FSE:G:Detecting and Preventing the Architectural Roots of Bugs

Lu Xiao
Drexel University
Philadelphia, PA, USA
lx52@drexel.edu

ABSTRACT

Existing research has shown that files that were buggy in the past are also likely to be buggy in the future, implying that buggy files are seldom truly fixed. Our research has revealed that, in most cases, buggy files seldom exist alone; hundreds of buggy files can be connected by just a few architectural "roots", created by implementation flaws that introduce unhealthy relations between files. These relations in fact *cause* groups of files to be buggy. Hence, *it is impossible to reduce error or change rates without first identifying and then fixing the flaws that cause errors to propagate.* Our objective is to locate these problematic files, and to identify the design flaws in them that cause bugs to arise, propagate, and persist. We contribute a novel architecture model and a *root* detection algorithm for this purpose. We have validated our approach on dozens of open-source and commercial projects. Despite their widely varying nature, the detected roots have been shown to have significant and persistent impact on software maintenance costs. If not fixed, these roots accumulate maintenance costs just as debts accumulate interest. Our approach has already been adopted in 4 major software organizations. Instead of examining hundreds of defective files in isolation, our collaborators now only need to examine a few architecture roots, fixing numerous defects simultaneously by removing their structural flaws, thus providing substantial long-term savings in maintenance costs.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design

Keywords

Software Architecture, Software Quality, Architecture Recovery

1. PROBLEM AND MOTIVATION

To help improve software quality, a great deal of effort has been spent to locate and predict buggy files. Researchers have found that files that have had many bugs in the past likely to be buggy in the future [4, 7, 13]. The predictive power of such history information implies a significant problem: *buggy files are seldom fixed completely*, otherwise history would not predict the future effectively.

Our recent research [8] has revealed the strong correlation between software structure and error/change proneness: although a software system may have hundreds of buggy files, these files always form just a few structurally connected groups that we call *architecture roots.* These roots exhibit design flaws that cause errors to propagate among source files. When there are flaws in the software structure (for example, cyclic dependencies, improper inheritance, modularity violations, and unstable interfaces) the files connected by these flawed structures are likely to be buggy [6, 8], and to remain buggy over the history of the project. This is not surprising. If, for example, a designer introduces a base class or a key interface that is not implemented properly, that is, it is subject to frequent changes or it is error-prone, then all the files depending on it are highly likely to be problematic as well. The more files that depend on this key interface, the more severe the problem.

These kinds of design flaws are easy for programmers to introduce—in fact, we conjecture that they are ubiquitous in software development—and their effects are both damaging and long-lasting [6, 8, 10] . We have documented these phenomena in dozens of large-scale projects, both open source and industrial, and they appear in projects of all domains, ages, and programming languages [6,8,10,15]. The problem is clear: *it is impossible to reduce error or change rates without first fixing the design flaws that cause errors to propagate.*

Our research objective is therefore to locate the *roots of bugginess*, and to identify the design flaws that cause bugs to arise, propagate, and persist. This will further enable us to rank the severity of roots by characterizing their consequences in terms of software quality and productivity. Our ultimate objective is to suggest effective refactoring solutions and reduce maintenance costs through strategic design improvement.

2. BACKGROUND AND RELATED WORK

There have been a large number of studies of bug localization and prediction in software systems, largely in three categories: 1) predicting defects using static code metrics [12]; 2) predicting defects using history data mined from software repositories [11] and 3) predicting defects using a mixture of predictors from both code metrics and history information [14]. Their objective is to prioritize and facilitate software testing and debugging. What has not been fully explored is how architectural connections among high-maintenance files contribute to their error-proneness, and how to reduce the overall bugginess through architecture improvement.

In the field of software architecture, various models and methods are proposed to describe and recover software architecture for the sake of understandability [3, 9]. Nevertheless, there is little work studying the direct relation between architecture structure and software quality, nor how to use architecture information to guide and facilitate maintenance activities.

Our work attempts to bridge the gap based on the concepts of design rules and modules proposed by Baldwin and Clark [1]. They proposed that a system is composed of *design rules*—architecturally important decisions, and *independent modules* defined and decoupled by design rules. Wong et. al [5]'s *design rule hierarchy algorithm* automatically clusters a system into design rules and modules based on the structure coupling of a system. Their work forms the background of our *design rule spaces* model introduced in the next section.

