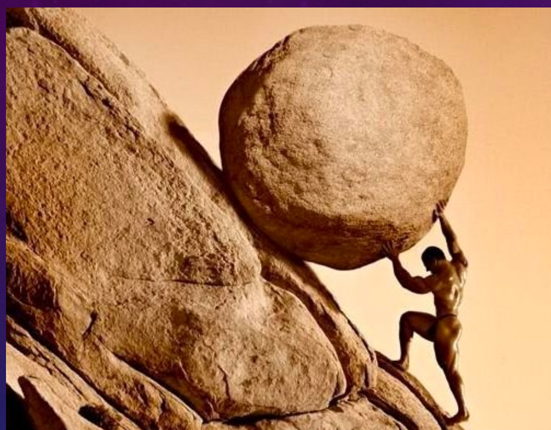


Challenges in Architecting Web 4.0 Systems

Rick Kazman

University of Hawaii

1

The State of the Practice in Software Development

2

The State of the Practice in Software Development

- The boat is leaking but you keep paddling!
- Why?
  - The illusion of progress.
  - The lack of measurements.
  - Design is largely invisible.

3

Architecture/Design Flaws

4

My "Grand Research Challenge"

- Design debt is the most pernicious form of technical debt.
- How to measure the health of an architecture?
- Can this be:
  - Automated?
  - Empirically justified?
  - Repeatable?

5

Isn't This a Solved Problem?

- Just use existing TD detection tools, e.g.

structure101 tangle free software

sonarqube

Designite http://www.designite-tools.com

6

Sadly, no…

- Results of a recent study:
  - TD detection tools disagree about basic (seemingly) objective measures due to different definitions of fundamental concepts.
  - The majority of what is reported by these tools is no more insightful than LOC.

[Lefever et al. ICSE 2021]

7

And it Gets Worse…

- Existing tools only analyze static relationships.
- But, increasingly, systems are being built from dynamic languages (e.g. Python, Ruby) and as a set of microservices.
- We called these DD (Dynamic and Distributed) Systems
- These are the architectures of Web 4.0 systems.

- How do we analyze these?
- And can this be automated, repeatable, etc. ?

8

Detecting Design Debt in "Traditional" Systems

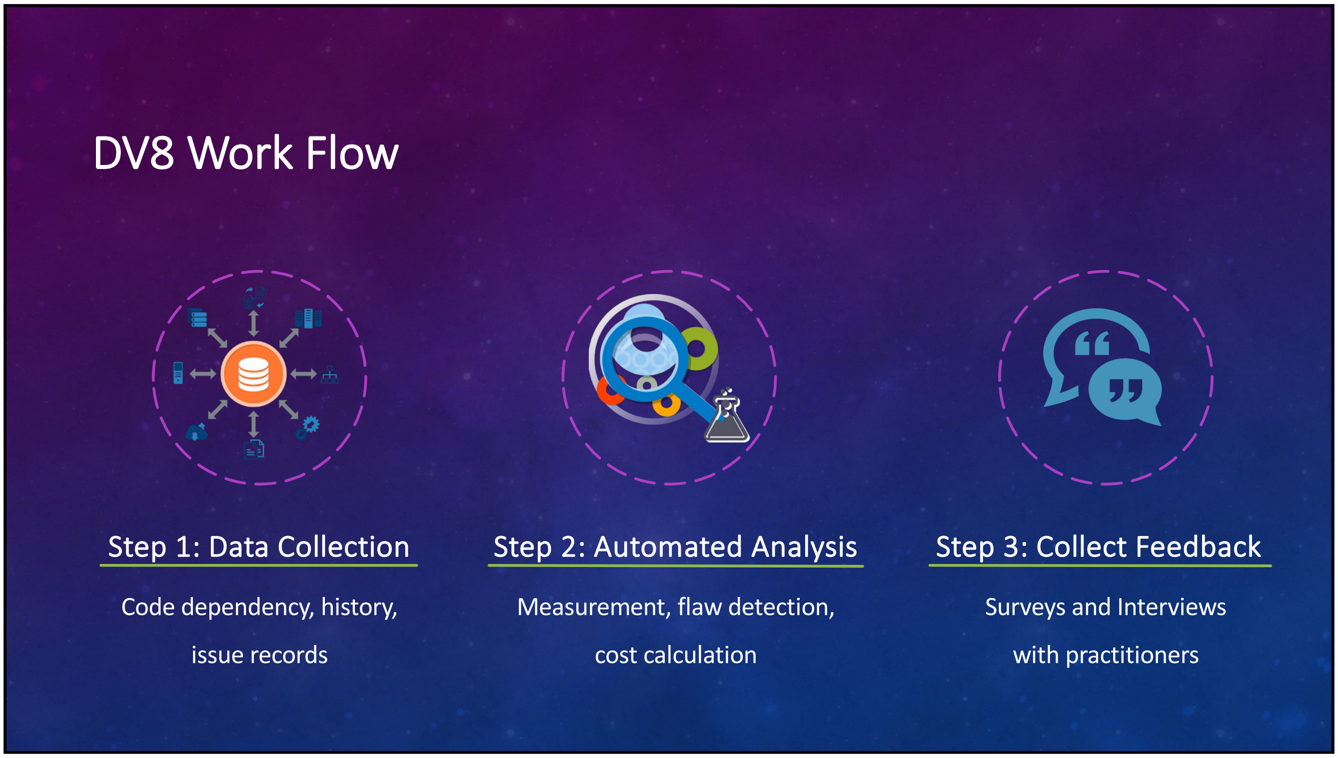
- Let us begin by reviewing the state of the art in design debt detection using DV8.

