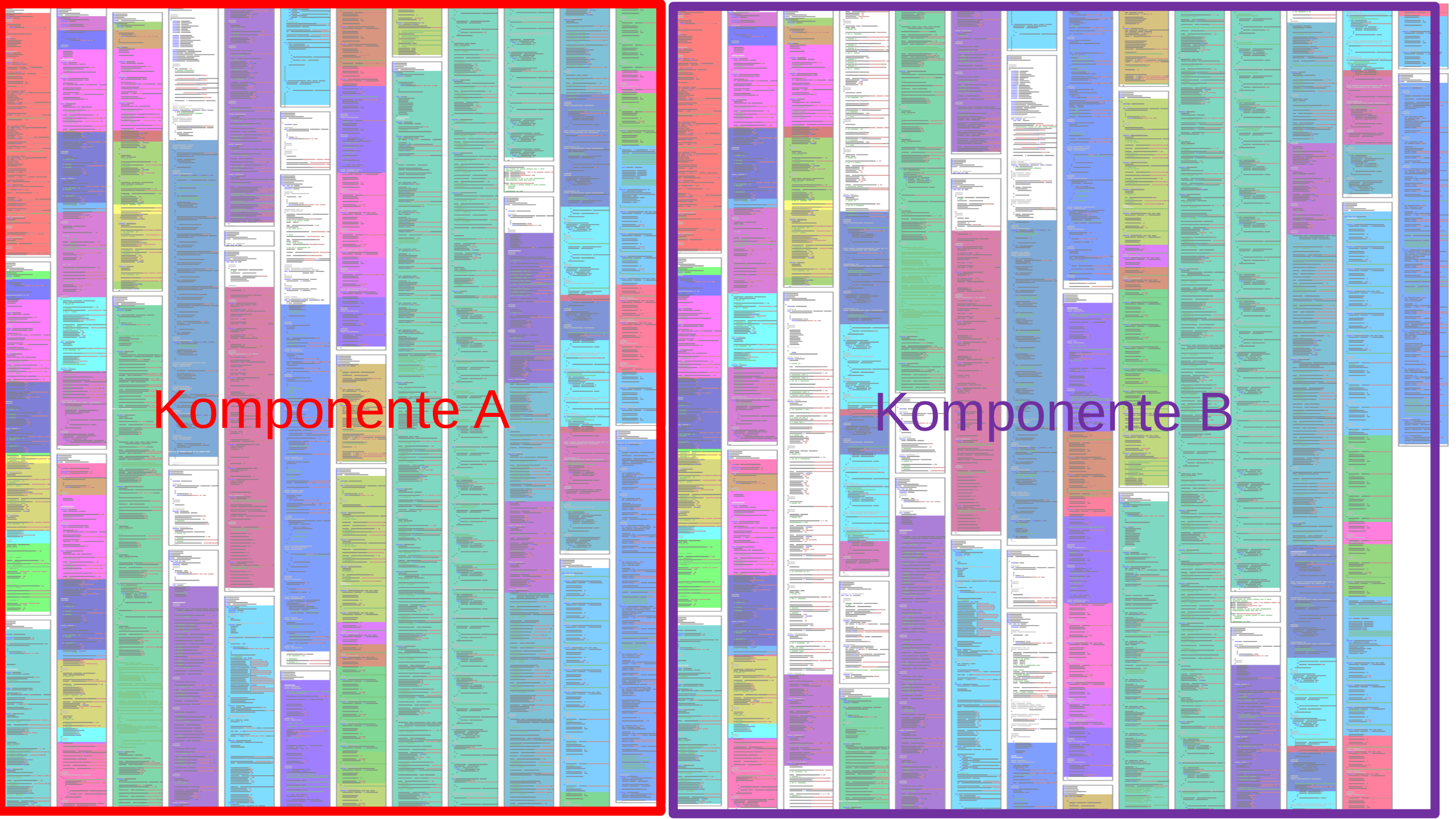
$$165\ \frac{Fehler}{Jahr} \times \textbf{Fehlerfolgekosten}\ \frac{PT}{Fehler}$$

$$165\ \frac{Fehler}{Jahr} \times 3\ \frac{PT}{Fehler}$$

$$495 \frac{PT}{Jahr}$$

$$500 \frac{PT}{Jahr}$$

Munich Re spart durch Einsatz von Clone Management jährlich ca. 500 PT Aufwand für Fehlerbehebung


Komponente A
Komponente B

How Much is a Clone?

Elmar Juergens, Florian Deissenboeck
Institut für Informatik, Technische Universität München, Germany
{juergens,deissenb}@in.tum.de

Abstract

Real-world software systems contain substantial amounts of cloned code. While the negative impact of cloning on software maintenance has been shown in principle, we currently cannot quantify it in terms of increased maintenance costs. However, as long as its economic impact cannot be quantified, control of cloning is probable to be neglected in practice. This paper presents an analytical cost model to estimate the maintenance effort increase caused by code cloning. The cost model can be used to assess the economic impact of cloning in a system and to evaluate investments in clone management tool support. To show its applicability, we report on a case study that instantiates the cost model for 11 industrial software systems.

1. Introduction

Code cloning abounds in real world software. Numerous studies report substantial amounts of cloning in open source and industrial systems [26, 32]. To name just a few, significant cloning was detected in GCC [14], X Windows [2], Linux and JDK [30]. Although the extent varies between systems, cloning occurs across programming languages, development contexts and application domains. Furthermore, it is not limited to source code. Recent studies have discovered substantial amounts of cloning in models [10] and requirements specifications [12, 15]. Cloning thus has to be considered as a phenomenon that occurs across different software artifacts.

Substantial research effort on software clones has established the negative impact of cloning on software maintenance activities in general [17]. It, often unnecessarily, increases code size and thus effort required for size-related activities such as inspections. Since changes to a piece of code, such as a bug fix, often need to be performed to its duplicates as well, cloning increases modification effort. If duplicates are missed when cloned code is modified, inconsistencies can be introduced into the system that can lead to faults, or existing faults can fail to be removed from the system. A study we published in [22] uncovered over 100 faults in productive software through analysis of unintentionally inconsistent changes to cloned code.

While the negative consequences of cloning have firmly been established qualitatively, their *quantitative* impact remains unclear. We are currently unaware of how large the economic impact of cloning is w.r.t. the total maintenance effort. We thus cannot determine how harmful cloning is for a project in economic terms. Industrial software engineering is done with scarce resources—factors that cannot be quantified are probable to be neglected in favor of competing factors that can be quantified, such as open bug issues or change requests. As long as we do not know the costs cloning causes, clone control is prone to be neglected—even though cloning could be the root cause, and open bugs and change requests the symptoms.

Understanding of the costs caused by cloning is a necessary foundation to evaluate alternative clone management strategies. Do expected maintenance cost reductions justify the effort required for clone removal? How large are the potential savings that clone management tools can provide? We need a clone cost model to answer these questions.

Problem While the negative consequences of cloning are established qualitatively, the economic impact of cloning on maintenance is poorly understood. Consequently, we lack the foundation to assess the economic harmfulness of cloning and to evaluate alternative clone management strategies.

Contribution We propose an analytical cost model to estimate the impact that code cloning has on software maintenance efforts. We present a case study that instantiates the cost model for 11 industrial software systems and estimates maintenance effort increase and potential benefits achievable through clone management tool support. The paper presents work in progress—we point out shortcomings and directions of future research—but provides a step towards a more economically substantiated discussion of cloning.

2. Terms & Definitions

Clones are regions of similar code. In the literature, different definitions of similarity are employed [26, 32], mostly based on syntactical characteristics. In this paper, we require clones to be semantically similar in the sense that they implement one or more common concepts. This redundant concept implementations gives rise to change coupling: when the concept changes, all of its implementations—the clones—need to be changed. In addition, we require clones to be syntactically similar. While syntactic similarity is not required for change coupling, existing clone detection approaches rely on syntactic similarity to detect clones. Hence, we employ the term *clone* to denote syntactically similar code regions that contain redundant implementation of one or more concepts. A *clone group* is a set of clones. Clones in a single group are referred to as *siblings*.

We use the term *failure* to denote an incorrect output of a software visible to the user. A *fault* is the cause of a potential failure in the source code.

Lines of code (LOC) denote the sum of the lines of code of all source files, including comments and blank lines. *Source statements* (SS) are the number of all source code statements, *not* taking commented or blank lines and code formatting into account. *Redundancy free source statements* (RFSS) are the number of source statements, if cloned source statements are only counted once. It thus estimates the size of a system from which all cloning is perfectly removed. For example, if a file contains 100 statements (and no clones) in version 1, and 50 of them are duplicated in the file to create version 2, SS increases to 150, but RFSS remains at 100. *Overhead* denotes the ratio by which a system's size has increased due to cloning. It is computed as $\frac{SS}{RFSS} - 1$. For the above example, the resulting overhead is 0.5, denoting a cloning induced size increase by 50%.

3. Maintenance Process

This section introduces the software maintenance process on which the cost model is based. It *qualitatively* describes the impact of cloning for each process activity and discusses potential benefits of clone management tools. The process is loosely based on the IEEE 1219-standard [18] that describes the activities carried on single change requests (CRs) in a waterfall fashion. The successive execution of activities that, in practice, are typically carried out in an interleaved and iterated manner, serves the clarity of the model but does limit its application to waterfall-style processes.

Analysis (A) studies the feasibility and scope of the change request to devise a preliminary plan for design, implementation and quality assurance. Most of it takes place on the problem domain. Analysis is not impacted by code cloning, since code does not play a central part in it.[1]

Location (L) determines a set of change start points. It thus performs a mapping from problem domain concepts affected by the CR to the solution domain. Location does not contain impact analysis, that is, consequences of modifications of the change start points are not analyzed. Location involves inspection of source code to determine change start points. We assume that the location effort is proportional to the amount of code that gets inspected.

Cloning increases the size of the code that needs to be inspected during location and thus affects location effort. We are not aware of tool support to alleviate the consequences of code cloning on location.

Design (D) uses the results of analysis and location as well as the software system and its documentation to design the modification of the system. We assume that design is not impacted by cloning. This is a conservative assumption, since for a heavily cloned system, design could attempt to avoid modifications of heavily cloned areas.

Impact Analysis (IA) uses the change start points from location to determine where changes in the code need to be made to implement the design. The change start points are typically not the only places where modifications need to be performed—changes to them often require adaptations in use sites. We assume that the effort required for impact analysis is proportional to the number of source locations that need to be determined.

If the concept that needs to be changed is implemented redundantly in multiple locations, all of them need to be changed. Cloning thus affects impact analysis, since the number of change points is increased by cloned code. Tool support (clone indication) simplifies impact analysis of changes to cloned code. Ideal tool support could reduce cloning effect on impact analysis to zero.

Implementation (Impl) realizes the designed change in the source code. We differentiate between two classes of changes to source code. *Additions* add new source code to the system without changing existing code. *Modifications* alter existing source code and are performed to the locations determined by impact analysis. We assume that effort required for implementation is proportional to the amount of code that gets added or modified.

