

COSE

# **CQSE Software Intelligence Talk**

# Agenda

- Begrüßung
- Kosten-Nutzen-Berechnung von Qualitätsanalysen  
Erfahrungen bei der Munich Re  
Uwe Proft  
Münchener Rückversicherungs-Gesellschaft AG  
Dr. Elmar Juergens  
CQSE GmbH
- Q&A Session

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

Uwe Proft (Munich Re)

Elmar Jürgens (CQSE)



Requirement  
Engineers



User



## TreeAdministrationQuarterly.cs

```
93  /// </summary>
94  protected override bool DoLazyLoading(
95      UltraTreeNode node)
96  {
97      // Some types of segments always have the
98      // expand icon
99      if ( node.Tag is GaSegment ||
100         node.Tag is MainSegment ||
101         node.Tag is GeneralMainSegment ||
102         node.Tag is Branch )
103         return true;
104     // All others only if there are child nodes
105     return false;
106 }
107
108 /// <summary>
109 /// Structure segments return only loss or
110 /// premium segments according to
111 /// the mode.
112 /// </summary>
113 protected override IList GetChildSegments(
114     ISegment parent, ref bool alreadySorted )
115 {
116     if ( parent is StructureSegment )
117     {
118         // Show either premium or loss beyond
119         // structure segment
120         StructureSegment strucSeg = (StructureSegment
121             )parent;
122         if ( (TypeOfSgmtEnum) GetFilterValue( typeof(
123             TypeOfSgmtEnum) ) == TypeOfSgmtEnum.Loss
124             )
125             return strucSeg.GetLossProcessingSegments();
126         else
127             return strucSeg.GetPremiumProcessingSegments
128             ();
129     }
130 }
```

## TreeAdministrationYearly.cs

```
106  /// </summary>
107  protected override bool DoLazyLoading(
108      UltraTreeNode node)
109  {
110      // Some types of segments always have the
111      // expand icon
112      if ( node.Tag is GaSegment ||
113         node.Tag is MainSegment ||
114         node.Tag is GeneralMainSegment ||
115         node.Tag is Branch )
116         return true;
117     // All others only if there are child nodes
118     return false;
119 }
120
121 /// <summary>
122 /// Structure segments return only loss or
123 /// premium segments according to
124 /// the mode.
125 /// </summary>
126 protected override IList GetChildSegments(
127     ISegment parent, ref bool alreadySorted )
128 {
129     if ( parent is StructureSegment )
130     {
131         // Show either premium or loss beyond
132         // structure segment
133         StructureSegment strucSeg = (StructureSegment
134             )parent;
135         if ( (TypeOfSgmtEnum) GetFilterValue( typeof(
136             TypeOfSgmtEnum) ) == TypeOfSgmtEnum.Loss
137             )
138             return strucSeg.GetLossProcessingSegments();
139         else
140             return strucSeg.GetPremiumProcessingSegments
141             ();
142     }
143 }
```

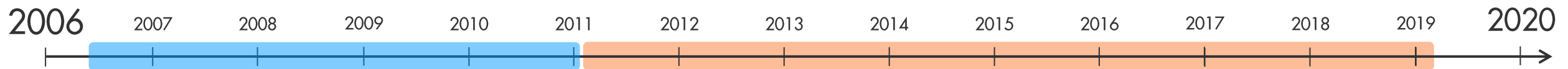




Requirement  
Engineers



User



**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



**TQE**

Static code analysis



**TGA**

Dynamic code analysis during test



**TSA**

Static analysis of manual tests

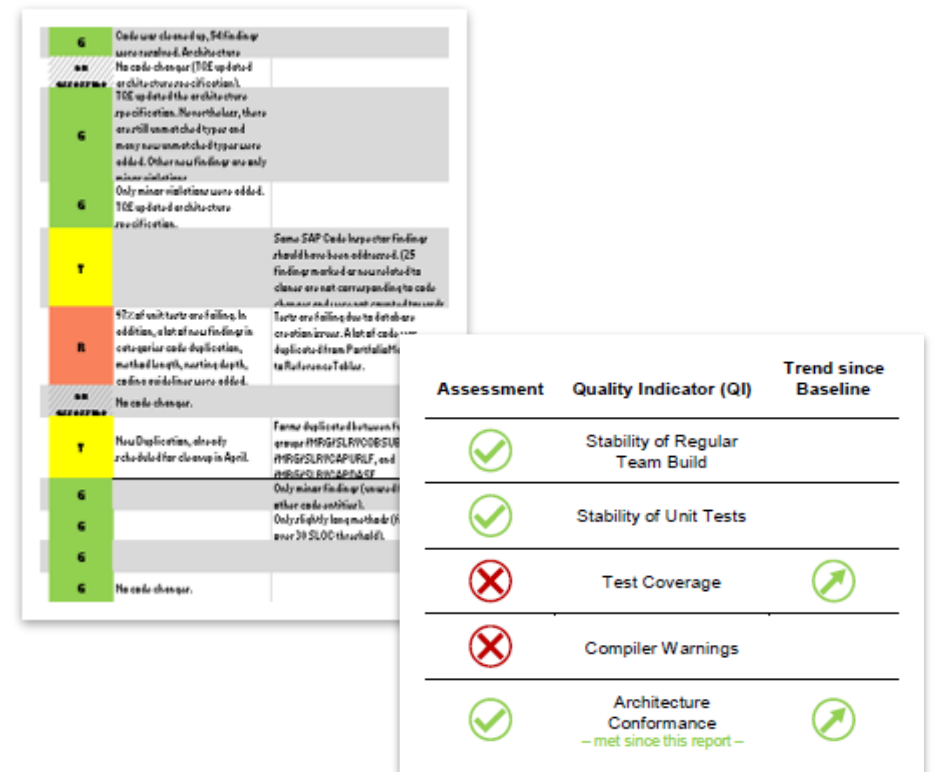


Project/Service/Line  
Mgr.  
IT Product Owner

# Munich Re Internal Services

## Dashboards, IDE Plugin, Azure DevOps

## Monthly Assessments, Reports





CodeParser.Expressions.cs | CodeParser.Preprocessor.cs | CodeParser.Statements.cs | CsParser.cs | CsToken.cs | ElementType.cs | FileHeader.cs | ICodePartExtensions.cs | QueryClauseType.cs

StyleCop.CSharp.CodeParser

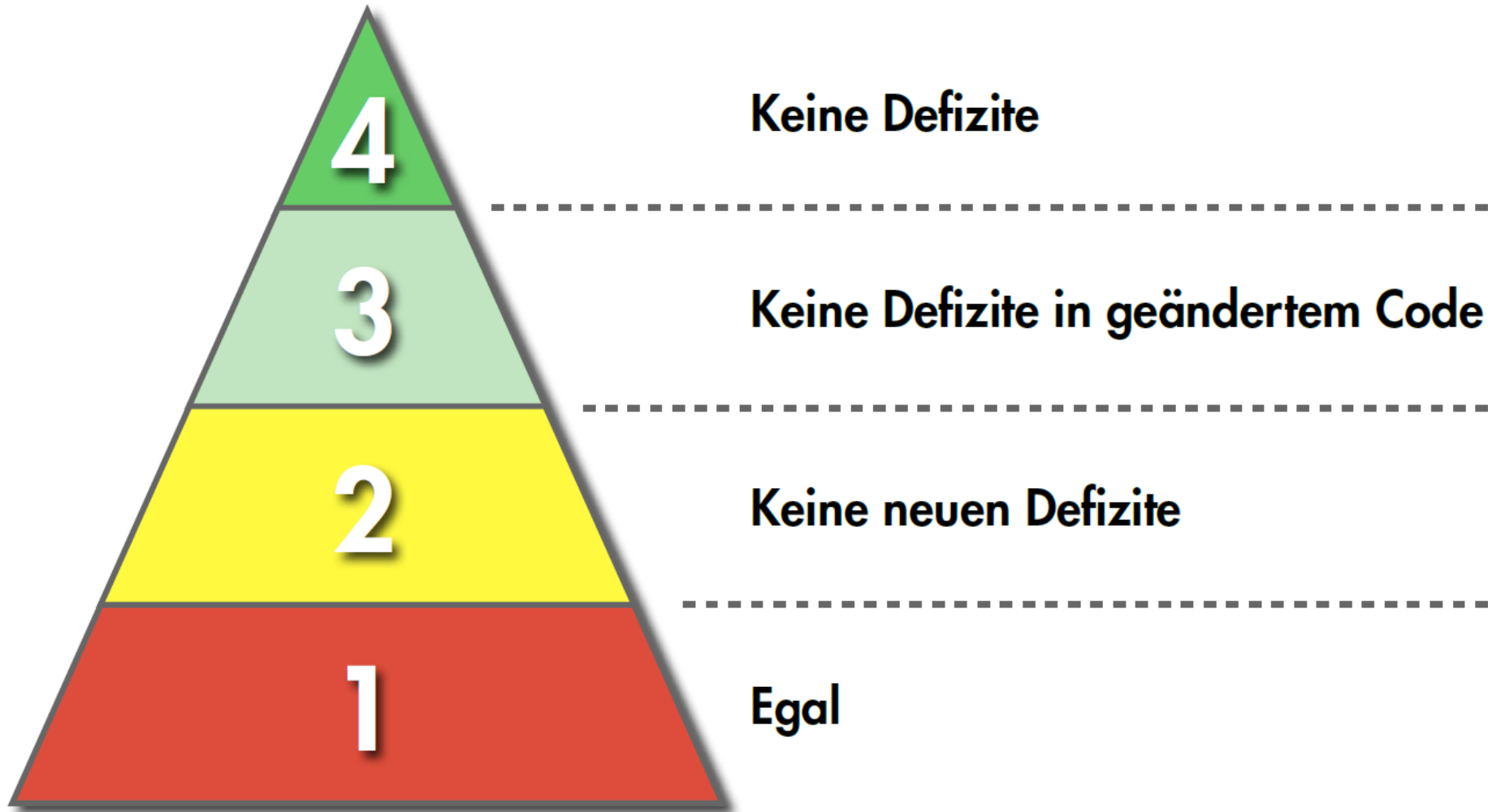
GetObjectInitializerExpression(bool unsafeCode)

```
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     if (symbol.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);
2771
```

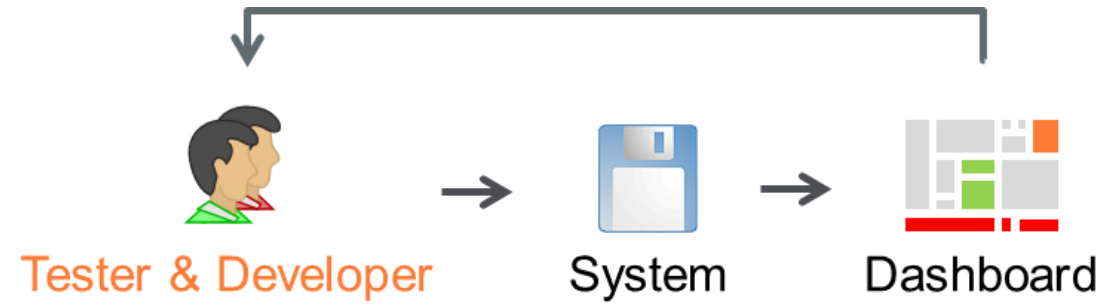
Clone with 2 instances of length 9

Redundancy  
Clones

# Qualitätsziele



# Prozess



# Portfolio Overview – Links to Dashboards & Monthly Assessments

Application	Dev	Test	TQE	TGA	TSA
<a href="#">iMyCG - Template Customizer</a>	<a href="#">iMyCG</a>	<a href="#">iMyCG</a>	<a href="#">2019-12</a>	<a href="#">x*</a>	<a href="#">Q</a>
<a href="#">iMyCG</a>	<a href="#">iMyCG</a>	<a href="#">iMyCG</a>	<a href="#">2019-12</a>	<a href="#">Q</a>	<a href="#">Q</a>
<a href="#">iMyCG</a>	<a href="#">iMyCG</a>	<a href="#">iMyCG</a>	<a href="#">2019-12</a>	<a href="#">2019-12</a>	<a href="#">2019-12</a>
<a href="#">iMyCG</a>	<a href="#">iMyCG</a>	<a href="#">iMyCG</a>	<a href="#">Q</a>	<a href="#">Q</a>	<a href="#">Q</a>
<a href="#">iMyCG</a>	<a href="#">iMyCG</a>	<a href="#">iMyCG</a>	<a href="#">2019-12</a>	<a href="#">Q</a>	<a href="#">Q</a>

