

Technische Schulden wirksam kommunizieren

TUM



CQSE



Technische Schulden

```

50
51 /**
52 90 } else {
53 91     levelChars.append(c[i]);
54 92 }
55 93 String[] tokens = tokens.get().split(" ");
56 94
57 95     for (int k = 0; k < tokens.length; k++) {
58 96         String token = tokens[k];
59 97         if (abbreviateThatIsSingleLetter) {
60 98             String[] dashes = token.split("-");
61 99
62 100             token = Arrays.asList(dashes).stream().map(BibtexNameFormatter::getFirstCharOfString)
63 101                 .collect(Collectors.joining("-"));
64 102         }
65 103
66 104         // Output token
67 105     } else {
68 106         sb.append(d[j]);
69 107     }
70 108 }
71 109 if (sb.length() > 0) {
72 110     boolean noDisTie = false;
73 111     if ((sb.charAt(sb.length() - 1) == '~') &&
74 112         ((BibtexNameFormatter.numberOfChars(sb.substring(groupStart, sb.length()), 4) >= 4) ||
75 113         ((sb.length() > 1) && (noDisTie = sb.charAt(sb.length() - 2) == '~')))) {
76 114         sb.deleteCharAt(sb.length() - 1);
77 115         if (!noDisTie) {
78 116             sb.append(' ');
79 117         }
80 118     }
81 119 }
82 120 } else if (c[i] == '}') {
83 121     if (warn != null) {
84 122         warn.warn("Unmatched brace in format string: " + format);
85 123     }
86 124 } else {
87 125     sb.append(c[i]); // verbatim
88 126 }
89 127 i++;
128 }
129 if ((braceLevel != 0) && (warn != null)) {
130     warn.warn("Unbalanced brace in format string for nameFormat: " + format);
131 }
132
133 return sb.toString();
134 }

```

```

157
158 ← ? for (int j = 0; j < d.length; j++) {
159
160     if (Character.isLetter(d[j]) && (bLevel == 1)) {
161         groupStart = sb.length();
162         if (!abbreviateThatIsSingleLetter) {
163             j++;
164         }
165         if (((j + 1) < d.length) && (d[j + 1] == '{')) {
166             StringBuilder interTokenSb = new StringBuilder();
167             j = BibtexNameFormatter.consumeToMatchingBrace(interTokenSb, d, j + 1);
168             interToken = interTokenSb.substring(1, interTokenSb.length() - 1);
169         }
170
171         for (int k = 0; k < tokens.length; k++) {
172             String token = tokens[k];
173             if (abbreviateThatIsSingleLetter) {
174                 String[] dashes = token.split("-");
175
176                 token = Arrays.asList(dashes).stream().map(BibtexNameFormatter::getFirstCharOfString)
177                     .collect(Collectors.joining("-"));
178             }
179
180             // Output token
181             sb.append(token);
182
183             if (k < (tokens.length - 1)) {
184                 // Output Intertoken String
185                 if (interToken == null) {
186                     if (abbreviateThatIsSingleLetter) {
187                         sb.append('.');
188                     }
189                     // No clue what this means (What the hell are tokens anyway???) 😊
190                     // if (lex_class[name_sep_char[cur_token]] = sep_char) then
191                     //     append_ex_buf_char_and_check (name_sep_char[cur_token])
192                     if ((k == (tokens.length - 2)) || (BibtexNameFormatter.numberOfChars(sb.substring(groupStart, sb.length()), 3) < 3)) {
193                         sb.append('~');
194                     } else {
195                         sb.append(' ');
196                     }
197                 } else {
198                     sb.append(interToken);
199                 }
200             }
201         }
202     } else if (d[j] == '}') {
203         bLevel--;
204         if (bLevel > 0) {
205             sb.append('}');
206         }
207     } else if (d[j] == '{') {
208         bLevel++;
209         sb.append('{');
210     } else {
211         sb.append(d[j]);
212     }
213 }
214 if (sb.length() > 0) {

```

```

70     i++,
71     c = finalResult.charAt(i);
72     String combody;
73     if (c == '{') {
74         String part = StringUtil.getPart(finalResult, i, false);
75         i += part.length();
76         combody = part;
77     } else {
78         combody = finalResult.substring(i, i + 1);
79     }
80     String result = OOPreFormatter.CHARS.get(command + combody);
81
82     if (result != null) {
83         sb.append(result);
84     }
85
86     incommand = false;
87     escaped = false;
88 } else {
89     // Are we already at the end of the string?
90     if ((i + 1) == finalResult.length()) {
91         String command = currentCommand.toString();
92         String result = OOPreFormatter.CHARS.get(command);
93         /* If found, then use translated version. If not,
94          * then keep
95          * the text of the parameter intact.
96          */
97         if (result == null) {
98             sb.append(command);
99         } else {
100             sb.append(result);
101         }
102     }
103 }
104 }
105 }
106 } else {

```

```

70     i++,
71     c = field.charAt(i);
72     String commandBody;
73     if (c == '{') {
74         String part = StringUtil.getPart(field, i, false);
75         i += part.length();
76         commandBody = part;
77     } else {
78         commandBody = field.substring(i, i + 1);
79     }
80     String result = HTML_CHARS.get(command + commandBody);
81
82     if (result == null) {
83         sb.append(commandBody);
84     } else {
85         sb.append(result);
86     }
87
88     incommand = false;
89     escaped = false;
90 } else {
91     // Are we already at the end of the string?
92     if ((i + 1) == field.length()) {
93         String command = currentCommand.toString();
94         String result = HTML_CHARS.get(command);
95         /* If found, then use translated version. If not,
96          * then keep
97          * the text of the parameter intact.
98          */
99         if (result == null) {
100             sb.append(command);
101         } else {
102             sb.append(result);
103         }
104     }
105 }
106 }
107 }
108 } else {

```

Wie kommunizieren?

Findings

<input checked="" type="checkbox"/> All	6793
<input checked="" type="checkbox"/> Architecture	1
<input checked="" type="checkbox"/> Architecture Conformance	1
<input checked="" type="checkbox"/> Code Anomalies	1344
<input checked="" type="checkbox"/> Bad practice	971
<input checked="" type="checkbox"/> Correctness	2
<input checked="" type="checkbox"/> Exception Handling	62
<input checked="" type="checkbox"/> General checks (built-in)	120
<input checked="" type="checkbox"/> Null pointer dereference	13
<input checked="" type="checkbox"/> Performance	36
<input checked="" type="checkbox"/> Unused code	93
<input checked="" type="checkbox"/> Unused variable or parameter	77
<input checked="" type="checkbox"/> Code Duplication	988
<input checked="" type="checkbox"/> Cloning	101
<input checked="" type="checkbox"/> Redundant Literals	887
<input checked="" type="checkbox"/> Documentation	3378
<input checked="" type="checkbox"/> Comment completeness	3236
<input checked="" type="checkbox"/> Task tags	142
<input checked="" type="checkbox"/> Formatting	6
<input checked="" type="checkbox"/> Code formatting	6
<input checked="" type="checkbox"/> Naming	110
<input checked="" type="checkbox"/> Java naming conventions	110
<input checked="" type="checkbox"/> Structure	966
<input checked="" type="checkbox"/> File Size	38
<input checked="" type="checkbox"/> Method Length	278
<input checked="" type="checkbox"/> Nesting Depth	650

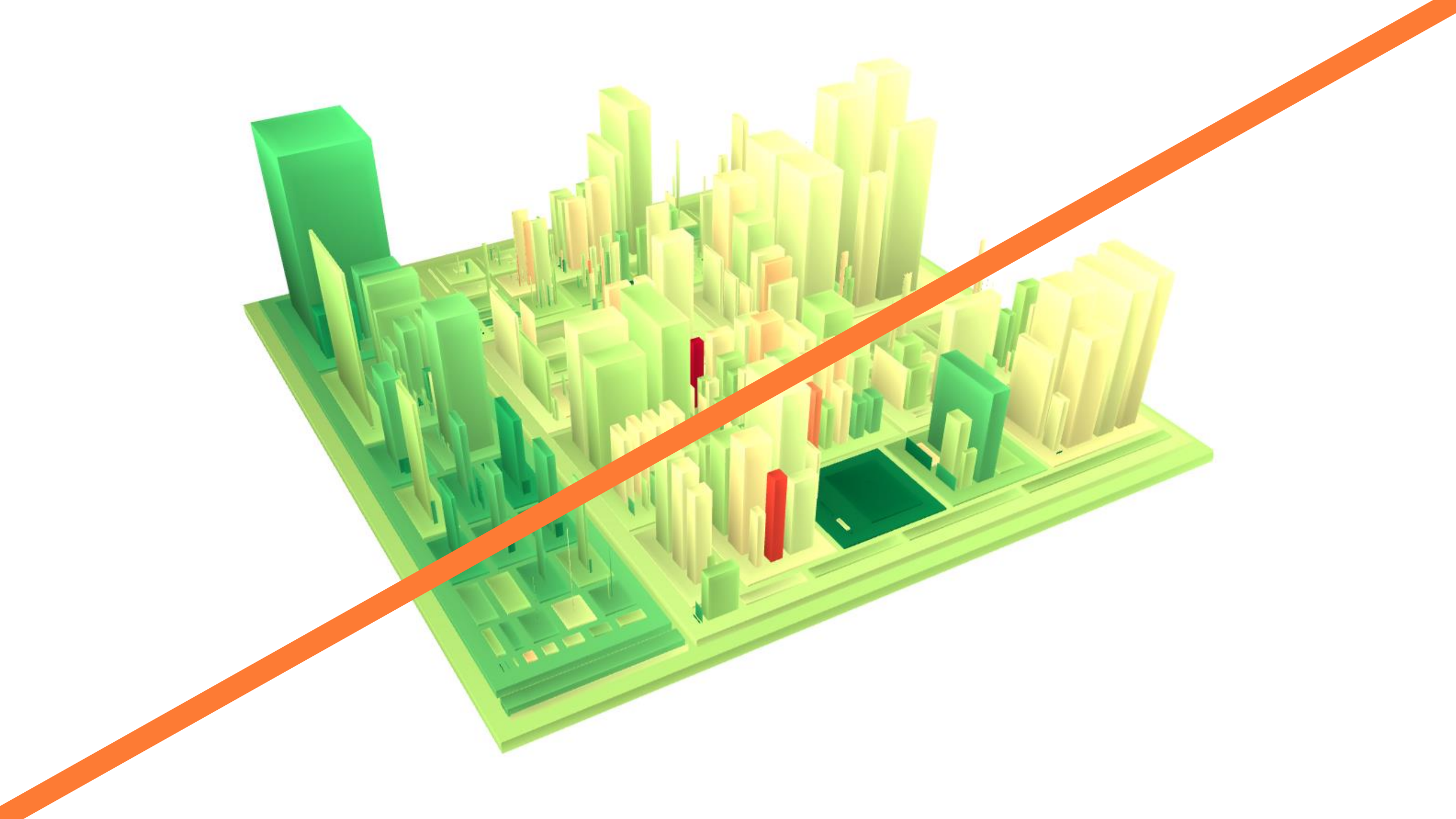
$$171 - 5.2 \cdot \ln(\text{avgHV}) - 0.23 \cdot \text{avgCC}(g) - 16.2 \cdot \ln(\text{avgLOC}) + 50 \cdot \ln(\text{sqrt}(2.4 \cdot \text{perCM}))$$