We assume that adding new code is unaffected by cloning in existing code. Implementation is still affected by cloning, since modifications to cloned code need to be performed multiple times. Linked editing tools could, ideally, reduce effects of cloning on implementation to zero.

Quality Assurance (QA) comprises all testing and inspection activities carried out to validate that the modification satisfies the change request. We assume a smart quality assurance strategy—only code affected by the change is processed. We do not limit the maintenance process to a specific quality assurance technique. However, we assume that quality assurance steps are systematically applied, *e. g.*, all changes are inspected or testing is performed until a certain test coverage is achieved on the affected system parts. Consequently, we assume that quality assurance effort is proportional to the amount of code on which quality assurance is performed.

We differentiate two effects of cloning on quality assurance: cloning increases the change size and thus the amount of modified code that needs to be quality assured. Second, just as modified code, added code can contain cloning. This also increases the amount of code that needs to be quality assured and hence the required effort. We are not aware of tool support that can substantially alleviate the consequences of cloning on quality assurance.

Other (O) comprises further activities, such as, *e. g.*, delivery and deployment, user support or change control board meetings. Since code does not play a central part in these activities, they are not affected by cloning.

4. Detailed Cost Model

This section introduces a detailed cost model that quantifies the impact of cloning on software maintenance. The model assumes each activity of the maintenance process to be completed. It is thus not suitable to model partial change request implementations that are aborted at some point.

4.1. General Approach

The *total* maintenance effort E is the sum of the efforts of individual change requests:

$$E = \sum_{cr \in CR} e\,(cr)$$

The scope of the cost model is determined by the population of the set CR: to compute the maintenance effort for a time span t, it is populated with all change requests that are realized in that period. Alternatively, if the total lifetime maintenance costs are to be computed, CR is populated with all change requests ever performed on the system. The model can thus scale to different project scopes.

The effort of a single change request $cr \in CR$ is expressed by $e\,(cr)$. It is the sum of the efforts of the individual activities performed during the realization of the cr. The activity efforts are denoted as e_X, where X identifies the activity. For brevity, we omit (cr) in the following:

$$e = e_A + e_L + e_D + e_{IA} + e_{Impl} + e_{QA} + e_O$$

In order to model the impact of cloning on maintenance efforts, we split e into two components: *inherent effort* e^- and *cloning induced overhead* e^+. e^- is independent of cloning. It captures the effort required to perform an activity on a hypothetical version of the software that does not contain cloning. e^+, in contrast, captures the effort penally caused by cloning. Total effort is expressed as the sum of the two:

$$e = e^- + e^+$$

The increase in efforts due to cloning, Δe^-, is captured by $\frac{e^-_{clone}}{e^-} - 1$, or simply $\frac{e^-}{e^-}$. The cost model thus expresses cloning induced overhead *relative* to the inherent effort required to realize a change request. The increase in total maintenance efforts due to cloning, ΔE , is proportional to the average effort increase per change request and thus captured by the same expression.

4.2. Activity Models

The activities *Analysis*, *Design*, and *Other* are not impacted by cloning. Their cloning induced effort, e^+, is thus zero. Their total efforts hence equal their inherent efforts.

Location effort depends on code size. Cloning increases code size. We assume that, on average, increase of the amount of code that needs to be inspected during location is proportional to the cloning-induced increase in the size of the entire code base. Size increase is captured by *overhead*:

$$e^+_L = e^-_L \cdot overhead$$

Impact analysis effort depends on the number of change points that need to be determined. Cloning increases the number of change points. We assume that e^+_{IA} is proportional to the cloning-induced increase in the number of source locations. This increase is captured by *overhead*:

$$e^+_{IA} = e^-_{IA} \cdot overhead$$

Implementation effort comprises both addition and modification effort: $e_{Impl} = e_{Impl_{add}} + e_{Impl_{mod}}$. We assume that effort required for additions is unaffected by cloning in existing source code. We assume that the effort required for modification is proportional to the amount of code that gets modified, *i. e.*, the number of source locations determined by impact analysis. Its cloning induced overhead is, consequently, affected by the same increase as impact analysis: $e^+_{Impl^-} = e^+_{Impl^-_{mod}} \cdot overhead$.

The modification ratio *mod* captures the modification-related part of the inherent implementation effort: $e^-_{Impl_{mod}} = e_{Impl} \cdot mod$. Consequently, $e^+_{Impl^-}$ is:

$$e^+_{Impl^-} = e^-_{Impl} \cdot mod \cdot overhead$$

Quality Assurance effort depends on the amount of code on which quality assurance gets performed. Both modifications and additions need to be quality assured. Since the measure *overhead* captures size increase of both additions and modifications, we do not need to differentiate between them, if we assume that cloning is, on average, similar in modified and added code. The increase in quality assurance effort is hence captured by the *overhead* measure:

$$e^+_{QA} = e^-_{QA} \cdot overhead$$

4.3. Maintenance Effort Increase Model

Based on the models for the individual activities, we model cloning induced maintenance effort e^+ for a single change request like this:

$$e^+ = overhead \cdot (e^-_L + e^-_{IA} + e^-_{Impl} \cdot mod + e^-_{QA}) + e^-_O$$

The relative cloning induced overhead is computed as follows:

$$\Delta e = \frac{overhead \cdot (e^-_L + e^-_{IA} + e^-_{Impl} \cdot mod + e^-_{QA})}{e^-_A + e^-_L + e^-_D + e^-_{IA} + e^-_{Impl} + e^-_{QA} + e^-_O}$$

This model allows to compute the relative effort increase in maintenance costs caused by cloning. It does not take consequences of cloning on program correctness into account. This is done in the next section.

4.4. Fault Increase

Quality assurance is not perfect. Even if performed thoroughly, faults may remain unnoticed and cause failures in production.

Quality assurance can be decomposed into two subactivities: fault detection and fault removal. We assume that, independent of the quality assurance technique, the effort required to detect a single fault in a system depends primarily on its fault density. We furthermore assume, that average fault removal effort for a system is independent of its size and fault density. These assumptions allow us to reason about the number of remaining faults in similar systems of different size but equal fault densities. If a QA procedure is applied with the same amount of available effort per unit of size, we expect a similar reduction in defect density, since the similar defect densities imply equal costs for fault location per unit. For these systems, the same number of faults can thus be detected and fixed per unit. For two systems A and B, with B having twice the size and available QA effort, we expect a similar reduction of fault density. However, since B is twice as big, the same fault density means twice the absolute number of remaining faults.

A system that contains cloning and its hypothetical version without cloning are such a pair of similar systems. We assume that fault density is similar between cloned code and non-cloned code—cloning duplicates both correct and faulty statements. Besides system size, cloning also increases the absolute number of faults contained in a system. If the amount of effort available for quality assurance is increased by *overhead* w.r.t. the system without cloning, the same reduction in fault density can be achieved. However, the *absolute* number of faults is still larger by *overhead*.

This reasoning assumes that developers are completely ignorant of cloning. That is, if a fault is fixed in one clone, it not immediately fixed in any of its siblings. Instead, faults in siblings are expected to be detected independently. Empirical data confirms that inconsistent bug fixes do frequently occur in practice [22]. However, it also confirms that clones are often maintained consistently. Both assuming entirely consistent or entirely inconsistent evolution is thus not realistic.

In practice, a certain amount of the defects that are detected in cloned code are hence fixed in some of the sibling clones. This reduces the cloning induced overhead in remaining fault counts. However, unless all faults in clones are fixed in all siblings, resulting fault counts remain higher than in systems without cloning. Therefore, cloning can have negative consequences on programm correctness beyond activity effort increase captured by the detailed model.

4.5. Tool Support

Clone management tools can alleviate the consequences of cloning on maintenance efforts. We adapt the detailed model to quantify the impact of clone management tools. We evaluate the upper bound of what two different types of clone management tools can achieve.

Clone Indication makes cloning relationships in source code available to developers, for example through *clone bars* in the IDE that mark cloned code regions. Examples for clone indication tools include CordQAT and CloneTracker [13]. Optimal clone indication thus lowers the effort required for clone discovery to zero. It thus simplifies impact analysis, since no additional effort is required to locate affected clones. Assuming perfect clone indicators, e^+_{IA} is reduced to zero, yielding this cost model:

$$\Delta e = \frac{overhead \cdot (e^-_L + e^-_{Impl} \cdot mod + e^-_{QA})}{e^-_A + e^-_L + e^-_D + e^-_{IA} + e^-_{Impl} + e^-_{QA} + e^-_O}$$

Linked Editing replicates edit operations performed on one clone to its siblings. Prototype linked editing tools include Codelink [35] and CReN [19]. Optimal linked editing tools thus lowers the overhead required for consistent modifications of cloned code to zero. Since linked editors typically also provide clone indication, they also simplify impact analysis. Their application yields the following model:

$$\Delta e = \frac{overhead \cdot (e^-_L + e^-_{Impl} \cdot mod)}{e^-_A + e^-_L + e^-_D + e^-_{IA} + e^-_{Impl} + e^-_{QA} + e^-_O}$$

We do not think that clone management tools can substantially reduce the overhead cloning causes for quality assurance. If the amount of changed code is larger due to cloning, more code needs to be processed by quality assurance activities. We do not assume that inspections or test executions can be simplified substantially by the knowledge that some similarities reside in the code—faults might still lurk in the differences.

However, we are convinced that clone indication tools can substantially reduce the impact that cloning imposes on the number of faults that slip through quality assurance. If a single fault is found in cloned code, clone indicators can point to all the faults in the sibling clones, assisting in their prompt removal. We assume that perfect clone indication tools reduce the cloning induced overhead in faults after quality assurance to zero.

5. Simplified Cost Model

This section introduces a simplified cost model. While less generally applicable than the detailed model, it is easier to apply.

Due to its number of factors, the detailed model requires substantial effort to instantiate in practice—each of its nine factors needs to be determined. Except for *overhead*, all of them quantify maintenance effort distribution across individual activities. Since in practice the activities are typically interleaved, without clear transitions between them, is difficult to get exact estimates on, *e. g.*, how much effort is spent on location and how much on impact analysis.

The individual factors of the detailed model are required to make trade-off decisions. We need to distinguish between, *e. g.*, impact analysis and location in order to evaluate the impact that clone indication tool support can provide, since impact analysis benefits from clone indication, whereas location does not. Before evaluating trade-offs between clone management alternatives however, a simpler decision needs to be taken: whether to do anything about cloning at all. Only then is it reasonable to invest the effort to determine accurate parameter values. If the cost model is not employed to assess clone management tool support, many of the distinctions between different factors are obsolete. We can thus aggregate them to reduce the number of factors and hence the effort involved in model instantiation.

