Technical Debt in Large Systems: Understanding the cost of software complexity

Dan Sturtevant, Ph.D.

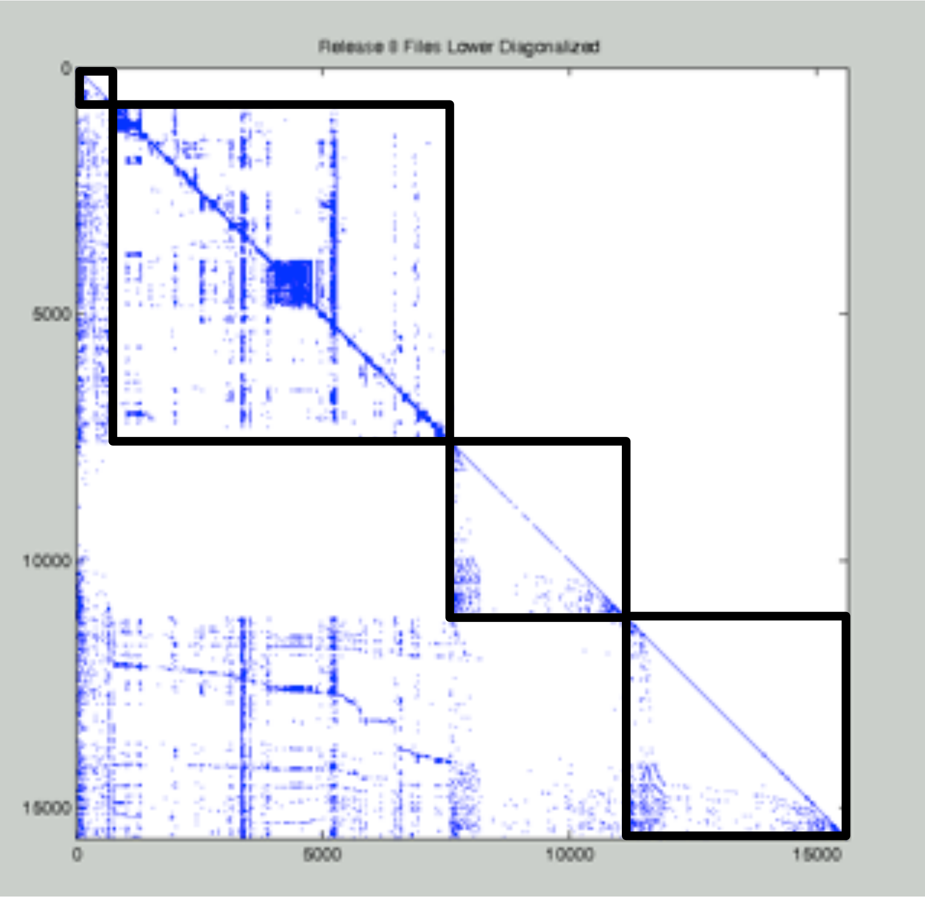
dan.sturtevant@sloan.mit.edu

INCOSE Webinar

June 12th, 2013

With thanks to:

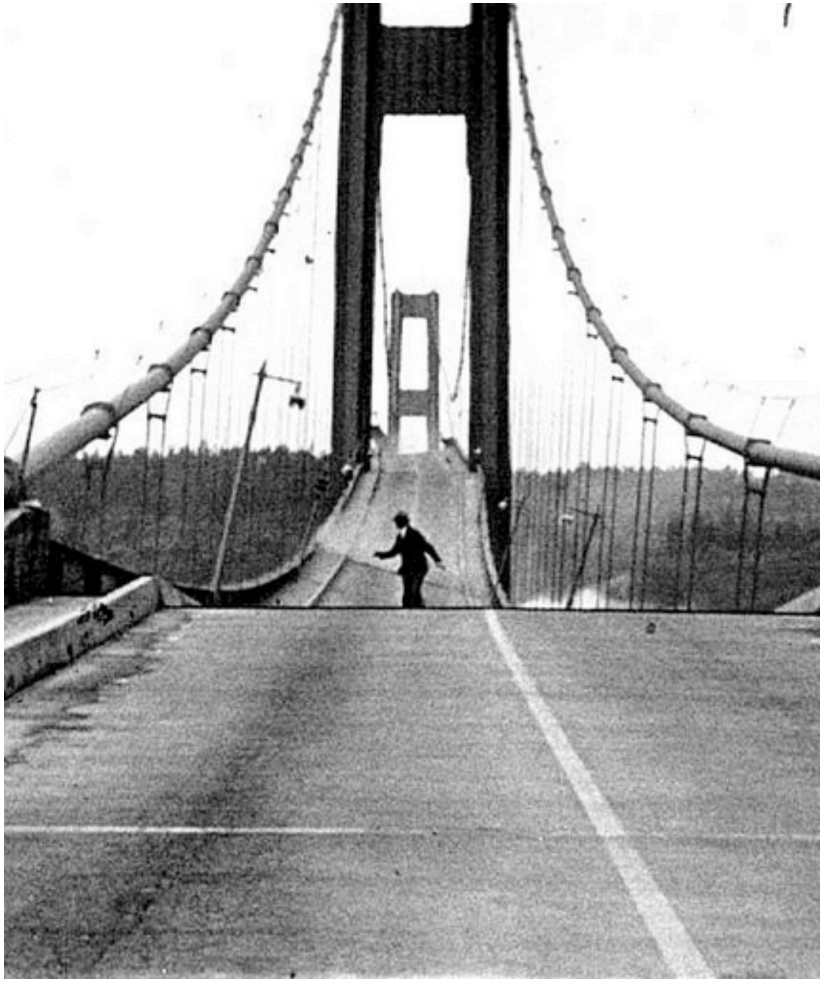
Alan MacCormack, Steven Eppinger, Chris Magee, Daniel Jackson, Carliss Baldwin



Background

Designing and Maintaining Large Systems is Really Hard

- Changing requirements
- Growth and scaling limits
- Changing environment
- Changing technology landscape
- Architectural lock-in
- Loss of information (esp. about design intent)
- Mismatch between organization and architecture
- Change propagation
- Design “decay”
- Emergent properties



Systems are Becoming Larger, Much of the Complexity Now in Code

Large systems are:

- ***Psychologically complex***: No single person can understand how they work. Design process must be split across teams.
- ***Inherently complex***: Whole does not behave in a manner that follows from the independent functioning of its parts.

Software especially so:

- "Software entities are more complex for their size than perhaps any other human construct because no two parts are alike… [they] differ profoundly from computers, buildings, or automobiles, where repeated elements abound" [Brooks]

Large Designs Can Easily Become Unmanageable

Regions within a system that are more ***architecturally complex*** have fewer hierarchical, modular, or layering structures mediating the relationships between system elements.

Regions with high complexity:

- May be initially designed to be integral or entropy may have eroded boundaries later.
- May have higher likelihood of side-effects or change propagation.

Architects Fight to Impose and Maintain Control

They:

- Decompose design into manageable chunks so that teams can act independently and coordinate across boundaries
- Identify the things that should be managed centrally, enforce “design rules.”
- Make sure the system delivers needed functionality, with good performance, at acceptable cost.
- Endow system with various beneficial non-functional properties (“illities”) such as maintainability, flexibility, evolvability, scalability, safety, etc.

They do this by building patterns into designs

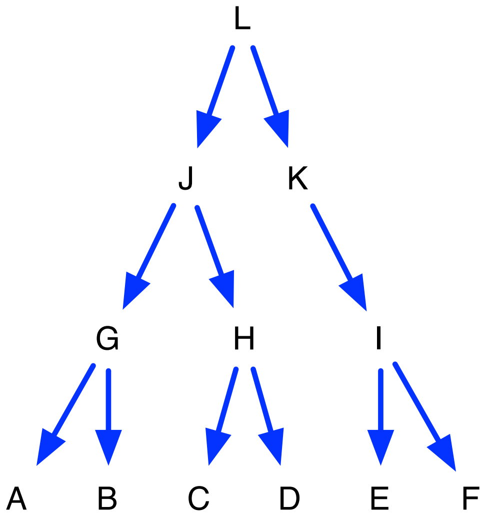
Design Patterns

Naturally evolved organisms and man-made systems are often made up of patterns that help them scale while keeping complexity under control:

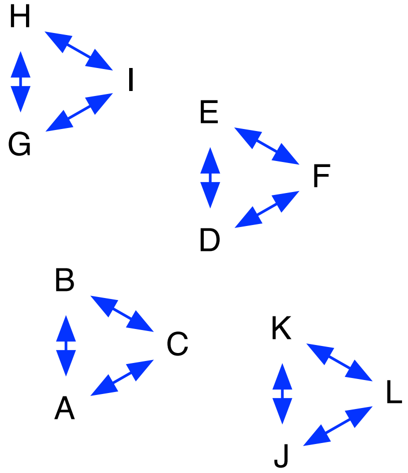
- From a macro-level they are **hierarchical**
- This hierarchy will be made up of **modules**
- This hierarchy may contain **layers** or **abstractions**
- Some components will be **reused**