HV: Halstead Volume CC: Cyclomatic Complexity
LOC: lines of code perCM: % Comment Lines

Maintainability [Measures](#)

589d A

Debt ?



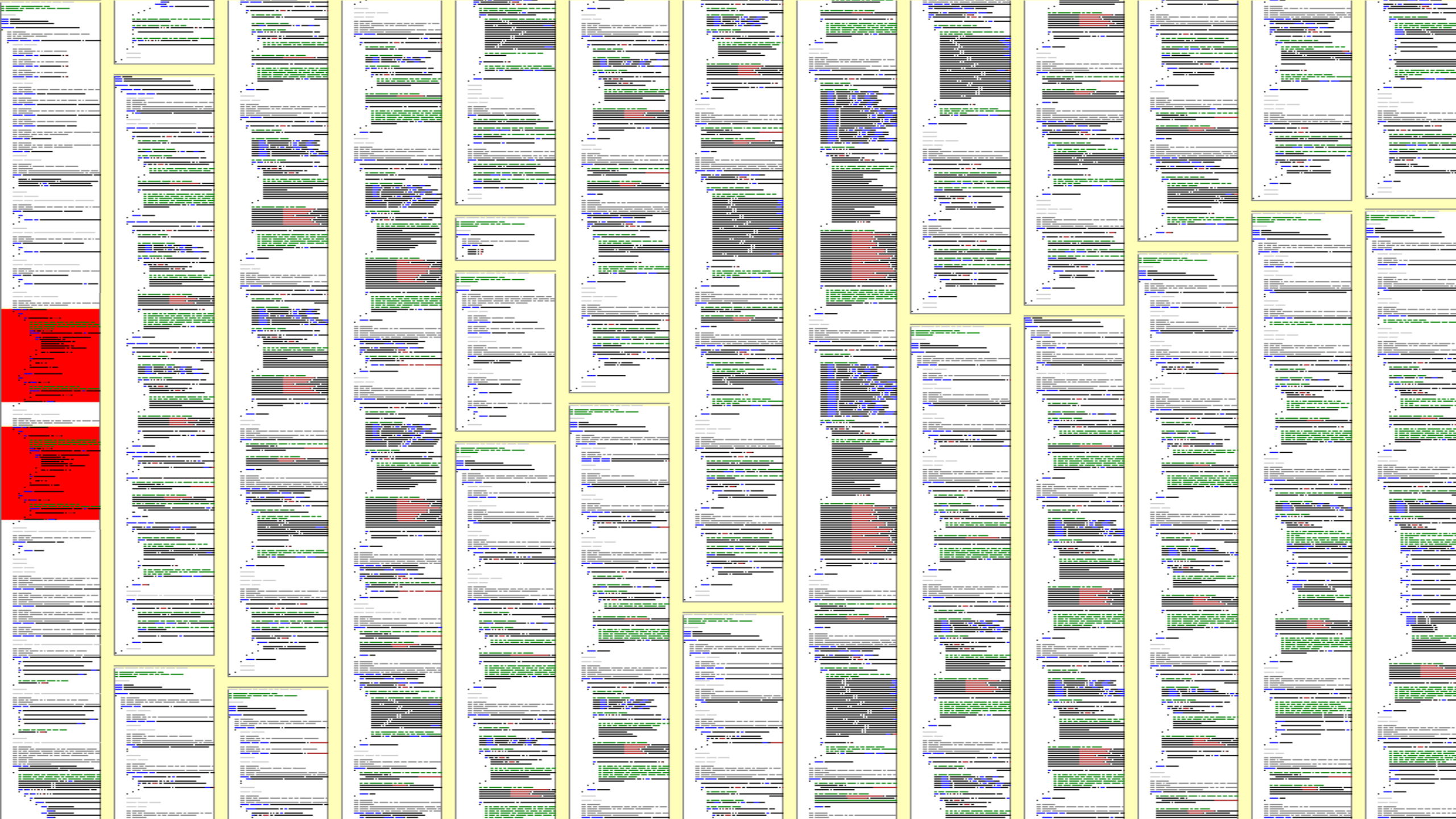
Best Practices

Konsequenzen für eigenes Projekt herausarbeiten

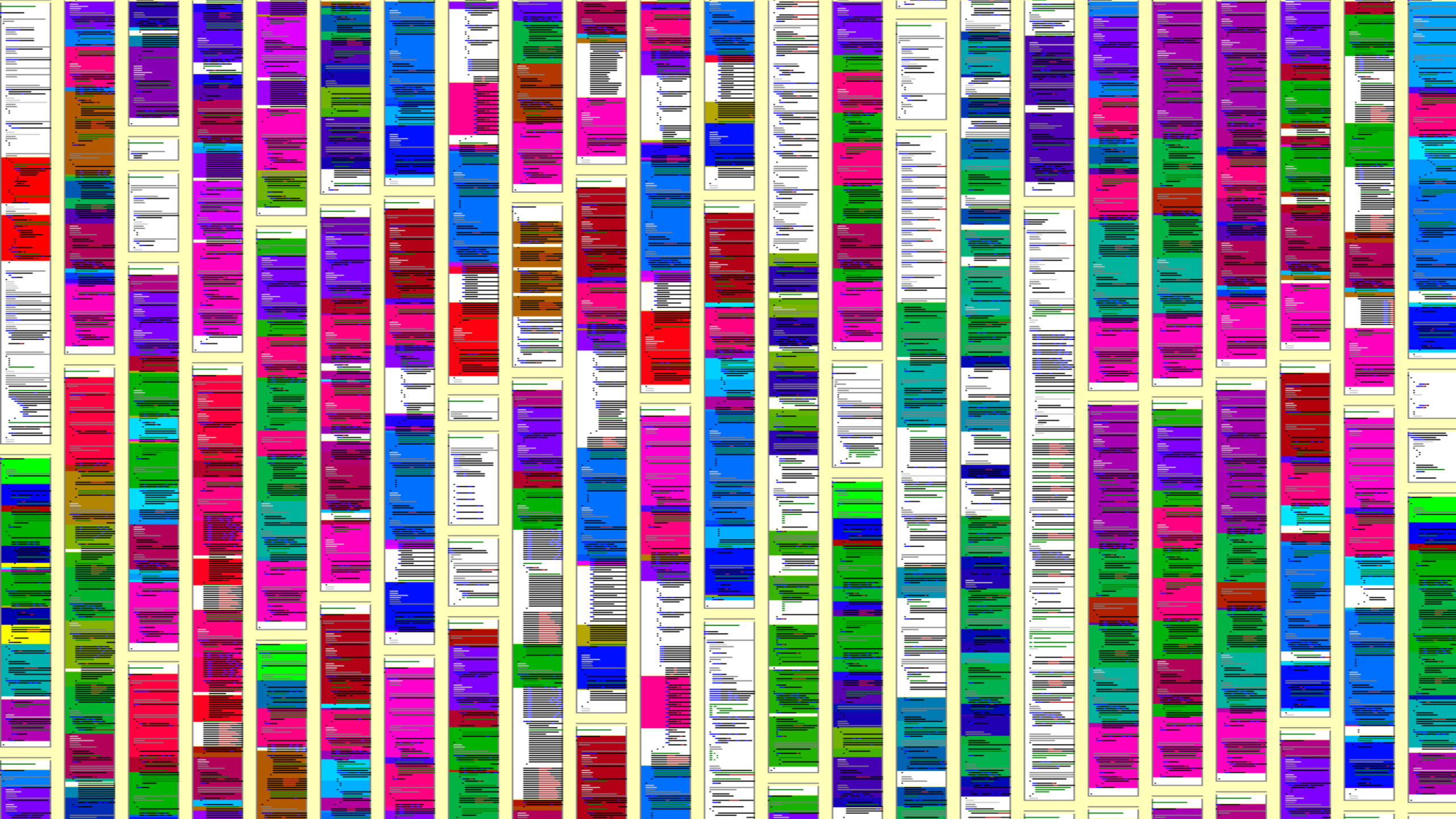
1. Jeweils auf einen Typ Qualitätsproblem fokussieren, da sich unterschiedliche Qualitätsprobleme (Fehlende Tests? Klone?) unterschiedlich auswirken.
2. Konsequenzen des Fortbestehens der technischen Schulden greifbar machen, und zwar in nicht-technischer Dimension
3. Daten aus eigenem System so visualisieren, dass Konsequenzen greifbar werden













```
// 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();
}
```