# Portfolio Overview – Trends

Application	Dev
MyCo Template Controller	MyCo POC IT
MyCo	MyCo POC MyCo IT1.2 (MyCo Technology)
MyCo	MyCo IT1.5 (MyCo)
MyCo	MyCo IT1.5 (MyCo)
MyCo	MyCo IT1.8 (MyCo)

## TQE assessment trend for

Assessment	Comment	QG relevant findings	Details
2019-09	Only one small finding in changed code	1	Show Details
<del>2019-08</del>	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:			
<ul style="list-style-type: none"> <li>Naming convention violations in <code>TmOneParam</code></li> <li>Method threshold violation in <code>method DeleteProcessYear</code> of class <code>ProcessYearsController</code></li> <li>Method threshold violation in a <code>lambda</code> in class <code>LossChartService</code></li> <li>Cloning between <code>ReverseTriangleGrid</code> and <code>CommissionTriangleGrid</code> (c.f. <a href="#">here</a>)</li> </ul>			
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

TGA	TSA
2019-12	2019-12
2019-12	2019-12
2019-12	<del>2019-12</del>



# Portfolio Overview – Trends

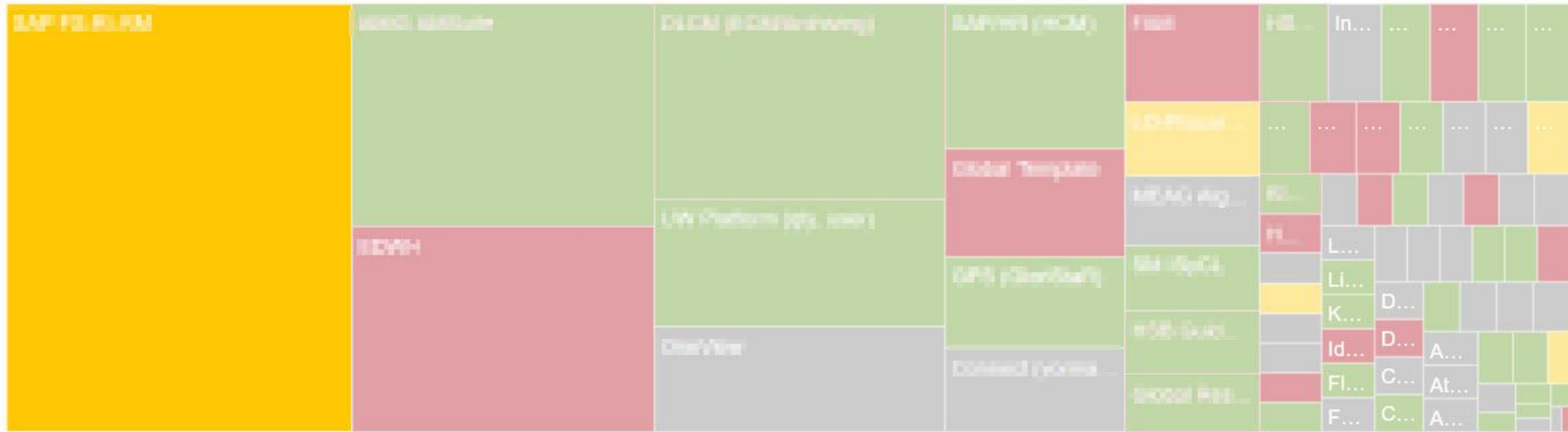
Application	Dev
MySQL Template Controller	MSDev, PMS, IT
MS-GEN	MSDev, Applian, PMS, Budget
IT1.2 (BAZ - Test/Training)	IT1.2 (BAZ - Test/Training)
MS-GEN	MSDev, IT1.2 (BAZ - Test/Training)
IT1.5 (MS)	IT1.5 (MS)
MS-GEN	MSDev, IT1.5 (MS)
IT1.5 (MS)	IT1.5 (MS)
MS-GEN	MSDev, IT1.8 (MS)

## 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:			
<ul style="list-style-type: none"> <li>Naming convention violations in <code>TmOneParam</code></li> <li>Method threshold violation in <code>method DeleteProcessYear</code> of class <code>ProcessYearsController</code></li> <li>Method threshold violation in a <code>lambda in class LossChartService</code></li> <li>Cloning between <code>ReverseTriangleGrid</code> and <code>CommissionTriangleGrid</code> (c.f. <a href="#">here</a>)</li> </ul>			
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

Assessment	Comment	QC relevant findings	Details
2019-09	Only few findings compared to the amount of change	93	Show Details
2019-08	Mostly minor violations given large amount of changes.	87	Show Details
2019-07	Mostly minor violations given large amount of changes.	131	Show Details
2019-06	Mostly minor violations.	117	
2019-05	Mostly minor violations.		Show Details
2019-04	Tolerable number of findings given amount of code changes	104	Show Details
2019-03	Tolerable but not significant amount of findings.	106	Show Details
2019-02	Tolerable but no significant amount of findings.	160	Show Details
2018-11	Tolerable but significant amount of findings.	173	Show Details
2018-12	Minor violations only.	77	Show Details
2018-11	Some findings that could have been avoided and resolved.	123	Show Details
2018-10	Minor violations only. Amount of findings tolerable with respect to code churn.	121	Show Details
2018-09	Minor violations only. Amount of findings tolerable with respect to code churn.	64	
2018-08	Minor violations only.	181	Show Details
2018-07	Duplicated code in <code>lean</code> and <code>spc</code> classes increased clone coverage significantly.	101	Show Details
2018-06	Minor violations only.	77	
2018-05	Tolerable amount of findings wrt to code change.	180	
2018-04	Tolerable amount of findings wrt to code change	99	Show Details
2018-03	Findings correspond to minor structural violations only.	97	
2018-02	New findings correspond to minor structural violations.	77	
2018-01	Goal reached. Amount of findings tolerable with respect to code churn.	122	Show Details
2017-12	Goal reached. Amount of findings tolerable with respect to code churn.	101	
2017-11	Goal reached. Amount of findings tolerable with respect to code churn.	37	
2017-10	High code churn with positive trend in most quality indications. Architecture specification not up-to-date anymore.	80	Show Details
2017-09	Code duplications, long methods, deeply nested code and coding guideline violations	92	Show Details
2017-08	Newly duplicated code, new long methods, new coding guideline violations. However tolerable, increase of system size by 10.000 code lines.	148	Show Details
2017-07	new long methods, new deeply nested code, CK wrt to file changes build/ test/ build and tests are running	8	
2017-06	New findings tolerable, new compiler warnings may also be from last month	48	
2017-05	SLOC +5050. Several clones between <code>box</code> and <code>normal</code> submission header broke up (intended?). Also some new clones. Other findings tolerable.	99	
2017-04	New cloned component (Treaty.Kbox.DataAccess)	142	
2017-03	several new clones affecting not related business entities, findings tolerable given growth of +10k SLOC	106	
2017-02	Given large amount of new code (+10k LOC), new findings ok, also resolved several old ones	40	
2017-01	excludes review related findings	53	Show Details
2016-12	Peer review findings have been subtracted	26	Show Details
2016-11			
2016-10	Given amount of development acceptable amount of new findings.	71	
2016-09	Most findings are either architecture related or target review findings or lead code. The remaining ones or minor clones or minor method length violations	297	
2016-08	Most findings concern the unfinished architecture spec. However several findings in new or modified code.	386	

# Monthly Assessment Results Portfolio Aggregation



Name	Tool	Assessment	Comments
TQE	TQE	GREEN	Only few violations
TGA	TGA	YELLOW	Some relevant test gaps
TSA	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?**

```
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) {
74
```

```
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     //AKN - Defect 224598 - 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) {
```

Project	QG	Assessment	Comment	Detailed Comment	QG-relevant Findings
[REDACTED]	2	Y	Four new security relevant findings should have been reviewed.	Missing authorization checks (1) at the beginning of report [REDACTED]_GOODSRECEIPT. (2) before CALL TRANSACTION ycl_p2p_invoice=>gc_ta_f90 in method create_post_values in class [REDACTED]_FIXEDASSET (3) before CALL TRANSACTION gc_ta_fbd1 in method start_recurring_entry in class [REDACTED]_INVOICE	4



Di 29.10.2019 18:01

[REDACTED] Munich-MR

AW: Security Code Scan - Monthly Assessment [REDACTED]

To  IT TQE-TGA (Pool) - Munich-MR; [REDACTED]

Cc  Proft Uwe - Munich-MR

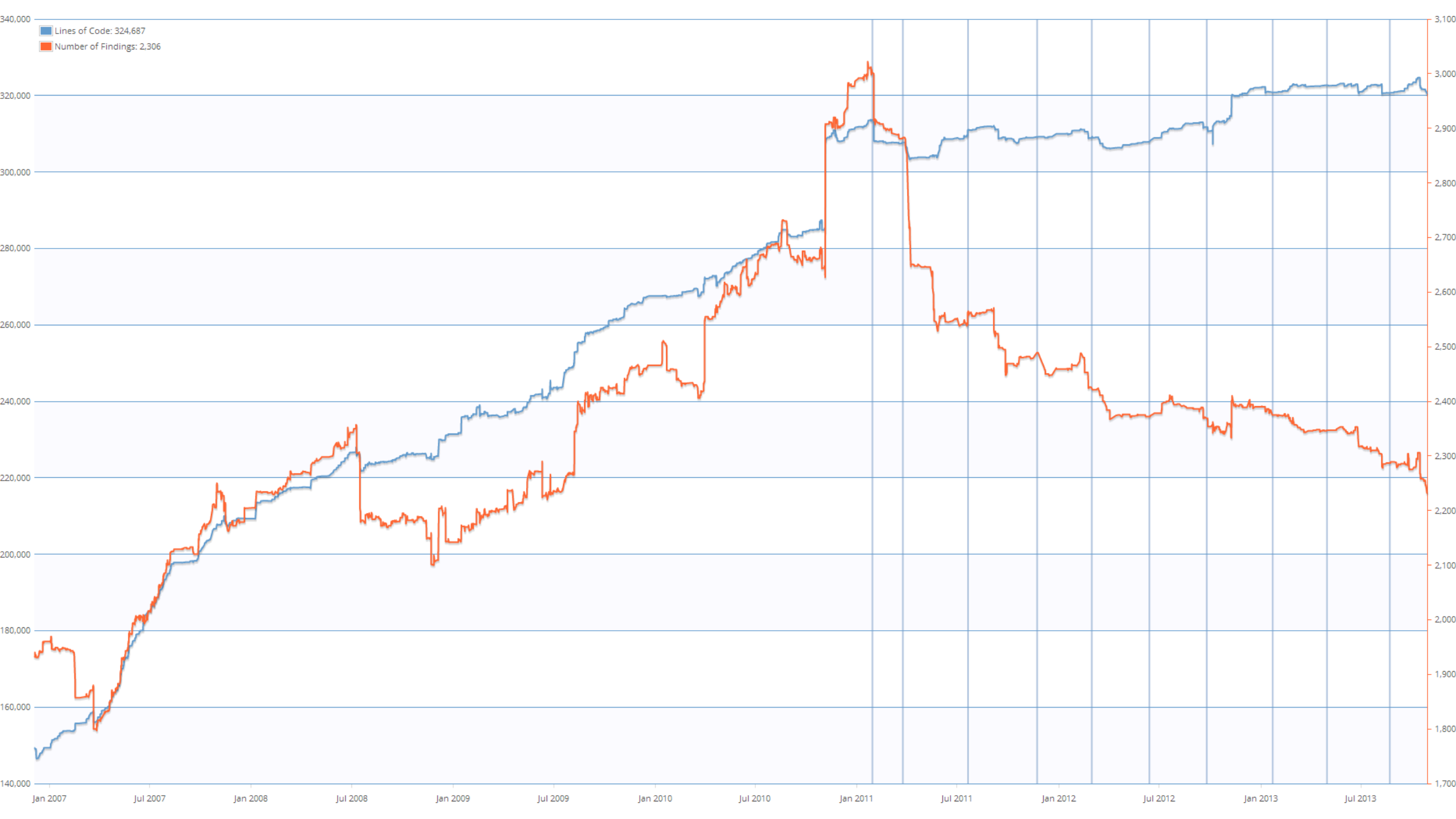
You replied to this message on 30.10.2019 15:06.