Written slightly different, the detailed model is:

$$\Delta e = overhead \cdot \frac{e^-_L + e^-_{IA} + e^-_{Impl} \cdot mod + e^-_{QA}}{e}$$

The fraction is the ratio of effort required for code comprehension ($e^-_L + e^-_{IA}$), modification of existing code ($e^-_{Impl} \cdot mod$) and quality assurance (e^-_{QA}) w.r.t. the entire effort required for a change request. We introduce the new parameter *cloning-affected effort* (CAE) for it:

$$CAE = \frac{e^-_L + e^-_{IA} + e^-_{Impl} \cdot mod + e^-_{QA}}{e}$$

If CAE is determined as a whole (without its constituent parameters), this simplified model provides a simple way to evaluate the impact of cloning on maintenance efforts:

$$\Delta e = overhead \cdot CAE$$

6. Instantiation

This section describes how the cost model is instantiated and gives results from a large scale industrial case study.

We apply our quality analysis toolkit ConQAT[2] for clone detection and measure computation. ConQAT is described in general in [9, 11], its application as a clone detection workbench in [21].

6.1. Parameter Determination

This section describes how the parameter values can be determined to instantiate the cost model in practice.

[2] Available as open source at `http://www.conqat.org`

Overhead Computation *Overhead* is computed on the clones detected for a system. We have developed an algorithm that computes the *overhead* and implemented it in ConQAT. It is computed as explained in Section 2. *Overhead* captures cloning induced size increase independent of whether the clones can actually be removed or not. Only a detector that ignores identifier names. According to our experience [22], after tailoring, clones exhibited change coupling, indicating their semantic relationship through redundant implementation of a common concept.

Determining activity efforts The distribution of the maintenance efforts depends on many factors, including the maintenance process employed, the maintenance environment, the personnel and the tools available [33]. To receive accurate results, the parameters for the relative efforts of the individual activities thus need to be determined for each software system individually.

Coarse effort distributions can be taken from project calculation, by matching engineer wages against maintenance process activities. This way, the relative analysis effort, *e. g.*, is estimated as the share of the wages of the analysts w.r.t. all wages. As we cannot expect engineer roles to match the activities of our maintenance process exactly, we need to refine the distribution. This can be done by observing development efforts for change requests to determine, *e. g.*, how much effort analysts spend on analysis, location and design, respectively. To be feasible, such observations need to be carried out on representative samples of the engineers and of the change requests. Stratified sampling can be employed to improve representativeness of results—sampled CRs can be selected according to the change type distribution, so that representative amounts of perfective and other CRs are analyzed.

Table 1. Effort distribution

Activity	[31]	[6]	[36]	Estimate
Analysis			26%	5%
Location	13%			8%
Design	30%	16%	19%	16%
Impact Analysis				5%
Implementation	22%	29%	26%	26%
Quality Assurance	22%	24%	17%	22%
Other	26%	18%	12%	18%

Since each study used a slightly different maintenance process, each being different from the one used in this paper, we cannot directly determine average values for activity distribution. For example, in [31], *design* subsumes *analysis* and *location*. In [6], *analysis* subsumes *location*. The estimated average efforts are depicted in the fourth row of Table 1. Since the definitions of *implementation*, *quality assurance* and *other* are similar between the studies and our process, we used the median as estimated value. For the remaining activities, the effort distributions from the literature are of little help, since the activities do not exist in their processes or are defined differently. We thus distributed the remaining 34% of effort according to our best knowledge, based on our own development experience and that of our industrial partners.—the distribution can thus be inaccurate.

To determine the ratio between modification and addition effort during implementation, we inspect the distribution of change request types. We assume that adaptive, corrective and preventive change requests mainly involve modifications, whereas perfective changes mainly involve additions. Consequently, we estimate the ratio between addition and modification by the ratio of perfective w.r.t. all other change types. Table 2 shows effort distribution across change types from the above studies. The fourth row depicts the median of all three—37% of maintenance efforts are spent on perfective CRs, the remaining 63% are distributed across the other CR types. Based on these values, we estimate the modification ratio to be 0.63.

Table 2. Change type distribution

Effort	[31]	[6]	[36]	Median
Adaptive	7%	5%	8%	7%
Corrective	27%	14%	23%	23%
Other	29%	20%	44%	29%
Perfective	37%	61%	25%	37%

Literature values for Activity efforts offer a simple way to instantiate the model. Unfortunately, the research community still lacks a thorough understanding of how the activity costs are distributed across maintenance activities [33]. Consequently, results based on literature values are less accurate. They can however serve for a coarse approximation based on which decision can be taken, whether effort for more accurate determination of the parameters is justified.

Several researchers have measured effort distribution across maintenance activities. Rombach et al. [31] report measurement results for three large systems, carried out over the course of three years and covering around 10,000 hours of maintenance effort. Lientz et al. [6] analyzed 25 releases each of 10 different projects, covering over 20,000 hours of effort. Both studies work on data that was recorded during maintenance. Yeh and Jeng [36] performed a questionnaire-based survey in Taiwan. Their data is based on 97 valid responses received for 1000 questionnaires distributed across Taiwan's software engineering landscape. The values of the three studies are depicted in Table 1.

6.2. Case Studies

This section presents the application of the clone cost model to several large industrial software systems to quantify the impact of cloning, and the possible benefit of clone management tool support, in practice.

Goal The case study has two goals. First, evaluation of the clone cost model. Second, quantification of the impact of cloning on software maintenance costs across different software systems, and the possible benefit of the application of clone management tools.

Study Objects We chose 11 industrial software systems as study objects. Since we require the willingness of developers to contribute in clone detection tailoring, we had to rely on our contacts with industry. However, we chose systems from different domains (finance, content management, convenience, power supply, insurance) from 7 different companies written in 5 different programming languages to capture a representative set of systems. For nondisclosure reasons, we termed the systems A-K. Table 3 gives an overview ordered by system size.

Table 3. Add caption

System	Language	kLOC	kSS	kRFSS	overhead	ΔE	ΔE_{Tool}
A	XSLT	31	15	6	150.0%	75.0%	67.5%
B	ABAP	51	21	15	40.0%	20.0%	18.0%
C	C#	154	41	35	17.1%	8.6%	7.7%
D	C#	326	108	95	13.7%	6.8%	6.2%
E	C#	360	73	59	23.7%	11.9%	10.7%
F	C#	423	96	87	10.3%	5.2%	4.7%
G	ABAP	461	208	155	34.2%	17.1%	15.4%
H	C#	657	242	210	15.2%	7.6%	6.9%
I	Cobol	1,005	400	224	78.6%	39.3%	35.4%
J	Java	1,347	368	265	38.9%	19.4%	17.5%
K	Java	2,179	733	556	31.8%	15.9%	14.3%

6.3. Challenges of project diversity

Many factors influence maintenance productivity [5, 7, 33]: the type of system and domain, development process, available tools and experience of developers, to name just a few. Since these factors vary substantially between projects, they need to be reflected by cost estimation approaches to achieve accurate absolute results. The more factors a cost model comprises, the more effort is required for both its creation and its associated factor lookup tables, and for its instantiation in practice. If an absolute value is required, such effort is unavoidable.

The assessment of the impact of cloning differs from the general cost estimation problem in two important aspects. First, we compare efforts for two systems—the actual one and the hypothetical one without cloning—for which most factors are identical, since our maintenance environment does not change. Second, relative effort increase w.r.t. the cloning overhead is sufficient to evaluate the impact of cloning. Since we do not need an *absolute* result value in terms of costs, and since most factors influencing maintenance productivity remain stable in both settings, they do not need to be contained in our cost model. In a nutshell, we deliberately chose a *relative* cost model to keep its number of parameters and involved instantiation effort at bay.

7.2. Assumptions

The cost model is based on a series of assumptions. It can only sensibly be applied for projects that satisfy them. We list and discuss them here to simplify their evaluation.

We assume that the significant part of the cost models for the maintenance process activities are linear functions on the size of the code that gets processed. For example, we assume that location effort is primarily determined by and proportional to the amount of code that gets inspected during location. In some situations, activity cost models might be more complicated. For example, if an activity has a high fixed setup cost, the cost model should include a fixed factor; diseconomy of scale could increase effort w.r.t. size in a super linear fashion. In such cases, the respective part of the cost model needs to be adapted appropriately. COCOMO II, *e. g.*, uses an exponential function to adapt size to diseconomy of scale.

We assume that changes to clones are coupled to a substantial degree. This is in accordance with our experiences from large scale application of clone detection in industrial contexts [11, 21, 22], if clone detection is tailored appropriately. In case clones are uncoupled, *e. g.*, because they are false positives or because parts of the system are no longer maintained, the model is not applicable.

We assume that each modification to a clone in a clone group requires the same amount of effort. We ignore that subsequent implementations of a single change to multiple clone instances could get cheaper, since the developer gets used to that particular clone group. In practice, most clone groups have size 2. The inaccuracy introduced by this simplification should thus be moderate.

8. Related Work

This section relates the proposed cost model to existing work on the consequences of code cloning and cost models.

8.1. Consequences of Cloning

Substantial research has been carried out to better understand the consequences of code cloning. A survey is given by Hordijk et al. in [17]:

Impact of cloning on program correctness is the subject of several studies. Li et al. [30] present an approach to detect bugs based on inconsistent renaming of identifiers between clones. Jiang, Su and Chiu [20] analyze different contexts of clones, such as missing *if* statements. Both papers report the successful discovery of bugs in released software. In [1] and [3], individual cases of bugs or inconsistent bug fixes discovered by analysis of clone evolution are reported for open source software. In earlier work [22], we inspected inconsistent clones together with the developers of the analyzed systems. In four open source and four industrial systems, analysis revealed 107 faults due to unintentionally inconsistent changes to cloned code. The above studies give strong indication that that cloning impacts correctness in practice and that developers cannot be assumed to have complete knowledge of cloning when fixing bugs.

Several researchers have investigated the impact of cloning on modification effort. In [25], Kim et al. report that a substantial amount of changes to code clones occur in a coupled fashion, indicating additional maintenance effort due to multiple change locations. Aversano et al. [1] confirm a high ratio of co-evolution in a separate study. Detected late propagations indicate that a substantial number of inconsistent changes to clones were later detected and corrected during quality assurance. These studies give strong indication that clone-related change complexity. Hotta et al. [16] examine coupling and Bril et al. evaluate architecture quality [8].

To the best of our knowledge, the cost model presented here is the first to focus on the impact of code cloning on development costs.

9. Conclusion

This paper proposes an analytical cost model to quantify the economic effect of cloning on maintenance efforts. It can be used as a basis to evaluate clone management alternatives. Instead of computing absolute costs, the model computes maintenance effort increase relative to a system without cloning. Since in a relative cost model many factors that are independent of cloning remain constant, they do not need to be reflected in it. This way, it only requires a small number of factors and can be instantiated with reasonable effort in practice.

We have instantiated the cost model on 11 industrial systems. Although result accuracy could be improved by using project specific instead of literature values for effort parameters, the results indicate that cloning induced impact varies significantly between systems and is substantial for some. Based on the results, some projects can achieve considerable savings by performing active clone management.