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

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(Object,
/
public 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.hasNext()
            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(Object,
/
public void assertEqualDataBoundBeans(List<?> lhs, List<?> rhs) throws Exception {
    assertEquals(lhs.size(), rhs.size());
}
    
```

```
54     orCriteria.compare(TerrorismTargetTier_Ext#County, Equals, polLocation.County)
55     orCriteria.compare(TerrorismTargetTier_Ext#County, Equals, null)
56 })
57 tierQuery.or(\orCriteria -> {
58     orCriteria.compare(TerrorismTargetTier_Ext#City, Equals, polLocation.City)
59     orCriteria.compare(TerrorismTargetTier_Ext#City, Equals, null)
60 })
61 tierQuery.or(\orCriteria -> {
62     orCriteria.compare(TerrorismTargetTier_Ext#ZipCode, Equals, polLocation.PostalCode)
63     orCriteria.compare(TerrorismTargetTier_Ext#ZipCode, Equals, null)
64 })
65
66 var results = tierQuery.select()
67 if(results.HasElements) {
68     var result = results.firstWhere( \ elt -> elt.ZipCode == polLocation.PostalCode and
69         elt.County == polLocation.County and elt.City == polLocation.City)
70     var result2 = results.firstWhere( \ elt -> elt.County == polLocation.County and elt.City == polLocation.City)
71     var result3 = results.firstWhere( \ elt -> elt.County == polLocation.County)
72
73     if(result != null) {
```