3. APPROACH AND UNIQUENESS

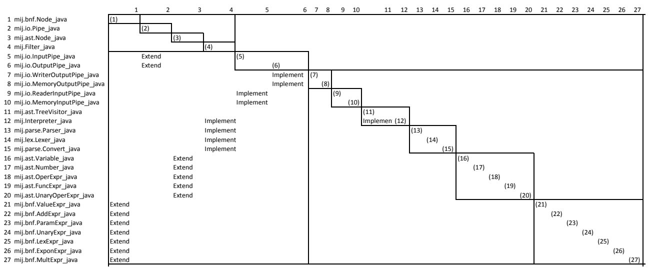
To detect architecture roots, we first propose a novel architecture model, call the *Design Rule Space* (DRSpace). We claim that any cluster of files, structurally connected–for example, to implement a system feature–can and should be viewed as a distinct design space. Accordingly, software architecture can be viewed as a set of overlapping design spaces. Furthermore, these connections are not just structural; we consider files that are *evolutionarily* coupled, that is, they have changed together frequently over the project's history, as another kind of design space that provides unique insights into maintenance effort. Our DRSpace model can be used to express all these views flexibly. And by calculating the interactions between various design spaces and recorded error-prone/change-prone files it is possible to reveal the flawed relations among these clusters of files, and hence to detect architecture roots.

3.1 A Novel Architecture Model

We use a simple calculator program, called `MIJ`, as a running example to illustrate the uniqueness and novelty of DRSpace models. `MIJ` has 34 files and applies multiple *design patterns*–widely used canonical solutions for recurring problems in software design, such as *Interpreter* and *Visitor*.

1. Separating files with different architectural importance. DRSpace modeling is unique first because it employs the design rule hierarchy algorithm to separate files with different architectural importance into *design rules* and *modules*. A small number of files–those that affect the implementation of other files–are the leading files (design rules) of a DRSpace. These are arranged at the top layers. Groups of files, led by (and dependent on) the design rules, form modules in lower layers.

Figure 1 depicts one `MIJ` DRSpace expressed as a design structure matrix (DSM) [1]. Each cell presents the dependency relation between two files. For example, the cell in row 5, column 2 (cell[r5,c2]), contains "Extend", indicating that the file in row 5 `InputPipe.java` extends the base class file in row 2, `Pipe.java`. The leading files are in the first layer of the hierarchy (files 1 to 4). In the third layer, files 7 to 27, following different leading files, are decoupled into 6 mutually independent *modules*.



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	mij.bnf.Node_java	(1)																										
2	mij.io.Pipe_java		(2)																									
3	mij.ast.Node_java			(3)																								
4	mij.Filter_java				(4)																							
5	mij.io.InputPipe_java		Extend			(5)																						
6	mij.io.OutputPipe_java		Extend				(6)																					
7	mij.io.WriterOutputPipe_java						Implement	(7)																				
8	mij.io.MemoryOutputPipe_java						Implement		(8)																			
9	mij.io.ReaderInputPipe_java					Implement				(9)																		
10	mij.io.MemoryInputPipe_java					Implement					(10)																	
11	mij.ast.TreeVisitor_java											(11)																
12	mij.Interpreter_java				Implement							Implemen	(12)															
13	mij.parse.Parser_java				Implement									(13)														
14	mij.lex.Lexer_java				Implement										(14)													
15	mij.parse.Convert_java				Implement											(15)												
16	mij.ast.Variable_java			Extend													(16)											
17	mij.ast.Number_java			Extend														(17)										
18	mij.ast.OperExpr_java			Extend															(18)									
19	mij.ast.FuncExpr_java			Extend																(19)								
20	mij.ast.UnaryOperExpr_java			Extend																	(20)							
21	mij.bnf.ValueExpr_java	Extend																				(21)						
22	mij.bnf.AddExpr_java	Extend																					(22)					
23	mij.bnf.ParamExpr_java	Extend																						(23)				
24	mij.bnf.UnaryExpr_java	Extend																							(24)			
25	mij.bnf.LexExpr_java	Extend																								(25)		
26	mij.bnf.ExponExpr_java	Extend																									(26)	
27	mij.bnf.MultExpr_java	Extend																										(27)