There is a definitive need for future work in this area. The assumptions the cost model is based on need to be validated. The consequences of cloning on the number of field failures needs to be modeled quantitatively, instead of qualitatively as currently done in the model. We plan to perform sensitivity analysis to determine the relative importance of the individual parameters to guide instantiation in practice. Furthermore, we intend to instantiate the model using project specific effort parameters. Lastly but most importantly, we need to validate the correctness of the results, *e. g.*, through comparing efforts on projects before and after clone consolidation with the predicted efforts.

References

[1] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *Proc. CSMR '07*. IEEE, 2007.
[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of WCRE 1995*.
[3] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *Proc. ICSM '07*. IEEE, 2007.
[4] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Commun. ACM*, 3, 1993.
[5] Barry, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, W. A. Brown, S. Chulani, and C. Abts. *Software Cost Estimation with Cocomo II*. Prentice Hall, January 2000.
[6] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett. Understanding and predicting the process of software maintenance release. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 464–474. IEEE CS Press, 1996.
[7] B. Boehm, C. Abts, and S. Chulani. Software development cost estimation approaches – a survey. *Ann. Softw. Eng.*, 2000.
[8] R. Bril, L. Feijs, A. Glas, R. Krikhaar, and M. Winter. Maintaining a legacy: towards support at the architectural level. *Journal of Software Maintenance*, 2000.
[9] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. Flexible architecture conformance assessment with ConQAT. *In ICSE'10: submitted for publication*, 2010.
[10] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *ICSE'08*, 2008.
[11] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Software*, 25(5):60–67, 2008.
[12] C. Domann, E. Juergens, and J. Streit. The curse of copy&paste – redundancy in requirements specifications. In *ESEM'09*, 2009.
[13] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. ICSE '07*. IEEE, 2007.
[14] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of ICSM 1999*.
[15] M. F. B. H. B. S. S. W. C. D. J. S. Elmar Juergens, Florian Deissenboeck. Can clone detection support quality assessments of requirements specifications? In *Accepted for publication at ICSE'10*, 2010.
[16] N. Göde. Evolution of type-1 clones. In *SCAM'09*, 2009.
[17] W. T. B. Hordijk, M. L. Ponisio, and R. J. Wieringa. Harmfulness of code duplication - a structured review of the evidence. In *EASE'06*, 2006.
[18] IEEE. Standard 1219 for software maintenance. Standard, IEEE, 1998.
[19] P. Jablonski and D. Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proc. Eclipse '07*. ACM, 2007.
[20] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proc. ESEC-FSE '07*. ACM, 2007.
[21] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective - A workbench for clone detection research. In *ICSE'09*, 2009.
[22] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE '09*, 2009.
[23] C. Kapser and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *IWPSE'04*, 2004.
[24] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful. In *Proc. WCRE '06*. IEEE, 2006.
[25] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. ESEC/FSE-13*. ACM, 2005.
[26] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings, 2007.
[27] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proc. WCRE '07*. IEEE, 2007.
[28] J. Krinke. Is cloned code more stable than non-cloned code? *SCAM'08*, 2008.
[29] D. Lanning and T. Khoshgoftaar. Modeling the relationship between source code complexity andmaintenance difficulty. *Computer*, 1994.
[30] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE TSE*, 2006.
[31] H. D. Rombach, B. T. Ulery, and J. D. Valett. Toward full life cycle control: Adding maintenance measurement to the SEL. *J. Syst. Softw.*, 18(2):125–138, 1992.
[32] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 541, Queen's University at Kingston, 2007.
[33] H. Saneed. A cost model for software maintenance & evolution. In *International Conference on Software Maintenance (ICSM)*. IEEE CS Press, 2004.
[34] R. Tiarks, R. Koschke, and R. Falke. An assessment of type-3 clones as detected by state-of-the-art tools. In *SCAM'09*, 2009.
[35] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VLHCC '04*, 2004.
[36] D. Yeh and J.-H. Jeng. An empirical study of the influence of departmentalization and organizational contexts on software maintenance. *J. Softw. Maint. Evol. Res. Pr.*, 14(1):65–82, 2002.

$$\Delta\text{Aufwand} = \%\text{BlowUp} \times \%\text{CloneAffectedEffort}$$

$$\Delta\text{Aufwand} = \textbf{\%BlowUp} \times \%\text{CloneAffectedEffort}$$

Blow-Up: 0%

20 Lines

20 Lines

Blow-Up: 50%

20 Lines

20 Lines

20 Lines

$$\Delta\text{Aufwand} = \%\mathbf{12} \times \%\text{CloneAffectedEffort}$$

$$\Delta\text{Aufwand} = \%12 \times \mathbf{\%CloneAffectedEffort}$$

%CloneAffectedEffort

Aktivitäten	Aufwändiger durch Cloning
▪ Analysis	-
▪ Location	Location
▪ Design	-
▪ Impact Analysis	Impact Analysis
▪ Implementation	Implementation
▪ Quality Assurance	Quality Assurance
▪ Other	-

Detaillierte Herleitung und Berechnung im Paper.

Wert für Berechnung: **51%**.

$$\Delta \text{Aufwand} = \%12 \times \mathbf{\%50}$$

$$\Delta \text{Aufwand} = \%12 \times \%50 = \mathbf{6}\%$$

Die Munich Re setzt
Clone Management seit
ca. 10 Jahren ein.

Wie sähe es ohne aus?

Continuous Software Quality Control in Practice

Daniela Steidl*, Florian Deissenboeck*, Martin Poehlmann*, Robert Heinke†, Bärbel Uhink-Mergenthaler†
* CQSE GmbH, Garching b. München, Germany
† Munich RE, München, Germany

***Abstract*—Many companies struggle with unexpectedly high maintenance costs for their software development which are often caused by insufficient code quality. Although companies often use static analyses tools, they do not derive consequences from the metric results and, hence, the code quality does not actually improve. We provide an experience report of the quality consulting company CQSE, and show how code quality can be improved in practice: we revise our former expectations on quality control from [1] and propose an enhanced continuous quality control process which requires the combination of metrics, manual action, and a close cooperation between quality engineers, developers, and managers. We show the applicability of our approach with a case study on 41 systems of Munich RE and demonstrate its impact.**

I. Introduction

Software systems evolve over time and are often maintained for decades. Without effective counter measures, the quality of software systems gradually decays [2], [3] and maintenance costs increase. To avoid quality decay, *continuous quality control* is necessary during development and later maintenance [1]: for us, quality control comprises all activities to monitor the system's current quality status and to ensure that the quality meets the quality goal (defined by the principal who outsourced the software development or the development team itself).

Research has proposed various metrics to assess software quality, including structural metrics[1] or code duplication, and has led to a massive development of analysis tools [4]. Much of current research focuses on better metrics and better tools [1], and mature tools such as ConQAT [5], Teamscale [6], or Sonar[2] have been available for several years.

In [1], we briefly illustrated how tools should be combined with manual reviews to improve software quality continuously, see Figure 1: We perceived quality control as a simple, continuous feedback loop in which metric results and manual reviews are used to assess software quality. A quality engineer – a representative of the quality control group – provides feedback to the developers based on the differences between the current and the desired quality. However, we underestimated the amount of required manual action to create an impact. Within five years of experience as software quality consultants in different domains (insurance companies, automotive manufacturers, or engineering companies), we frequently experienced that tool

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant EvoCon, 01IS12034A. The responsibility for this article lies with the authors.

[1] e. g., file size, method length, or nesting depth
[2] http://www.sonarqube.org/

Fig. 1. The former understanding of a quality control process

support alone is not sufficient for successful quality control in practice. We have seen that most companies cannot create an impact on their code quality although they employ tools for quality measurements because the pressure to implement new features does not allow time for quality assurance: often, newly introduced tools get attention only for a short period of time, and are then forgotten. Based on our experience, quality control requires actions beyond tool support.

In this paper, we revise our view on quality control from [1] and propose an enhanced quality control process. The enhanced process combines automatic static analyses with a significantly larger amount of manual action than previously assumed to be necessary: Metrics constitute the basis but quality engineers must manually interpret metric results within their context and turn them into actionable refactoring tasks for the developers. We demonstrate the success and practicability of our process with a running case study with Munich RE which contains 32 .NET and 9 SAP systems.

II. Terms And Definitions

- A *quality criterion* comprises a metric and a threshold to evaluate the metric. A criterion can be, *e. g.*, to have a clone coverage below 10% or to have at most 30% code in long methods (*e. g.*, methods with more than 40 LoC).
- *(Quality) Findings* result from a violation of a metric threshold (*e. g.*, a long method) or from the result of a static code analysis (*e. g.*, a code clone).
- *Quality goals* describe the abstract goal of the process and provide a strategy how to deal with new and existing findings during further development: The highest goal is to have no findings at all, *i. e.*, all findings must be removed immediately. Another goal is to avoid new findings, *i. e.*, existing findings are tolerated but new findings must not be introduced. (III-B will provide more information).

III. The Enhanced Quality Control Process

Our quality control process is designed to be *transparent* (all stakeholders involved agree on the goal and consequences

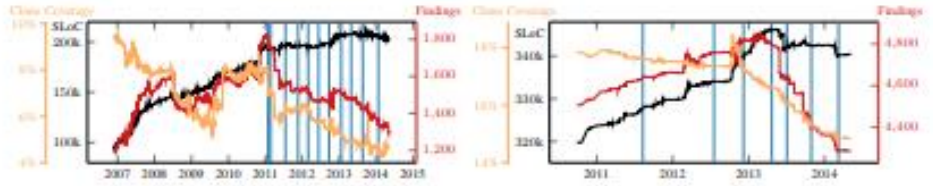


Fig. 3. System *A*

Fig. 5. System *C*

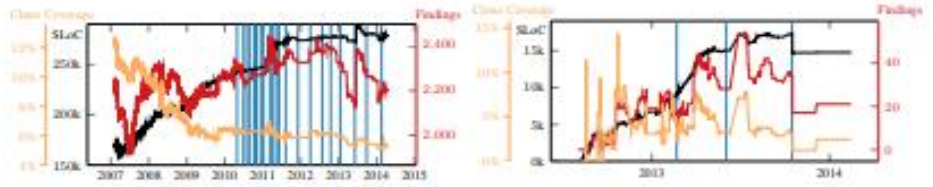


Fig. 4. System *B*

Fig. 6. System *D*

continuously in the available history (Figure 4). The number of findings, however, increases until mid 2012. In 2012, the project switched from QG2 to QG3. After this change, the number of findings decreases and the clone coverage settles around 6%, which is a success of the quality control. The major increase in the number of findings in 2013 is only due to an automated code refactoring introducing braces that led to threshold violations of few hundred methods. After this increase, the number of findings start decreasing again, showing the manual effort of the developers to remove findings.

For System C (Figure 5), the quality control process shows a significant impact after two years: Since the end of 2012, when the project also switched from QG2 to QG3, both the clone coverage and the overall number of findings decline. In the year before, the project transitioned between development teams and, hence, we only wrote two reports (July 2011 and July 2012).