9

Empirical Basis

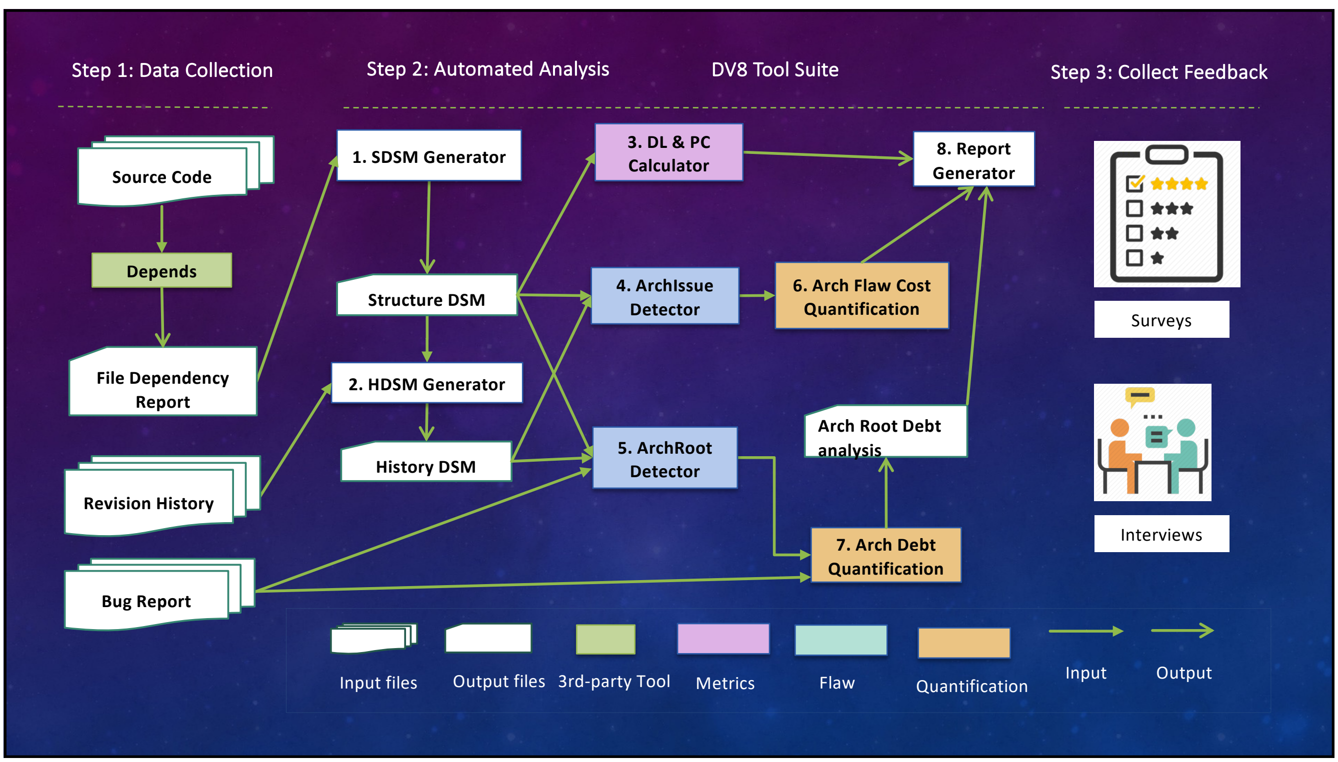
- >300 Open Source Projects
- >50 Industrial projects

10



DV8 Work Flow
Step 1: Data Collection
Code dependency, history, issue records
Step 2: Automated Analysis
Measurement, flaw detection, cost calculation
Step 3: Collect Feedback
Surveys and Interviews with practitioners

11



Step 1: Data Collection
Step 2: Automated Analysis
DV8 Tool Suite
Step 3: Collect Feedback
Source Code
Depends
File Dependency Report
Revision History
Bug Report
1. SDSM Generator
Structure DSM
2. HDSM Generator
History DSM
3. DL & PC Calculator
4. ArchIssue Detector
5. ArchRoot Detector
6. Arch Flaw Cost Quantification
7. Arch Debt Quantification
Arch Root Debt analysis
8. Report Generator
Surveys
Interviews
Input files
Output files
3rd-party Tool
Metrics
Flaw
Quantification
Input
Output

12

Step 1: Data Collection

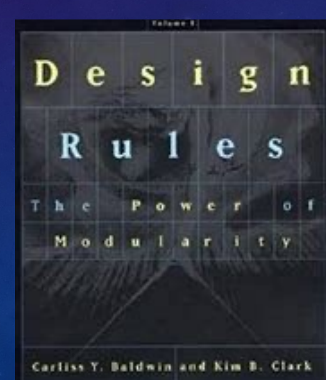
- Dependency information
- History information
- Issue information

13

Design Rule Space (DRSpace)

A DRSpace is composed of a **meaningful subset** of a system's files and the **architectural connections** among these files.

- Any subset of files may form a design space
- Architectural connections
  - Structural couplings: call, inherit, aggregate, etc.
  - Evolutionary couplings
  - Implicit or explicit



[Cai et al, TSE 2019]
[Xiao et al, ICSE 2014]

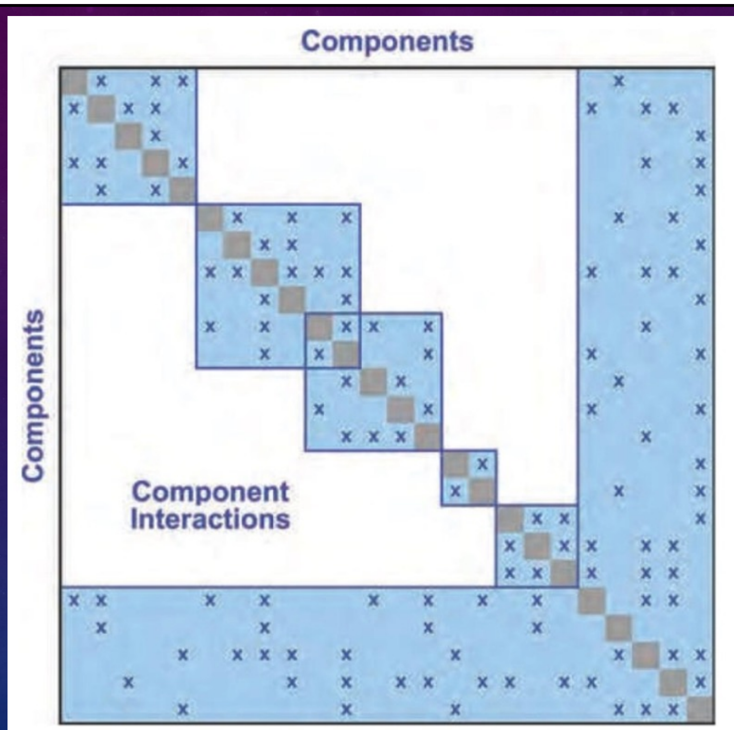
14

Design Rule Space (DRSpace)

- Non-trivial software system contain multiple design spaces:
  - each feature implemented
  - each pattern applied
  - each concern addressed
- Each file can participate in multiple DRSpaces
- Architectures can be modeled as overlapping DRSpaces
- We visualize each DRSpace as a ***Design Structure Matrix*** (DSM)