Figure 1: Inheritance DRSpace of MIJ

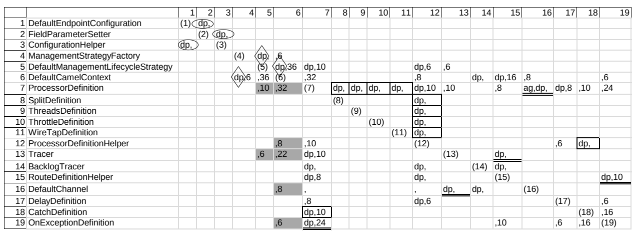
2. Viewing architecture as multiple, overlapping design spaces. Unlike existing work, which views an architectural structure in a single diagram, we view software architecture as multiple, overlapping design spaces. Figure 1 depicts one of many DRSpaces within the architecture formed by the inheritance relation. We have shown [8] that each type of structure relation, such as "call" or "aggregate", and each design pattern can form a meaningful but dramatically different design space.

3. Expressing structural and history dependences simultaneously. Another novel feature of DRSpace modeling is the simultaneous viewing of both structural and evolutionary relations. Figure 2 depicts a DRSpace extracted from the Apache Camel project. In this DSM, the cells with only a number, n, indicate that the two files changed together n times, but they have no structural relationship. A cell containing both letters and numbers, such as cell[r5, c6] with "dp, 36", means that these files have a structural relation and changed together. These numbers indicate the amount of effort spent on these files. This DRSpace is calculated using the architecture root detection algorithm, which we will introduce next.

3.2 Architecture Root Detection

To describe our algorithm and results we must first define several terms and measurements. First, a *Bug Space* is an ordered list of the project's buggy files, ordered by the number of times each file was changed in response to a bug. Then we define two parameters: *dsb* and *bsc* to reflect how error-prone a DRSpace is. *Design Space Bugginess(dsb)* is the percentage of files in a DRSpace that are also in a bug space at a given level of bugginess, N. *Bug Space Coverage(bsc)* (at bug level N) is the percentage of files in a project's bug space that are also in a given DRSpace.

Our algorithm takes a project's bug space and set of structural dependencies between files as inputs and outputs a minimal set of DRSpaces that contain the files in the given



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	DefaultEndpointConfiguration	(1)	dp,																	
2	FieldParameterSetter		(2)	dp,																
3	ConfigurationHelper	dp,		(3)																
4	ManagementStrategyFactory				(4)	dp,	,6													
5	DefaultManagementLifecycleStrategy					(5)	dp,36	dp,10					dp,6	,6						
6	DefaultCamelContext				dp,6	,36	(6)	,32					,8		dp,	dp,16	,8			,6
7	ProcessorDefinition					,10	,32	(7)	dp,	dp,	dp,	dp,	dp,10	,10		,8	ag,dp,	dp,8	,10	,24
8	SplitDefinition								(8)				dp,							
9	ThreadsDefinition									(9)			dp,							
10	ThrottleDefinition										(10)		dp,							
11	WireTapDefinition											(11)	dp,							
12	ProcessorDefinitionHelper						,8	,10					(12)					,6	dp,	
13	Tracer					,6	,22	dp,10						(13)		dp,				
14	BacklogTracer							dp,					dp,		(14)	dp,				
15	RouteDefinitionHelper							dp,8					dp,			(15)				dp,10
16	DefaultChannel						,8	,					,	dp,	dp,		(16)			
17	DelayDefinition							,8					dp,6					(17)		,6
18	CatchDefinition							dp,10											(18)	,16
19	OnExceptionDefinition						,6	dp,24								,10		,6	,16	(19)

Figure 2: Camel Root DRSpace Part1

bug space. The *dsb* and *bsc* of each DRSpace are computed. The output DRSpaces are the architecture roots that aggregate and propagate errors.

The DSM in Figure 2 is calculated by applying this algorithm to the Camel project. We observed numerous architecture issues in this DRSpace. We marked the files involved in these issues as follows: the cells marked with the oval, diamond, rectangle, and double underline shapes all form dependency cycles. For example, the diamond cells indicate a dependency cycle consisting of *ManagementStrategyFactory* $\rightarrow$ *DefaultManagementLifecycleStrategy* $\rightarrow$ *DefaultCamelContext* $\rightarrow$ *ManagementStrategyFactory*. In particular, *ProcessDefinition* participates in at least five dependency cycles. The history coupling shows that *ProcessDefinition* changes together with many other participants of these dependency cycles. For example, it changes with *OnExceptionDefinition*, *CatchDefinition*, *Tracer*, *RouteDefinitionHelper* and *DelayDefinition* 8 to 24 times. This shows that the cyclic dependencies are incurring high maintenance costs.