These features can be reasoned about as specific types of networks or matrices

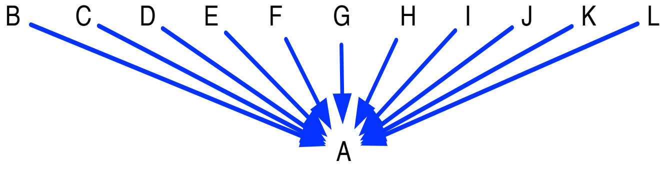
Hierarchies



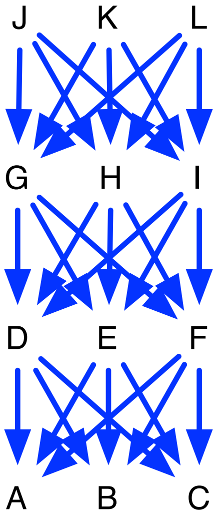
Modules



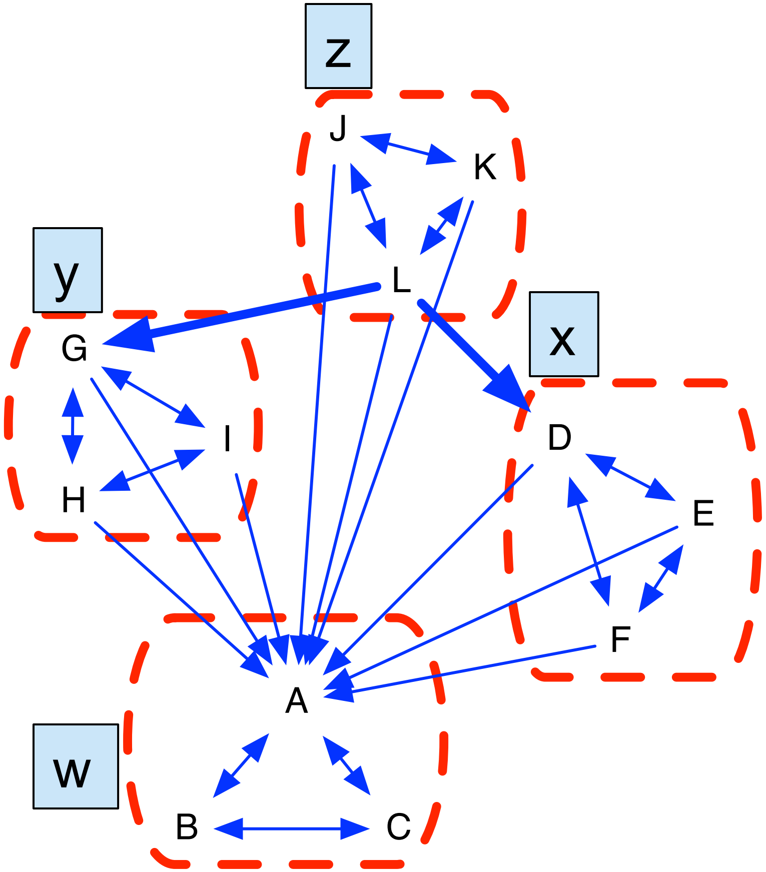
Reuse

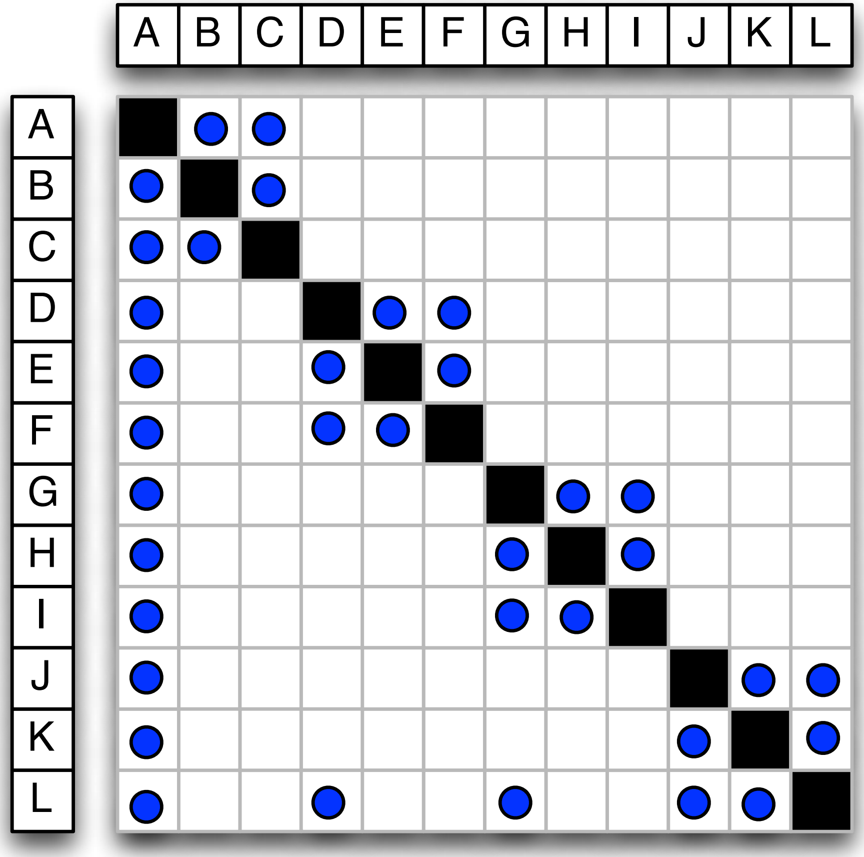


Layers



Combining Hierarchy, Modularity, and Reuse

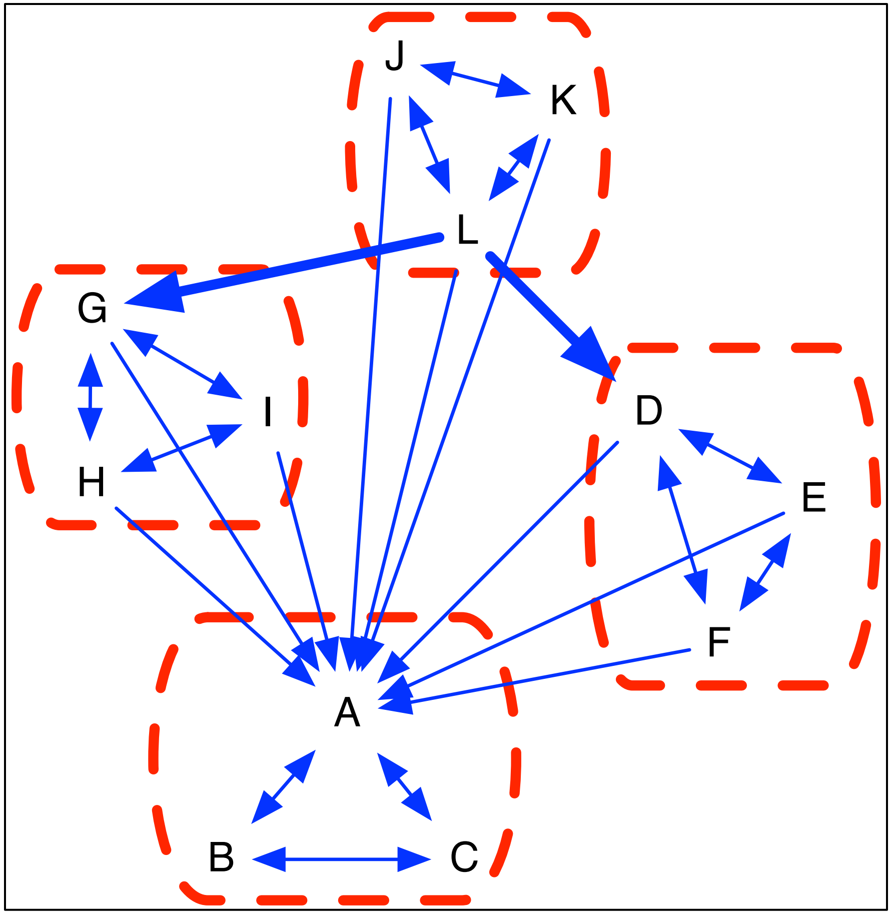


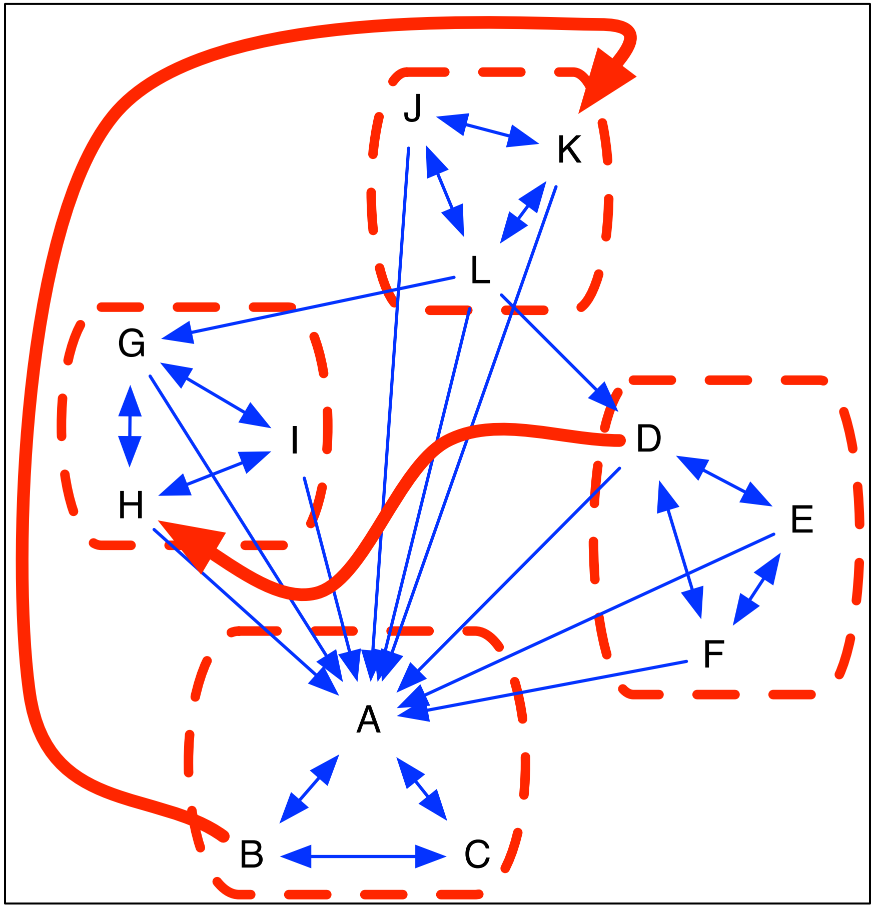


	A	B	C	D	E	F	G	H	I	J	K	L
A	■	●	●									
B	●	■	●									
C	●	●	■									
D	●			■	●	●						
E	●			●	■	●						
F	●			●	●	■						
G	●						■	●	●			
H	●						●	■	●			
I	●						●	●	■			
J	●									■	●	●
K	●									●	■	●
L	●			●			●			●	●	■

Network **Design Structure Matrix**

Why Does This Control Complexity?