Hi all,

the findings down below has been fixed, next assessment status should be green again.

Best regards

[REDACTED]



**Wie können wir den  
Nutzen quantifizieren?**

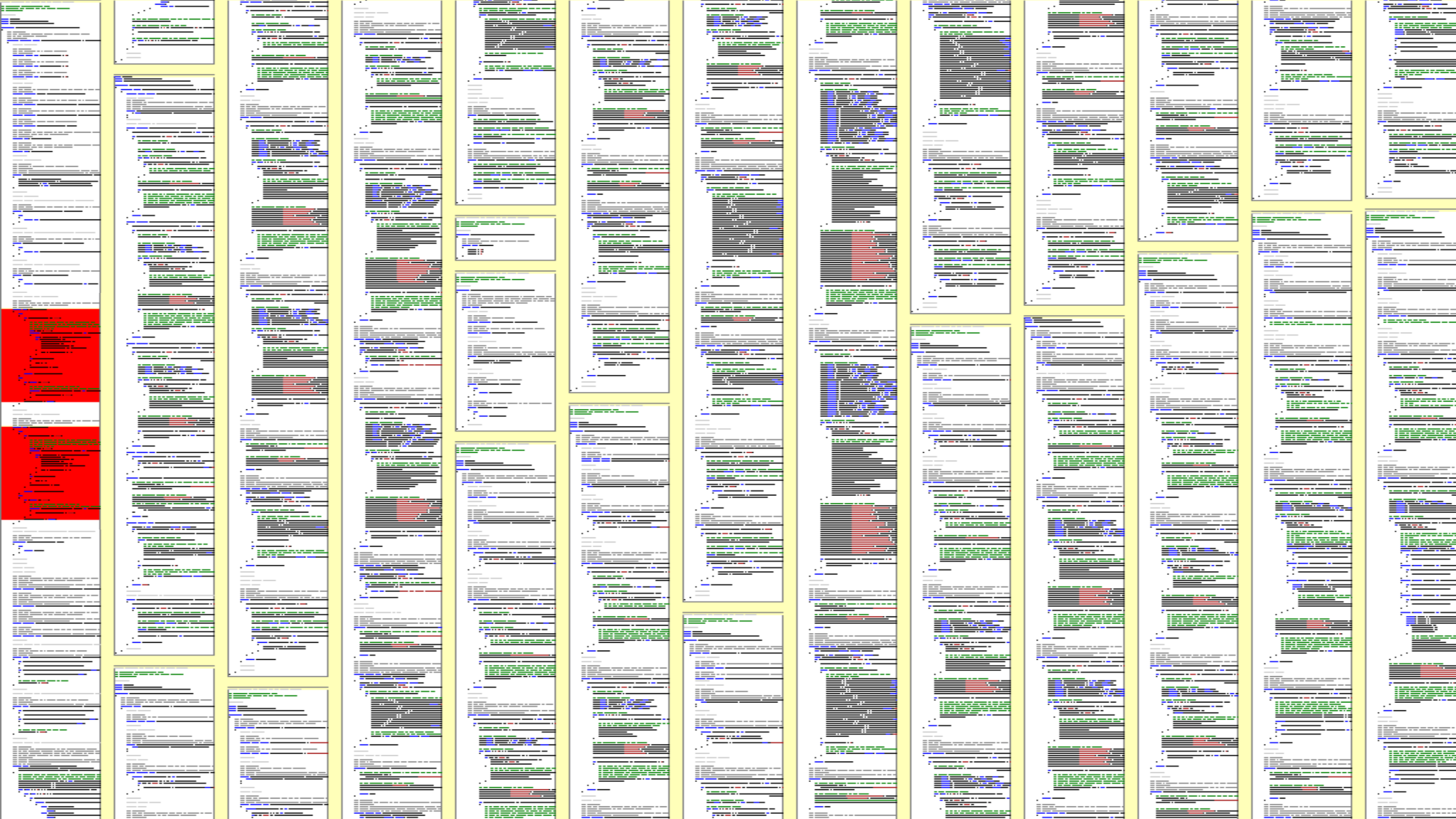
```
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
    boolean found = false;
    StringBuffer res = new StringBuffer();
    if (elements != null) {
        Index.getInstance().setCurrentRenderer(
            FlatReferenceRenderer.getInstance());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

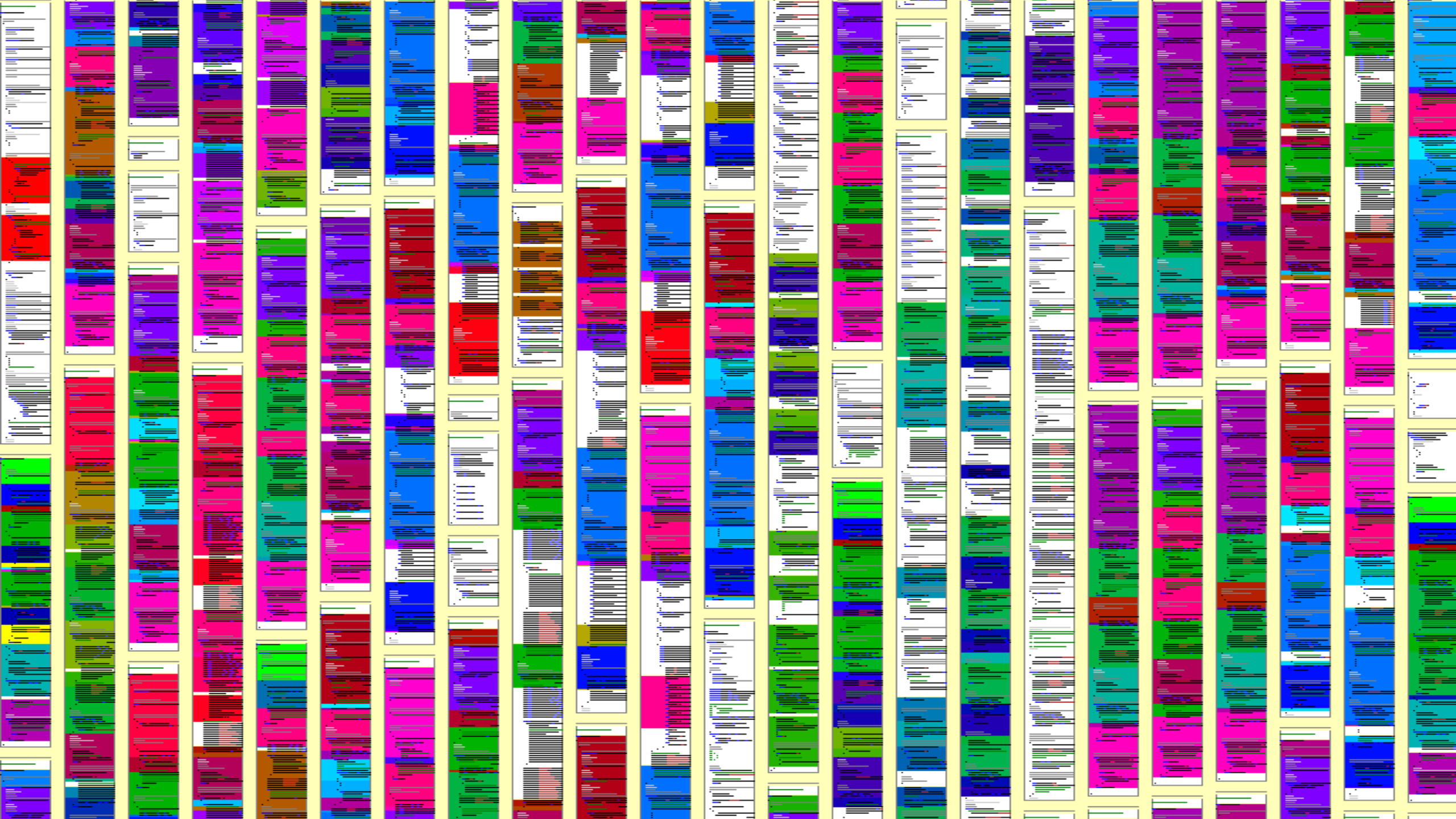


```
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
    boolean found = false;
    StringBuffer res = new StringBuffer();
    if (elements != null) {
        Index.getInstance().setCurrentRenderer(
            FlatReferenceRenderer.getInstance());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