In Figure 2, we highlighted files that belong to different modules, and have no structural dependencies but which change together often, using a dark background. For example, *DefaultCamelContext* changed together with *ProcessorDefinition* 32 times and with *Tracer* 22 times. We call this phenomenon *modularity violations*, which usually indicates shared secrets between files, suggesting a need for better encapsulation of such shared concepts [15].

4. RESULTS AND CONTRIBUTIONS

We developed a tool chain Titan [16] which automates the creation, representation, manipulation, and analysis of DRSpaces, to support our approach. This section presents some of the results that we obtained using this tool chain.

4.1 Results

We have investigated 9 open source projects: Avro, Camel, Cassandra, CXF, HBase, Ivy, OpenJPA, PDFBox and Wicket, and one commercial project that must remain anonymous. These projects differ in application domain, length of history (from 9 months to 6 years, with 1480 to 27427 commits and 630 to 5032 bug tickets), number of snapshots (from 11 to 92 snapshots) and scale (from 137 to 3073 files). Our studies have shown: 1) The majority of buggy files in all projects were concentrated in at most 5 architecture roots over every snapshot; 2) In 8 out of 10 projects we studied, there are long-lived DRSpaces that are consistently the most error-prone, over multiple snapshots; 3) Long-lived architecture roots connect ever more buggy files over time; 4) Architecture roots contain architecture issues that are the likely root causes of bugginess.

1. High-maintenance files are strongly connected. Using Titan, we extracted the architecture roots of the top 30% buggiest files for each snapshot of each project. Table 1 shows a summary of the 10 projects. The column "#Top30%BugFiles" shows the average and standard deviation of the number of the top 30% buggiest files over all snapshots. The third column shows the average and standard deviation of the percentage of the top 30% buggiest files that are concentrated in just one architecture root over all snapshots. The forth and fifth columns show the same information as the third column, but for three and five DRSpaces respectively. This data implies: 1) the majority of buggy files are concentrated in just a few architecture roots, and 2) the above claim holds over different snapshots during the life cycle of a project. For example, column "5 DRSpace Cover" shows that five architecture roots concentrate from 52% to 88% of the top 30% buggy files with low variation over all snapshots of the projects.

Table 1: Top 30% Bug Coverage

Projects	#Top30%BugFiles		1 DRSpace Cover		3 DRSpace Cover		5 DRSpace Cover	
	avg	stdev	avg	stdev	avg	stdev	avg	stdev
Avro	34.2	4.2	64.00%	1.40%	83.00%	2.00%	88.00%	2.60%
Camel	207.2	19.8	56.00%	2.20%	82.00%	3.30%	87.00%	2.90%
Cassandra	139.9	12	53.00%	6.60%	67.00%	2.50%	71.00%	2.80%
CXF	541.7	69.7	41.00%	1.20%	61.00%	1.30%	65.00%	1.10%
HBase	177.3	95.9	43.00%	4.00%	58.00%	5.10%	65.00%	4.80%
Ivy	68.0	0	70.00%	0.70%	84.00%	0.70%	87.00%	0.70%
OpenJPA	776.5	20.6	33.00%	0.90%	49.00%	0.80%	52.00%	0.70%
PDFBox	98.3	2.7	62.00%	3.50%	80.00%	1.10%	84.00%	1.30%
Wicket	241.6	10	55.00%	14.60%	65.00%	9.00%	69.00%	7.40%
Commercial	60.0	16	35%	7.60%	61.00%	4.00%	70.00%	4.40%

2. The impact of architecture roots is persistent and significant. We have determined that not only are hundreds of buggy files concentrated in just a few architecture roots, but also that these roots persist over a project's lifetime, and hence should merit special attention. Table 2 shows some of the longest-lived architecture roots we observed in 8 out of 10 projects. The column "Leading Files" gives the leading file of each long-lived architecture root. "B.rank" is the bugginess ranking of each leading file. Column "#Snap" is the number of snapshots in which the DRSpace, led by the files in column 2, is detected as the most error prone architecture root and the total number (in parenthesis) of snapshots we studied. "A.size" gives the average number of files in the architecture root over different snapshots. "A.bsc" is the average *bsc* of the root over different snapshots. "R.bsc" is the average *bsc* of a random DRSpace, with size equal to the architecture root, over different snapshots. "A.dsb" is the average *dsb* of the architecture root over different snapshots. "R.dsb" is the average *dsb* of a randomly selected DRSpace, with size equal to the architecture root, over different snapshots.