Imagine that two people add links which violate design rules

	A	B	C	D	E	F	G	H	I	J	K	L	
A	■	🔵	🔵										3
B	🔵	■	🔵										3
C	🔵	🔵	■										3
D	🔵			■	🔵	🔵							4
E	🔵			🔵	■	🔵							4
F	🔵			🔵	🔵	■							4
G	🔵						■	🔵	🔵				4
H	🔵						🔵	■	🔵				4
I	🔵						🔵	🔵	■				4
J	🔵									■	🔵	🔵	4
K	🔵									🔵	■	🔵	4
L	🔵			🔵			🔵			🔵	🔵	■	6
	12	3	3	4	3	3	4	3	3	3	3	3	47

	A	B	C	D	E	F	G	H	I	J	K	L	
A	■	🔵	🔵										3
B	🔵	■	🔵										3
C	🔵	🔵	■										3
D	🔵	🔵	🔵	■	🔵	🔵							6
E	🔵	🔵	🔵	🔵	■	🔵							6
F	🔵	🔵	🔵	🔵	🔵	■							6
G	🔵	🔵	🔵				■	🔵	🔵				6
H	🔵	🔵	🔵				🔵	■	🔵				6
I	🔵	🔵	🔵				🔵	🔵	■				6
J	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	■	🔵	🔵	12
K	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	■	🔵	12
L	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	■	12
	12	12	12	6	6	6	6	6	6	3	3	3	81

Good

Direct

Indirect

Bad

	A	B	C	D	E	F	G	H	I	J	K	L	
A	■	🔵	🔵										3
B	🔵	■	🔵								🔴		4
C	🔵	🔵	■										3
D	🔵			■	🔵	🔵		🔴					5
E	🔵			🔵	■	🔵							4
F	🔵			🔵	🔵	■							4
G	🔵						■	🔵	🔵				4
H	🔵						🔵	■	🔵				4
I	🔵						🔵	🔵	■				4
J	🔵									■	🔵	🔵	4
K	🔵									🔵	■	🔵	4
L	🔵			🔵			🔵			🔵	🔵	■	6
	12	3	3	4	3	3	4	4	3	3	4	3	49

	A	B	C	D	E	F	G	H	I	J	K	L	
A	■	🔵	🔵	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	12
B	🔵	■	🔵	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	12
C	🔵	🔵	■	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	12
D	🔵	🔵	🔵	■	🔵	🔵	🔴	🔴	🔴	🔴	🔴	🔴	12
E	🔵	🔵	🔵	🔵	■	🔵	🔴	🔴	🔴	🔴	🔴	🔴	12
F	🔵	🔵	🔵	🔵	🔵	■	🔴	🔴	🔴	🔴	🔴	🔴	12
G	🔵	🔵	🔵	🔴	🔴	🔴	■	🔵	🔵	🔴	🔴	🔴	12
H	🔵	🔵	🔵	🔴	🔴	🔴	🔵	■	🔵	🔴	🔴	🔴	12
I	🔵	🔵	🔵	🔴	🔴	🔴	🔵	🔵	■	🔴	🔴	🔴	12
J	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	■	🔵	🔵	12
K	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	■	🔵	12
L	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	■	12
	12	12	12	12	12	12	12	12	12	12	12	12	144

Architectural Complexity and the Power of Indirect Links

	A	B	C	D	E	F	G	H	I	J	K	L	
A	■	🔵	🔵	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	12
B	🔵	■	🔵	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	12
C	🔵	🔵	■	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	12
D	🔵	🔵	🔵	■	🔵	🔵	🔴	🔴	🔴	🔴	🔴	🔴	12
E	🔵	🔵	🔵	🔵	■	🔵	🔴	🔴	🔴	🔴	🔴	🔴	12
F	🔵	🔵	🔵	🔵	🔵	■	🔴	🔴	🔴	🔴	🔴	🔴	12
G	🔵	🔵	🔵	🔴	🔴	🔴	■	🔵	🔵	🔴	🔴	🔴	12
H	🔵	🔵	🔵	🔴	🔴	🔴	🔵	■	🔵	🔴	🔴	🔴	12
I	🔵	🔵	🔵	🔴	🔴	🔴	🔵	🔵	■	🔴	🔴	🔴	12
J	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	■	🔵	🔵	12
K	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	■	🔵	12
L	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	🔵	■	12
	12	12	12	12	12	12	12	12	12	12	12	12	144

Research Question

What costs does architectural complexity within a software system impose on the firm that develops and maintains it?

Three Costs Drivers Considered

1. **Does complexity increase defect density?**
2. **Does complexity impair software developer productivity?**
3. **Does complexity increase the probability of development staff turnover?**

Significance of Research

If we can

- Reliably estimate the architectural complexity of different regions within a software system’s design
- Quantitatively estimate the costs that a firm must shoulder while developing or maintaining complex regions of that code

Then we could

- Make better tradeoffs between time to market, system performance, and complexity management
- Estimate the potential dollar-value of redesign
- Have more success managing refactoring
- Perform due-diligence - audit systems prior to acceptance or acquisition

Analysis Approach

1. Case study of successful firm: “Iron Bridge Software”
2. Selected 8 successive software versions developed in fixed release cycles.
   - Measured complexity from source code
   - Measured development activity during development windows. Extracted info about significant cost / waste drivers
3. Tested relationship between cost and complexity using regression analysis
4. Performed isolated simulations to determine the size of the impact

Data and Data Sources

Source code examined:

- 8 historical releases
- All C++, other significant languages.

Understand Static Analysis Tool:

- McCabe cyclomatic complexity
- File size and other file-based metrics
- Dependency structure, DSMs
  - for C++ code only

Version control system:

- Age of files
- Patches to files, changesets
- Lines changed per patch
  - lines added + deleted
- Link to change tracking ID
- Login for person who submitted patch

Change tracking system:

- Determine if changeset / patch was for enhancement, task, bug fix
  - patches with multiple IDs split contribution among types
- Bug subtypes: Critical, Market

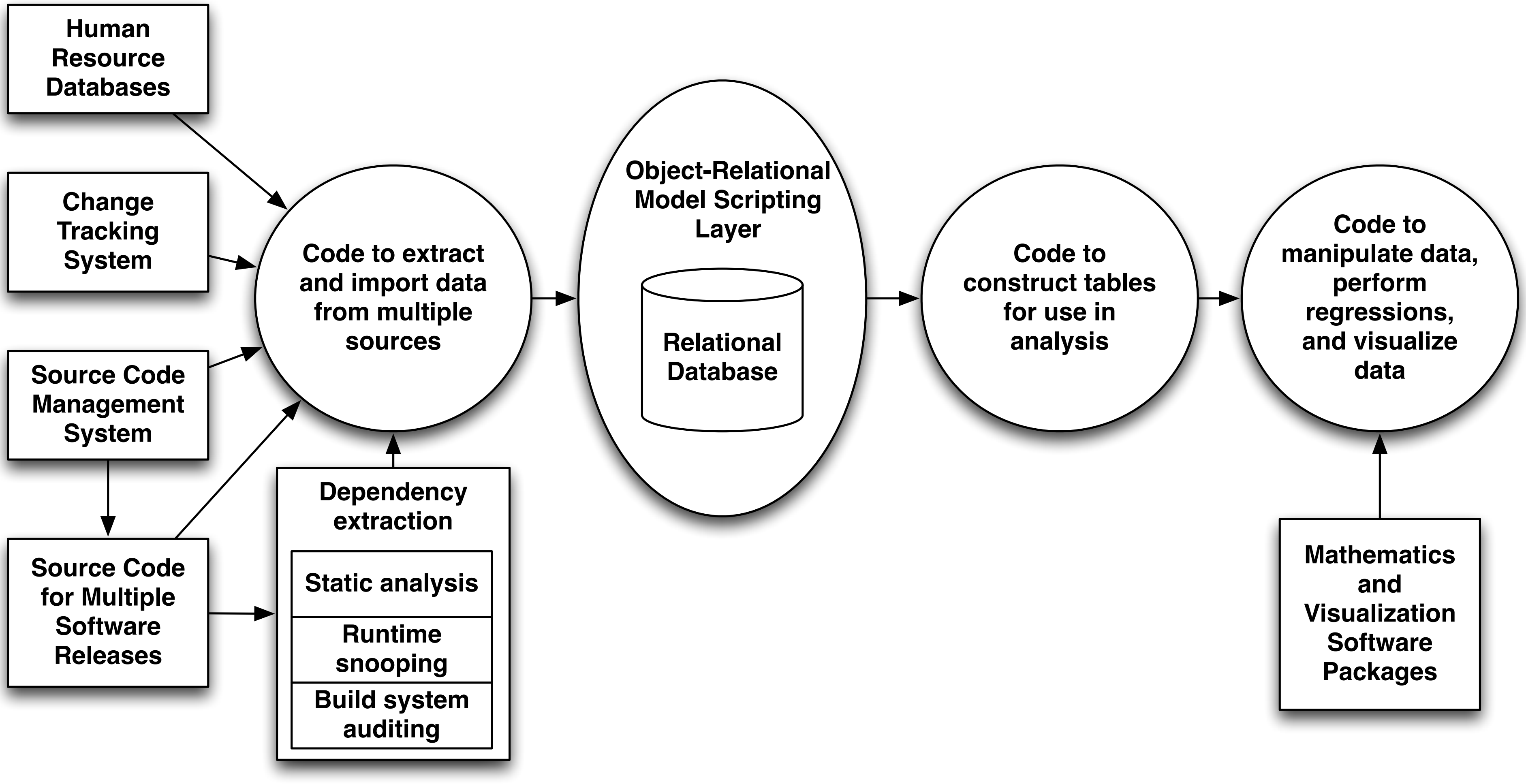
HR Databases:

- Identify software developers
  - distinguish from testers, consultants, etc
- Determine length of employment
- Determine if manager

MATLAB, R, STATA, Lattix, and Ruby graph library code:

- Network manipulation
- Visualization
- Statistical routines

Data Management and Analysis Software Created for this Investigation


Human Resource Databases
Change Tracking System
Source Code Management System
Source Code for Multiple Software Releases
Code to extract and import data from multiple sources
Dependency extraction
Static analysis
Runtime snooping
Build system auditing
Object-Relational Model Scripting Layer
Relational Database
Code to construct tables for use in analysis
Code to manipulate data, perform regressions, and visualize data
Mathematics and Visualization Software Packages

Measuring Complexity and Cost

- Architectural Complexity
- McCabe Cyclomatic Complexity
- More defects
- Lower productivity
- Higher staff turnover

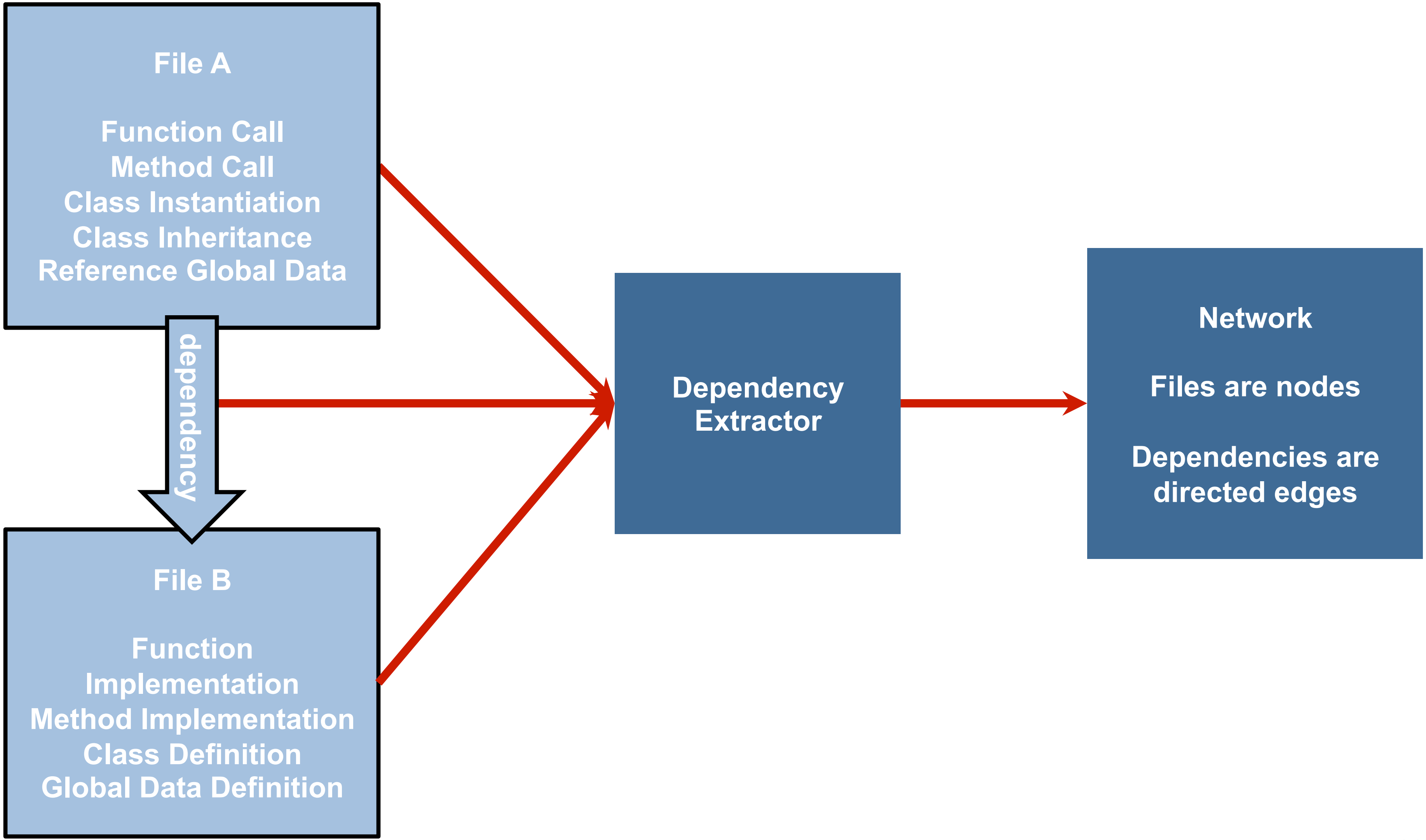
Measuring Complexity and Cost

- **Architectural Complexity**
- McCabe Cyclomatic Complexity
- More defects
- Lower productivity
- Higher staff turnover

The MacCormack, Baldwin, & Rusnak Approach To Architectural Classification

1. Extract dependencies between source code files and construct a network graph
2. Compute the indirect dependency (transitive closure) graph
3. Get “visibility scores” for each file from the indirect dependency graph
4. Classify each file as peripheral, utility, control, or core based on its visibility scores.

Step 1: Extract Dependencies Between Files and Construct a Network Representation



File A

Function Call
Method Call
Class Instantiation
Class Inheritance
Reference Global Data

File B

Function Implementation
Method Implementation
Class Definition
Global Data Definition