System *D* (Figure 6) almost fulfills QG4 as after 1 year of development, it has only 21 findings in total and a clone coverage of 2.5%. Technically, under QG4, the system should have zero findings. However, in practice, exactly zero findings is not feasible as there are always some findings (*e. g.*, a long method to create UI objects or clones in test code) that are not a major threat to maintainability. Only a human can judge based on manual inspection of the findings whether a system still fulfills QG4, if it does not have exactly zero findings. In the case of System D, we consider 21 findings to be few and minor enough to fulfill QG4.

To summarize, our trends show that our process leads to actual measurable quality improvement. Those trends go beyond anecdotal evidence but are not sufficient to scientifically proof our method. However, Munich RE decided only recently to extend our quality control from the .NET area to all SAP development. As Munich RE develops mainly in the .NET and SAP area, most application development is now supported by quality control. The decision to extend the scope of quality control confirms that Munich Re is convinced by the benefit of quality control. Since the process has been established, maintainability issues like code cloning are now an integral part of discussions among developers and management.

V. Conclusion

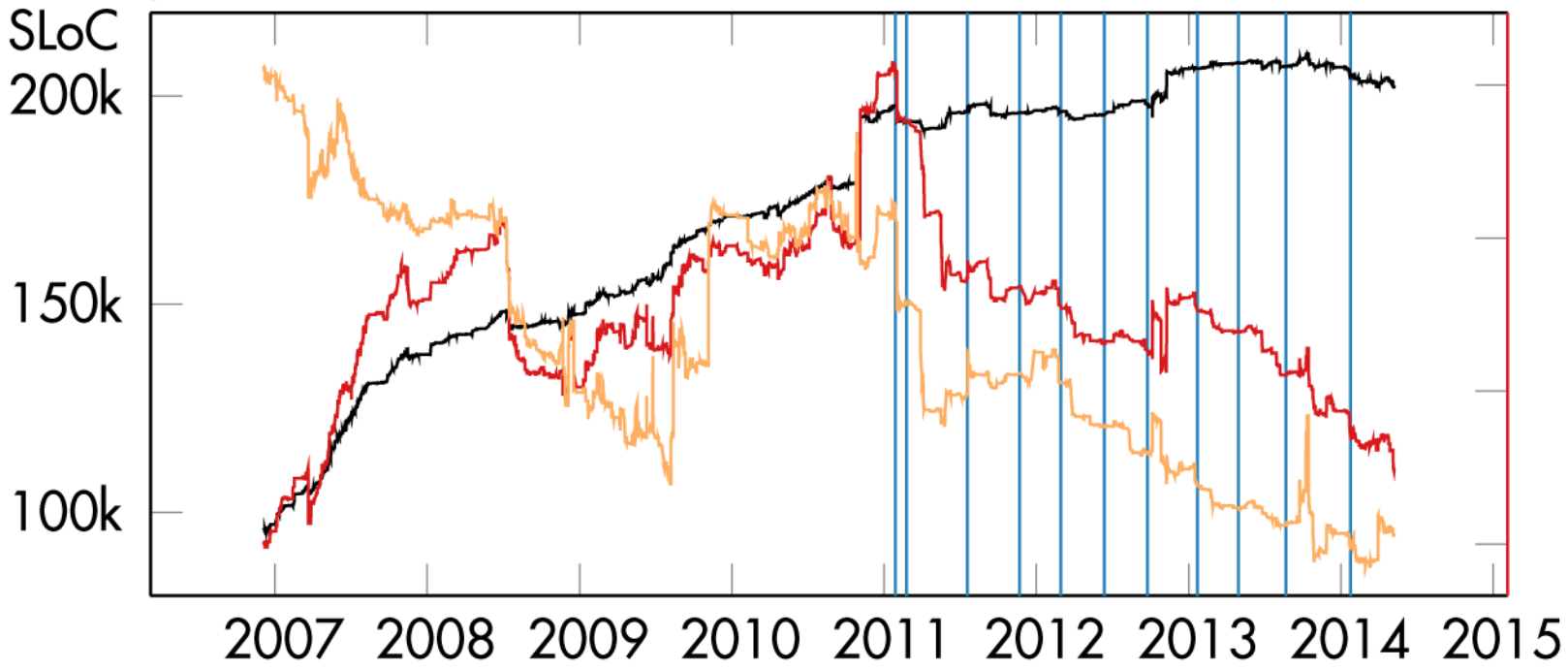
Quality analyses must not be solely based on automated measurements, but need to be combined with a significant amount of human evaluation and interaction. Based on our experience, we proposed a new quality control process for which we provided a running case study of 41 industry projects. With a qualitative impact analysis at Munich RE we showed measurable, long-term quality improvements. Our process has led to measurable quality improvement and an increased maintenance awareness up to management level at Munich Re.

References

[1] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka, "Tool support for continuous quality control," in *IEEE Software*, 2008.
[2] D. L. Parnas, "Software aging," in *ICSE '94*.
[3] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Trans. Software Eng.*, 2001.
[4] P. Johnson, "Requirement and design trade-offs in hackystat: An in-process software engineering measurement and analysis system," in *ESEM'07*.
[5] F. Deissenboeck, M. Pizka, and T. Seifert, "Tool support for continuous quality assessment," in *STEP'05*.
[6] L. Heinemann, B. Hummel, and D. Steidl, "Teamscale: Software quality control in real-time," in *ICSE'14*.
[7] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008.

Steidl, Heinke, Uhink: Continuous Software Quality Control in Practice? *ICSME* 2014

Einsparung durch Clone Detection



Menge an geklontem Code hat sich seit der Einführung von Clone Management halbiert.
Ohne Clone Management wäre der Clone Blow-Up daher vorraussichtlich doppelt so groß.

Ersparnis Aufwand = **6%**

Munich Re spart durch Einsatz von Clone Detection jährlich 6% Aufwand durch vermiedene Redundanz ein.

Findings

All	**6793**
Architecture	**1**
Architecture Conformance	1
Code Anomalies	**1344**
Bad practice	971
Correctness	2
Exception Handling	62
General checks (built-in)	120
Null pointer dereference	13
Performance	36
Unused code	93
Unused variable or parameter	47
Code Duplication	**988**
Cloning	101
Redundant Literals	887
Documentation	**3378**
Comment completeness	3236
Task tags	142
Formatting	**6**
Code formatting	6
Naming	**110**
Java naming conventions	110
Structure	**966**
File Size	38
Method Length	278
Nesting Depth	650