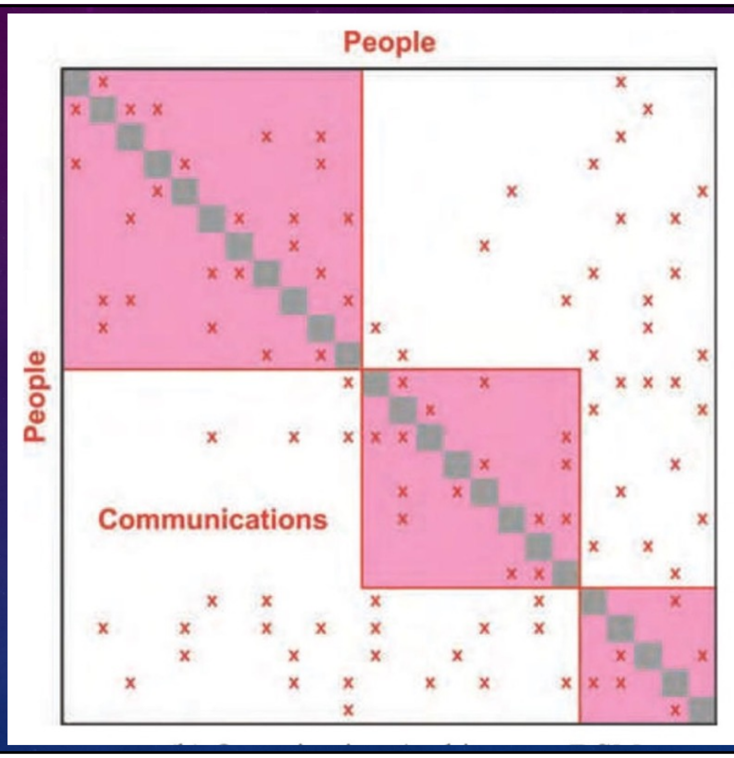
15

DSMs



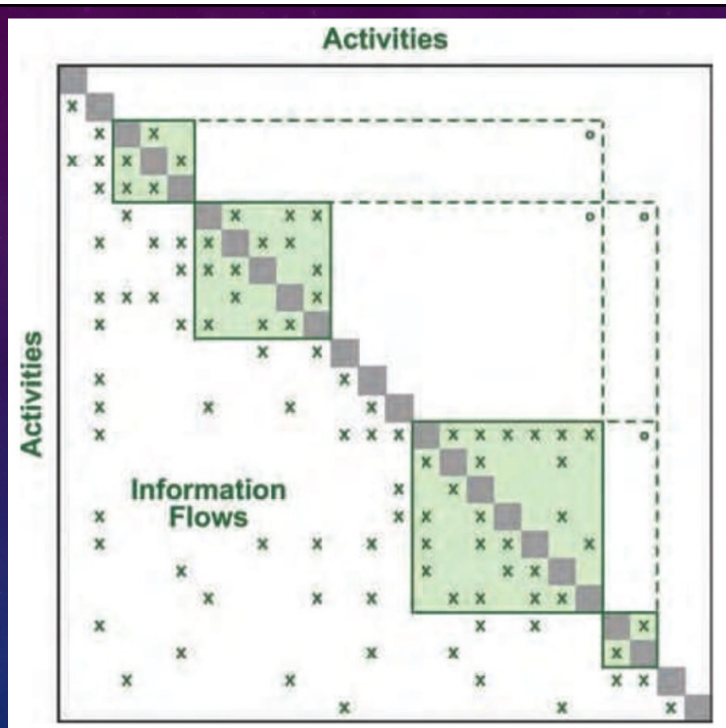
16

DSMs



17

DSMs



18

Step 2: Automated Architecture Analysis

Step 2.1 Measure and Monitor

Compare, Contrast, and Monitor

Step 2.2: Pinpoint Flaws

Detection and Visualization

Step 2.3: Quantify Design Debt

Costs and benefits

19

Step 2.1: Measure and Monitor

- **Decoupling Level (DL):**
  an options-based metric, measuring the system's ability to generate options
- **Propagation Cost (PC):**
  a DSM-based metric, measuring how tightly coupled a system is

[Mo et al. ICSE 2016]

20

Decoupling Level (DL): Rationale

- A true module should be
  - Small
  - Independent
- A highly modularized system should
  - Have large numbers of true modules…
  - connected by design rules

[Mo et al. ICSE 2016]

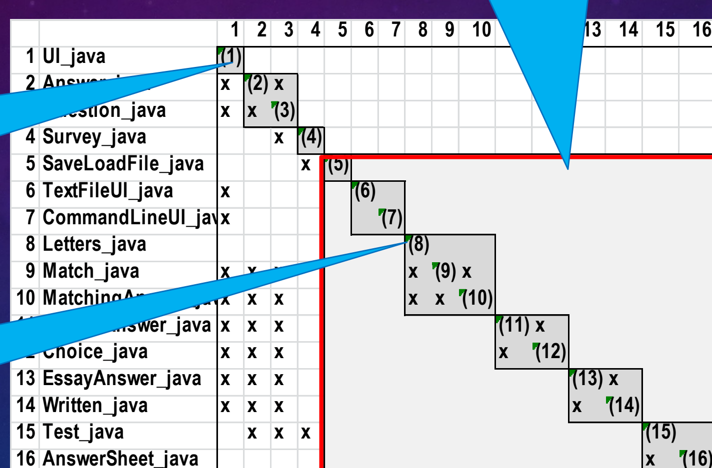
21

Decoupling Level (DL)

The more files are clustered into true modules, the higher the value

Upper Layer modules:
- The fewer dependents, the higher the value
- The smaller the module, the higher the value

True modules:
- The smaller a true module, the higher the value
- The more true modules, the higher the value



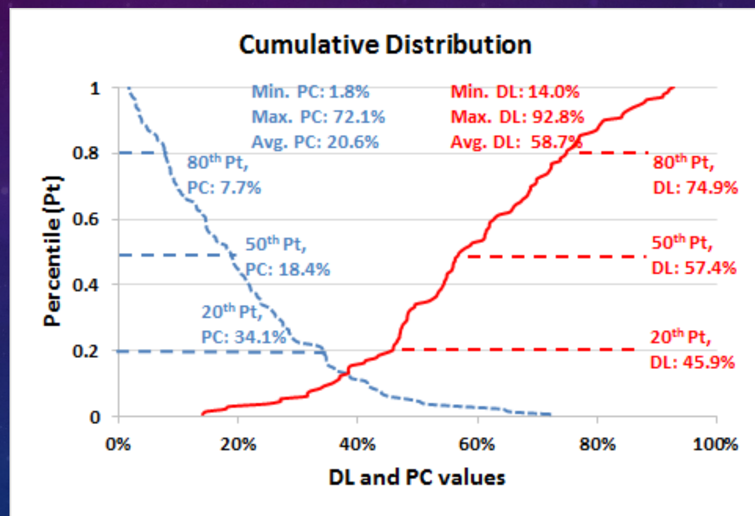
[Mo et al. ICSE 2016]