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



# #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)

? PT

## Aufwand für Reparatur

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

? PT

# 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  $\frac{PT}{Jahr}$

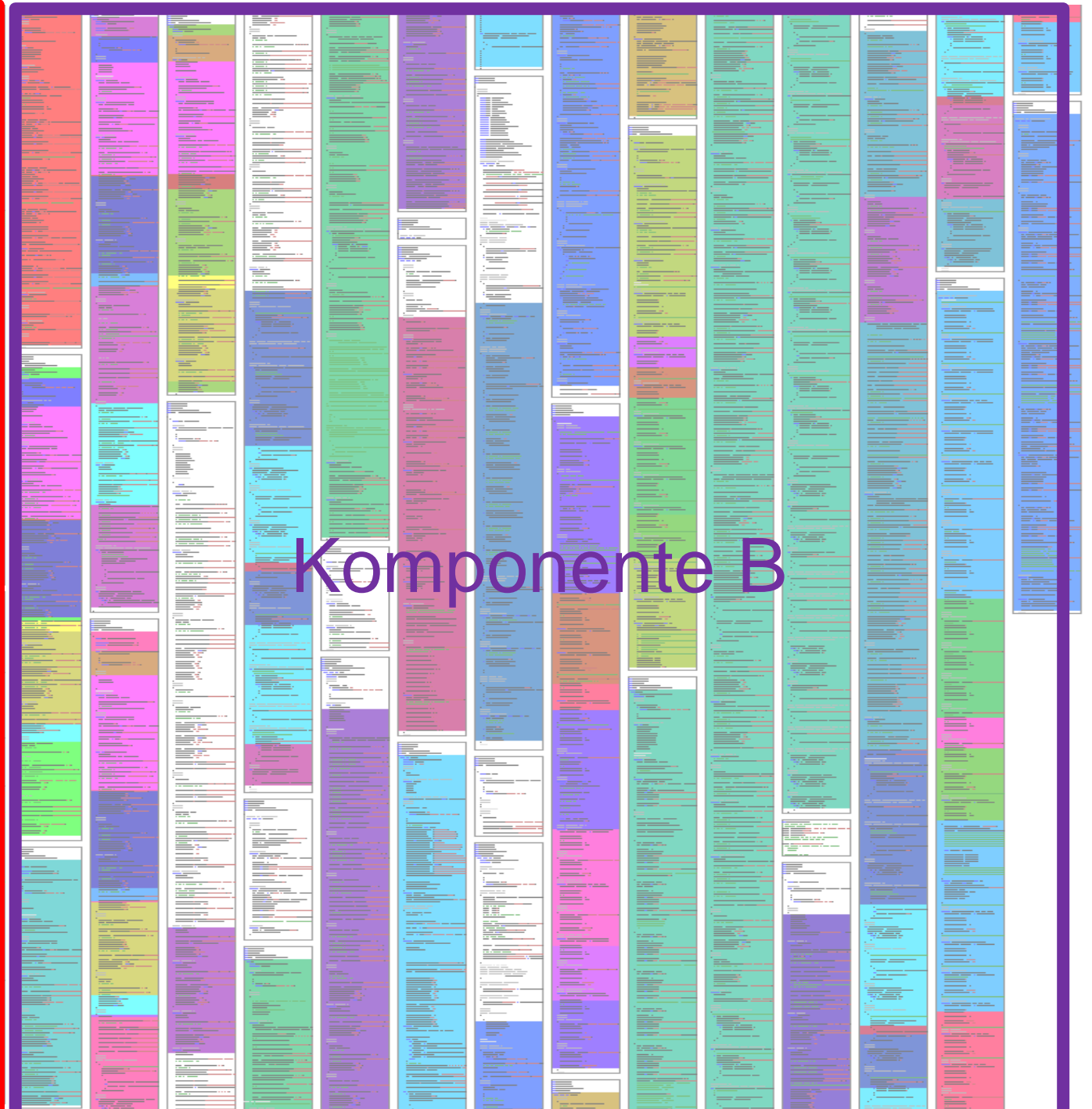
$$500 \frac{PT}{Jahr}$$

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



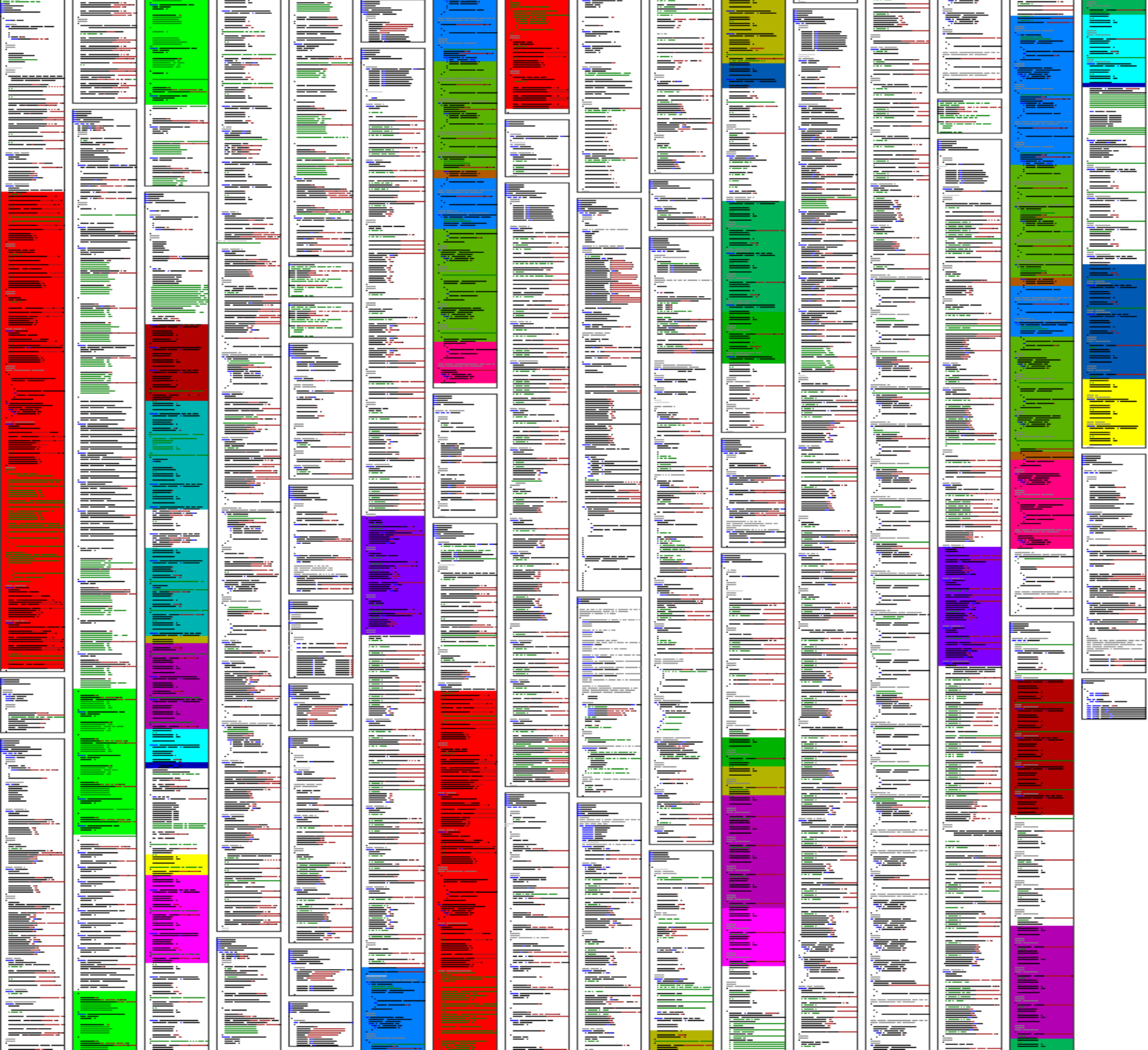


Komponente A



Komponente B







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

Blow-Up: 0%



20 Lines

20 Lines

Blow-Up: 50%



20 Lines

20 Lines

20 Lines

Project	BlowUp	SLOC
	1,54%	3658
	9,47%	8294
	14,77%	99852
	25,77%	33592
	14,21%	291411
	7,57%	33764
	44,78%	43092
	0,95%	12211
	20,90%	18910
	15,92%	6579
	22,49%	57939
	0,35%	7791
	4,62%	15942
	12,94%	14885
	4,19%	22239
	14,36%	139295
	2,60%	2046
	4,57%	2963
	15,11%	26633
	54,92%	3754
	11,01%	55582
	24,19%	14500
	15,72%	6496
	6,44%	48572
	1,67%	14759
	24,39%	51610
	1,62%	15432
	5,94%	27758

	27,07%	5171
	0,83%	12147
	9,01%	513884
	20,10%	15411
	21,91%	50805
	7,23%	58685
	3,47%	89744
	9,54%	14323
	20,07%	19498
	4,97%	10023
	7,65%	1033692
	6,10%	50753
	26,70%	133212
	1,79%	14098
	21,47%	25105
	27,31%	13304
	24,23%	21789
	10,68%	20748
	19,42%	138605
	18,99%	25826
	18,62%	216704
	3,37%	7060
	15,70%	230588
	27,02%	362546
	4,18%	269356
	3,51%	9449
	5,04%	48447
	22,34%	83027
	3,93%	192236

	25,26%	259587
	8,20%	8763
	5,41%	118943
	9,55%	337006
	7,62%	121976
	1,57%	34942
	10,28%	25523
	2,35%	66736
	17,35%	250101
	2,91%	76358
	7,48%	62847
	2,48%	16202
	0,65%	52059
	4,29%	123482
	10,24%	16220
	13,00%	212391
	8,00%	54948
	10,53%	243324
	4,04%	136083
	13,28%	112395
	1,09%	5326
	24,65%	16805
	11,49%	101958
	5,86%	34248
	17,18%	25810
	9,64%	908518
	1,10%	45147
	2,05%	5348

Project	BlowUp	SLOC
	1,54%	3658
	9,47%	8294
	14,77%	99852
	25,77%	33592
	14,21%	291411
	7,57%	33764
	44,78%	43092
	0,95%	12211
	20,90%	18910
	15,92%	6579
	22,49%	57939
	0,35%	7791
	4,62%	15942
	12,94%	14885
	4,19%	22239
	14,36%	139295
	2,60%	2046
	4,57%	2963
	15,11%	26633
	54,92%	3754
	11,01%	55582
	24,19%	14500
	15,72%	6496
	6,44%	48572
	1,67%	14759
	24,39%	51610
	1,62%	15432
	5,94%	27758

	27,07%	5171
	0,83%	12147
	9,01%	513884
	20,10%	15411
	21,91%	50805
	7,23%	58685
	3,47%	89744
	9,54%	14323
	20,07%	19498
	4,97%	10023
	7,65%	1033692
	6,10%	50753
	26,70%	133212
	1,79%	14058
	21,47%	5105
	27,31%	13304
	24,23%	21789
	10,68%	20748
	19,42%	138605
	18,99%	25826
	18,62%	216704
	3,37%	7060
	15,70%	230588
	27,02%	362546
	4,18%	269356
	3,51%	9449
	5,04%	48447
	22,34%	83027
	3,93%	192236

Blow-Up: Ø12%

	25,26%	259587
	8,20%	8763
	5,41%	118943
	9,55%	337006
	7,62%	121976
	1,57%	34942
	10,28%	25523
	2,35%	66736
	17,35%	250101
	2,91%	76358
	7,48%	62847
	2,48%	16202
	0,65%	52059
	4,29%	123482
	10,24%	16220
	13,00%	212391
	8,00%	54948
	10,53%	243324
	4,04%	136083
	13,28%	112395
	1,09%	5326
	24,65%	16805
	11,49%	101958
	5,86%	34248
	17,18%	25810
	9,64%	908518
	1,10%	45147
	2,05%	5348

Project	BlowUp	SLOC
	1,54%	3658
	9,47%	8294
	14,77%	99852
	25,77%	33592
	14,21%	291411
	7,57%	33764
	44,78%	43092
	0,95%	12211
	20,90%	18910
	15,92%	6579
	22,49%	57939
	0,35%	7791
	4,62%	15942
	12,94%	14885
	4,19%	22239
	14,36%	139295
	2,60%	2046
	4,57%	2963
	15,11%	26633
	54,92%	3754
	11,01%	55582
	24,19%	14500
	15,72%	6496
	6,44%	48572
	1,67%	14759
	24,39%	51610
	1,62%	15432
	5,94%	27758

	27,07%	5171
	0,83%	12147
	9,01%	513884
	20,10%	15411
	21,91%	50805
	7,23%	58685
	3,47%	89744
	9,54%	14323
	20,07%	19498
	4,97%	10023
	7,65%	1033692
	4,10%	59359
	26,70%	133212
	1,79%	14098
	21,47%	25105
	27,11%	18601
	24,23%	21189
	10,68%	20748
	19,42%	138605
	18,99%	25826
	18,62%	216704
	3,37%	7060
	15,70%	230588
	27,02%	362546
	4,18%	269356
	3,51%	9449
	5,04%	48447
	22,34%	83027
	3,93%	192236

Blow-Up: Ø12%  
(Das ist wenig)

	25,26%	259587
	8,20%	8763
	5,41%	118943
	9,55%	337006
	7,62%	121976
	1,57%	34942
	10,28%	25523
	2,35%	66736
	17,35%	250101
	2,91%	76358
	7,48%	62847
	2,48%	16202
	0,65%	52059
	4,29%	123482
	10,24%	16220
	13,00%	212391
	8,00%	54948
	10,53%	243324
	4,04%	136083
	13,28%	112395
	1,09%	5326
	24,65%	16805
	11,49%	101958
	5,86%	34248
	17,18%	25810
	9,64%	908518
	1,10%	45147
	2,05%	5348



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

# %CloneAffectedEffort

## Aktivitäten

- Analysis
- Location
- Design
- Impact Analysis
- Implementation
- Quality Assurance
- Other

## Aufwändiger durch Cloning

- 
- Location
- 
- Impact Analysis
- Implementation
- Quality Assurance
- 

Detaillierte Herleitung und Berechnung im Paper.

Wert für Berechnung: **51%**.

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

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

**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 Pöhlmann\*, Robert Heinke<sup>†</sup>, Bärbel Uhink-Mergenthaler<sup>‡</sup>  
\* CQSE GmbH, Garching b. München, Germany  
<sup>†</sup> 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<sup>1</sup> 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<sup>2</sup> 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 (BMBWF), grant EcoCon, 01IS12034A. The responsibility for this article lies with the authors.