```
54     orCriteria.compareIgnoreCase(TerrorismTargetTier_Ext#County, Equals, polLocation.County)
55     orCriteria.compare(TerrorismTargetTier_Ext#County, Equals, null)
56 })
57 tierQuery.or(\orCriteria -> {
58     orCriteria.compare(TerrorismTargetTier_Ext#City, Equals, polLocation.City)
59     orCriteria.compare(TerrorismTargetTier_Ext#City, Equals, null)
60 })
61 tierQuery.or(\orCriteria -> {
62     orCriteria.compare(TerrorismTargetTier_Ext#ZipCode, Equals, polLocation.PostalCode)
63     orCriteria.compare(TerrorismTargetTier_Ext#ZipCode, Equals, null)
64 })
65
66 var results = tierQuery.select()
67 if(results.HasElements) {
68     var result = results.firstWhere( \ elt -> elt.ZipCode == polLocation.PostalCode and
69         elt.County == polLocation.County and elt.City == polLocation.City)
70     var result2 = results.firstWhere( \ elt -> elt.County == polLocation.County and elt.City == polLocation.City)
71     // - Defect 224 - Fix for defaulting target tier value irrespective of lower and upper
72     var result3 = results.firstWhere( \ elt -> elt.County.equalsIgnoreCase(polLocation.County))
73
74     if(result != null) {
```

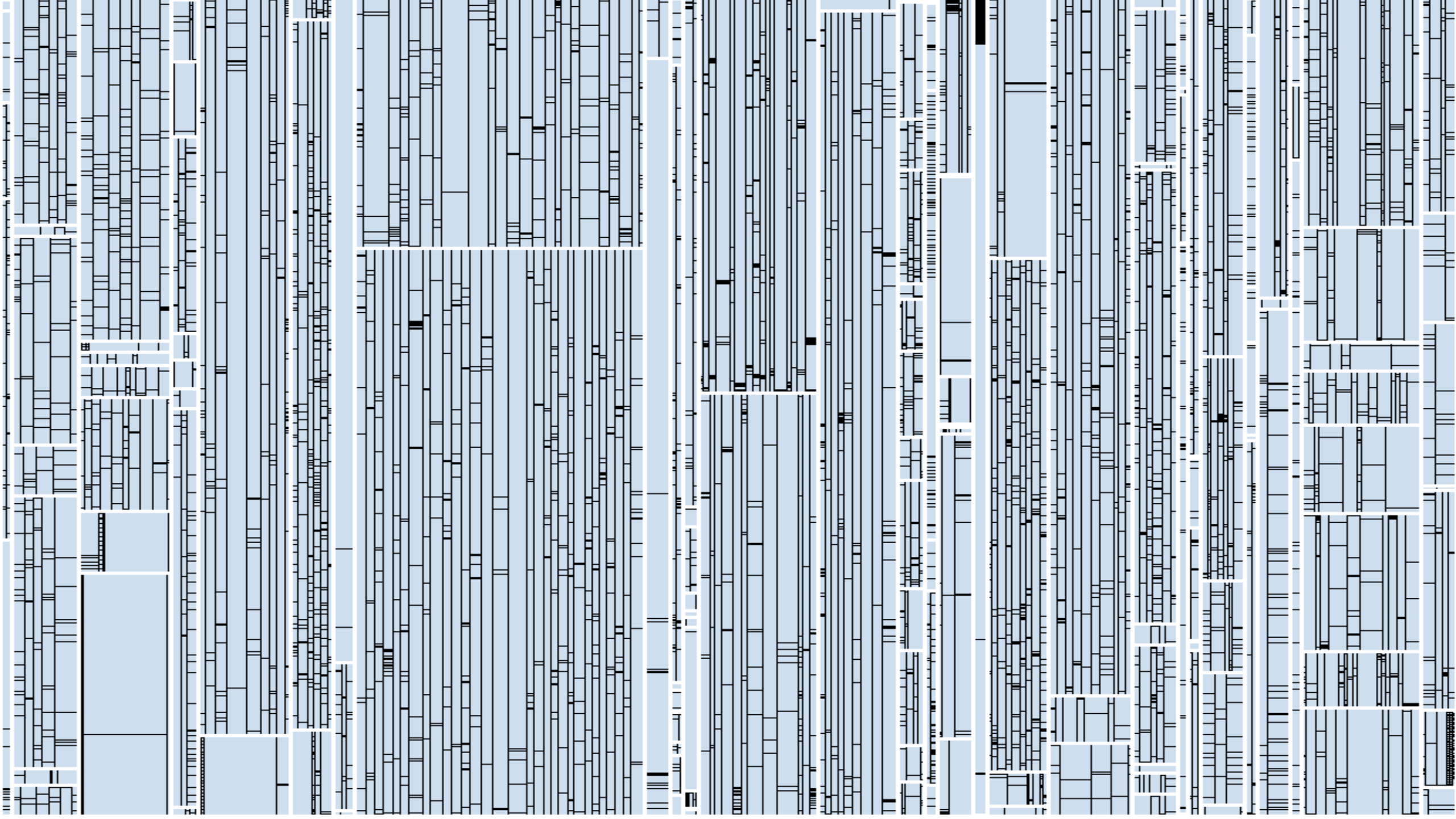

UI Controls

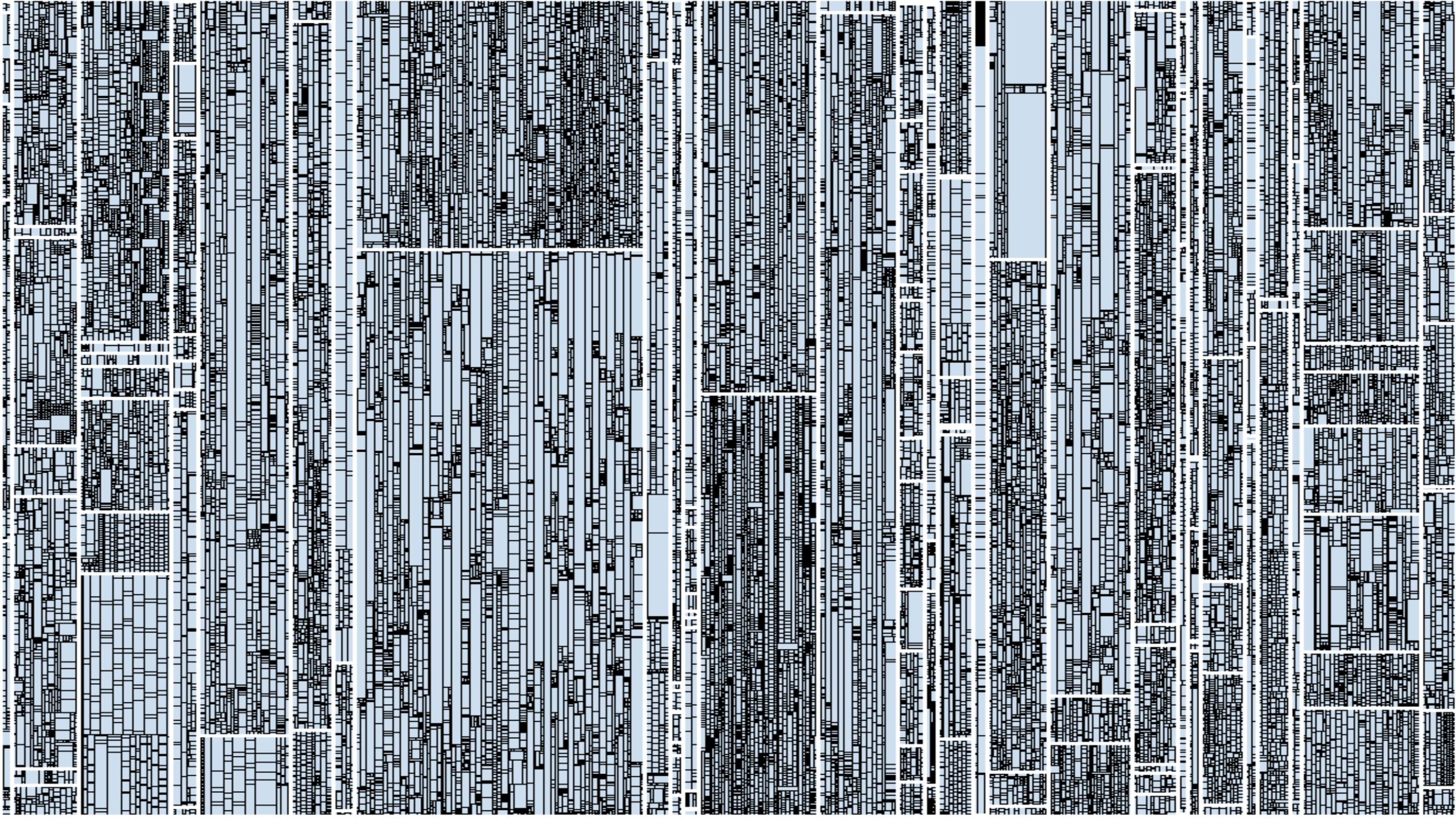
GUI.Dialogs

GUI.Base

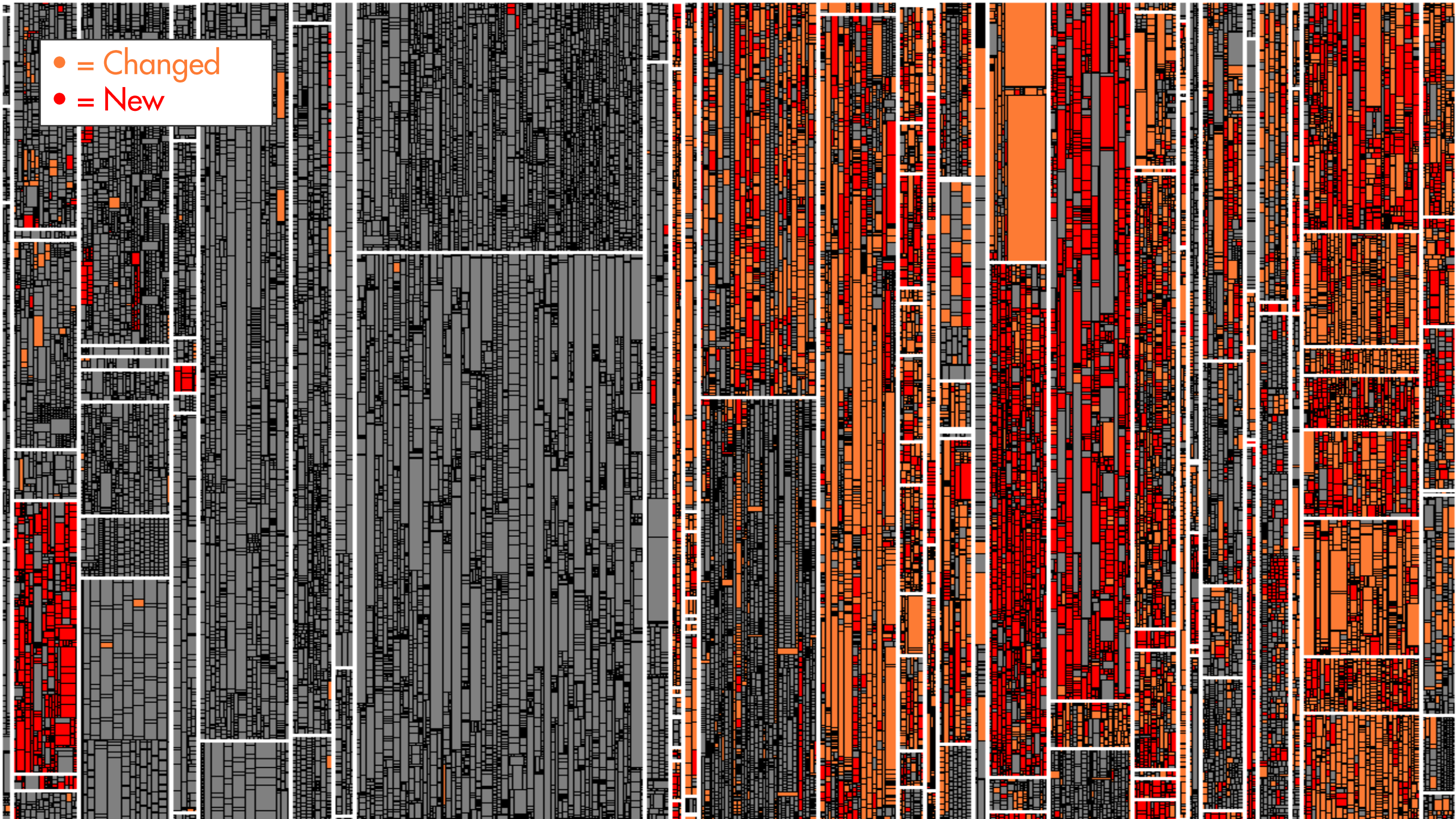
Authentication

Data Validation

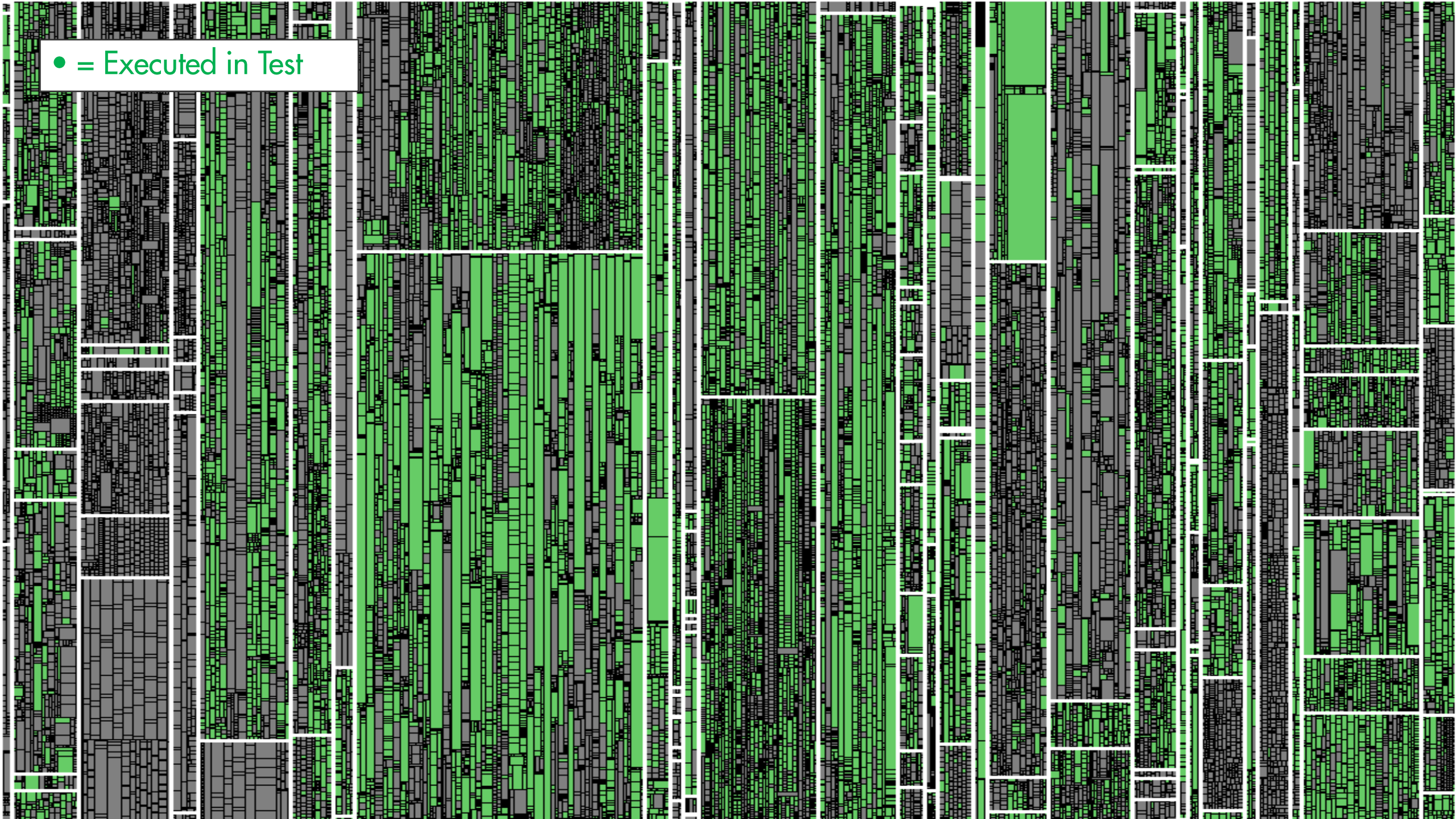




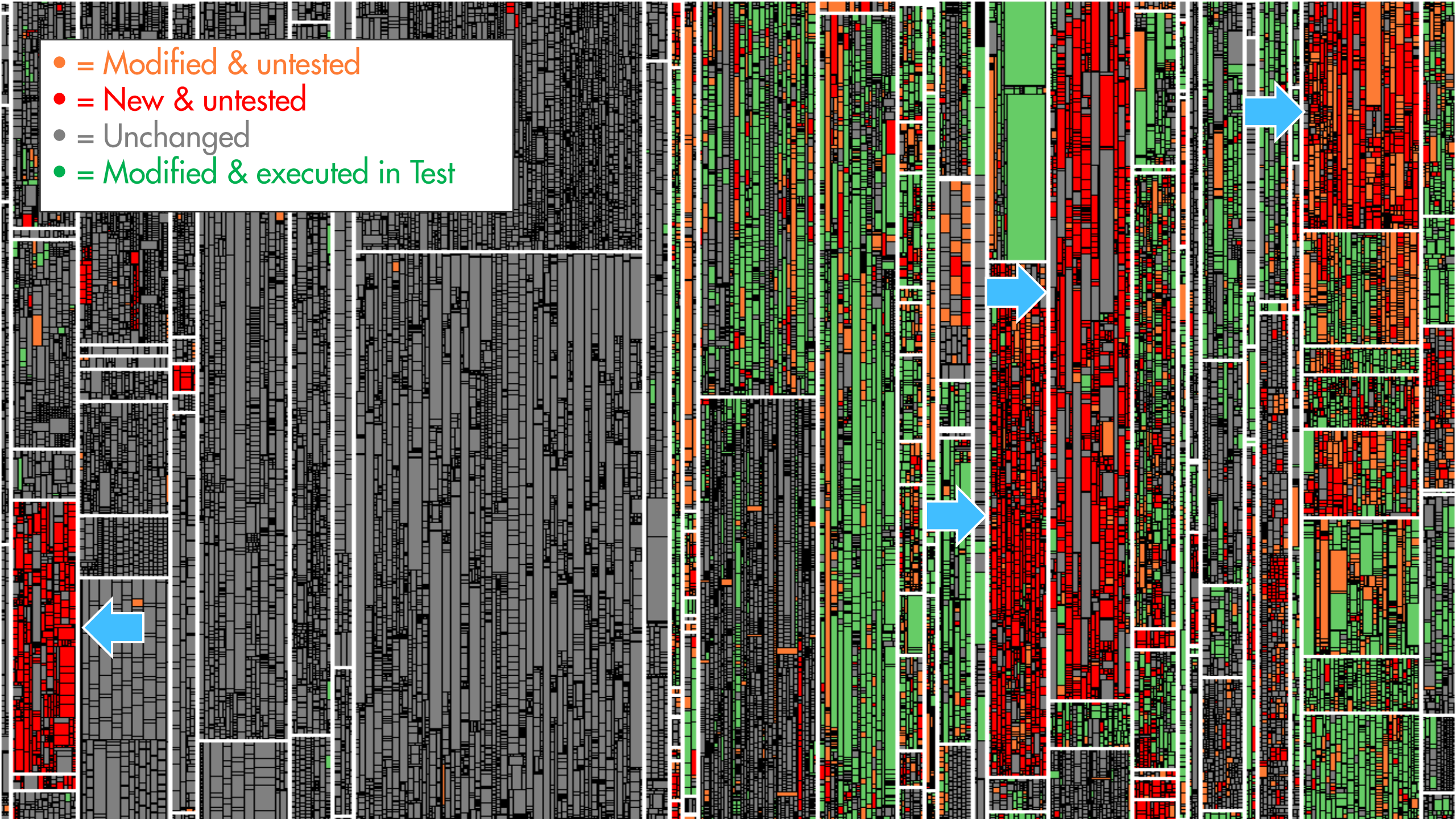
- = Changed
- = New



● = Executed in Test



- = Modified & untested
- = New & untested
- = Unchanged
- = Modified & executed in Test



Kosten-Nutzen Berechnung



Requirement
Engineers



User

2006

2007

2008

2009

2010

2011

2012

2013

2014

2015

2016

2017

2018

2019

2020

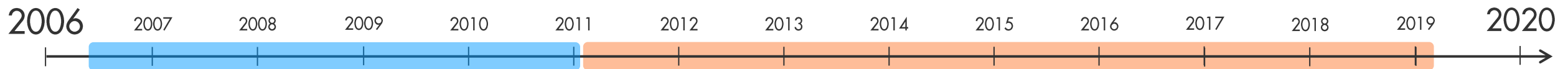


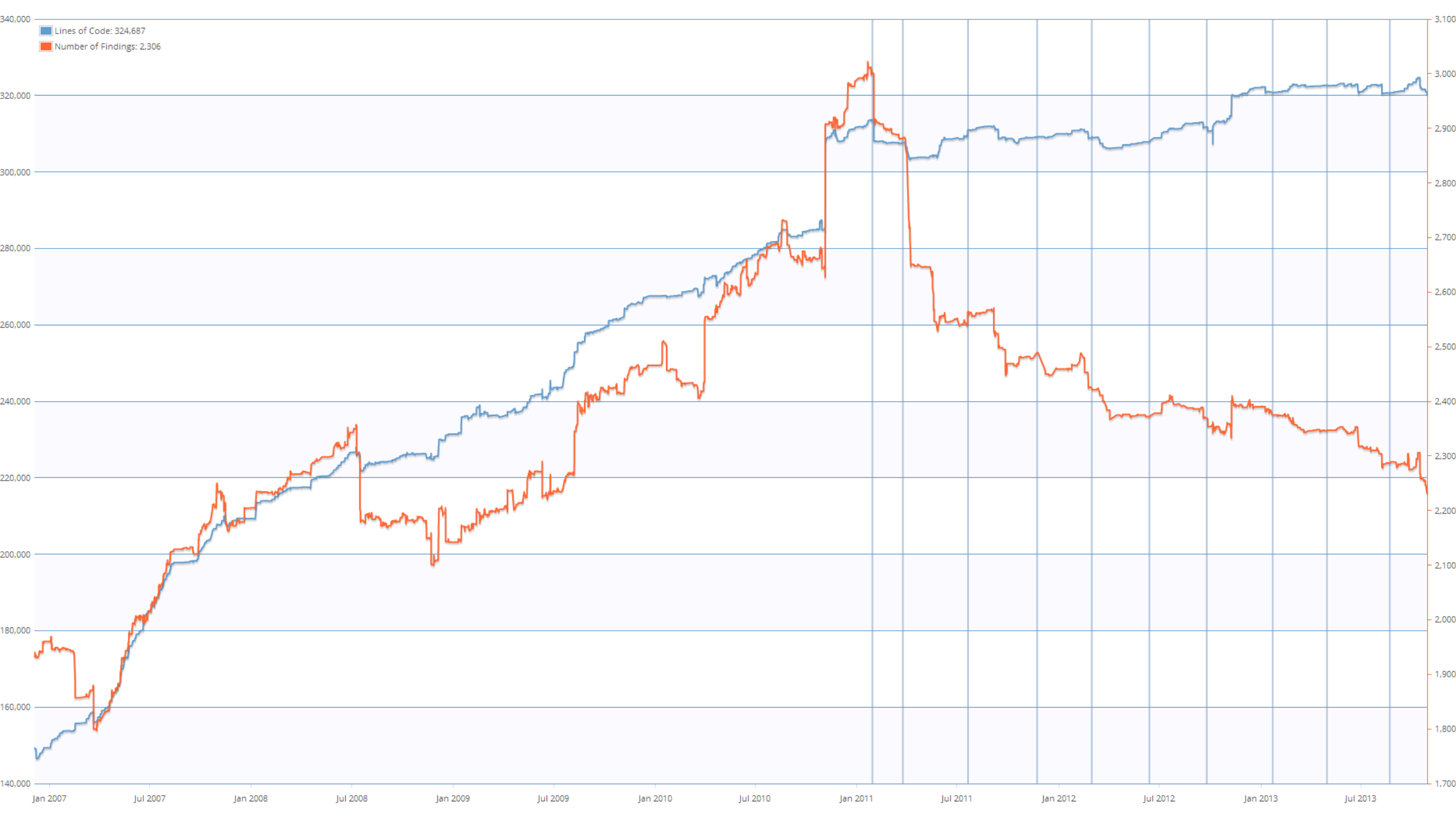


Requirement
Engineers



User





Kosten-Nutzen Berechnung von Clone Detection

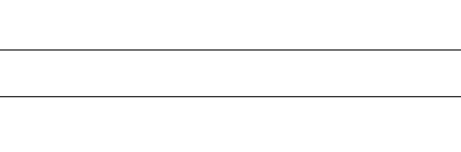