22

Decoupling Level (DL) and Propagation Cost (PC)

Data from 129 projects:

- 108 open source
- 21 industrial



[Mo et al. ICSE 2016]

23

DL and PC "Health Chart"

	Open Source		Commercial	
	DL	PC	DL	PC
Avg	60	20	54	21
Median	58	18	56	20
Max	92	72	93	50
Min	14	2	15	2
20th Pt	47	8	36	6
40th Pt	55	14	46	17
60th Pt	66	21	59	24
80th Pt	75	34	65	35

Pt: Percentile

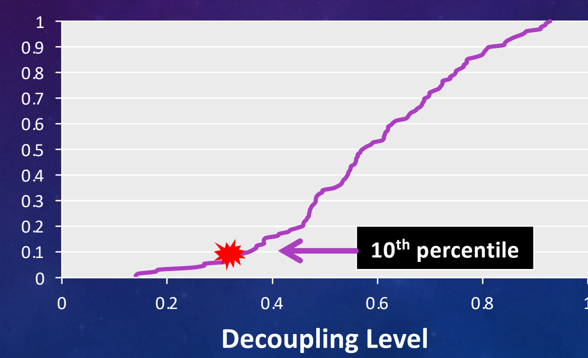
Best DL (93%) is from industry

Worst DL (14%) is from open source

24

DL and PC "Health Chart"

- An industrial project:
  - DL: 29%, 10th percentile: Confirmed to have severe maintenance difficulty



25

Step 2.2: Flaw Detection

- We automatically identify 6 types of design flaws
  1. Unstable interface
  2. Modularity violation
  3. Crossing
  4. Improper inheritance
  5. Cliques among files
  6. Package cycles
- These flaws are highly correlated with bugs, changes, and churn

[Mo et al. WICSA 2015]

26

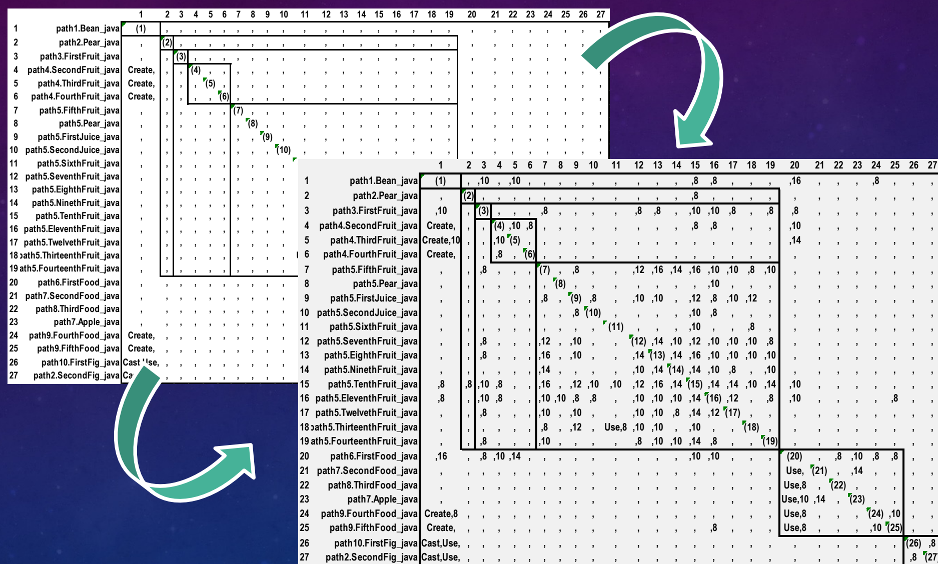
Flaw Type 1: Unstable Interface

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	config.DatabaseDescriptor	(1)	dp,44	,14	,10	,10	,6	,14	,36	,118	,12	,	,16	,12	,42	,52	,4	,18	,30
2	utils.FBUtilities	dp,44	(2)	,40	,4	,6	,10	,6	,12	,38	,28	,12	,8	,14	,24	,46	,6	,18	,28
3	utils.ByteBufferUtil	,14	dp,40	(3)	,		,	,	,4	,10	,20	,4	,4		,10	,26		,12	,4
4	service.WriteResponseHandler	,10	dp,4	,2	(4)	,4	,6	,18	dp,22	,						,6	,		,
5	locator.TokenMetadata	,10	,6		,4	(5)	,4	,10	dp,24	,	,8					,4	,6	,4	,
6	locator.NetworkTopologyStrategy	,6	dp,10	,2	,6	dp,4	(6)	,10	ih,22	,4	,	,				,16	,		,8
7	service.DatacenterWriteResponseHandler	dp,14	dp,6	,2	ih,18	,10	dp,10	(7)	,20		,	,				,6	,6		,
8	locator.AbstractReplicationStrategy	,36	dp,12	,4	dp,22	ag,24	,22	dp,20	(8)	,6						,16	,10	,	,10
9	config.CFMetaData	,118	dp,38	dp,10	,	,	,4		,6	(9)			,16		,36	,46			,56
10	dht.RandomPartitioner	,12	dp,28	dp,20		,8	,	,			(10)	dp,4			,4	,16		,50	
11	utils.GuidGenerator	,	dp,12	,4			,	,			,4	(11)				,4		,	
12	io.sstable.SSTable	,16	,8	dp,4						ag,16			(12)	,4	dp,68	,10			,
13	utils.CLibrary	,12	dp,14										,4	(13)	,12	,			
14	io.sstable.SSTableReader	dp,42	,24	dp,10						,36	,4		ih,68	dp,12	(14)	,22	,4	,	,10
15	cli.CliClient	,52	dp,46	dp,26	,6	,4	,16	,6	,16	,46	,16	,4	,10	,	,22	(15)	,6	,14	,48
16	locator.PropertyFileSnitch	,4	dp,6		,	dp,6	,	,6	,10						,4	,6	(16)		,4
17	dht.OrderPreservingPartitioner	dp,18	dp,18	dp,12		,4			,		,50	,			,	,14		(17)	
18	thrift.ThriftValidation	dp,30	,28	dp,4	,	,	,8	,	dp,10	dp,56			,		,10	,48	,4		(18)