We found each project in Table 2 has at least one persistent architecture root, that 1) is the most error prone DRSpace in at least half of the snapshots studied. For example, in CXF the DRSpace led by LogUtil is the most error prone root over all 92 snapshots; 2) is led by very error-prone files. For example, the long-lived architecture roots of 7 projects are led by at least one file from the top 3% most buggy files; 3) is more error-prone than a random DRSpace of the same size: each root has roughly double the *bsc* and *dsb* compared to a random DRSpace of the same size, except for the root of OpenJPA, which has only slightly higher *bsc* compared to a random DRSpace. We did not find long-lived architecture roots in Wicket or the commercial project. We plan to explore the reason for this in our future work.

In summary, there exist long-lived architecture roots, led by very error prone files, which contain and aggregate a significant portion of a project's error-prone files over time.

3. Maintenance cost "grows" with architecture roots. We further investigated how those persistent architecture roots reported in Table 2 evolved over time. Table 3 shows the Pearson correlation coefficient between the number of top 30% buggy files versus the number of top 30% buggy files contained in the architecture root in different snapshots. In all but one case (Ivy) the data shows that these two data sets are highly correlated, meaning that as the number of buggy files in a system grows, the number of buggy files

Table 2: Persistence of Architecture Roots

Project	Leading Files	B. rank	#Snap	A. size	A.bsc	R.bsc	A.dsb	R.dsb
Avro	Schema	2.4%	22(22)	73	0.6	0.33	0.3	0.15
Camel	CamelContext	3.3%	23(46)	323	0.6	0.27	0.4	0.17
	Exchange	2.2%	23(46)	333	0.6	0.28	0.3	0.17
Cassandra	FBUtilities Token	2.9% 27%	45(46)	180	0.5	0.2	0.4	0.16
PDFBox	COSName	3%	13(13)	204	0.6	0.38	0.3	0.17
CXF	LogUtils	10%	92(92)	470	0.4	0.19	0.5	0.21
HBase	Bytes	2.7%	17(21)	194	0.5	0.29	0.5	0.2
Ivy	ModuleRevisionId IvyContext	34.4% 21.3%	11(11)	181	0.7	0.32	0.3	0.12
OpenJPA	Localizer ResourceBundleProvider J2DoPrivHelper	6.6% 29.6% 2.1%	17(17)	427	0.3	0.24	0.6	0.44

within these architecture roots grows with high probability. For project Ivy, the correlation is not calculated because the bug space size and bug files in the roots remain stable over different snapshots.

Table 3: #Bug Files in Snapshot vs #Bug Files in Root

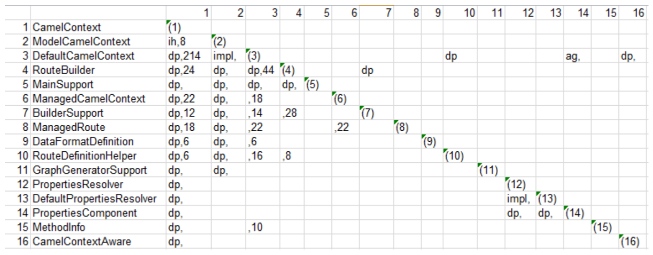
Project	Correlation	R-square
Avro.Schema	0.99	0.98
Camel.Exchange	0.97	0.94
Camel.CamelContext	0.83	0.68
Cassandra.FBUtilities Cassandra.Token	0.82	0.68
CXF.LogUtils	0.99	0.98
HBase.Bytes	0.99	0.99
Ivy.ModuleRevisionId Ivy.IvyContext	na	na
OpenJPA.Localizer OpenJPA.ResourceBundleProvider OpenJPA.J2DoPriHelper	0.79	0.63
PDFBox.COSName	0.97	0.95