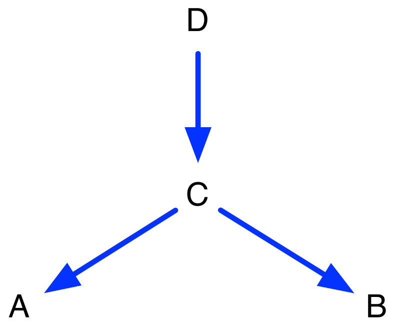
Dependency Extractor

Network

Files are nodes

Dependencies are directed edges

Step 2: Compute the Transitive Closure of the Graph

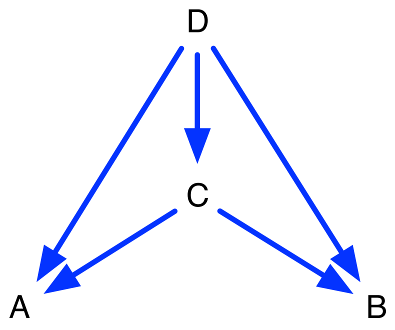


Direct Dependencies

	A	B	C	D
A	■			
B		■		
C	●	●	■	
D			●	■

Traditional Network View

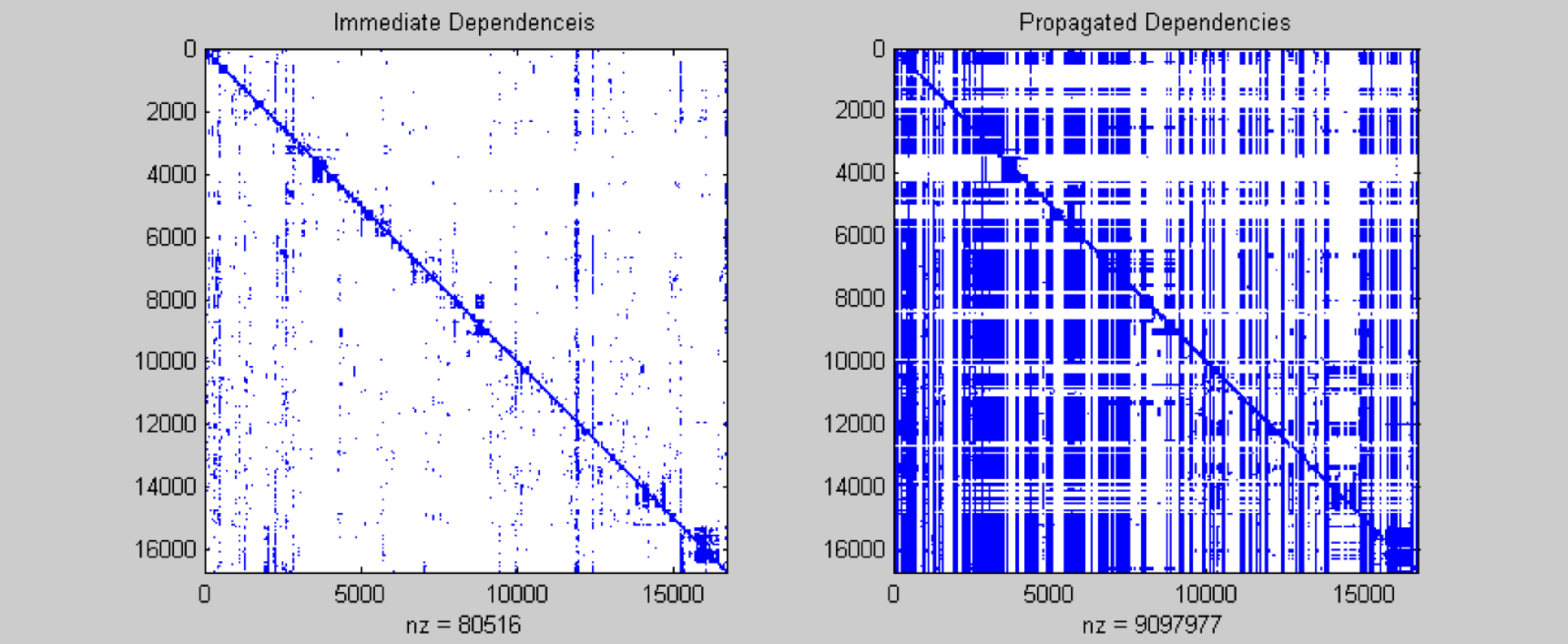
Design Structure Matrix



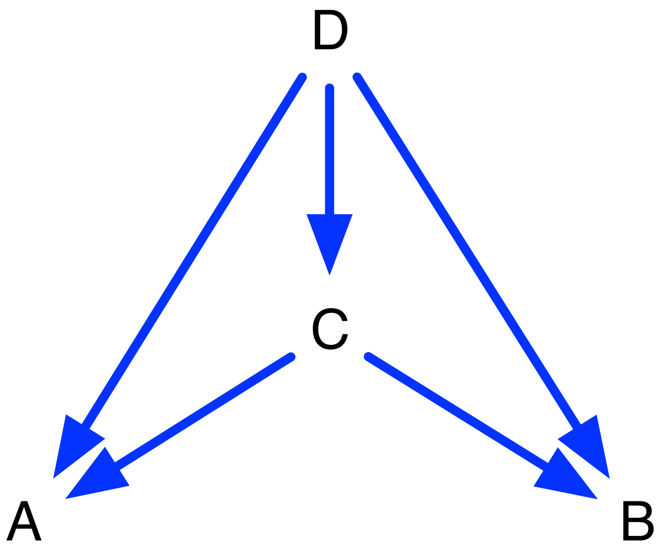
Indirect Dependencies

	A	B	C	D
A	■			
B		■		
C	●	●	■	
D	●	●	●	■

Example: Direct & Indirect Dependencies for a Commercial Software System



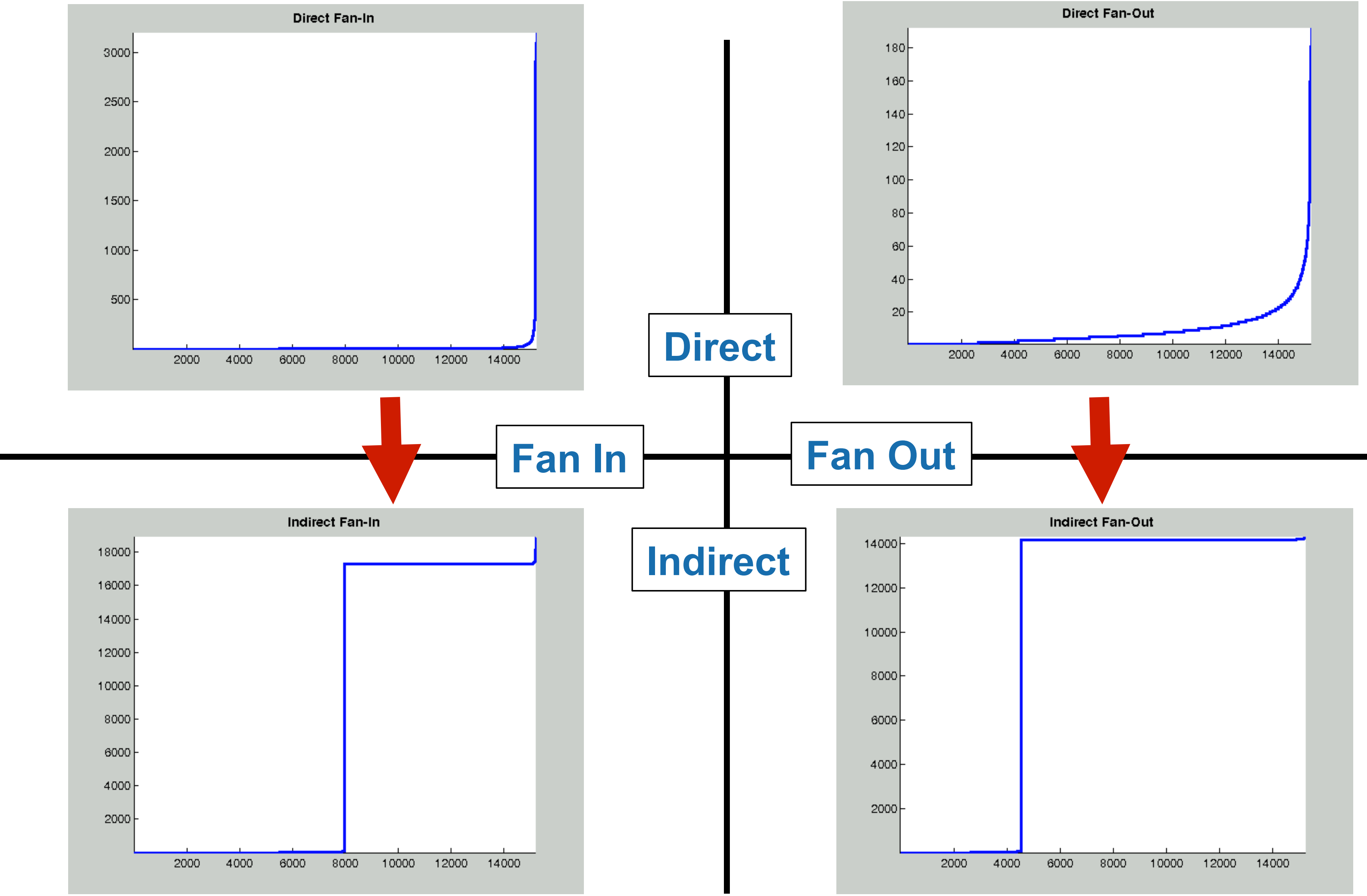
Step 3: Get “Visibility Scores” for Each File From Indirect Dependency Graph