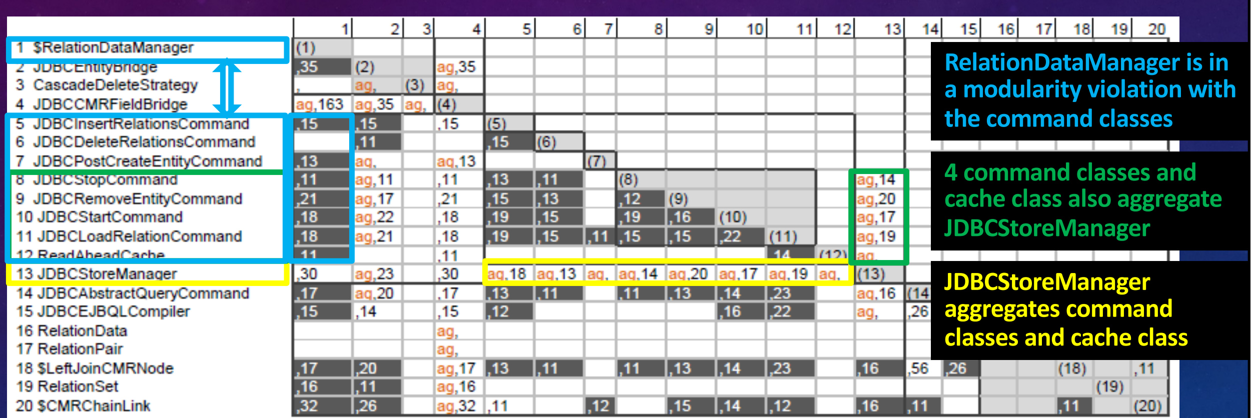
Flaw Type 2: Crossing

		1	2	3	4	5	6	7	8	9	10	11	12	13
1	path1.File1_h	(1)	, 2		, 6	, 11	, 3	, 2	, 8	, 6	, 5		, 2	
2	path1.File2_h	d, 2	(2)			, 2								
3	path1.File3_h	d		(3)		, 2			, 2				, 2	
4	path2.File1_h	, 6			(4)	, 5	, 2	, 2	, 5	, 5	, 4		, 2	
5	path2.File2_h	d, 11	d, 2	d, 2	d, 5	(5)	, 3	, 2	, 10	, 6	, 5	, 2	, 3	, 2
6	path2.File3_h	d, 3			, 2	d, 3	(6)	d, 2	, 2	, 3	, 2		, 2	
7	path3.File1_h	d, 2			, 2	d, 2	d, 2	(7)	, 2	, 2	, 2		, 2	
8	path3.File2_c	d, 8		, 2	, 5	d, 10	d, 2	d, 2	(8)	, 6	d, 5		, 3	
9	path3.File3_c	d, 6			, 5	d, 6	d, 3	d, 2	d, 6	(9)	, 8		, 2	
10	path4.File1_c	d, 5			, 4	d, 5	d, 2	d, 2	d, 5	d, 8	(10)		, 2	
11	path4.File2_c	d				d, 2						(11)		, 7
12	path4.File3_c	d, 2		, 2	, 2	d, 3	d, 2	d, 2	, 3	, 2	, 2		(12)	
13	path5.File1_c	d	d	d		d, 2	d	d	d			d, 7	d	(13)

Flaw Type 3: Modularity Violation



Sample Flaws from JBoss



Jboss JDBCCMRFieldBridge DRSpace

Sample Flaws from Cassandra

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1 cassandra.config.DatabaseDescriptor	(1)	dp,44	,14	,10	,10	,6	,14	,36	**,118**	,	,12	,12	,16	,42	,52	,30	,18	,4
2 cassandra.utils.FBUtilities	**dp,44**	2)	,40	,4	,6	,10	,6	,12	,38	,12	,28	,14	,8	,24	,46	,28	,18	,6
3 cassandra.utils.ByteBufferUtil	**,14**	dp,40	(3)	,	,	,	,	,4	,10	,4	,20	,	,4	,10	,26	,4	,12	,
4 cassandra.service.WriteResponseHandler	**,10**	dp,4	,	(4)	,4	,6	,18	dp,22	,	,	,	,	,	,	,6	,	,	,
5 cassandra.locator.TokenMetadata	**,10**	6	,	,4	(5)	,4	,10	dp,24	,	,	,8	,	,	,	,4	,	,4	,6
6 cassandra.locator.NetworkTopologyStrategy	,6	dp,10	,	,6	dp,4	(6)	,10	ih,22	,4	,	,	,	,	,	,16	,8	,	,
7 cassandra.service.DatacenterWriteResponseHandler	**dp,14**	dp,6	,	ih,18	,10	dp,10	(7)	,20	,	,	,	,	,	,	,6	,	,	,6
8 cassandra.locator.AbstractReplicationStrategy	**,36**	dp,12	,4	dp,22	ag,24	,22	dp,20	(8)	,6	,	,	,	,	,	,16	,10	,	,10
9 cassandra.config.CFMetaData	**,118**	dp,38	dp,10	,	,	,4	,	,6	(9)	,	,	,	,16	,36	,46	,56	,	,
10 cassandra.utils.GuidGenerator	,	dp,12	,4	,	,	,	,	,	,	(10)	,4	,	,	,	,4	,	,	,
11 cassandra.dht.RandomPartitioner	**,12**	dp,28	dp,20	,	,8	,	,	,	,	dp,4	(11)	,	,	,4	,16	,	,50	,
12 cassandra.utils.CLibrary	**,12**	dp,14	,	,	,	,	,	,	,	,	,	(12)	,4	,12		,	,	,
13 cassandra.io.sstable.SSTable	**,16**	8	dp,4	,	,	,	,	,	ag,16	,	,	,4	(13)	**dp,68**	10	,	,	,
14 cassandra.io.sstable.SSTableReader	**dp,42**	24	dp,10	,	,	,	,	,	,36	,	,4	dp,12	**ih,68**	(14)	22	,10		,4
15 cassandra.cli.CliClient	**,52**	dp,46	dp,26	,6	,4	,16	,6	,16	,46	,4	,16	,	,10	,22	(15)	,48	,4	,6
16 cassandra.thrift.ThriftValidation	**dp,30**	28	dp,4	,	,	,8	,	dp,10	dp,56	,	,	,	,	,10	,48	(16)	,	,4
17 cassandra.dht.OrderPreservingPartitioner	**dp,18**	dp,18	dp,12	,	,4	,	,	,	,	,	,50	,	,	,	,14	,	(17)	,
18 cassandra.locator.PropertyFileSnitch	,4	dp,6	,	,	dp,6	,	,6	,10	,	,	,	,	,	,4	,6	,4	,	(18)

31

Do Design Flaws Really Matter?

Research Question: If a file is involved in greater numbers of architecture flaws, it is more error-prone/change-prone than average files?