```
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
    boolean found = false;
    StringBuffer res = new StringBuffer();
    if (elements != null) {
        Index.getInstance().setCurrentRenderer(
            FlatReferenceRenderer.getInstance());
        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();
}
```

```
// 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();
}
```

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

Abstract
 Code cloning is not only assumed to inflate maintenance costs but also identified as a major cause for inconsistency. Consequently, the identification of duplicated code, clone detection, has been a very active area of research in recent years. Up to now, however, no substantial investigation of the consequences of code cloning on program correctness has been carried out. In this case study we analyzed three commercial systems written in C#, one written in Cobol and one open-source system written in Java. To conduct the study we developed a novel detection algorithm that enables us to detect inconsistent clones. We manually inspected about 900 code groups to locate the inevitable false positives and discovered each of the over 700 inconsistent clone groups with the developers on the respective systems to determine if the inconsistencies are intentional and if they represent faults. Altogether, around 1800 individual clone group assessments were manually performed in the course of the case study. This study laid the foundations of 107 faults that have been confirmed by the system's developers.

- 1. Clones and correctness**
 Research in software maintenance has shown that many programs contain a significant amount of duplicated (cloned) code. Such cloned code is considered harmful for two reasons: (1) multiple, possibly inconsistent, duplicate code increase maintenance costs and, (2) inconsistent changes to cloned code can create faults and, hence, lead to incorrect program behavior [19, 20]. While clone detection has been a very active area of research in recent years up to now, there is no thorough understanding of the degree of harmfulness of code cloning. In fact, some researchers even started to doubt the harmfulness of cloning at all [16]. This is because they have investigated clones generated automatically and that such changes can represent faults. Second, they propose a novel software tree-based algorithm for the detection of inconsistent clones. In contrast to other detection programs for the detection of inconsistent clones, our tool suite is made available for other researchers as open source.



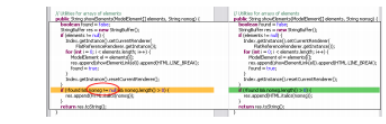
Syntypish The open source system Syntypish¹ is developed at the Technische Universität München (TUM) but none of the authors who executed the detection were involved in its development. It constitutes a collaboration environment for distributed software development projects. The inclusion of an IDE and the tooling makes it possible to work on the system. It does not make sense to analyze inconsistent clones if they are a pure phenomenon.

EV 1871 The Lebensversicherung von 1871 AG is a Munich-based life insurance company. The EV 1871 developers developed several commercial software systems for maintainers and PCs. In this study, we analyze a maintenance system constructed from several software systems in Cobol (EV 1871) employed by about 150 users.

Table 1. Summary of the analyzed systems

System	Development	Lang.	Age (years)	Lines (KLOC)
A	March Re	C#	6	317
B	March Re	C#	6	324
C	March Re	Java	2	495
D	EV 1871	Cobol	197	192
Syntypish	TUM	C#	8	281

¹http://syntypish.tum.de/ ²http://www.ev1871.com/ ³http://www.ev1871.com/



2. Terms and definitions
 The literature provides a wide variety of different definitions of clones and clone related terms [19, 28]. To avoid ambiguity, we describe the terms as used in this paper. Code is interpreted as a sequence of units, such as for example statements or lines of code, which are subject to independent evolution and are thus prone to misclassification. The reason to allow normalization of units in this step is, that often pieces of code are considered equal even despite differences in comments or naming, which can be resolved by the normalization. An exact Clone is thus a (conservative) subset of the code that approximates at least two in the (non-normalized) code. Thus our definition of a clone is purely syntactic, but catches exactly the idea of copy/paste, while excluding clones that are the result of a copy/paste with modifications.

3. Related work
 A substantial amount of research has been dedicated to clone cloning in recent years. The detailed survey by Koschützki [9] or Roy and Gordy [24] provide a comprehensive overview of existing work. Since this paper targets consequences of cloning and detection of inconsistent clones, we detail existing work in these areas.

4.2. Detecting inconsistent clones
 This section explains the approach used for detecting inconsistent clones in large amounts of code. Our approach is done on a code level, which usually is sufficient for finding copy-pasted code, while at the same time being efficient. The algorithm works by constructing a suffix tree of the code and then for each possible suffix an approximate search based on the edit distance in this tree is performed. Our clone detector is organized as a pipeline, which is depicted in Figure 2. The files under analysis are loaded and then fragmented by the analyzer, yielding a stream of tokens, which is filtered to exclude comments and generated code (recognized by our provided patterns). From the token stream, which consist of single keywords, identifiers, operators, and so on, the normalizer normalizes statements. This stage performs normalization, such that

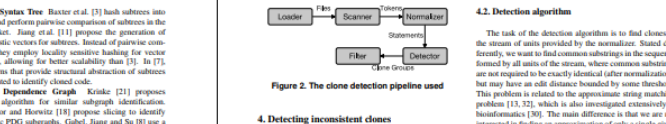
a small fraction of inconsistent clones becomes consistent again through later changes, potentially inducing a larger degree of independence of clones than hitherto believed. Geiger et al. [9] report that a relation between clone configurations and code clones could, contrary to expectations, not be statistically verified. Lotz and Werninger [20] report that no systematic relationship between code cloning and changeability could be established. The effect of cloning on maintainability and correctness is thus not clear. Furthermore, the above listed publications suffer from a few or more shortcomings that limit the transferability of the reported findings.