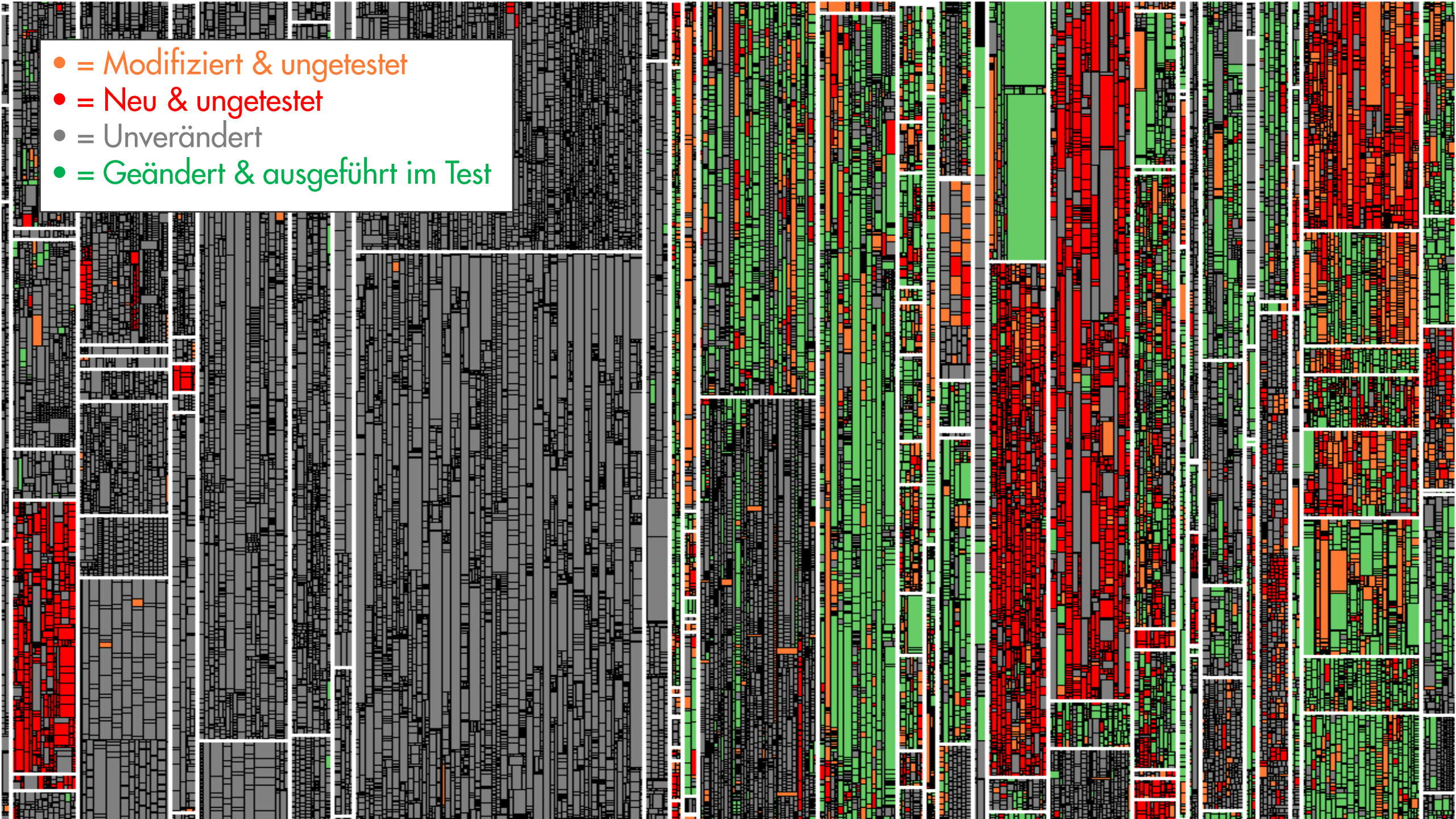
$$500 \frac{PT}{Jahr}$$

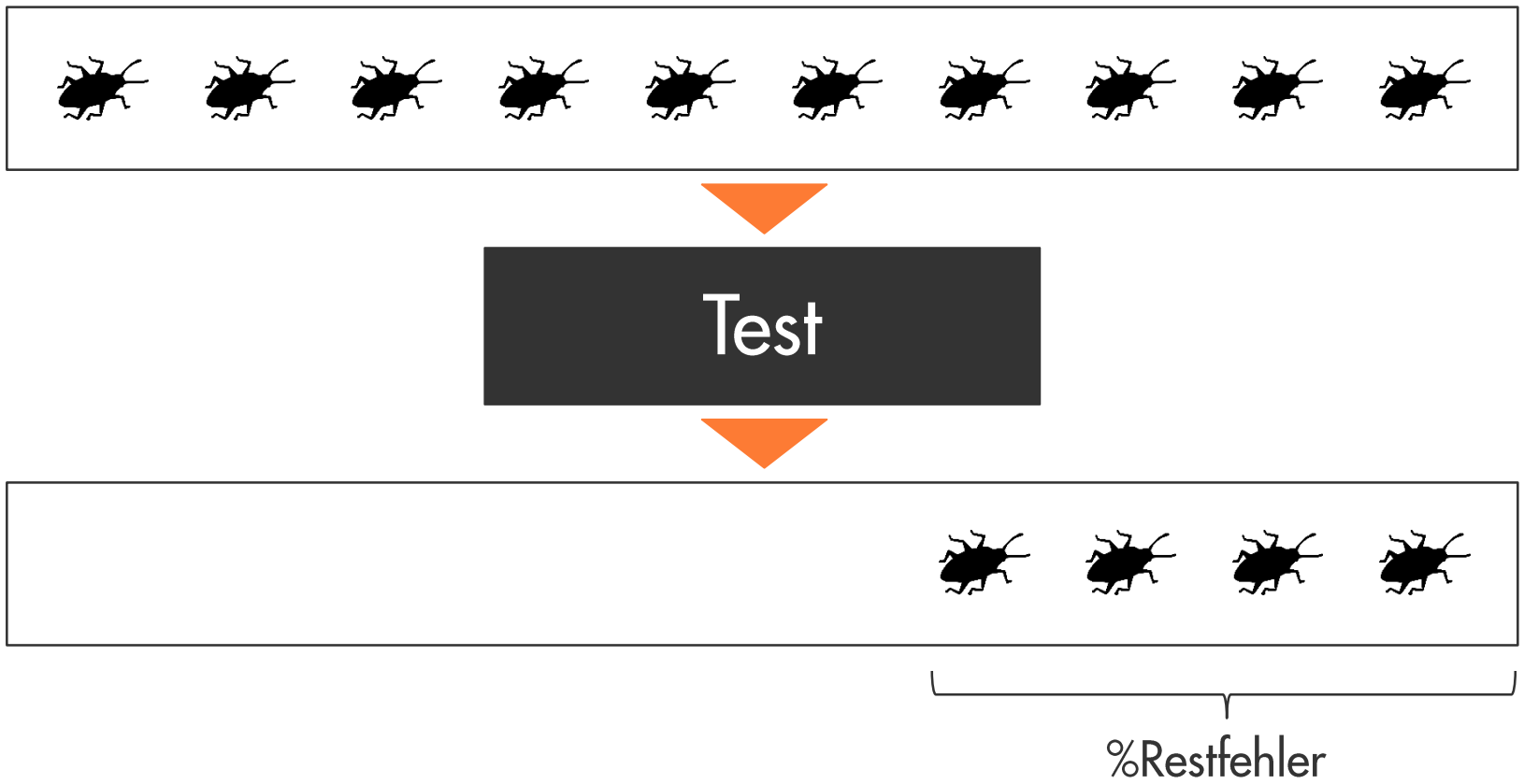
Munich Re spart durch Einsatz von Clone Detection jährlich ca. 500 PT Aufwand für Fehlerbehebung

$$\text{Ersparnis Aufwand} = \mathbf{6}\%$$

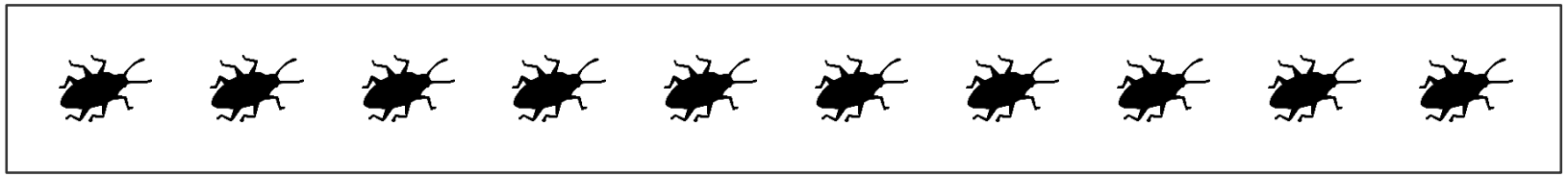
Munich Re spart durch Einsatz von Clone Detection jährlich 6% Aufwand durch vermiedene Redundanz ein.

Kosten-Nutzen von Test-Gap-Analyse

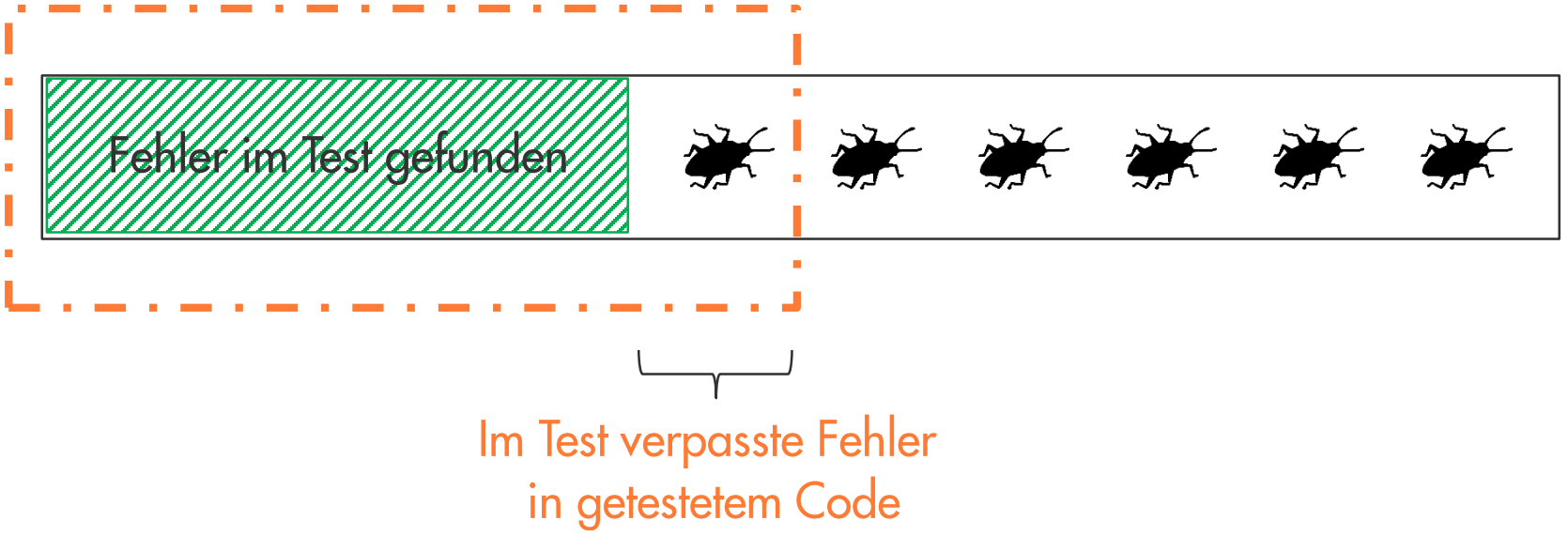

• = Modifiziert & ungetestet
• = Neu & ungetestet
• = Unverändert
• = Geändert & ausgeführt im Test


Test
%Restfehler

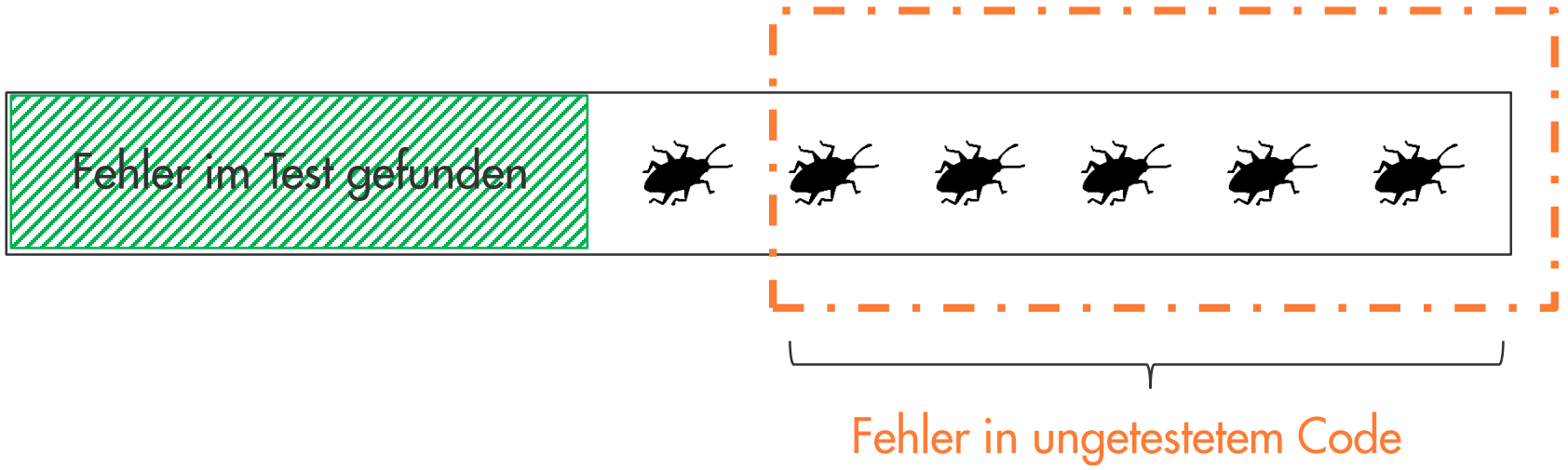
$$\%\text{Restfehler} = \%\text{Getestet} * \text{Testineffektivität} + \%\text{Testgap}$$



$$\%\text{Restfehler} = \%\text{Getestet} * \text{Testineffektivität} + \%\text{Testgap}$$



$$\%\text{Restfehler} = \%\text{Getestet} * \text{Testineffektivität} + \%\text{Testgap}$$



Did We Test Our Changes? Assessing Alignment between Tests and Development in Practice

Sebastian Eder, Benedikt Hauptmann, Maximilian Junker
Technische Universität München, Germany

Elmar Juergens
CQSE GmbH, Germany

Rudolf Vaas, Karl-Heinz Prommer
Munich Re Group, Germany

***Abstract*—Testing and development are increasingly performed by different organizations, often in different countries and time zones. Since their distance complicates communication, close alignment between development and testing becomes increasingly challenging. Unfortunately, poor alignment between the two threatens to decrease test effectiveness or increases costs.**

In this paper, we propose a conceptually simple approach to assess test alignment by uncovering methods that were changed but never executed during testing. The paper's contribution is a large industrial case study that analyzes development changes, test service activity and field faults of an industrial business information system over 14 months. It demonstrates that the approach is suitable to produce meaningful data and supports test alignment in practice.