	A	B	C	D
A	■			
B		■		
C	●	●	■	
D	●	●	●	■

	Visibility Fan In	Visibility Fan Out
File A	3	1
File B	3	1
File C	2	3
File D	1	4

Example: Scores for Release 7 C++ Files



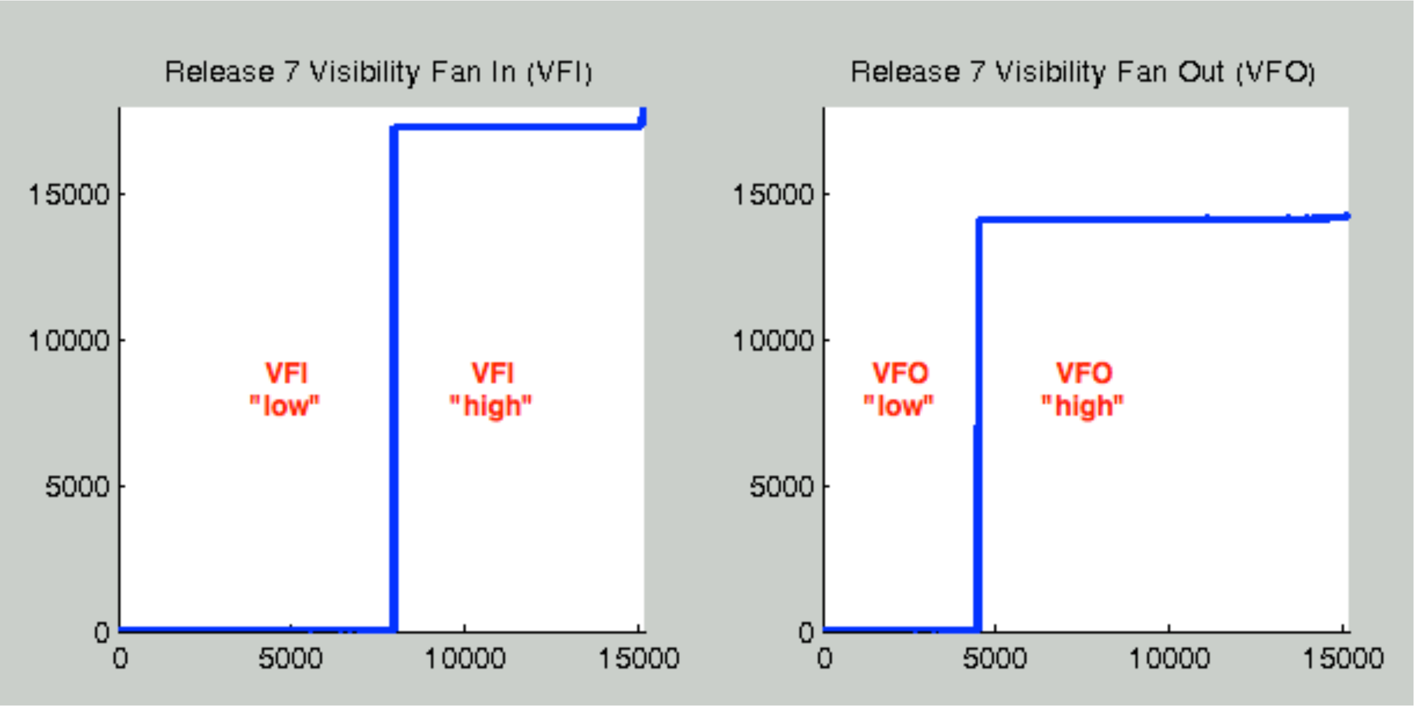
Direct

Fan In

Fan Out

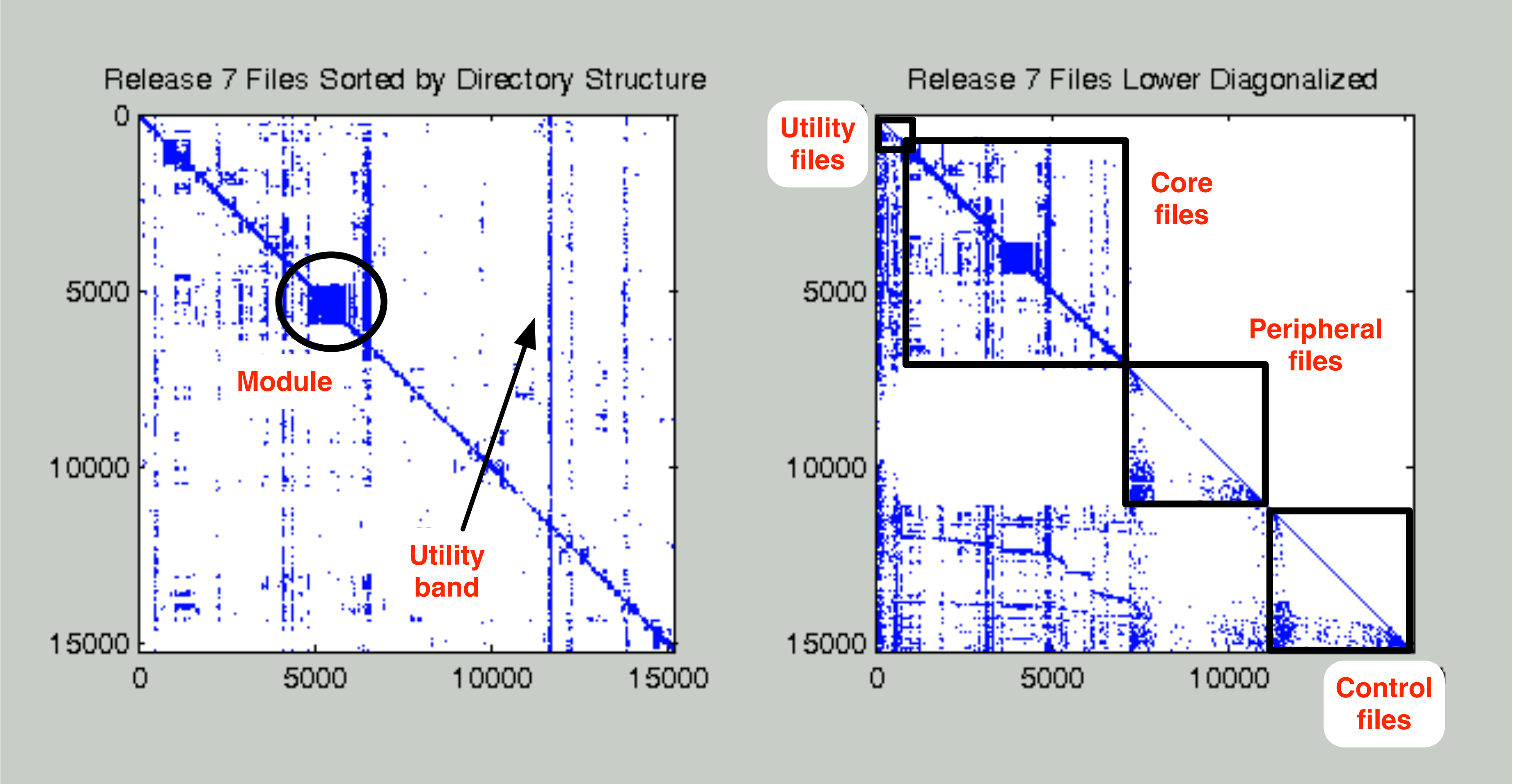
Indirect

Step 4: Classify Files by Indirect Scores



If a file has	VFO "Low"	VFO "Low"	VFO "High"	VFO "High"
and	VFI "Low"	VFI "High"	VFI "Low"	VFI "High"
Then the file is considered	**Peripheral**	**Utility**	**Control**	**Core**

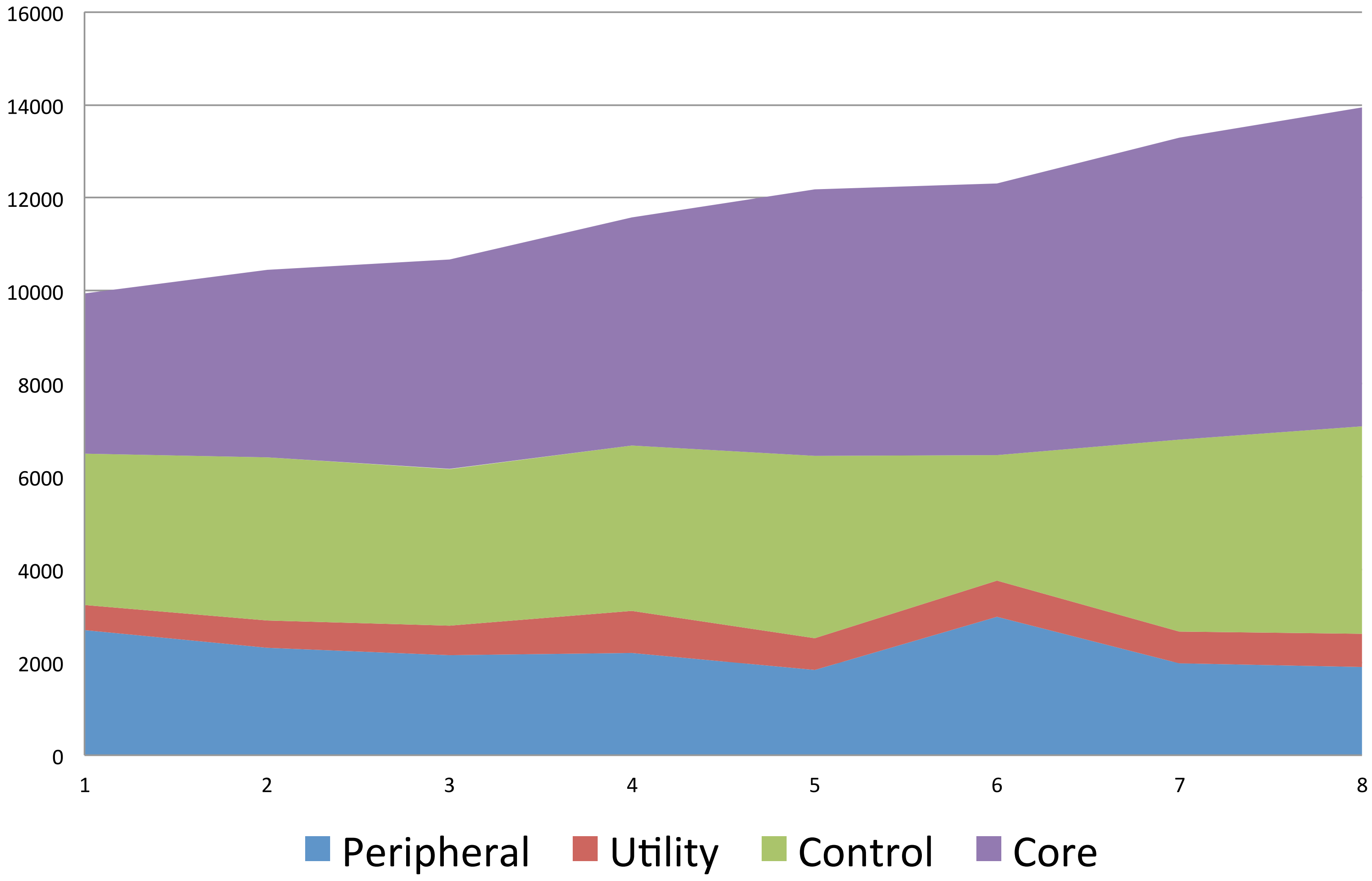
Example: Release 7 C++ Direct DSM File-system (left) and Sorted (right)



Meaning of Architecture Categories

- **Peripheral** files do not influence and are not influenced by much of the rest of the system.
- **Utility** files are relied upon (directly or indirectly) by a large portion of the system but do not depend upon many other files themselves. They have the potential to be self-contained and stable.
- **Control** files invoke the functionality or accesses the data of many other nodes. They may coordinate collective behavior so as to bring about the system level function.
- **Core** files connect to form highly integral clusters, often containing large cycles in which components are directly or indirectly co-dependent. These regions are hard to decompose into smaller parts and may be unmanageable if they become too large.