To better understand how the bugginess of the architecture roots grows over different snapshots, we numbered each snapshot, starting from 1, in chronological order for each project. Next we built a regression module between the snapshot number and the number of top 30% buggy files in each root for each snapshot of each project. We found that the bugginess of root spaces for four projects fits a linear regression model with the snapshot numbers, meaning the number of buggy files grows with each snapshot. The result is shown in Table 4. The equation for the root space of PDFBox led by $COSName$ is $b = 5v + 92.6$, where b is the number of top 30% buggy files in the DRSpace during each snapshot v. This equation means the number of top 30% buggy files contained in the root space increased by 5 in each snapshot. The $Rsquare$ is 0.93, meaning this linear regression model fits very well. We can make similar observations for other root spaces in Table 4. To summarize, as the project evolves, more and more buggy files "grow" out of these roots. However, the data from other projects do not fit the regression model so well. Why and under what conditions the buggy files size in a root DRSpace grows with versions will be studied in our future work.

4. Architecture Issues within Roots. Using the Titan tool chain, we can automatically detect architecture "issues" or "flaws", e.g. cyclic dependencies, modularity violations, improper inheritance and unstable interfaces. In section 3.2, we discussed the cyclic dependencies and modularity violations observed in the root space of Camel in Figure 2. We can also observe improper inheritance and unstable interfaces in the root space of Camel shown in Figure 3.

Table 4: #Most Buggy DRSpaces over Time

Project	DRSpace Leading Files	Equation	R-2
Camel	CamelContext	b=0.5v+118	0.86
	Exchange	b=2.3v+82.3	0.79
PDFBox	COSName	b=5v+92.6	0.93
CXF	LogUtils	b=1.2v+170	0.88
HBase	Bytes	b=6.8v+30.6	0.83



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	CamelContext	(1)															
2	ModelCamelContext	ih,8	(2)														
3	DefaultCamelContext	dp,214	impl,	(3)							dp				ag,		dp,
4	RouteBuilder	dp,24	dp,	dp,44	(4)			dp									
5	MainSupport	dp,	dp,	dp,	dp,	(5)											
6	ManagedCamelContext	dp,22	dp,	,18			(6)										
7	BuilderSupport	dp,12	dp,	,14	,28			(7)									
8	ManagedRoute	dp,18	dp,	,22			,22		(8)								
9	DataFormatDefinition	dp,6	dp,	,6						(9)							
10	RouteDefinitionHelper	dp,6	dp,	,16	,8						(10)						
11	GraphGeneratorSupport	dp,	dp,									(11)					
12	PropertiesResolver	dp,											(12)				
13	DefaultPropertiesResolver	dp,											impl,	(13)			
14	PropertiesComponent	dp,											dp,	dp,	(14)		
15	MethodInfo	dp,		,10												(15)	
16	CamelContextAware	dp,															(16)

Figure 3: Camel Root DRSpace Part 2

In a software system following basic object-oriented design principles, a client of an inheritance hierarchy should only depend on the base class but not the concrete class. In Figure 3, $DefaultCamelContext$ implements $ModelCamelContext$ and $ModelCamelContext$ inherits from $CamelContext$, which form an inheritance hierarchy. Two classes—$RouteBuilder$ and $MainSupport$—depend on all three classes in the inheritance tree, including the concrete class $DefaultCamelContext$. This is an inappropriate inheritance because the client of an inheritance hierarchy should depend on the abstract class, not the concrete classes, so that the concrete classes and the clients could evolve independently. The number in cell[r4:c3] indicates that $RouteBuilder$ changes frequently with $DefaultCamelContext$ (44 times). This demonstrates that this inappropriate inheritance incurs high maintenance cost.

In a software project, when changes are made to files with high impact, such as super classes, interfaces and classes with many dependents, the impacted files have to accommodate these changes. Hence, files with high impact should remain *stable*, otherwise they trigger severe ripple effects and high costs. In Figure 3, the super class $CamelContext$ changes frequently with five of the classes that depend on it. In particular, it changes with $DefaultCamelContext$ 214 times! This indicates that $CamelContext$ is an unstable interface.

In summary, architecture roots contain issues—architectural flaws—that cause high maintenance costs and which are, we claim, the *root causes* of their bugginess.