***Index Terms*—Software testing, software maintenance, dynamic analysis, untested code**

I. Introduction

A substantial part of the total life cycle costs of long-lived software systems is spent on testing. In the domain of business-information systems, it is not uncommon that successful software systems are maintained for two or even three decades. For such systems, a substantial part of their total lifecycle costs is spent on testing to make sure that new functionality works as specified, and—equally important—that existing functionality has not been impaired.

During maintenance of these systems, test case selection is crucial. Ideally, each test cycle should validate all implemented functionality. In practice, however, available resources limit each test cycle to a subset of all available test cases. Since selection of test cases for a test cycle determines which bugs are found, this selection process is central for test effectiveness.

A common strategy is to select test cases based on the changes that were made since the last test cycle. The underlying assumption is that functionality that was added or changed recently is more likely to contain bugs than functionality that has passed several test cycles unchanged. Empirical studies support this assumption [1], [2], [3], [4].

If development and testing efforts are not aligned well, testing might focus on code areas that did not change, or—more critically—substantial code changes might remain untested. Test alignment depends on communication between testing and development. However, they are often performed by different teams, often located in different countries and time-zones. This distance complicates communication and thus challenges test alignment. But how can we assess test alignment and expose areas where it needs to be improved?

Problem: We lack approaches to determine alignment between development and testing in practice.

Proposed Solution: In this paper, we propose to assess test alignment by measuring the amount of code that was changed but not tested. We propose to use *method-level change coverage* information to support testers in assessing test alignment and improving test case selection.

Our intuition is that changed, but untested methods are more likely to contain bugs than either unchanged methods or tested ones. However, our intuition might be dead wrong: method-level churn could be a bad indicator for bugs, since methods can contain bugs although they have not changed in ages.

Contribution: This paper presents an industrial case study that explores the meaningfulness and helpfulness of method-level change coverage information. The case study was performed on a business information system owned by Munich Re. System development and testing were performed by different organizations in Germany and India. The case study analyzed all development changes, testing activity, and all field bugs, for a period of 14 months. It demonstrates that field bugs are substantially more likely to occur in methods that were changed but not tested.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A". The responsibility for this article lies with the authors.

II. Related Work

The proposed approach is related to the fields of defect prediction, selective regression testing, test case prioritization, and test coverage metrics. The most important difference to the named topics is the simplicity of the proposed approach and the fact that change coverage *assesses* the executed subsets of test suites, but does not give hints to improve them.

Defect prediction is related to our approach, because we identify code regions that were changed, but remained untested, with the expectation that there are more field bugs.

therefore useful for maintainers and testers to identify relevant gaps in their test coverage.

B. Study Object

We perform the study on a business information system at Munich Re. The analyzed system was written in C# and its size are 340 kLOC. In total, we analyzed the system for 14 months. The system has been successfully in use for nine years and is still actively used and maintained. Therefore, there is a well implemented bug tracking and testing strategy. This allows us to gain precise data about which parts of the system were changed and why they were changed.

We analyzed two consecutive releases of the system. Release 1 was developed in five iterations in two months, and release 2 was developed in ten iterations in four months. Both releases were deployed to the productive environment due to hot fixes five times and were in productive use for six months. Note that one deployment may concern several bugs and changes in the system. The system contained 22123 (release 1) respectively 22712 (release 2) methods.

For both releases, test suites containing 65 system test cases covering the main functionality were executed three times.

C. Study Design and Execution

For all research questions, we classify methods according to the categories shown in Figure 2: Tested or untested, changed or unchanged, and whether methods contain field bugs.

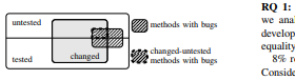


Fig. 2. Method categories used to evaluate change coverage

Study Design: First, we collect coverage and program data, then we answer RQ 1 and RQ 2 based on the collected data.

For answering RQ 1, we build method genealogies and identify changes during the development phase and relate usage data to these genealogies. With this information, we identify method genealogies that are changed-untested.

For answering RQ 2, we calculate the probability of field defects for every category of methods by detecting changes in the productive phase of the system in retrospective. This is valid for the analyzed system, since only severe bugs are fixed directly in the productive environment, which is defined by the company's processes.

We gain our results by identifying methods that are changed in the productive phase, which means they were related to a bug. We then categorize methods by change and coverage during the development phase. Based on this, we calculate the bug probability in the different groups of methods.

Study Execution: We used tool support, which consists of three parts: An ephemeral [18] profiler that records which methods were called within a certain time interval, a database that stores information about the system under consideration,

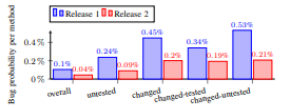


Fig. 3. Probability of fixes in both releases

and a query interface that allows retrieving coverage, change, and change coverage information. The same tool support was used in earlier studies [17], [19].

Validity Procedures: We focus on validity procedures and not on threats to validity due to space limitations.

We conducted manual inspections to ensure that every bug that is identified by our tool support is indeed a bug.

To confirm the correctness of method genealogies we build based on locality and signatures, we conducted manual inspections of randomly chosen method genealogies. We found no false genealogies and have therefore a high confidence in the correctness of our technique. We also used the algorithm in our former work [17], which provided suitable results as well.

D. Results

RQ 1: Untested methods account for 34% in both releases we analyzed. 15% of all methods were changed during the development phase of the system, also in both releases. The equality of the numbers for both releases is a coincidence.

8% respectively 9% of all methods were changed-untested. Considering only changed methods, only 44% were tested in release 1 and 45% of these methods were tested in release 2. These numbers constitute that there are gaps in the test coverage of changed code in the analyzed system.

RQ 2: We found 23 fixes in release 1 and 10 fixes in release 2. The distribution of the bugs over the different change and coverage categories of methods is shown in Table I. The biggest part of bugs occurred in methods categorized as changed-untested with 43% of all bugs in release 1 and 40% of all bugs in release 2. In both releases, there are considerably less bugs in unchanged regions than in changed regions.

The probabilities of bugs are shown in Figure 3. With 0.53% in release 1 and 0.21% in release 2, the probability of bugs is higher in the group of methods that were changed-untested. This confirms that tested code or code that was not changed in the development phase is less likely to contain field defects.

E. Discussion

RQ 1: With 15% of all methods being changed and 34% of all methods being not tested, untested code and changed code plays a considerable role in the analyzed system. The high amount of changed methods results from newly developed features, which means that many methods were added during the development phase of both releases.

There are several models for defect prediction [5]. In contrast to these models, we measure only changes in the system and the coverage by tests and do not predict bugs, but assess test suites and use the probability of bugs in changed, but untested code as validation of the approach.

The proposed approach is related to [6], which uses series of changes "change bursts" to predict bugs. The good results that were achieved by using change data for defect prediction encourage us to combine similar data with testing efforts.

Selective regression testing techniques target the selection of test cases from changes in source code and coverage information. [7], [8], [9].

In contrast to these approaches, the paper at hand focuses on the assessment of already executed test suites, because often experts decide which tests to execute to cover most of the changes made to a software system [10]. However, their estimations contain uncertainties and therefore possibly miss some changes. Our approach aims at identifying the resulting uncovered code regions. Therefore, our approach can only be used if testing activities were already performed.

Compared to [11], we are validating our approach by measuring field defects, and do not take defects into account that were found during development.

Test coverage metrics give an overview of what is covered by tests. Much research has been performed in these topics [12] and there is a plethora of tools [13] and a number of metrics available, such as statement, branch, or path coverage [14]. In contrast to these metrics, we focus on the more coarse grained method coverage. Furthermore, we do not only consider static properties of the system under test, but changes.

Empirical studies on related topics focus to the best of our knowledge mainly on the effectiveness of test case selection and prioritization techniques [9], [15]. In our study, we assess test suites by their ability to cover changes of a software system, but do not consider sub sets of test suites.

III. Context and Terms

In this work, we focus on *system testing* according to the definition of IEEE Std 610.12-1990 [16] to denote *"testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements"*. System tests are often used to detect bugs in existing functionality after the system has been changed. In our context, many tests are executed manually and denoted in natural language.

Our study uses *methods* as they are known from programming languages such as Java or C#. Methods form the entities of our study and can be regarded as units of functionality of a software system. They are defined by a signature and a body. To compare different releases of a software system over time, we create *method genealogies* which represent the evolution of a single method over time. A genealogy connects all releases of a method in chronological order [17].

In the context of our work, the life cycle of a software system consists of two alternating phases (see Figure 1). In the *development phase*, existing functionality is maintained

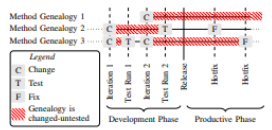


Fig. 1. Development life-cycle

or new features are developed. Development usually occurs in *iterations* which are followed by *test runs* which are the execution of a selection of tests aiming to test regressions as well as the changed or added code. A development phase is completed by a *release* which transfers the system into the *productive phase*. In the productive phase, functionality is usually neither added nor changed. If critical malfunctions are detected, *hot fixes* are deployed in the productive phase.

We consider a method as *tested* if it has been executed during a test run. If a method has been changed or added and been tested afterwards before the system is released we consider it as *changed-tested*. If a method change or addition has not been tested before the system is transferred in the productive phase, we consider the method as *changed-untested* (see genealogy 1 and 3 in Figure 1).

IV. Change Coverage

To quantify the amount of changes covered by tests, we introduce the metric *change coverage (CC)*. It is computed by the following formula and ranges between [0,1].

$$\text{change coverage} = \frac{\#\text{methods changed-tested}}{\#\text{methods changed}}$$

A change coverage of 1 ($CC = 1$) means that all methods which have been changed since the last test run have been tested after their last change. On the contrary, a coverage of 0 ($CC = 0$) indicates that none of the changed methods have been covered by a test.

V. Case Study

A. Goal and Research Questions

The goal of the study is to show whether change coverage is a useful metric for assessing the alignment between tests and development. We formulate the following research questions.

RQ 1: *How much code is changed, but untested?* The goal of this research question is to investigate the existence of changed, but untested code, to justify the problem statement of this work. Therefore, we quantify changed and untested code.

RQ 2: *Are changed-untested methods more likely to contain field bugs than unchanged or tested methods?* The goal of this research question is to decide whether change coverage can be used as a predictor for bugs in large code regions and is

TABLE I
Distribution of fixes over the different categories

	Release 1		Release 2	
Category	Absolute	Relative	Absolute	Relative
changed-tested	5	22%	3	30%
changed-untested	10	43%	4	40%
unchanged-tested	0	0%	0	0%
unchanged-untested	8	35%	3	30%

43% respectively 40% of the changed methods were not tested in the analyzed system. These high numbers also result from features that are newly developed during the development phase. For these new features, there was only a very limited number of test cases.

RQ 2: With a probability of bugs in untested-changed methods of 0.53% respectively 0.21%, this group of methods contains most of the bugs. This means that the system itself contains few bugs at the current stage of development and bugs are brought into the system by changes.

Furthermore, the probability of bugs in untested code is, in both releases, less than half of the probability in changed-untested code. Hence, we conclude that only considering test coverage is not as efficient as considering change coverage.