<sup>1</sup>e.g., file size, method length, or nesting depth

<sup>2</sup><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

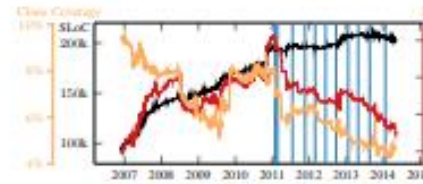


Fig. 3. System A

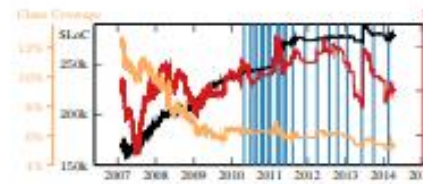


Fig. 4. System B

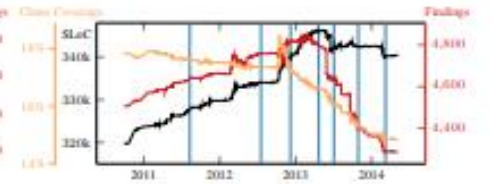


Fig. 5. System C

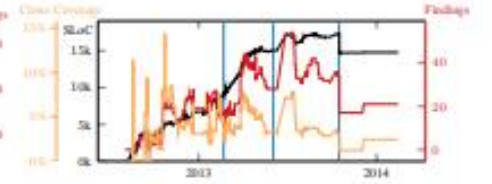


Fig. 6. System D

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 prove 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.

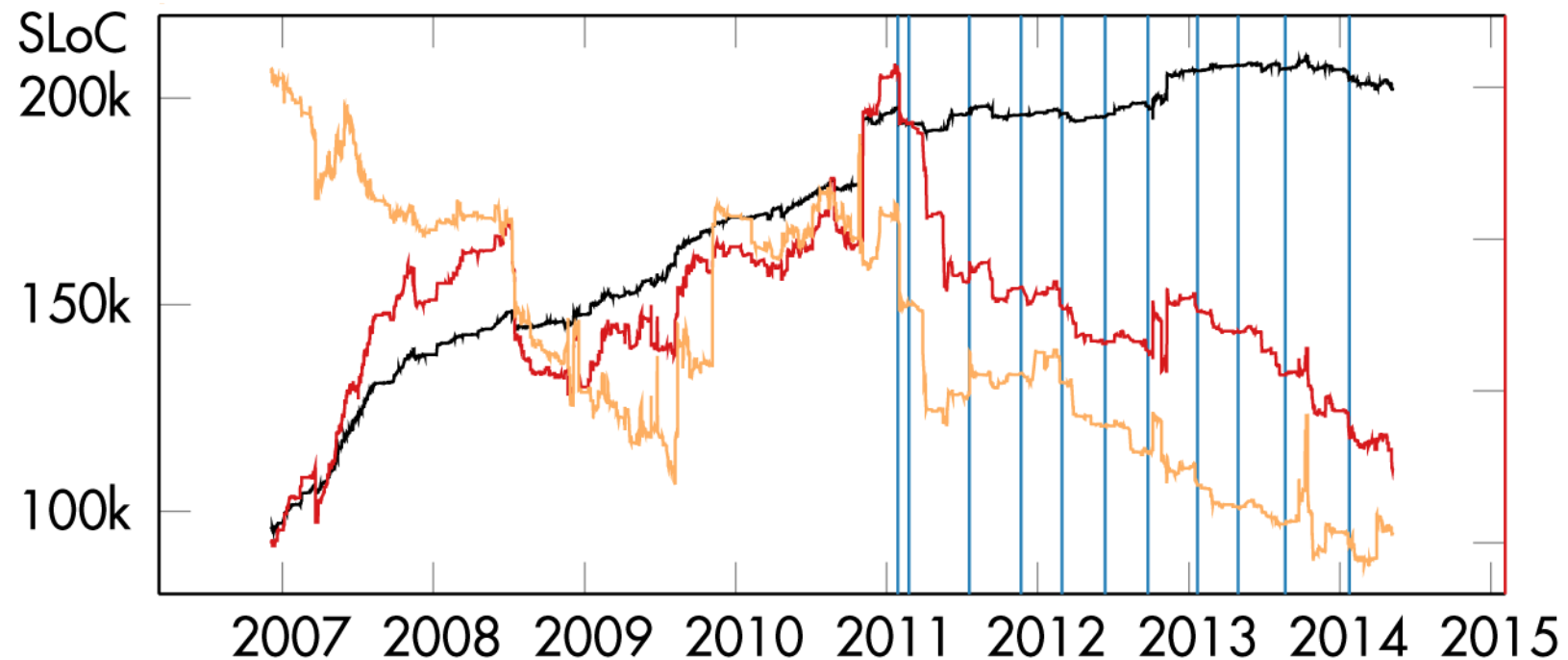
## V. CONCLUSION

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. Pirka, "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 backstair: An in-process software engineering meta-statement and analysis system," in *ESEM'07*.
- [5] F. Deissenboeck, M. Pirka, and T. Seifart, "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.

# 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

<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	47
<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

500  $\frac{PT}{Jahr}$

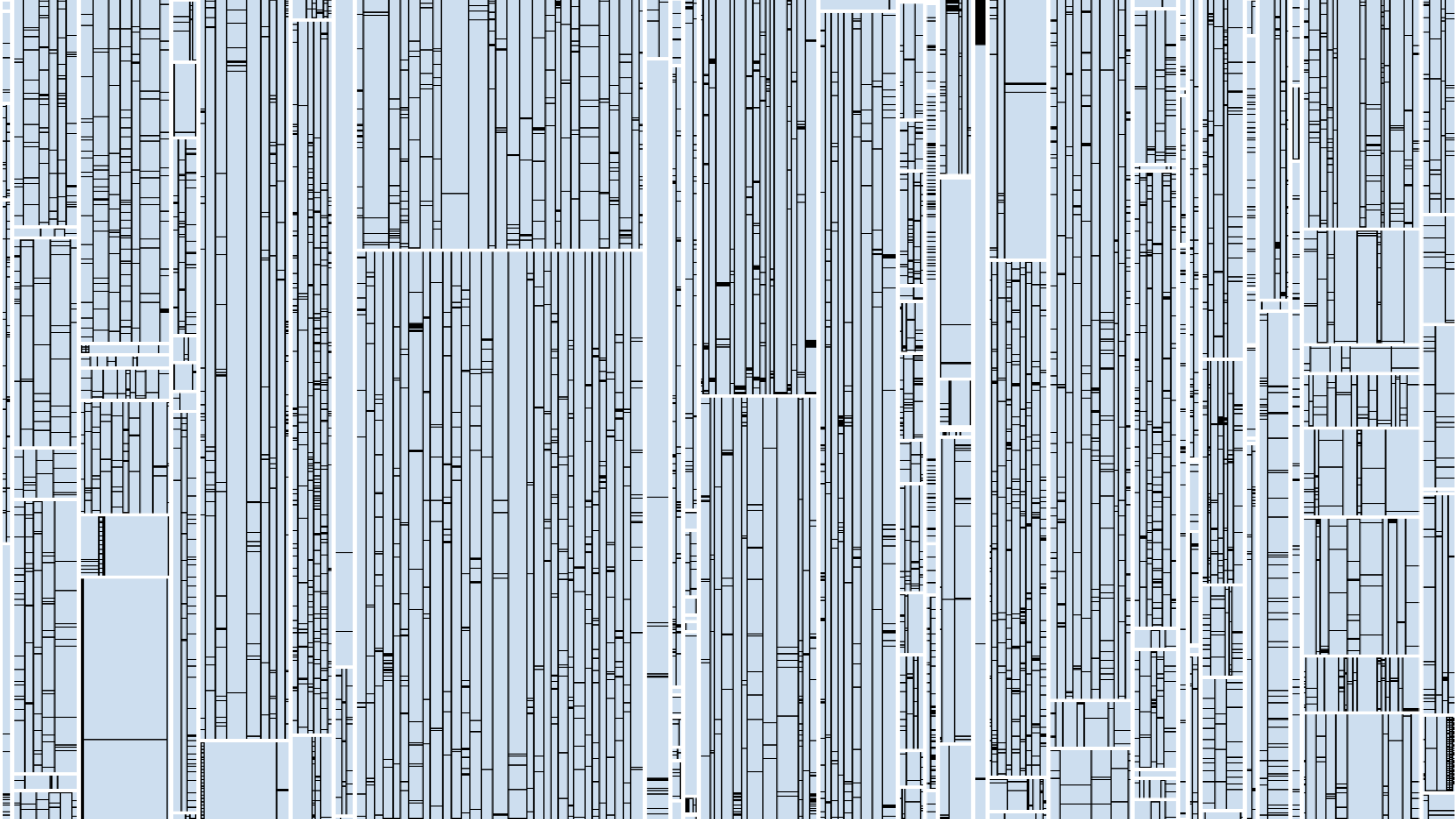
Munich Re spart durch Einsatz von Clone Detection jährlich ca. 500 PT Aufwand für Fehlerbehebung