[Mo et al. WICSA 2015]

32

Data Set

Subject	#Month	#Snap	#Comm	#Issue	#File
Avro 1.7.6	47	22	1480	630	145-298
Camel 2.11.1	53	46	17706	2326	528-1203
Cassandra 1.0.7	24	46	6738	3645	419-786
CXF 2.7.10	70	92	27247	3400	1426-3073
HBase 0.94.16	70	21	14858	5032	347-2142
Ivy 2.3.0	52	11	3799	839	418-607
OpenJPA 2.2.2	68	17	6736	1574	1216-1761
PDFBox 1.8.4	46	13	1798	1279	458-589
Wicket 1.5.5	57	55	18004	3359	1099-1549
Commercial	9	13	6000	800	137-599

33

Analysis

We counted the architecture flaws in these 10 projects and compared these to:

- Bug frequency
- Bug churn
- Change frequency
- Change churn

34

Results

Avro-1.7.6

#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.1	3.7	0.5	29.0
1	0.4	3.9	0.9	26.2
2	1.6	12.6	5.2	376.7
3	7.9	124.5	21.6	628.5
4	16.5	255.0	33.5	1220.0
PC	0.91	0.89	0.94	0.95

Camel-2.11.1

#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.5	7.9	2.2	58.2
1	1.2	18.5	5.6	131.5
2	3.7	56.6	14.4	304.7
3	8.4	141.5	33.9	681.3
4	13.9	204.7	50.9	1043.5
PC	0.96	0.96	0.97	0.97

Cassandra-1.0.7

#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.4	7.1	1.0	32.6
1	1.1	17.4	4.8	106.4
2	5.3	84.5	21.2	559.1
3	12.8	245.8	45.7	1202.0
4	18.8	364.9	65.7	1909.4
PC	0.97	0.96	0.98	0.97

CXF-2.7.10

#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.8	21.0	2.8	86.9
1	2.9	62.3	9.4	262.5
2	8.6	164.8	23.1	592.0
3	20.2	390.9	52.5	1232.4
4	54.1	890.2	142.3	3326.0
PC	0.90	0.92	0.89	0.89

Hadoop-2.2.0

#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.4	12.7	1.0	56.8
1	1.5	24.8	4.2	167.7
2	5.3	173.6	13.8	558.3
3	26.0	725.1	58.0	1959.6
4	13.7	237.9	26.8	1252.0
PC	0.76	0.63	0.72	0.83

HBase-0.94.16

#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.7	10.4	0.9	53.0
1	4.8	236.7	8.3	614.6
2	9.9	418.5	17.2	2083.6
3	47.8	1335.1	87.6	3158.7
4	76.7	2370.4	135.1	6019.0
PC	0.93	0.94	0.93	0.97

Ivy-2.3.0

#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.2	4.5	1.1	31.8
1	1.1	22.8	3.3	79.6
2	2.9	54.6	8.4	251.9
3	7.0	119.9	20.9	646.2
4	6.4	204.6	18.6	792.3
PC	0.94	0.96	0.93	0.97

OpenJPA-2.2.2

#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	1.8	10.0	1.1	36.8
1	3.2	31.1	3.7	111.5
2	4.6	64.5	7.5	229.8
3	10.8	408.6	22.4	862.5
4	25.1	981.0	52.5	2301.1
PC	0.90	0.88	0.90	0.88

Pdfbox-1.8.4

#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.5	27.1	1.1	92.0
1	1.4	35.9	2.9	136.5
2	1.5	64.1	3.4	259.9
3	8.1	495.0	13.7	861.3
4	12.2	669.5	18.4	1254.4
PC	0.92	0.92	0.94	0.94

Commercial Project

#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.1	2.25	2.7	102.5
1	0.2	4.6	5.9	200.4
2	0.8	3.24	10.3	372.0
3	2.8	36.8	19.8	884.7
4	6.0	21	29.0	549.0
PC	0.91	0.73	0.98	0.81

35

Avro-1.7.6

#Flaws	BF_avg	BC_avg	CF_avg	CC_avg
0	0.1	3.7	0.5	29.0
1	0.4	3.9	0.9	26.2
2	1.6	12.6	5.2	376.7
3	7.9	124.5	21.6	628.5
4	16.5	255.0	33.5	1220.0
PC	***0.91***	***0.89***	***0.94***	***0.95***

36

More Consequences of Design Flaws

Research Question: If a file is involved in greater numbers of architecture flaws, it is involved in more *security* bugs/changes than average files?

[Feng et al. WICSA 2016]

37

Answer

We counted the architecture flaws in these 11 projects and compared these to:

- Security bug frequency
- Security change frequency
- …as well as the original measures (bugs, changes, bug churn, change churn)

38

Answer

Project	Bug/Flaw Correlation	Change/Flaw Correlation	Security Bug/Flaw Correlation
Avro	0.845	0.923	0.861
Camel	0.956	0.959	0.958
Cassandra	0.830	0.869	0.808
Chrome	0.987	0.988	0.979
CXF	0.896	0.910	0.939
Derby	0.938	0.917	0.897
Hadoop	0.752	0.902	0.862
HBase	0.894	0.932	0.961
httpd	0.710	0.688	0.885
PHP	0.929	0.987	0.923
Tomcat	0.901	0.776	0.920

39

Step 2.3: Quantification

- Calculate the costs of each root, each flaw and each type of flaw
- Calculate ROI (Return on Investment)

[Kazman et al. ICSE 2015]

[Xiao et al. ICSE 2016]

40

Industrial Experience: ROI Calculation

Penalty Caused by Architecture Debt | Refactoring Cost | Expected Savings

	A	B	C	D	E	F	G	I	J	K	L	M	N
1	***DRSpace Leading File***	**DRSpace Size**	**Norm Size**	**Current Defects/Yr**	**Norm Defects**	**Current Changes/Yr**	**Norm Changes/Yr**	**Tot LOC Changed**	**Norm LOC Changed**	**Refactor Cost (PM)**	**Norm Exp Defects/Yr**	**Norm Exp Changes/Yr**	**Norm Exp LOC Changed**
2	*Pear.java*	139	119.33	166	142.5	1068	839.2	49,171	42,213	5.5	39	346	20,281
3	*Apple.java*	158	133.83	63	53.4	607	451.7	25,603	21,686	7	44	388	22,745
4	*Bean.java*	65	37.83	72	41.9	429	207.2	17,807	10,364	1.5	12	110	6,429
5													
6	**DRSpace Total**		**290.99**		**237.8**		**1498**		**74,263**		**96.0**	**843.871**	**49,455**
7	**Project Total**	**797**		**265**		**2332**		**135,453**		**14**			
8	**Savings**										**142**	**654**	**24,808**
9													
10													
11	Base defect rates	0.33											
12	Base change rates	2.9										**Exp PM save**	**41.35**
13	Base LOC/file	169.95											
14	LOC/PM	600											

Result: ~300% ROI in the first year alone!