The probability of bugs in changed code regions is also considerably higher than in untested regions. But the combination of both metrics, test coverage and changed methods points to code regions that are more likely to contain bugs than others.

Is Change Coverage Helpful in Practice? We employed the proposed approach also in the context of Munich Re in currently running development phases. We showed the results to developers and testers by presenting code units, like types or assemblies ordered by change coverage. During the discussion of the results, we conducted open interviews with developers to gain knowledge about how helpful information about change coverage is during maintenance and testing.

Developers identified meaningful methods in changed but untested regions by using the static call graph to find methods they know. With these methods, the developers were able to identify features that remained untested. For example the processing of excel sheets in a particular calculation was changed, but remained untested afterwards. In this case, among some others, the (re-)execution of particular test cases and the creation of new test cases were issued. This increased the change coverage considerably for the code regions where the features are located. This shows that change coverage is helpful for practitioners.

VI. Conclusion and Future Work

We presented an automated approach to assess the alignment of test suites and changes in a simple and understandable way. Instead of using rather complex mechanisms to derive code units that may be subject to changes, we are focusing on changed but untested methods and calculate an expressive metric from these methods. The results show that the use of change coverage is suitable for the assessment of the alignment of testing and development activities.

We also showed that change coverage is suitable for guiding testers during the testing process. With information about change coverage, testing efforts can be assessed and redirected if necessary, because the probability of bugs is increased in changed-untested methods. Furthermore, we presented our tool support that allows us to utilize our technique in practice.

However, the number of bugs we found is too small to derive generalizable results. Therefore, we plan to extend our studies to other systems to increase external validity. But the first results that we presented in this work point out that the consideration of code regions that are modified, but not very well tested is important. This motivates future work on the topic and the inference of improvement goals.

One challenge is the identification of suitable test cases from code regions to give hints to testers and developers which test case to execute to cover more changed, but untested methods. Therefore, we plan to evaluate techniques related to trace link recovery to bridge the gap to test cases.

References

[1] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE*, 2005.
[2] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality," in *ICSE*, 2008.
[3] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, 2000.
[4] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ISSTA*, 2004.
[5] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, 2012.
[6] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *ISSRE*, 2010.
[7] V. Channakeshava, V. K. Shanbhag, A. Panigrahi, R. Sisodia, and S. Lakshmanan, "Safe subset-regression test selection for managed code," in *ISEC*, 2008.
[8] Y.-F. Chen, D. Rosenblum, and K.-P. Vo, "Testtube: a system for selective regression testing," in *ICSE*, 1994.
[9] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," in *ICSE*, 1998.
[10] M. Harrold and A. Orso, "Retesting software during development and maintenance," in *FoSM*, 2008.
[11] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *ISSTA*, 2002.
[12] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, 1997.
[13] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage based testing tools," in *AST*, 2006.
[14] Y. Malaiya, M. Li, J. Bieman, and R. Karcich, "Software reliability growth with test coverage," *IEEE Trans. Rel.*, vol. 51, no. 4, 2002.
[15] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, 2001.
[16] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," New York, USA, 1990.
[17] S. Eder, M. Junker, E. Jurgens, B. Hauptmann, R. Vaas, and K. Prommer, "How much does unused code matter for maintenance?" in *ICSE*, 2012.
[18] O. Traub, S. Schechter, and M. D. Smith, "Ephemeral instrumentation for lightweight program profiling," School of engineering and Applied Sciences, Harvard University, Tech. Rep., 2000.
[19] E. Juergens, M. Feilkas, M. Herrmannsdoerfer, F. Deissenboeck, R. Vaas, and K. Prommer, "Feature profiling for evolving systems," in *ICPC*, 2011.

Paper published at Workshop on Automation of Software Test, 2013.

Wieviele Änderungen sind ungetestet?

Studie: C# System @ Munich Re

Release A:

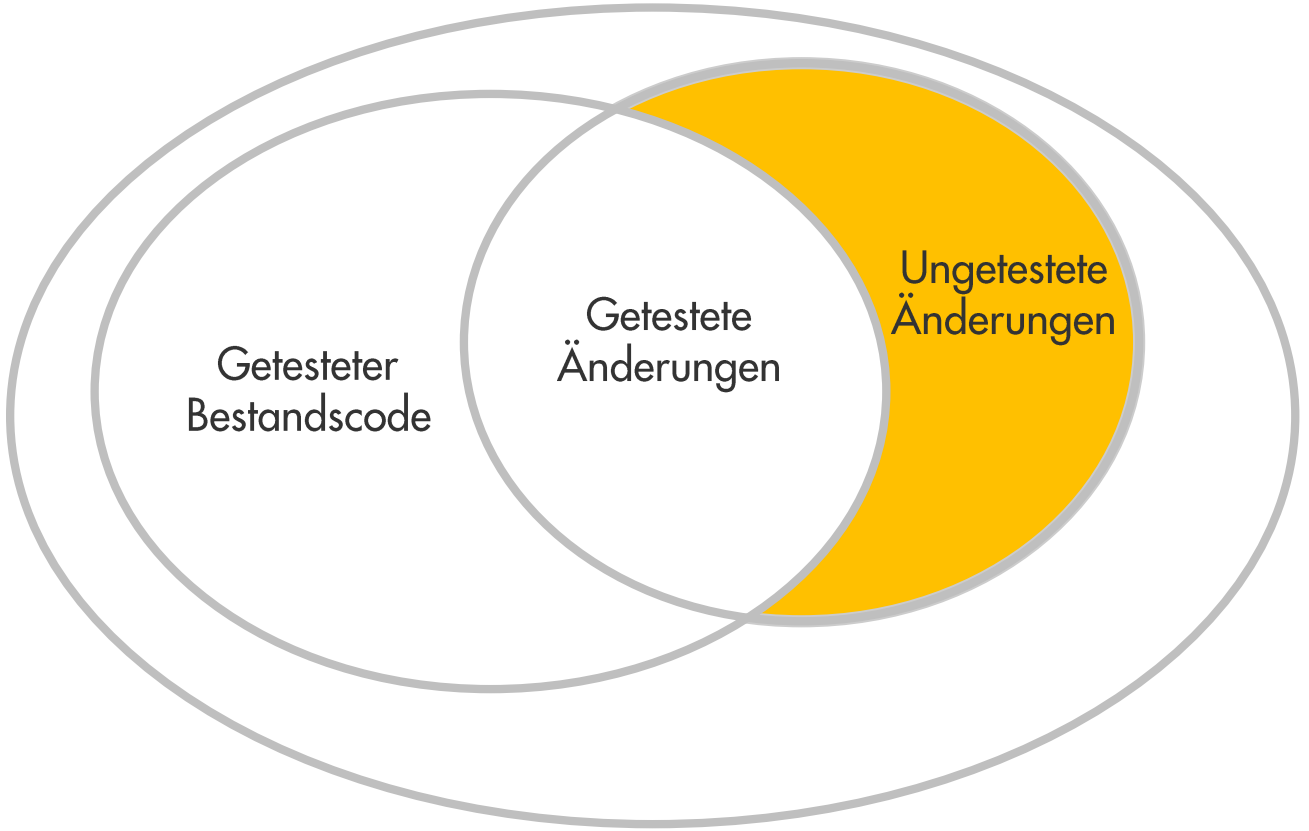
15% Code neu/geändert,

>50% ungetestet

Release B:

15% Code neu/geändert,

>60% ungetestet

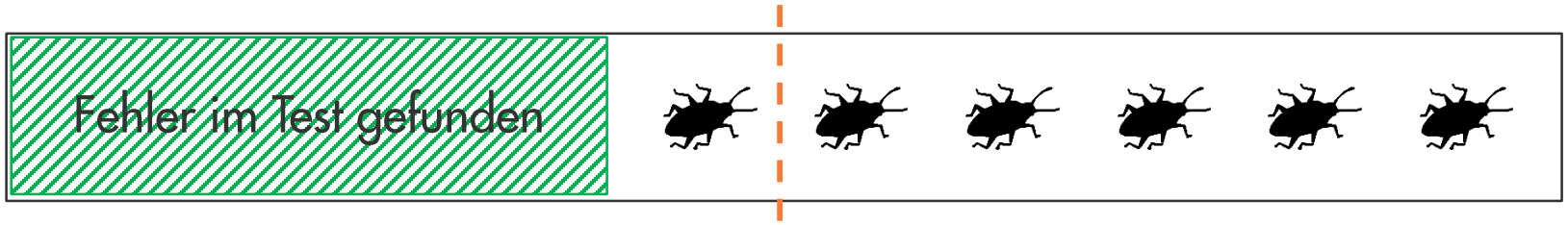


Feldfehlerwahrscheinlichkeit 5x höher für ungetestete Änderungen!

Eder, Jürgens, … *Did We Test Our Changes? Assessment btw. Tests & Development in Practice*, AST@ICSE 2013

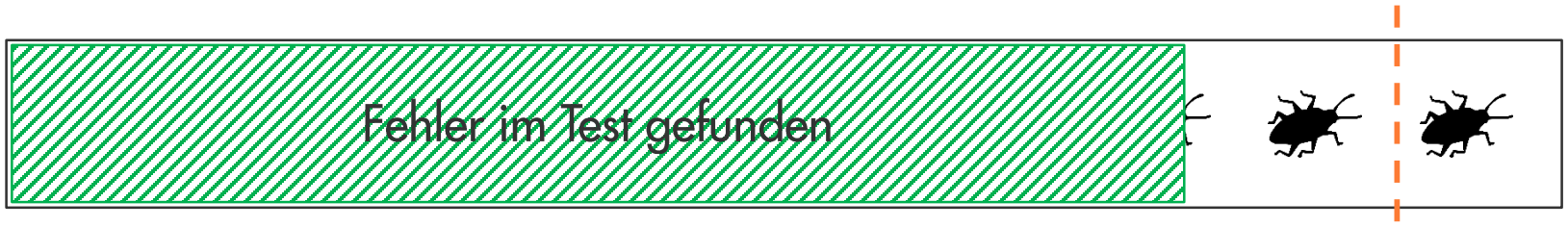
Ohne Test-Gap-Analyse

%Restfehler = 60%



Mit Test-Gap-Analyse

%Restfehler = 28%



Reduzierte Feldfehler = **50%**

Reduzierte Feldfehler = **50%**

Test-Gap-Analyse reduziert Feldfehler in den Applikationen der Munich Re um ⅓

Fazit

- Conformance Costs << Costs of Non-Conformance

- Mit der Nutzenargumentation im Rücken konzentrieren uns auf umfassende Nutzung der Tools und Prozesse.

- Tools ***und*** Prozesse wichtig, etabliert und fest verankert.
- Internes Change Management („1/3“) notwendig.
- Sichtbarmachen von Qualität ist essentiell.

Kontakt – Wir freuen uns auf Diskussionen ☺

Uwe Proft uproft@munichre.com

Dr. Elmar Jürgens juergens@cqse.eu