4.2 Contributions

This research has made a number of unique contributions to software engineering.

A new way of modeling and analyzing software architecture. We model software architecture as multiple, overlapping DRSpaces, which reveal design rules and the modules that these rules lead. Evolutionary coupling—a special form of dependency—is analyzed together with structural coupling to reveal architecture flaws in DRSpaces.

A new way of bug localization and quality improvement. We bridge the gap between software architecture and quality by automatically detecting architecture roots that aggregate and connect large numbers of buggy files. We have observed, over many projects, a strong correlation between architecture flaws in those roots and high rates of change-proneness and bugginess, leading us to postulate that such flaws are the *root causes* of bugginess. To increase the quality and reduce the overall bugginess of a system, the architecture flaws must be removed. Software architects must therefore pay attention to the architecture flaws that are the causes, in many cases, of individual buggy files. Furthermore, we recommend that architects consider refactoring to remove these architecture flaws and hence address bugginess more fundamentally.

5. INDUSTRY IMPACT

Our approach has been applied and adopted by 4 major commercial software organizations. The applications in large-scale industrial software systems have demonstrated the effectiveness of our approach in giving valuable architecture feedback and potential refactoring suggestions to the developer teams. In the case study by Schwanke [15], we measured the architecture and identified architecture issues in an agile industrial software project by combining software structure and evolution history. The developer team confirmed the issues and implemented a refactoring proposal justified by the case study. In the case study by Kazman [6] with SoftServe, we identified the architecture roots which were confirmed as the cause of technical debt by the developers. We were able to quantify the maintenance costs incurred by each root in terms of the estimated additional numbers of bugs and changed lines of code caused by flawed relations among files. We proposed refactoring suggestions to pay off these debts and quantitatively estimated the return on investment for this refactoring using simple economic models. The project manager immediately decided to invest on our refactoring proposal. Other industry case studies, that we have not yet published, have also showed that our approach has great potential in that it identifies the architectural root causes of bugginess and high maintenance costs: instead of examining hundreds of defective files in isolation, our collaborators now only need to examine a few architecture roots, fixing numerous defects simultaneously by removing their structural flows, thus providing substantial savings in maintenance costs.

6. REFERENCES

[1] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity.* MIT Press, 2000.

[2] D. Garlan. *Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events.* Springer Berlin Heidelberg, 2003.

[3] M. Gethers and D. Poshyvanyk. Using relational topic models to capture coupling amoung classes in object-oriented software systems. In *Proc. 26th IEEE International Conference on Software Maintenance*, pages 1–10, Sept. 2010.

[4] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[5] S. Huynh, Y. Cai, and K. Sethi. Design rule hierarchy and model transformations. Presented at *Student Research Forum* of 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Nov. 2008. (Best Student Poster Award).

[6] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyevy, V. Fedaky, and A. Shapochkay. A case study in locating the architectural roots of technical debt. In *Proc. 37th International Conference on Software Engineering*, 2015.

[7] S. Kim, T. Zimmermann, J. Whitehead, and A. Zeller. Predicting faults from cached history. In *Proc. 29st International Conference on Software Engineering*, pages 489–498, May 2007.

[8] Y. C. Lu Xiao and R. Kazman. Design rule spaces: A new form of architecture insight. In *Proc. 36th International Conference on Software Engineering*, 2014.

[9] S. Mancoridis, B. S. Mitchell, Y.-F. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proc. 15th IEEE International Conference on Software Maintenance*, pages 50–59, Aug. 1999.

[10] R. Mo, Y. Cai, R. Kazman, and L. Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Proc. 15th Working IEEE/IFIP International Conference on Software Architecture*, May 2015.

[11] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. 27th International Conference on Software Engineering*, pages 284–292, May 2005.

[12] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. 28th International Conference on Software Engineering*, pages 452–461, 2006.

[13] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *Proc. 13thACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 86–96, July 2004.

[14] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[15] R. Schwanke, L. Xiao, and Y. Cai. Measuring architecture quality by structure plus history analysis. In *Proc. 35rd International Conference on Software Engineering*, pages 891–900, May 2013.

[16] L. Xiao, Y. Cai, and R. Kazman. Titan: A toolset that connects software architecture with quality analysis. In *Proc. 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE 2014, pages 763–766, 2014.