Files Counts By Architectural Complexity Type



Measuring Complexity and Cost

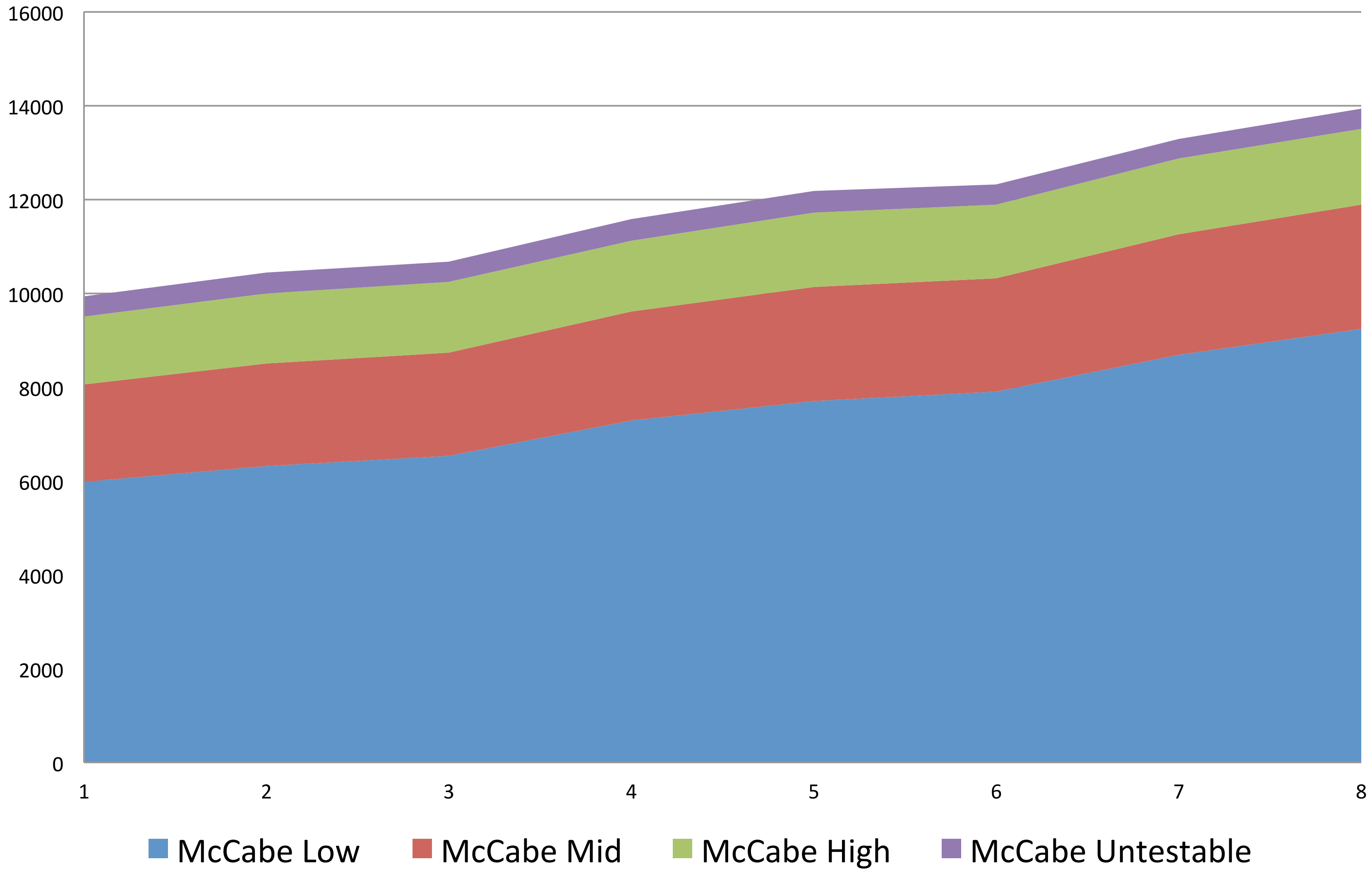
- Architectual Complexity
- **McCabe Cyclomatic Complexity**
- More defects
- Lower productivity
- Higher staff turnover

Measuring Cyclomatic Complexity For a File

- Find the McCabe score for the most complex function contained in a file
- Classify the file based on its score:

McCabe Score	McCabe Classification
1-10	Low
11-20	Mid
21-50	High
51-Inf	Untestable

Files By McCabe Type



Measuring Complexity and Cost

- Architectual Complexity
- McCabe Cyclomatic Complexity
- **More defects**
- Lower productivity
- Higher staff turnover

Analyzing Complexity & Quality

94,364 source files observed over 8 software releases
For each:

Measure

- Architectural complexity
- McCabe complexity

Count

- Number of changes made to fix bugs.
- Number of lines changed to fix those bugs.

Control for

- Number of changes made to implement features or do other non-bug related tasks
- File size
- File age
- Software version being released

Regression Models

Defects go:

- Up with file size
- Up with development activity in file
- Down with file age
- Up with McCabe complexity
- **Up with Architectural complexity**

Negative Binomial regressions used because dependent variable is count data that is overdispersed

Regression Model Details

Predicting LOC changed in a file to fix bugs. (Negative binomial model)

Parameter	**Model 1: controls**	**Model 2: cyclomatic complexity**	**Model 3: architectural complexity**	**Model 4: combined**
LOC in file	0.00156486***	0.0011712***	0.00143183***	0.00104115***
Non-bug lines change	0.00372536***	0.00353601***	0.00355368***	0.00335322***
File age	-0.10050305***	-0.11730352***	-0.1026859***	-0.11853279***
Cyclomatic: mid		0.774729***		0.70392074***
Cyclomatic: high		0.93363115***		0.95513134***
Cyclomatic: very high		0.91923347***		0.96444595***
Architectural: utility			0.2018549*	0.35797922***
Architectural: control			0.94111466***	0.84721344***
Architectural: core			1.14823521***	1.14683088***
Residual Deviance	30370	30418	30428	30475
Degrees of Freedom	94353	94350	94350	94347
AIC	227861	227512	227403	227079
Theta	0.030212	0.030692	0.030836	0.031295
Std-err	0.000285	0.00029	0.000291	0.000295
2 x log-lik	-227837.302	-227482.025	-227373.406	-227042.861

N = 94364 files observations (from 8 releases)

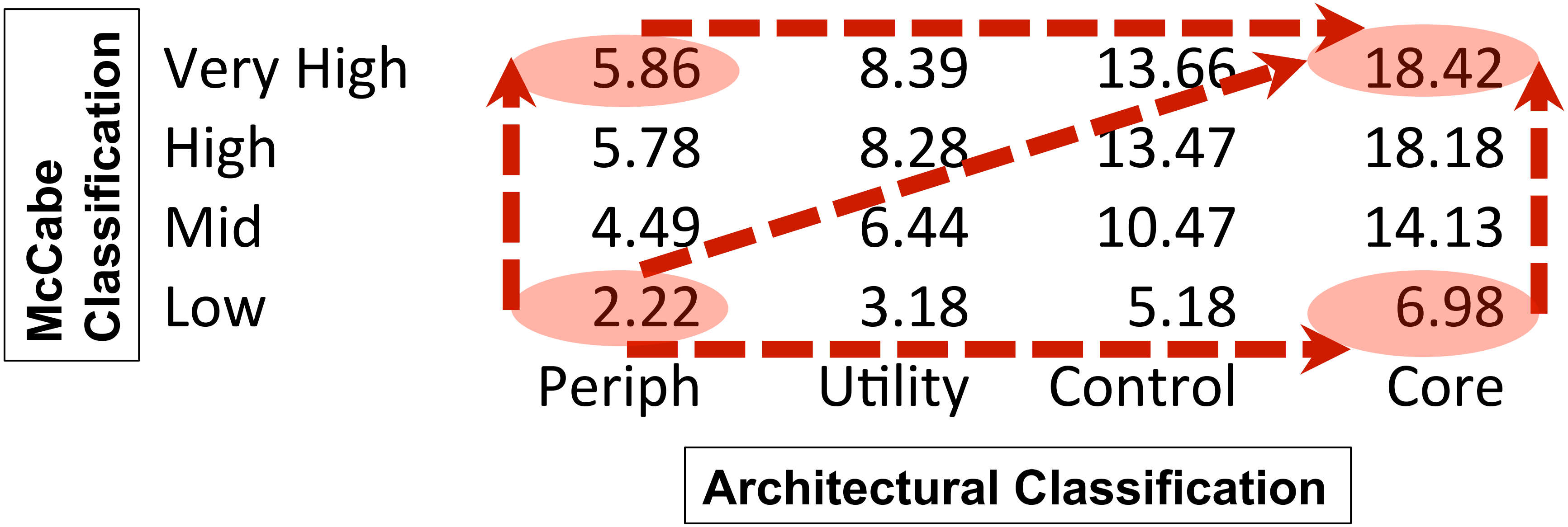
Dummy variables for each of 8 releases omitted.

*Significance codes: .<0.1, *<0.05, **<0.01, ***<0.001*

Using Simulations to Interpret Results

- Once regression complete, run simulations holding control variables constant and test impact of varying predictors
- Control variables set to mean values:
  - File size: 550 LOC
  - Non bug-fix patches per file: 0.47
  - Non bug-fix LOC submitted per file: 33
  - File age: 4.198 years
- Test all combinations of complexity scores:
  - McCabe: Low, Mid, High, Untestable
  - Architectural: Peripheral, Utility, Control, Core
- See how bugs counts are affected

Interpreting Results via Simulation: Defect Density



McCabe Classification	Periph	Utility	Control	Core
Very High	5.86	8.39	13.66	18.42
High	5.78	8.28	13.47	18.18
Mid	4.49	6.44	10.47	14.13
Low	2.22	3.18	5.18	6.98