[Kazman et al. ICSE 2015]

41

Industrial Experience: Analyzing 8 ABB Projects

- Using 3 complementary techniques:
  - Architecture-level maintainability metrics
  - Architecture flaw analysis
  - Cost and benefit analysis
- 8 projects developed at multiple locations (India, USA, Switzerland) differing in age, domain, and size.
- We reported the results back to each project and collected feedback

[Mo et al. ASE 2018]

42

Results

Participants of all 8 projects verified that the information provided was useful in closing the understanding gap with management. They have begun the refactoring process.

All participants said the report gave them quantifiable results with which to judge their project. The comparison with industrial benchmarks made it clear that maintenance difficulty caused by degrading architecture is common.

Six of the eight projects planned to or already started refactoring to address the detected flaws. The project with the lowest DL score is undergoing a major rewrite.

43

Industrial Experience: Huawei

- Developed a set of architecture measures based on DL and architecture flaws
  - Adopted as a corporate standard
  - Now used in over 100 projects
- Quantified architecture debt
- 24 out of 29 projects studied showed a positive correlation between these measures and productivity

[Wu et al. ECSA 2018]

44

Industrial Experience: BrightSquid

- Analyzed BrightSquid's secure communication platform (6/16 – 5/17)
- Identified many areas of architecture debt—the "before" state—and recommended a refactoring plan to pay down the debt (7/17)
- Architecture was refactored (1/18 – 3/18)
- Analyzed the "after" state (3/18 – 8/18)

[Nayebi et al. ICSE 2019]

45

BrightSquid Results

General information	Before	After
# of files	1713	711
# of roots covering 80% of bugs	5	3
# of files in roots covering 80% of bugs	296	295
# of files covering 80% of bugs	17%	37%
Architectural Metrics	**Before**	**After**
Decoupling level	86%	83%
Propagation cost	6%	6%
Architectural flaws	**Before**	**After**
# of cliques	17	10
# of files influenced by cliques	71	26
# of unhealthy inheritance	60	30
# of files influenced by unhealthy inheritance	222	102
# of unstable interface	12	8
# of files influenced by unstable interface	471	59
# of crossings	29	6
# of files influenced by crossings	387	47
# of package cycles	34	19
# of files influenced by package cycles	242	94

46

Industrial Experience: BrightSquid

- The refactoring activities were recorded as 106 change requests, which consumed 563.8 person hours.
- After refactoring, the size of the code base shrunk by 41.5%
- The average time needed to close issues before and after refactoring was reduced by 72%.
- The average bug-fixing churn per issue dropped by 2/3: from 102 LOC before refactoring to 34 LOC after refactoring
- The average bug-fixing duration reduced 30%, dropping from 10 days before to 7 days

47

But What About DD Systems?

- Web 4.0 architectures are primarily expected to be DD systems.
- Consider this example from a dynamic language:

		1	2	3	4	5	6	7	8
1	CarElement.java	(1)				X			
2	Body.java	X	(2)			X			
3	Wheel.java	X		(3)		X			
4	Engine.java	X			(4)	X			
5	**CarElementVisitor.java**		X	X	X	(5)	X		
6	Car.java	X	X	X	X	X	(6)		
7	CarElementPrintVisitor.java		X	X	X	X	X	(7)	
8	CarElementDoVisitor.java		X	X	X	X	X		(8)

(a) The DSM recovered from Java code

		1	2	3	4	5	6	7	8
1	CarElement.py	(1)							
2	Body.py	X	(2)						
3	Wheel.py	X		(3)					
4	Engine.py	X			(4)				
5	**CarElementVisitor.py**					(5)			
6	Car.py	X	X	X	X		(6)		
7	CarElementDoVisitor.py					X		(7)	
8	CarElementPrintVisitor.py					X			(8)

(b) The DSM recovered from Python code

48

Types Need to be Inferred

- Fortunately, we can resolve most of these "possible dependencies" using type inference (Duck typing).

49

Types Need to be Inferred: Preliminary Study

- We summarized possible dependencies and explicit dependencies from 105 Python projects.
- On average, 75.72% of all syntactic dependencies are explicit, and 24.28% are possible dependencies.
- Among the possible dependencies, the majority (14.28%) are "P1" dependencies.
- Good news! This means that 90% (75.72%+14.28%) of syntactic dependencies can be unambiguously determined using static analysis.

50

Types Need to be Inferred: Consequences

- On average, a file involved in possible dependencies requires 30% more maintenance effort than a file involved in explicit dependencies.
- => maintainability impact imposed by these possible dependencies is surprisingly high compared with explicit dependencies.

[Jin et al, 2021]

51

But What About DD Systems?

- Currently DV8 ingests source code dependencies and co-change information extracted from a project's revision history.
- But in *distributed*, microservice systems, services are typically created and maintained in separate repositories by distinct teams.
- Hence, a poorly designed microservice system may be much harder to analyze and maintain–there is no single place to analyze.

52

Research Challenge

- For dynamic systems we need to extract:
  - Compile-time dependencies, including explicit and possible dependencies.
  - API dependencies among components.
  - Data-dependency and semantic-dependency.
  - Build-time dependencies.
  - Run-time dependencies, which will be extracted from execution logs.
  - Ownership relations.

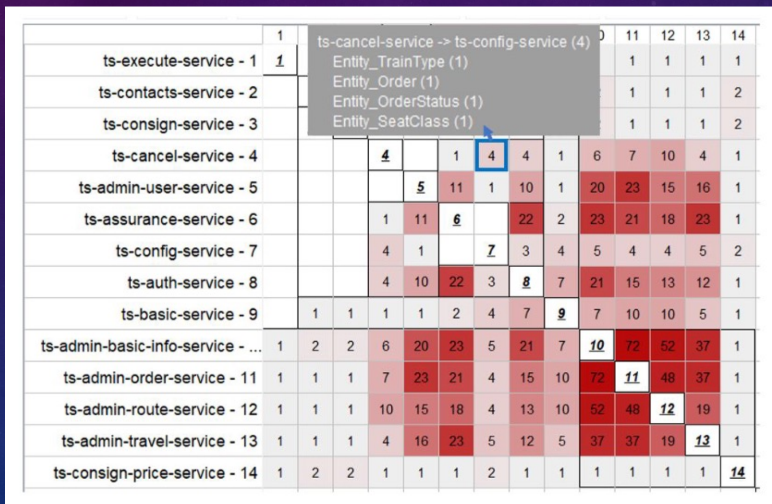
53

Early Results: Team-based Interactions

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Infrastructure - 1	1		27					1		17	18	412	415	990		5514
Team1 - 2	21	2														1
Team2 - 3	248		3													14
Team3 - 4	11			4												
Team4 - 5	8				5											
Team5 - 6	205					6										8
Team6 - 7	9						7									1
Team7 - 8	83							8								4
Team8 - 9	107								9				1			
Team9 - 10	176									10				1		8
Team10 - 11	433										11		1			24
Team11 - 12	3179										9	12	6	147		54
Team12 - 13	387										3		13	26	9	17
Team13 - 14	665											2	3	14	2	45
Team14 - 15	131										1		3	2	15	10
unknown - 16	3779									2				6		16