4.3. Pre-processing and filtering
 During and after detection, the clones that are reported are subject to filtering. Filtering is usually performed as early as possible, so no memory is wasted with storing clone groups that are not considered relevant. Using these filters, we discard clone groups whose clones overlap with each other and groups whose clones are contained in other clone groups. Additionally, we exclude not only an absolute ratio for the detection of inconsistent clones, but also a relative one, i.e., we filter clone groups where the number of inconsistent clones in the clone relative to the clone's length exceeds a certain amount. Moreover, we merge clone groups which share a common clone. While this leads to clone groups with non-related clones (so our definition of an inconsistent clone is not transitive), for practical purposes it is preferred to know of these individual relations, too.



4.4. Detection algorithm
 The task of the detection algorithm is to find clones in the stream of units provided by the normalizer. Stated differently, we want to find common substrings in the sequence formed by all units of the stream, where common substrings are not expected to be exactly identical (after normalization), but may have an edit distance below some threshold. This problem is related to the approximate string matching problem [13, 22], which is also investigated extensively in bioinformatics [30]. The main difference is that we are not interested in an approximation of only a single given word in the suffix, but rather are looking for a substring approximately occurring more than once in the entire sequence.

4.5. Scalability and performance
 Due to the huge implementation details, the worst case complexity is hard to analyze. Additionally, for practical reasons as Open Source at <http://openclone.de/en/def/> (last visited: 04.05.2016)



4.6. Detecting inconsistent clones
 This section explains the approach used for detecting inconsistent clones in large amounts of code. Our approach is done on a code level, which usually is sufficient for finding copy-pasted code, while at the same time being efficient. The algorithm works by constructing a suffix tree of the code and then for each possible suffix an approximate search based on the edit distance in this tree is performed. Our clone detector is organized as a pipeline, which is depicted in Figure 2. The files under analysis are loaded and then fragmented by the analyzer, yielding a stream of tokens, which is filtered to exclude comments and generated code (recognized by our provided patterns). From the token stream, which consist of single keywords, identifiers, operators, and so on, the normalizer normalizes statements. This stage performs normalization, such that

4.7. Internal variability
 As we ask the developers for their expert opinion on whether an inconsistency is intentional or unintentional and faulty or non-faulty, it is likely that the developers do not judge this correctly. One case is that the developer assesses something as non-faulty which actually is faulty. This case only reduces the chance to positively answer the research questions. The second case is that the developer rates something as faulty which is not faulty. We mitigated this threat by only rating an inconsistency as faulty if the developer was completely sure. Otherwise it was postponed and the developer consulted colleagues that know the corresponding part of the code better. Inconsistent candidates were ranked as intentional and non-faulty. Hence, again only the chance to answer the research questions positively was reduced.

4.8. Constructive
 The configuration of the clone detector has a strong influence on the detection results. We calibrated the parameters through pre-study and user experience with clone detection in general. The configuration also varies over the different programming languages encountered, so that their differences are taken into account. However, this should not strongly affect the detection of inconsistent clones because we spent great care to configure the tool in a way that the resulting clones are sensible.

4.9. Conclusion
 In this paper we provide strong evidence that inconsistent clones constitute a major source of faults, which means that cloning can be a substantial problem during development. It is thus interesting to investigate whether the same is also true in other systems and their evolution. Our results suggest that every second unintentionally inconsistent clone in our study leads to a fault. Furthermore, we provide a scalable algorithm for finding such inconsistent clones as well as suitable tool support for future experiments.

5. 52% oder ungewollten unterschiede fehlerhaft

Kritisch Nutzersichtbar Nicht nutzlersichtbar

Table 1. Summary of the analyzed systems

System	Development	Lang.	Age (years)	Lines (KLOC)
A	March Re	C#	6	317
B	March Re	C#	6	324
C	March Re	Java	2	495
D	EV 1871	Cobol	197	192
Syntypish	TUM	C#	8	281

5.1. Study description
 In order to gain a valid insight into the effects of inconsistent clones, we use a study design with 3 objects and 3 research questions that guide the investigations.

5.2. Study objectives
 We chose 2 objects and 1 open source project as sources of software systems. This resulted in 3 analyzed projects. In total, these systems written in different languages, by different teams in different companies and with different functionalities to analyze the transferability of the study results. These objects included 3 systems written in C#, a Java system as well as a long-lived Cobol system. All these systems are already in production. For non-disclosure reasons we give the commercial systems names from A to D. An overview is shown in Table 1.

5.3. Research questions
 The underlying problem that we analyze are clones and especially their inconsistencies. In order to investigate this question, we answer the following 3 more detailed research questions. Our main research questions are:

RQ 1: Are clones created unintentionally?