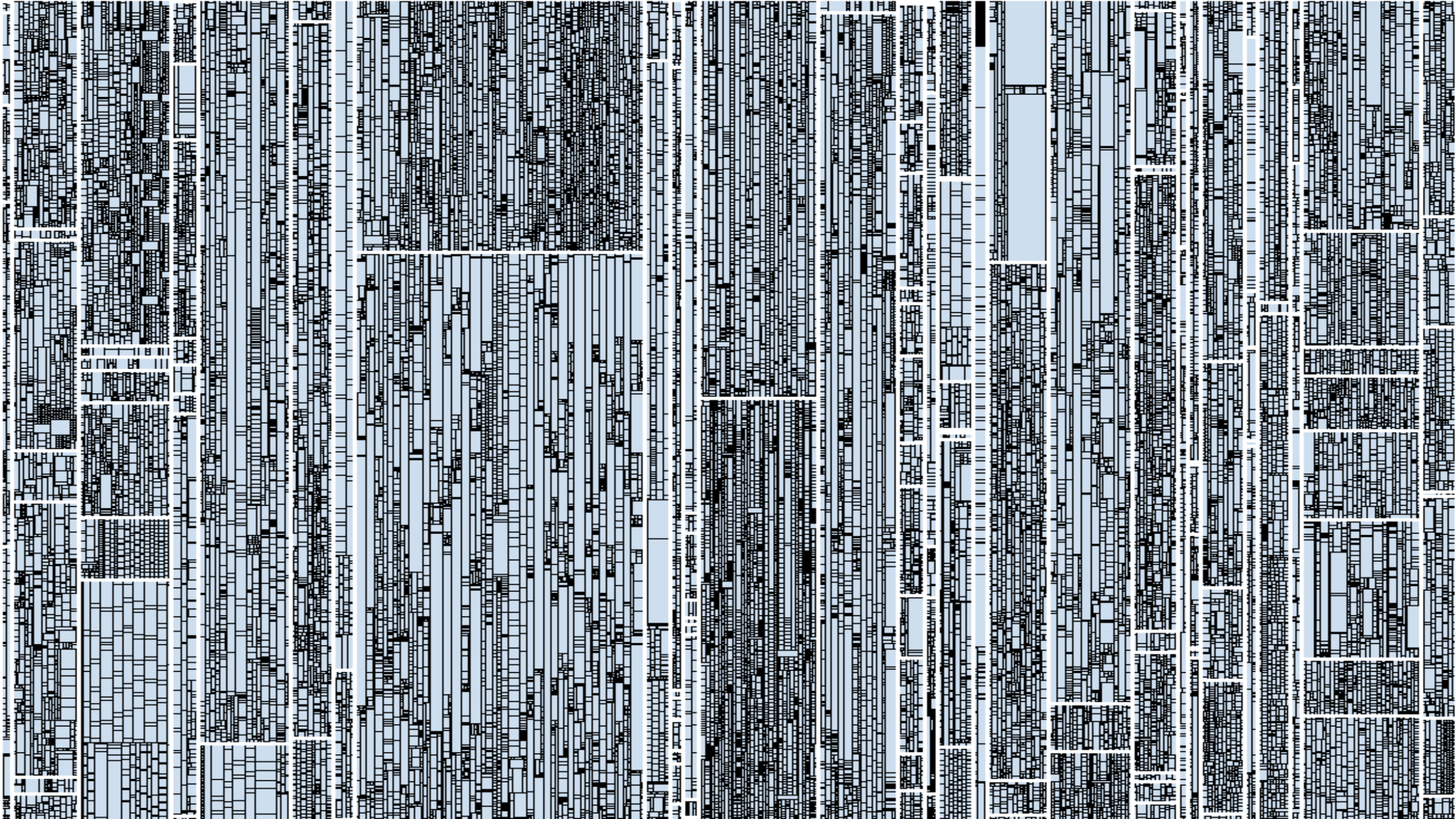
Ersparnis Aufwand = 6%

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

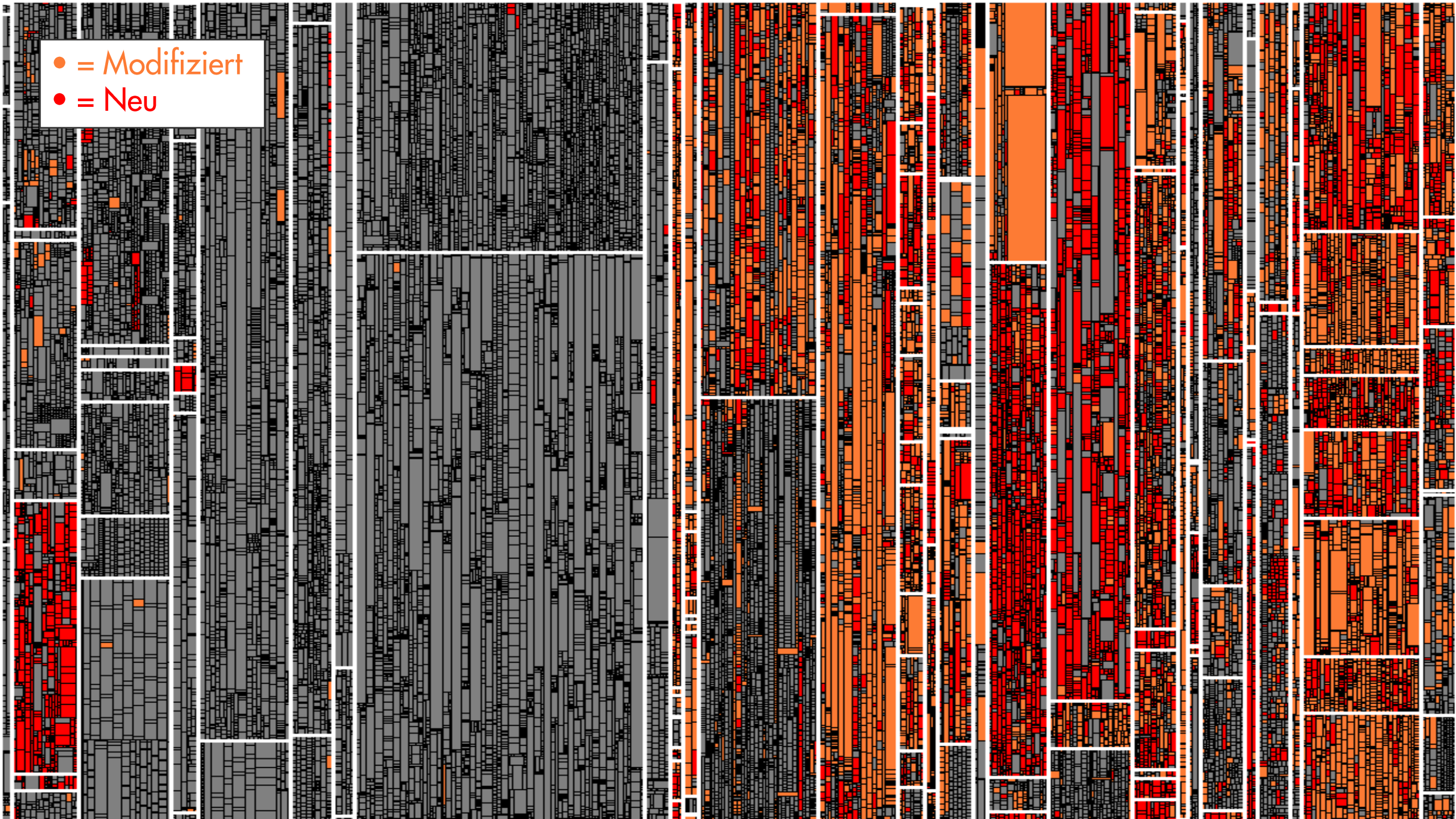
# Kosten-Nutzen von Test-Gap-Analyse







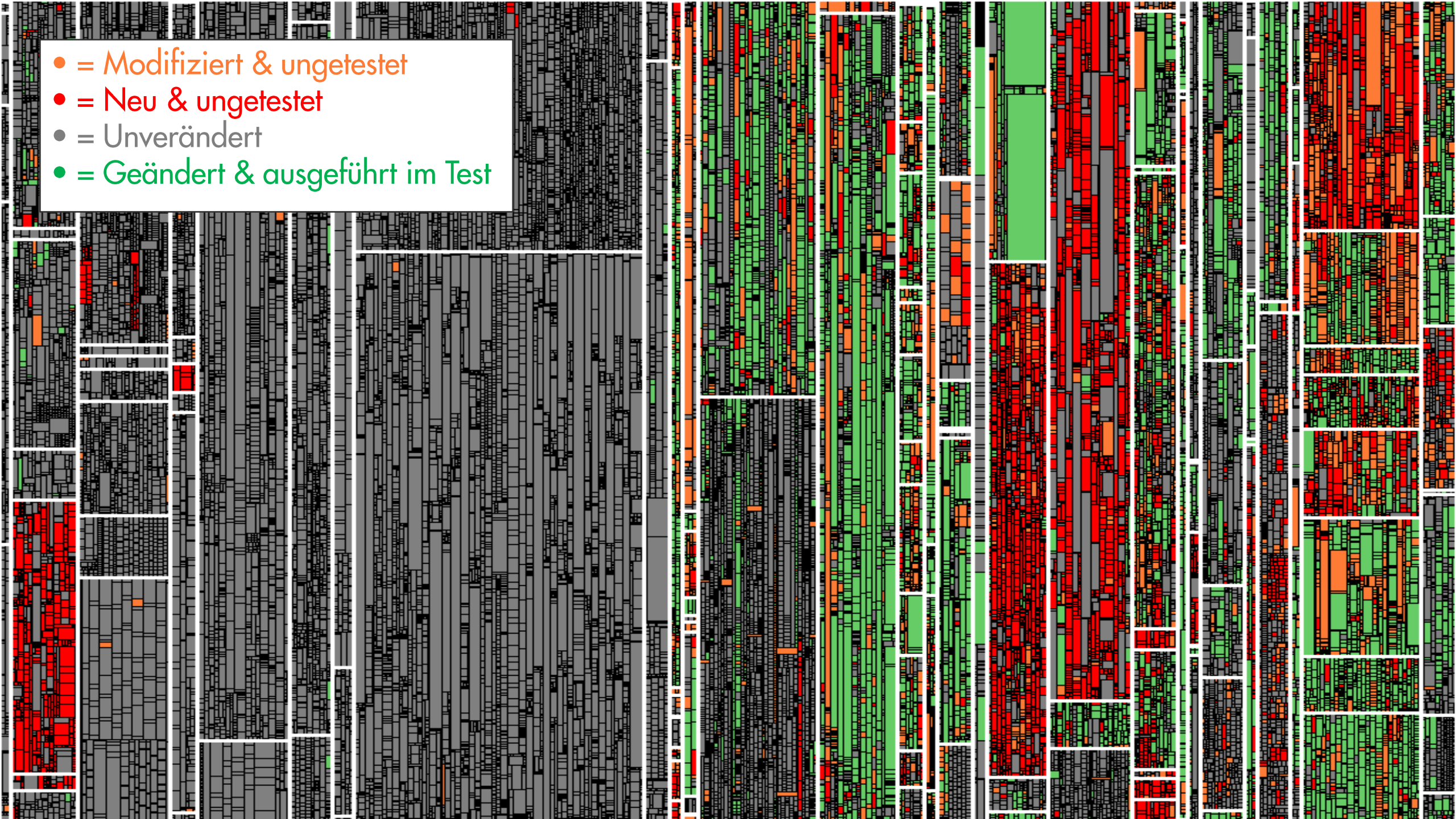
- = Modifiziert
- = Neu



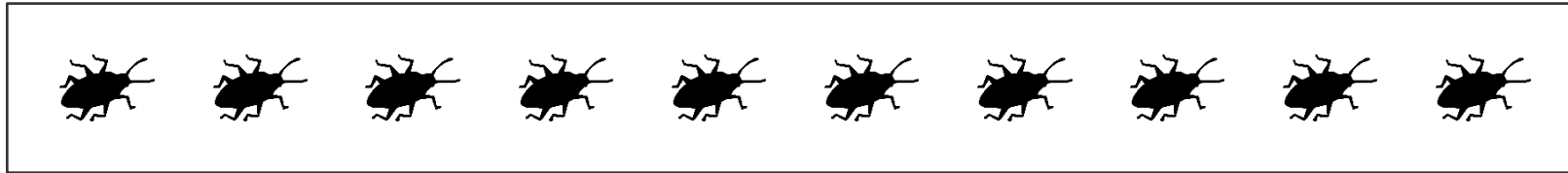
● = Getestet

Manual & automated Tests

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





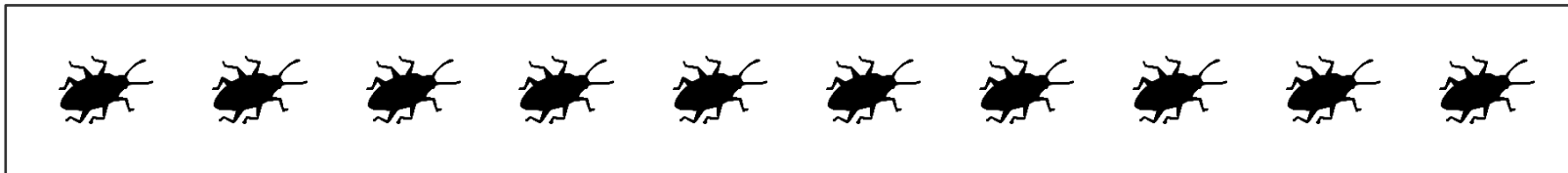


Test

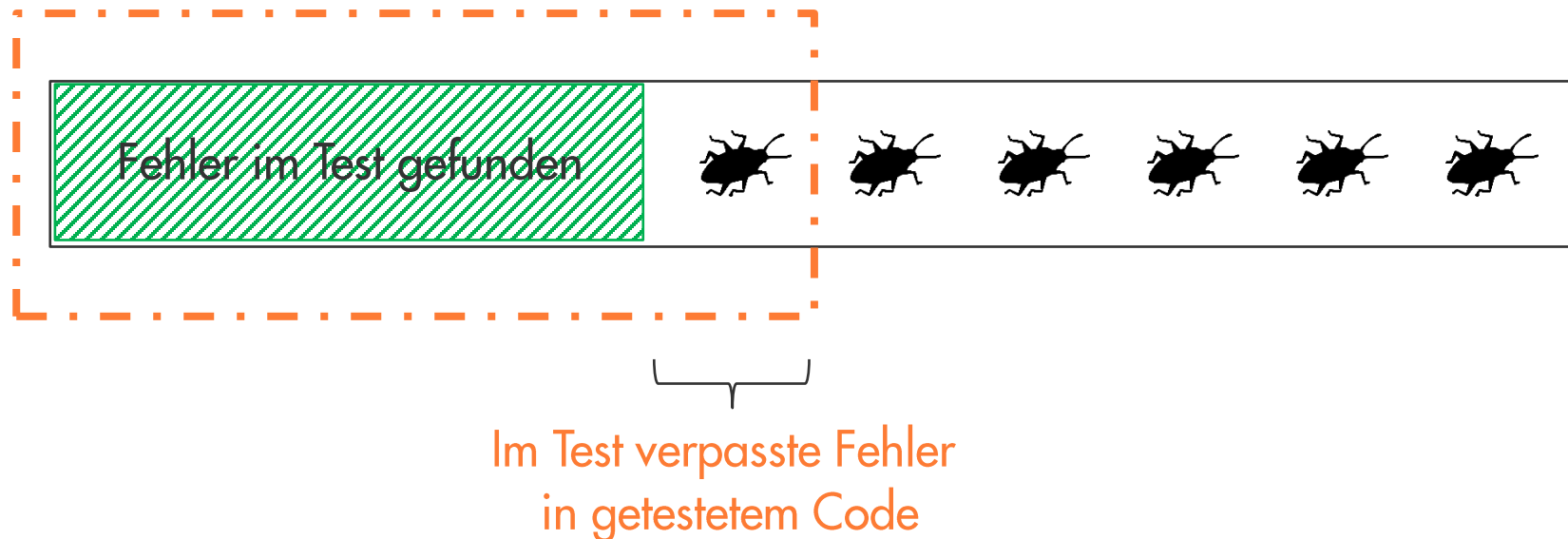


%Restfehler

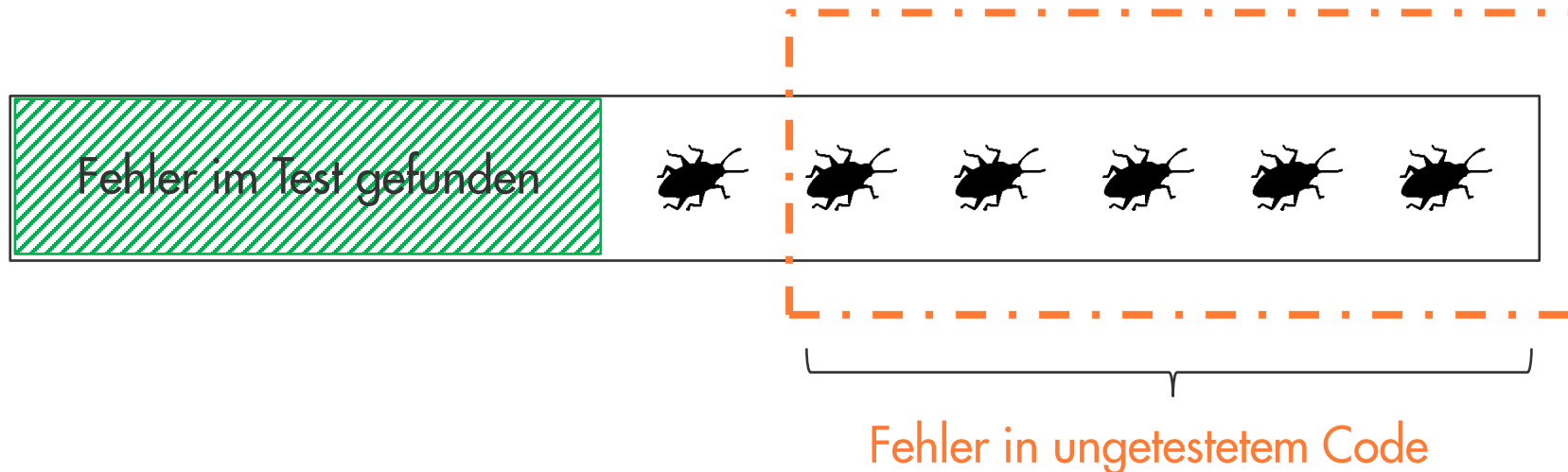
$$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivität} + \% \text{Testgap}$$



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



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



# 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 "TivCon, 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

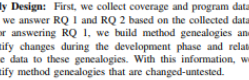
We perform the study on a business information system at Munich Re. The analyzed system was written in C# and its size are 340 KLOC. In total, we analyzed the system for 14 months. The system has been successfully in use for nine years and is still actively used and maintained. Therefore, there is a 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.



**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 efforts 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

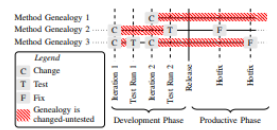


Fig. 1. Development life-cycle

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 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] J. Graves, A. Karr, J. Marion, and H. Sny, "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 *ICSE*, 2004.