54

Early Results: Data Coupling



	1										11	12	13	14
ts-execute-service - 1	*1*										1	1	1	1
ts-contacts-service - 2											1	1	1	2
ts-consign-service - 3											1	1	1	2
ts-cancel-service - 4				*4*		1	4	4	1	6	7	10	4	1
ts-admin-user-service - 5					*5*	11	1	10	1	20	23	15	16	1
ts-assurance-service - 6				1	11	*6*		22	2	23	21	18	23	1
ts-config-service - 7				4	1		*7*	3	4	5	4	4	5	2
ts-auth-service - 8				4	10	22	3	*8*	7	21	15	13	12	1
ts-basic-service - 9		1	1	1	1	2	4	7	*9*	7	10	10	5	1
ts-admin-basic-info-service - ...	1	2	2	6	20	23	5	21	7	*10*	72	52	37	1
ts-admin-order-service - 11	1	1	1	7	23	21	4	15	10	72	*11*	48	37	1
ts-admin-route-service - 12	1	1	1	10	15	18	4	13	10	52	48	*12*	19	1
ts-admin-travel-service - 13	1	1	1	4	16	23	5	12	5	37	37	19	*13*	1
ts-consign-price-service - 14	1	2	2	1	1	1	2	1	1	1	1	1	1	*14*

55

Candidate DD System Anti-Patterns

- Team Coupling
- Data coupling
- Evolutionary Coupling
- Crossing API
- Retiring Components
- Repetitive Components

56

Lessons Learned

- There is enormous design debt in today's software.
- Yes, in your software.
- That's the bad news.
- The good news: we can do something about it.
- More good news: It is possible to automatically and objectively assess and quantify architecture quality – to find and fix the debt. And we have reason to believe this can be extended to DD systems.

57

Lessons Learned

- And it is possible to bridge the gap. Prior DV8 results were enthusiastically received by the industrial projects.
- Most projects embarked on major refactorings.
- Several companies have incorporated DV8 into their development processes/pipelines.
- There is hope for Web 4.0 architectures.

58

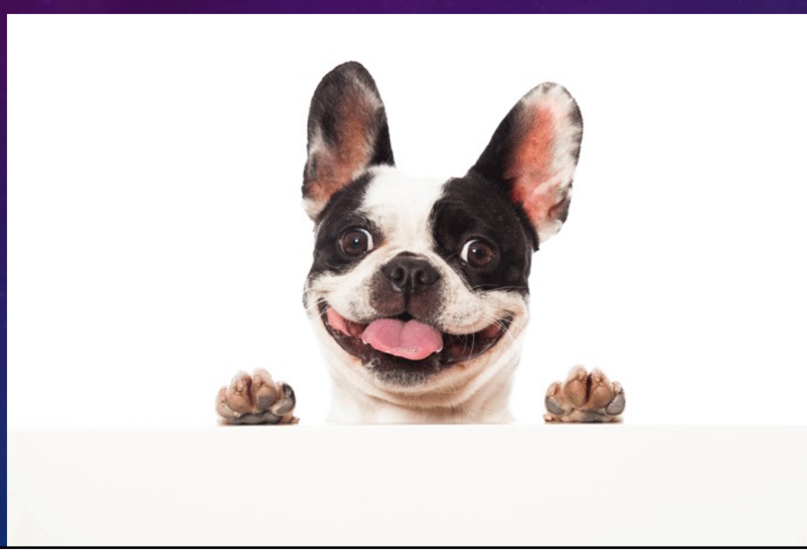
Final Thoughts

- You can't manage it if you don't measure it. Quantification is key.
- If the measurement is not automated it won't be done, or won't be repeatable.
- Incorporating these techniques into build processes and runtime ensures rapid feedback with supporting data.
- This measurement, detection, and quantification practice leads to improved architectures.
- Results must be accompanied by ROI measures, to aid in adoption.

You can get the software—free for academic use—at: https://archdia.com/

59

Thank You!



60

References

[Xiao 2022] L. Xiao, R. Kazman, Y. Cai, R. Mo, Q. Feng, “Detecting the Locations and Predicting the Costs of Compound Architectural Debts”, *IEEE Transactions on Software Engineering*, to appear, 2022.

[Lefever 2021] Jason Lefever, Yuanfang Cai, Humberto Cervantes, Rick Kazman, Hongzhou Fang, “On the Lack of Consensus Among Technical Debt Detection Tools”, ICSE 2021 SEIP

[Jin 2021] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, Q. Zheng, “Service Candidate Identification from Monolithic Systems based on Execution Traces”, *IEEE Transactions on Software Engineering*, 47:5, May, 2021.

[Mo 2021] R. Mo, Y. Cai, R. Kazman, L. Xiao, Q. Feng “Architecture Anti-patterns: Automatically Detectable Violations of Design Principles”, IEEE Transactions on Software Engineering, 47:5, May, 2021.

[Cai 2019] Y. Cai, L. Xiao, R. Kazman, R. Mo, Q. Feng, “Design Rule Spaces: A New Model for Representing and Analyzing Software Architecture”, IEEE Transactions on Software Engineering, 45:7, July, 2019.

References

[Nayebi 2019] Maleknaz Nayebi, Yuanfang Cai, Rick Kazman, Guenther Ruhe, Qiong Feng, Chris Carlson, Francis Chew: "A Longitudinal Study of Identifying and Paying Down Architectural Debt“, ICSE SEIP 2019.

[Feng 2019] Q. Feng, Y. Cai, R. Kazman, D. Cui, T. Liu. H. Fang. "Active Hotspot: An Issue-Oriented Model to Monitor Software Evolution and Degradation“, ASE 2019.

[Mo 2018] Ran Mo, Will Snipes, Yuanfang Cai, Srini Ramaswamy, Rick Kazman, Martin Naedele: " Experiences Applying Automated Architecture Analysis Tool Suites“, ASE 2018.

[Mo 2016] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, Qiong Feng, "Decoupling level: a new metric for architectural maintenance complexity", ICSE 2016: 499-510

[Kazman 2015] Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, Andriy Shapochka, "A Case Study in Locating the Architectural Roots of Technical Debt", ICSE 2015: 179-188

[Xiao 2014] Lu Xiao, Yuanfang Cai, Rick Kazman, "Design rule spaces: a new form of architecture insight", ICSE 2014: 967-977.