 The first question we need to answer is whether inconsistent clones appear at all in real-world systems. This not only means whether we can find them at all but also whether they can be considered a significant part of the code of the system. It does not make sense to analyze inconsistent clones if they are a pure phenomenon.

RQ 2: Are inconsistent clones created unintentionally?
 Having established that there are inconsistent clones in real systems, we need to analyze whether these inconsistent clones were created intentionally or not. It can be questioned whether it is sensible to change a clone so that it becomes inconsistent to its counterpart because if the other conform to it, whether the developer is aware of this change, to which the inconsistency is intentional.

RQ 3: Can inconsistent clones be indicators for faults in real systems?
 The first question we need to answer is whether inconsistent clones appear at all in real-world systems. This not only means whether we can find them at all but also whether they can be considered a significant part of the code of the system. It does not make sense to analyze inconsistent clones if they are a pure phenomenon.

#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

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

Ø Fehlerfolgekosten 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

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

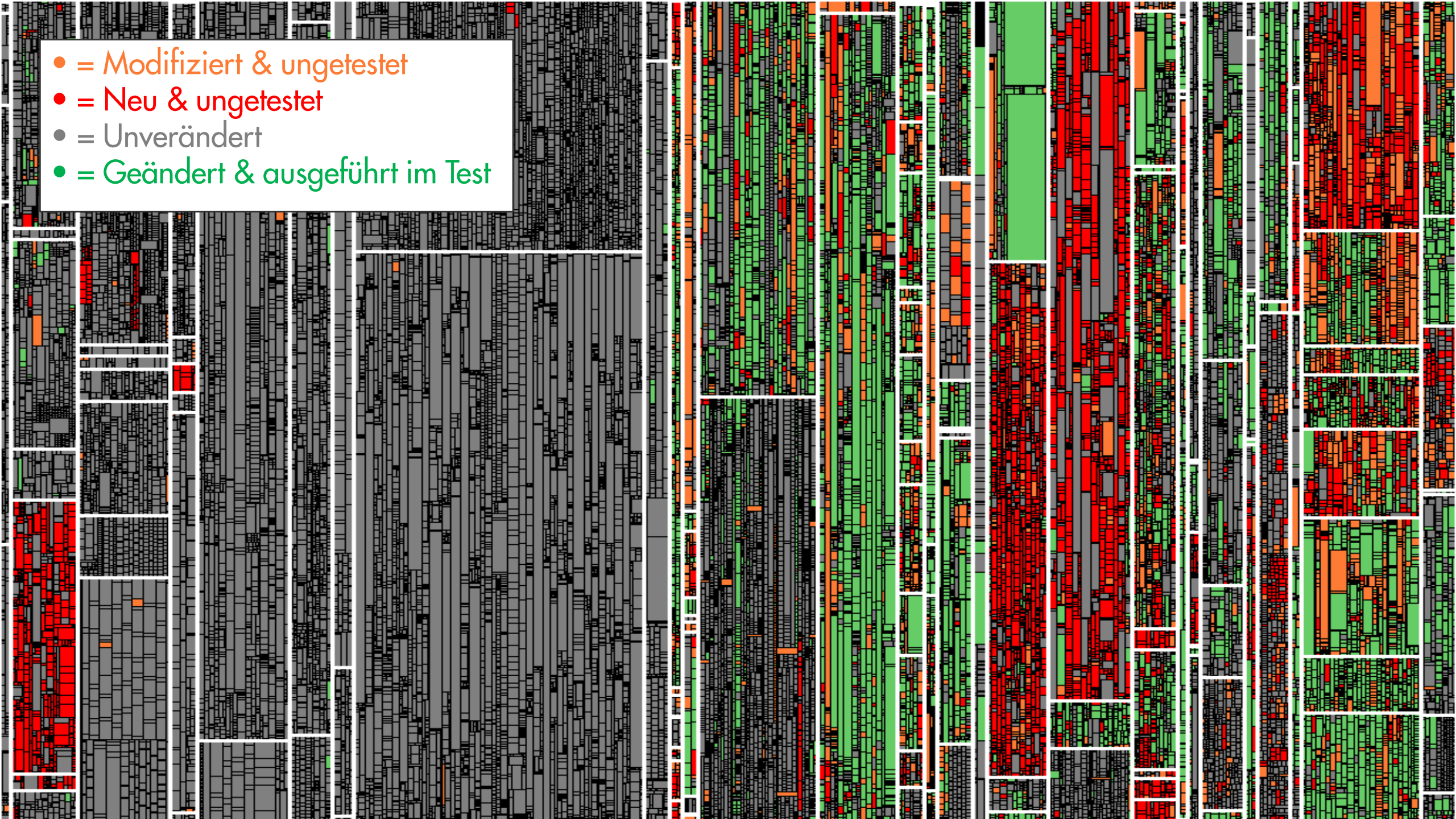
495 $\frac{PT}{Jahr}$

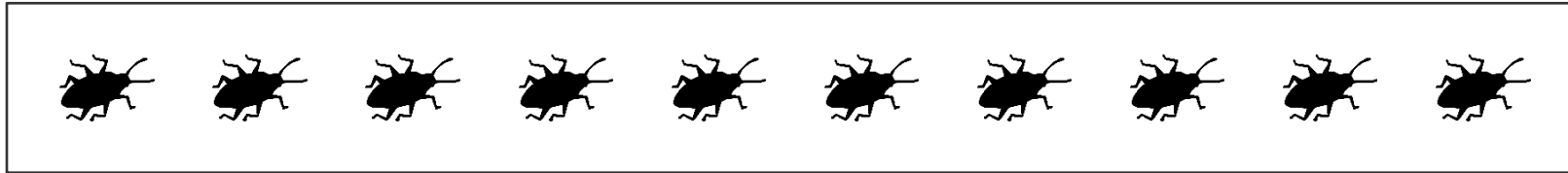
$$500 \frac{PT}{Jahr}$$

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

Kosten-Nutzen Berechnung von Test-Gap-Analyse

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



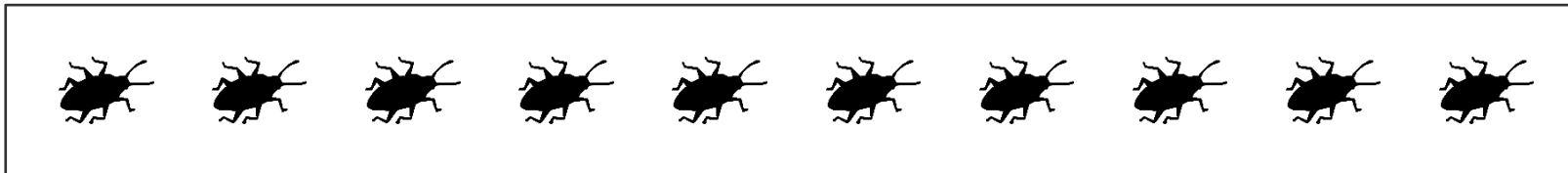


Test

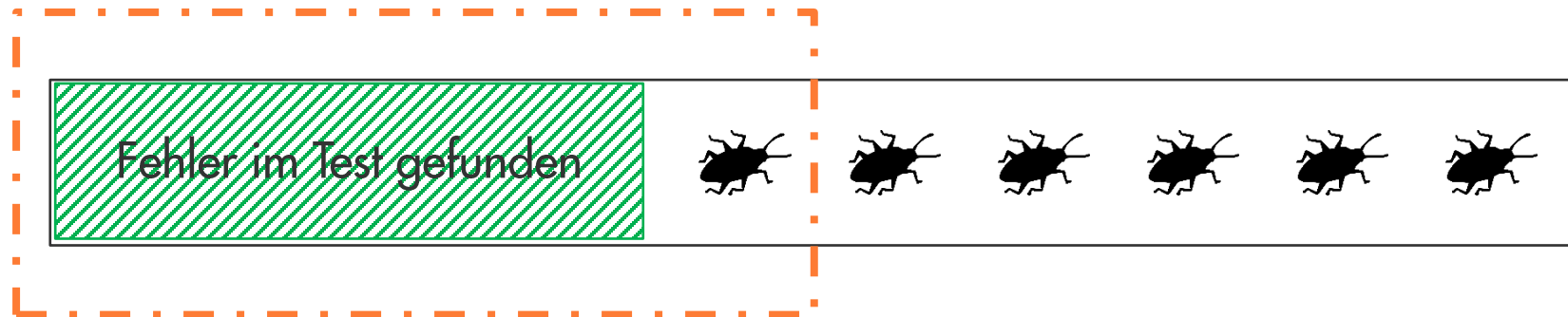


%Restfehler

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

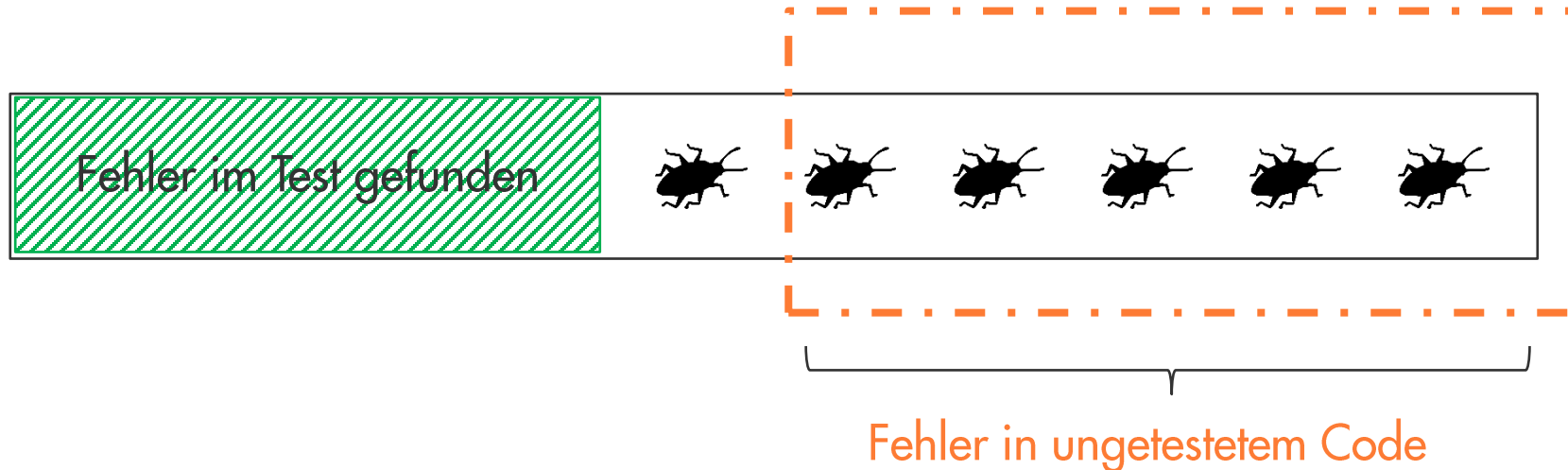


$$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivität} + \% \text{Testgap}$$



Im Test verpasste Fehler
in getestetem Code

$$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivitat} + \% \text{Testgap}$$



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
COSE GmbH
Germany

Rudolf Vass, 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 increase 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.

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

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.

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 but 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 contained field bugs.

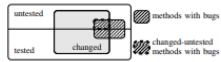


Fig. 2. Method categories used to evaluate change coverage

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 fitness 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 metric 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

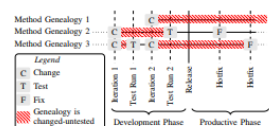


Fig. 1. Development life-cycle

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-untested*. 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

Category	Release 1		Release 2	
	Absolute	Relative	Absolute	Relative
changed-untested	5	22%	3	30%
changed-untested	10	43%	4	40%
unchanged-untested	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 other things, 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 J. 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] J. Graves, A. Karz, J. Marton, and H. Siv, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, 2003.
- [4] T. J. Ostrand, E. J. Weisker, and R. M. Bell, "Where the bugs are," in *ISSTA*, 2004.
- [5] H. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsel, "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 in defect prediction," in *ISSTSE*, 2010.
- [7] V. Chankeslavava, V. K. Shabbag, A. Patragari, R. Soodala, and S. Lakshmanan, "Safe subset-regression test selection for managed code," in *ISST*, 2008.
- [8] Y.-F. Chen, D. Rosenblum, and K.-P. Vo, "Testmate: 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 *ISMM*, 2008.
- [11] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environments," in *ISSTA*, 2002.
- [12] M. Zine, 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 *ISST*, 2006.
- [14] Y. Malaya, M. Li, J. Bierman, and R. Karach, "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. Eide, M. Jankel, E. Jorgens, B. Hauptmann, R. Vass, and K. Prommer, "How much does untested code matter for maintenance?" in *ICSE*, 2012.
- [18] O. Trush, S. Schuster, and M. D. Smith, "Empirical instrumentation for lightweight program profiling," School of engineering and Applied Sciences, Harvard University, Tech. Rep., 2010.
- [19] E. Juergens, M. Fellias, M. Hermannsdorfer, F. Deisselboeck, R. Vass, and K. Prommer, "Feature profiling for evolving systems," in *ICPC*, 2011.

Wieviele Änderungen sind ungetestet?

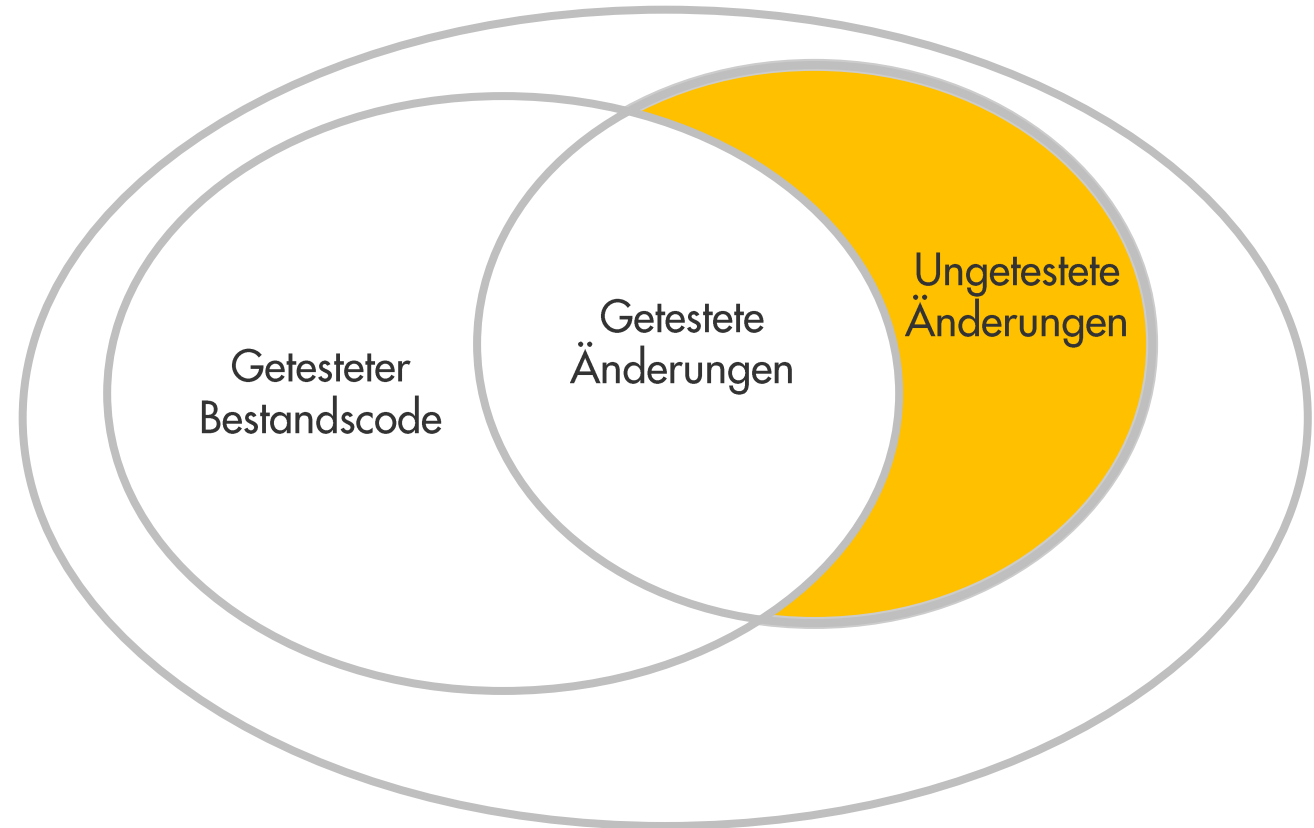
Studie: C# System @ Munich Re

Release A:

15% Code neu/geändert, **>50% ungetestet**

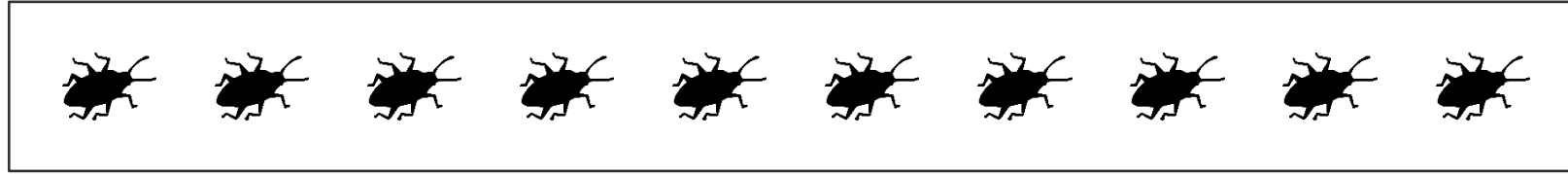
Release B:

15% Code neu/geändert, **>60% ungetestet**

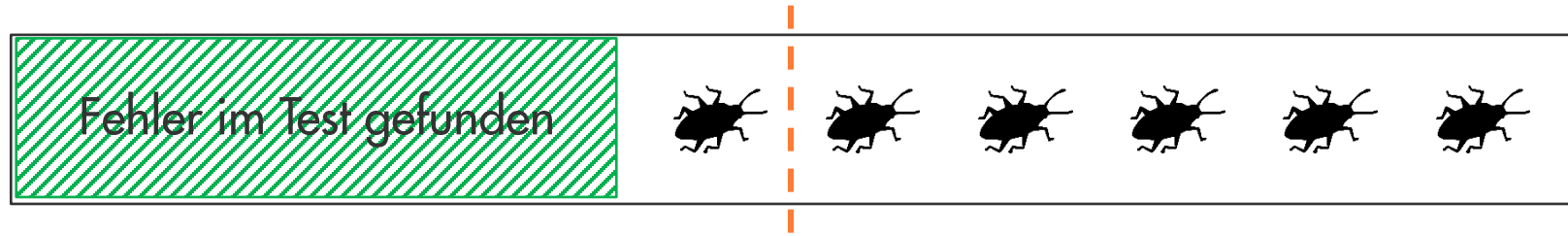


Feldfehlerwahrscheinlichkeit 5x höher für ungetestete Änderungen!

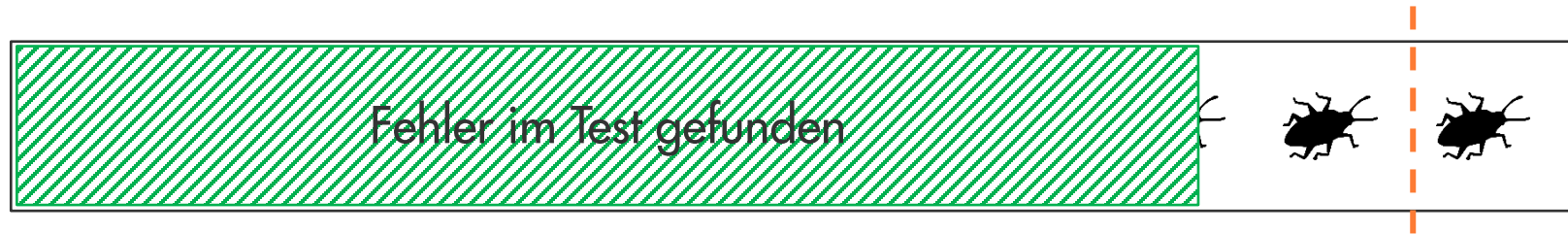
$$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivität} + \% \text{Testgap}$$



$\% \text{Restfehler} = 60\%$



$\% \text{Restfehler} = 28\%$



Reduzierte Feldfehler = **50%**

Test-Gap-Analyse reduziert Feldfehler in den Applikationen der Munich Re um 1/2

Fazit

Technische Schulden sammeln sich in allen langlebigen Softwaresystemen.

Um die Konsequenzen von technischen Schulden zu kommunizieren, müssen wir ihre Auswirkungen auf unsere Aktivitäten und Kosten im eigenen Projekt herausarbeiten. Das ist unsere Aufgabe als Entwickler und Architekten.

Eine zentrale Abstraktionsebene sind hierbei die Aktivitäten, auf die sich technische Schulden negativ auswirken. Es hat sich für uns bewährt, Aktivitäten einzeln zu betrachten und im Zweifel konservative Abschätzungen zu treffen. Das macht die Abschätzung der Auswirkungen nachvollziehbar und überzeugend.

Wir stehen gerne für Diskussionen und zum Austausch zur Verfügung!

Kontakt – Ich freue mich auf Diskussionen 😊



Dr. Elmar Jürgens · juergens@cqse.eu · +49 179 675 3863

CQSE GmbH
Lichtenbergstraße 8
85748 Garching bei München
www.cqse.eu