- [5] H. Hall, S. Boehm, D. Bovee, 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 as defect predictors," in *ICSE*, 2010.
- [7] V. Chamaklavala, V. K. Shabbag, A. Patgabri, R. Soodia, and S. Lakshmanan, "Safe sub-regression test selection for managed code," in *ICSE*, 2008.
- [8] Y.-F. Chen, D. Rosenblum, and K.-P. Vo, "Testcase: a system for selective regression testing," in *ICSE*, 1994.
- [9] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothmel, "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 *FSTT*, 2002.
- [12] H. Zou, 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. Malaya, M. Li, J. Bierman, and R. Karach, "Software reliability growth with test coverage," *IEEE Trans. Rel.*, vol. 51, no. 4, 2002.
- [15] G. Rothmel, R. Linck, 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. Prossner, "How much does untested 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., 2010.
- [19] E. Juergens, M. Fellman, M. Hermannsdorfer, F. Deisselboeck, R. Vass, and K. Prossner, "Feature profiling for evolving systems," in *ICPC*, 2011.

# Wieviele Änderungen sind ungetestet?

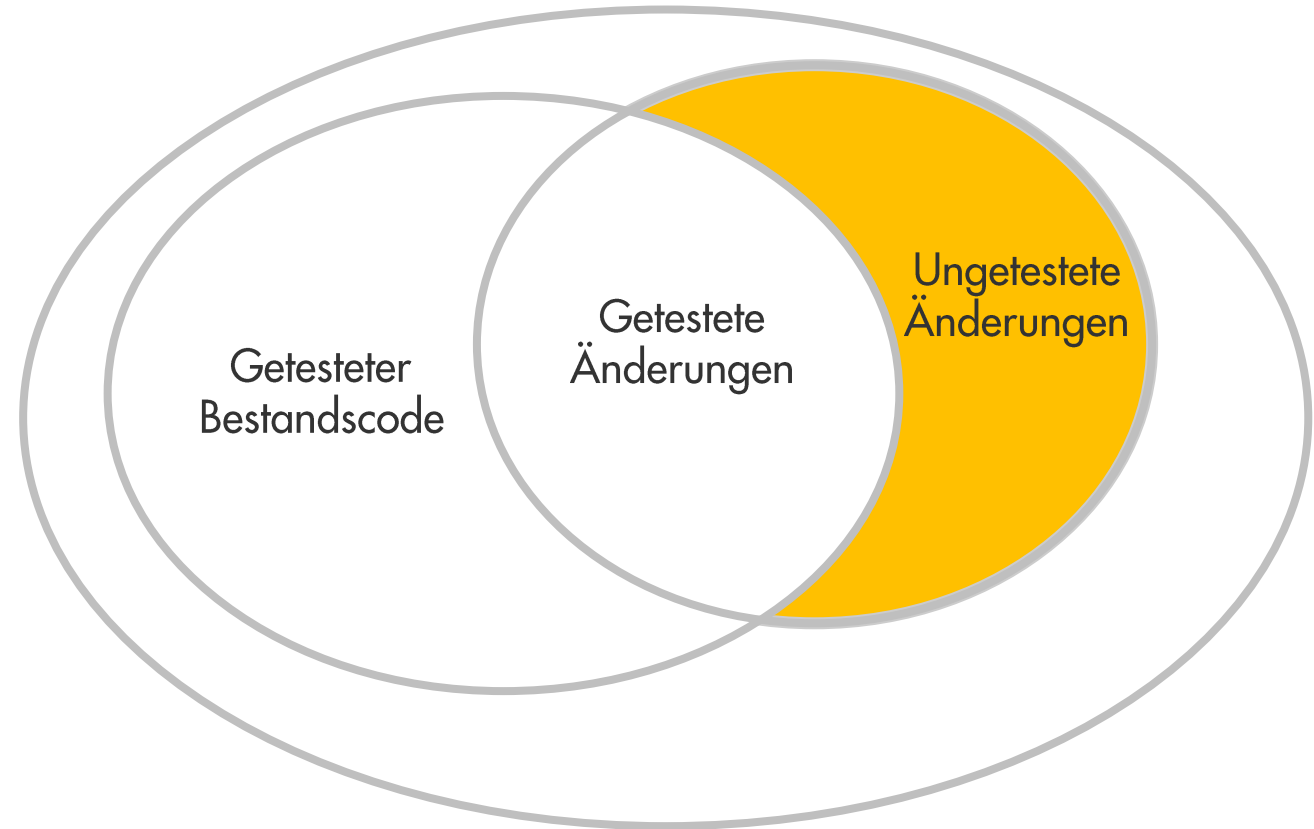
Studie: C# System @ Munich Re

## Release A:

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

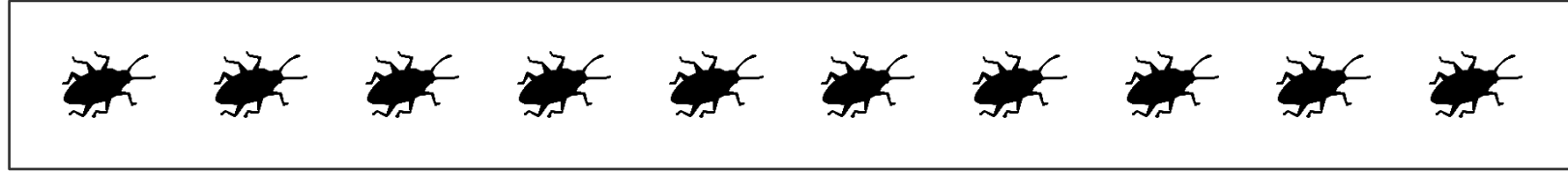
## Release B:

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



**Feldfehlerwahrscheinlichkeit 5x höher für ungetestete Änderungen!**

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

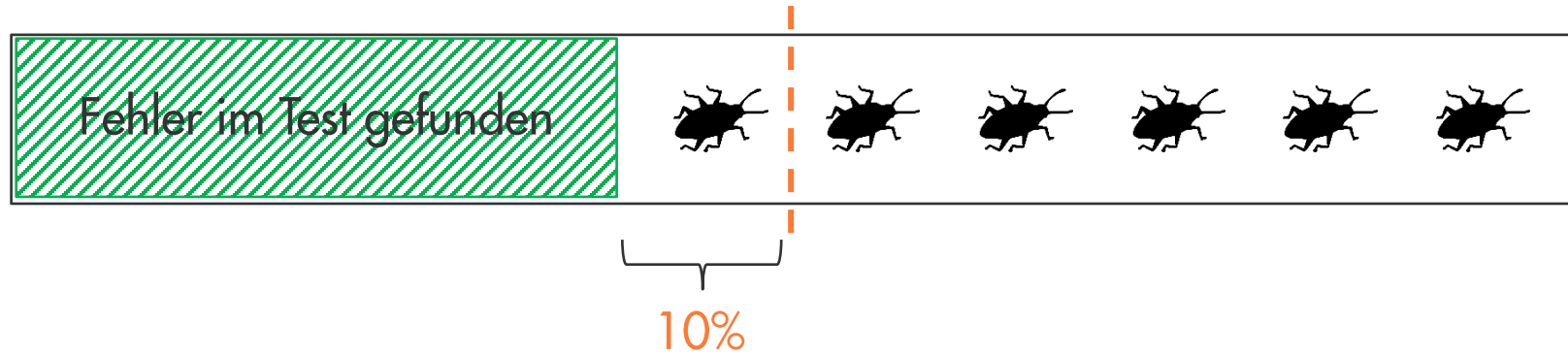


$$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivität} + \% \text{Testgap}$$

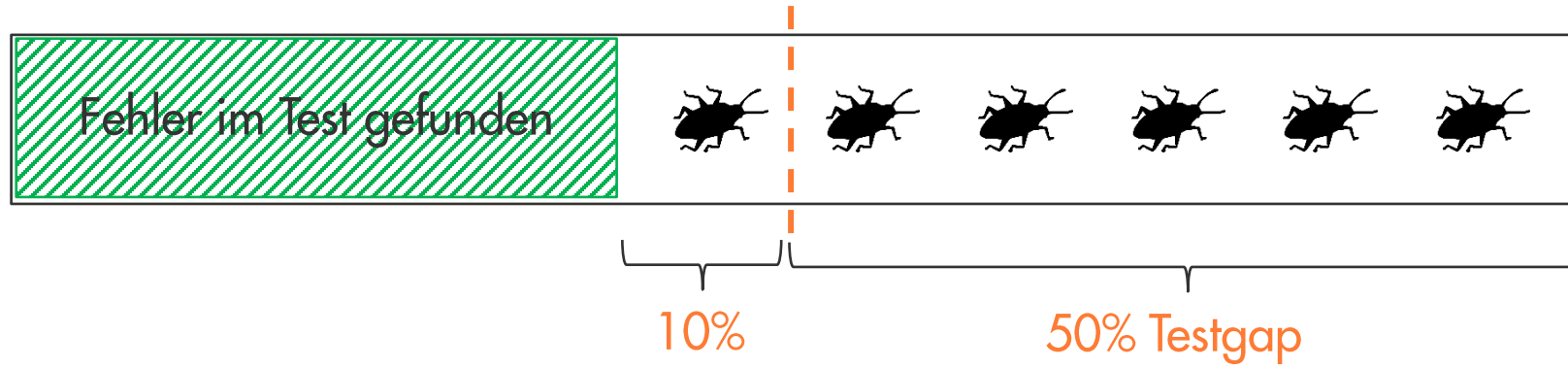




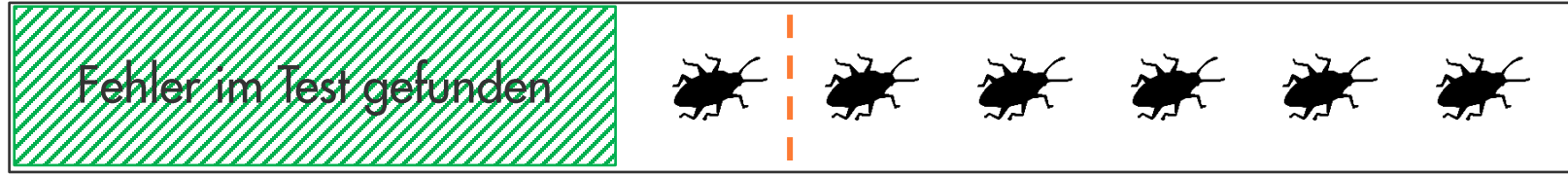
$$\% \text{Restfehler} = 10\% + \% \text{Testgap}$$



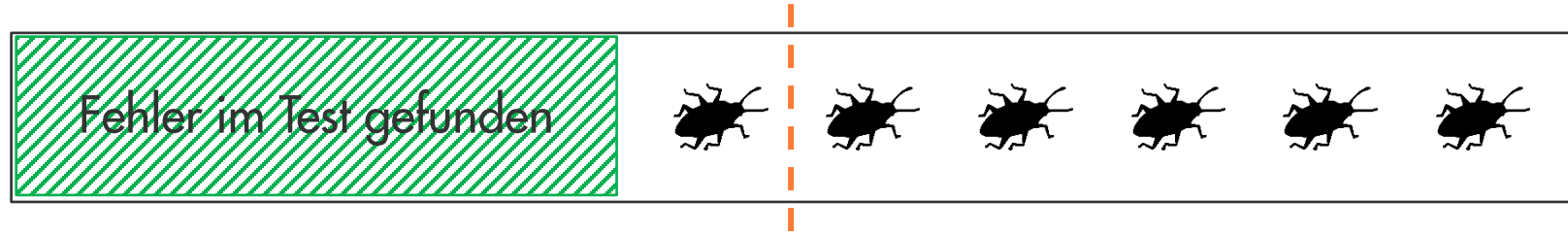
$$\% \text{Restfehler} = 10\% + 50\%$$



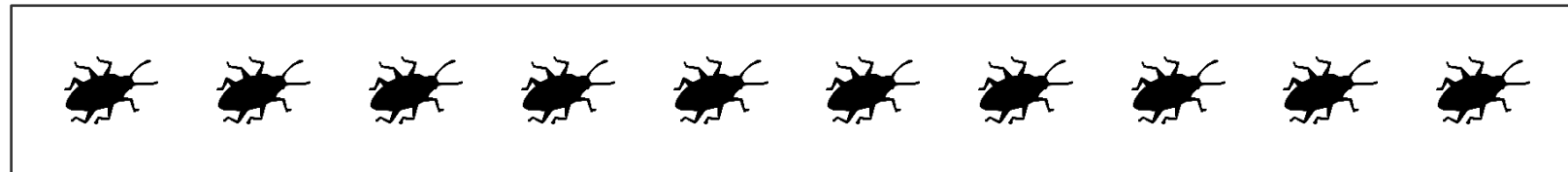
%Restfehler = 60%



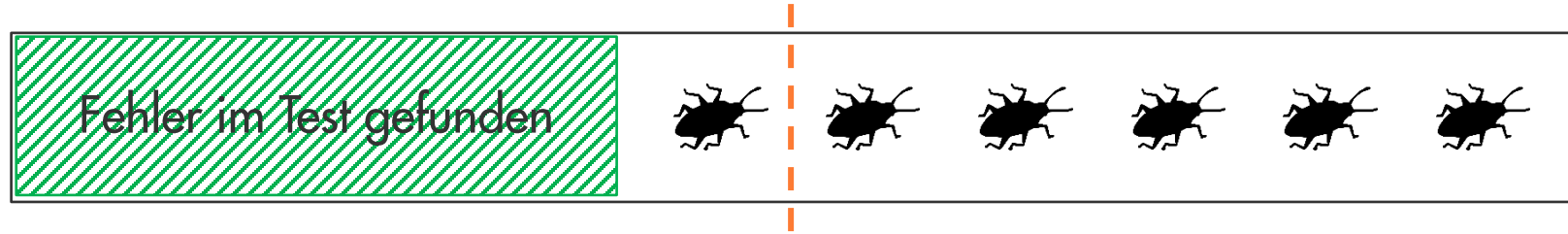
%Restfehler = 60%



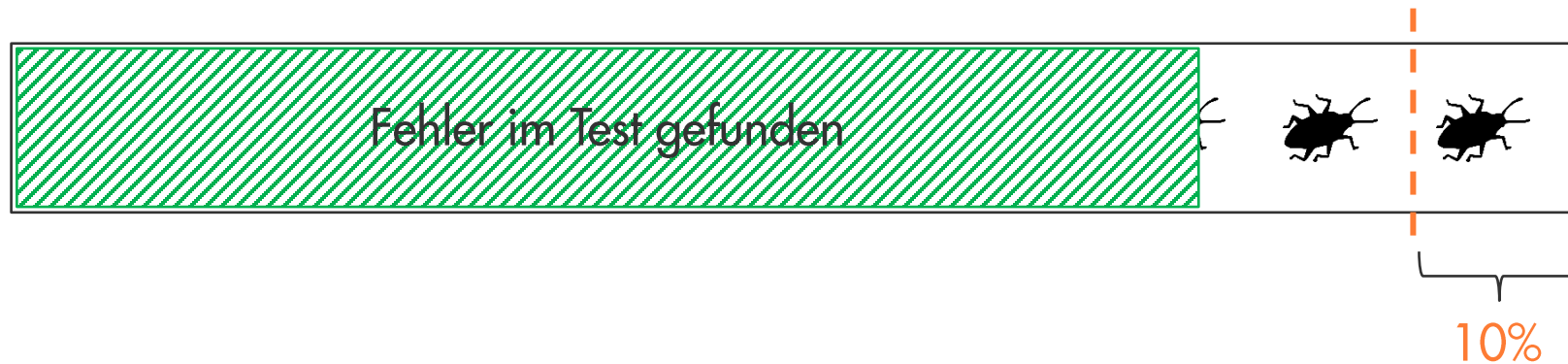
%Restfehler = %Getestet \* Testineffektivität + %Testgap



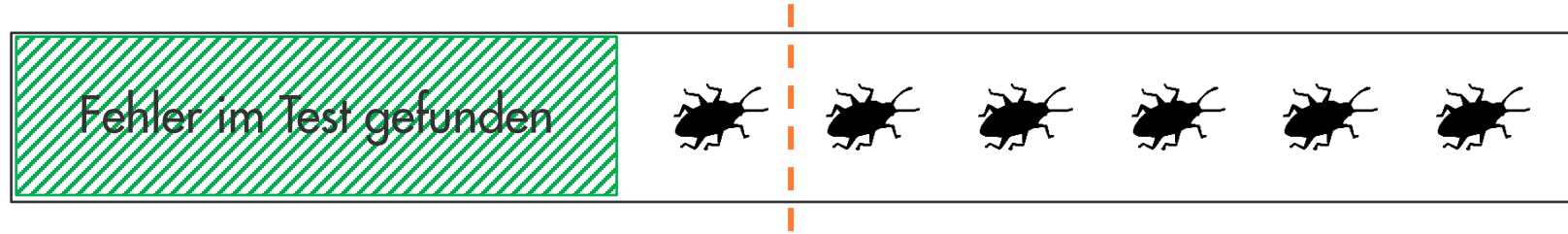
$$\% \text{Restfehler} = 60\%$$



$$\% \text{Restfehler} = 90\% * 20\% + 10\%$$



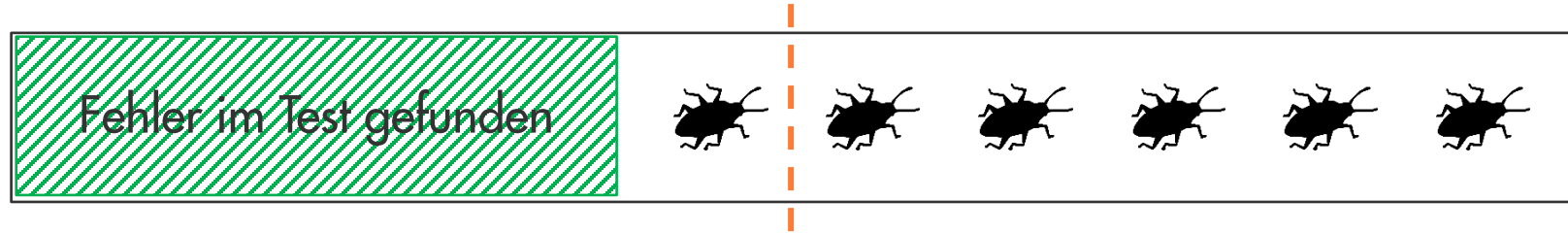
$\% \text{Restfehler} = 60\%$



$\% \text{Restfehler} = 18\% + 10\%$



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



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



Reduzierte Feldfehler = **50%**



Reduzierte Feldfehler = **50%**

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

# 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.

# Vielen Dank für Ihre Teilnahme!

**Feedback** zum  
Software Intelligence Talk 2020-1



[cqse.eu/si-talks/1/feedback](https://cqse.eu/si-talks/1/feedback)

**Anmeldung** zum  
Software Intelligence Talk 2021-2  
(Passwort: 20210519\_SI)



[cqse.eu/si-talks/2](https://cqse.eu/si-talks/2)