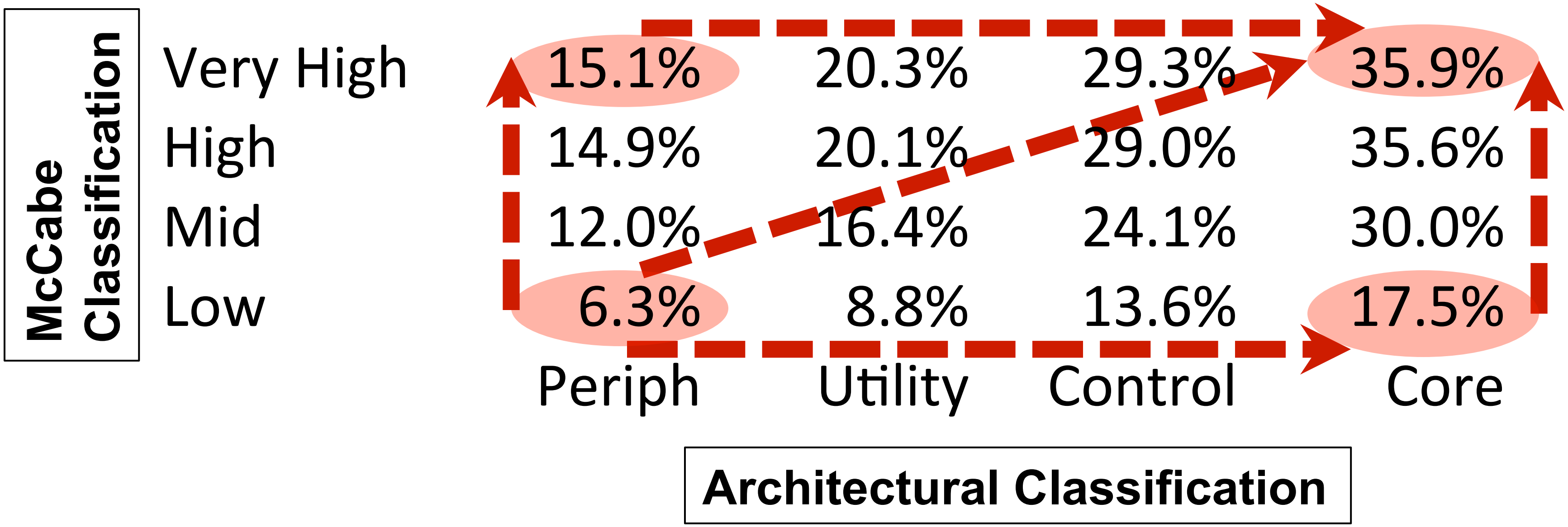
Architectural Classification

McCabe: 2.6X bugs

Architectural: 3.1X bugs

Combined: 8.3X bugs

Interpreting Results via Simulation: Defect Density



McCabe Classification	Periph	Utility	Control	Core
Very High	15.1%	20.3%	29.3%	35.9%
High	14.9%	20.1%	29.0%	35.6%
Mid	12.0%	16.4%	24.1%	30.0%
Low	6.3%	8.8%	13.6%	17.5%

Architectural Classification

McCabe: 2.6X bugs

Architectural: 3.1X bugs

Combined: 8.3X bugs

Measuring Complexity and Cost

- Architectural Complexity
- McCabe Cyclomatic Complexity
- More defects
- **Lower productivity**
- Higher staff turnover

Analyzing Complexity & Developer Productivity

Sample: 478 developer-releases, 178 unique people
For each:

Measure

- % effort working in files with high architectural complexity ("Core" files)
- % effort working in files with high cyclomatic complexity

Count

- Number of lines of code contributed during the release

Control for

- Time with company
- Is a manager?
- % effort working in new files
- % effort fixing bugs
- Software version being released
- Person-specific dummy

Regression Models

Productivity goes:

- Up with years employed
- Up with work in new (rather than legacy files)
- Down with work on bug fixes (rather than features or tasks)
- **Down with work in architecturally complex files**
- No relationship found with cyclomatic complexity

Negative Binomial fixed-effects panel data regressions used because:

- *Dependent variable is count data that is overdispersed*
- *Tests differences **within** the same developer over multiple releases.*

Regression Model Details

Predicting LOC produced by a developer to implement enhancements for one release. (Negative binomial panel data model)

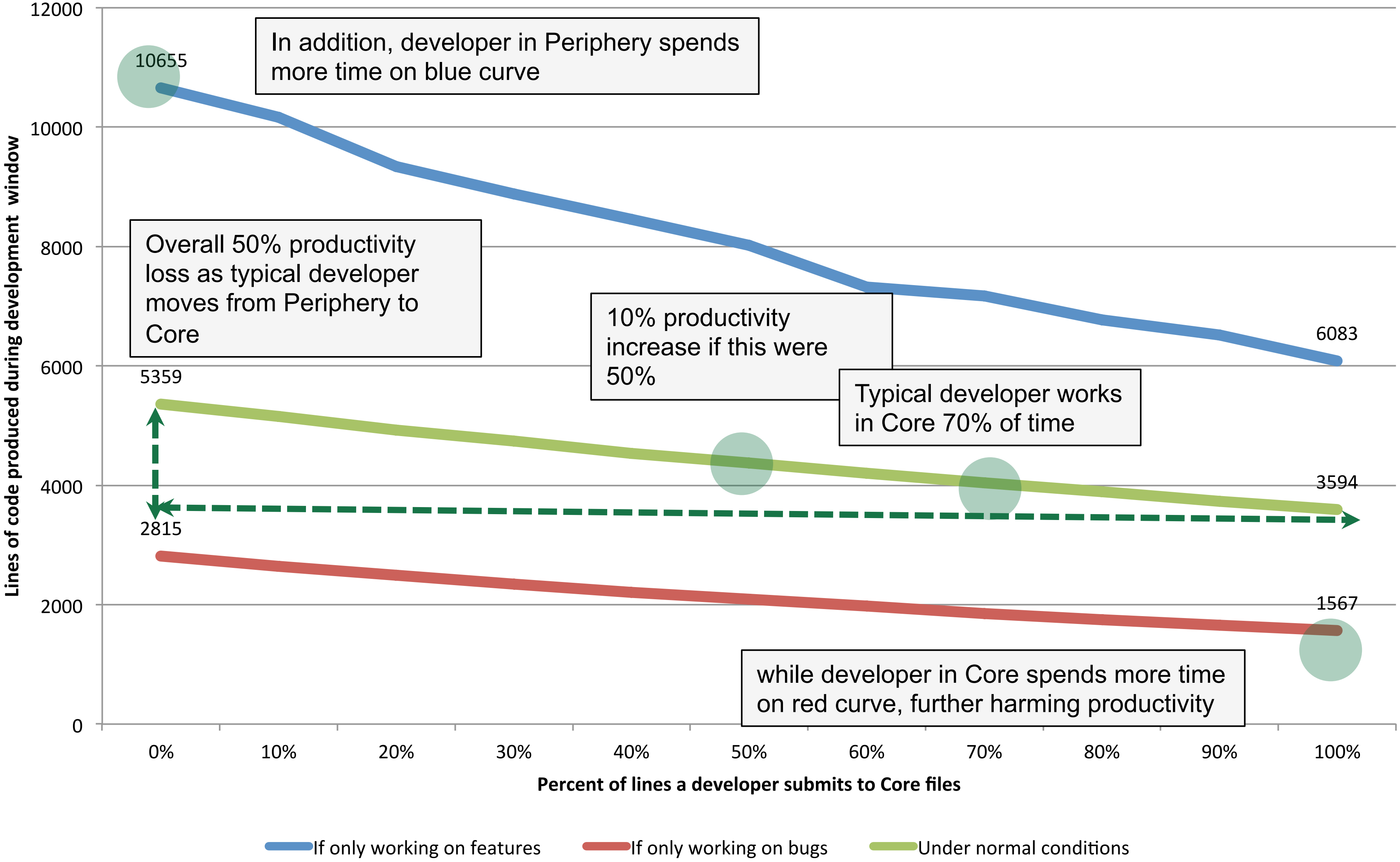
Parameter	Model 1: developer attributes	Model 2: type of work	Model 3: cyclomatic complexity	Model 4: all controls	Model 5: architectural complexity	Model 6: combined
Lines for bug fixes	-0.000071	-0.000068	-0.000060	-0.000067	-0.000077 .	-0.000078 .
Log(years employed)	0.279600			0.492500		0.483700
Is manager?	-0.283000			-0.251600		-0.292900
Pct lines in new files		1.801000 ***		1.699000 ***		1.714000 ***
Pct lines high cyclomatic			-1.166011 ***	-0.648300 .		-0.613000 .
Pct lines in core					-0.610943 .	-0.618600 *
Residual Deviance	560.77	558.46	560.60	558.32	560.71	558.13
Degrees of Freedom	290.00	291.00	291.00	288.00	291.00	287.00
AIC	8170.66	8135.14	8162.14	8136.78	8166.87	8135.75
Theta	0.85	0.90	0.86	0.91	0.85	0.92
Std-err	0.05	0.05	0.05	0.05	0.05	0.05
2 x log-lik	-7792.66	-7759.14	-7786.14	-7754.78	-7790.87	-7751.75

N = 478 developer/releases

Dummy variables for each of 8 releases omitted. Dummy variables for each of 178 developers omitted.

*Significance codes: .<0.1, *<0.05, **<0.01, ***<0.001*

Interpreting Developer Productivity Results via Simulation



In addition, developer in Periphery spends more time on blue curve

Overall 50% productivity loss as typical developer moves from Periphery to Core

10% productivity increase if this were 50%

Typical developer works in Core 70% of time

while developer in Core spends more time on red curve, further harming productivity

Measuring Complexity and Cost

- Architectual Complexity
- McCabe Cyclomatic Complexity
- More defects
- Lower productivity
- **Higher staff turnover**

Analyzing Complexity & Staff Turnover

Sample of 108 people. For each:

Measure

- % effort working in files with high architectural complexity (“Core” files)
- % effort working in files with high cyclomatic complexity

Determine

- Whether person left the company (voluntarily or involuntarily) over 8 year period

Control for

- Length of employment
- Managerial status
- % effort developing in new files rather than working in legacy code
- % effort fixing defects rather than implementing features or doing other non-bug related coding tasks

Regression Models

Staff turnover goes:

- Down with productivity
- Down with managerial status *(marginal, P value is 11%)*
- **Up with work in architecturally complex files**

Did not establish a link for these factors:

- Years employed
- Bug fix vs. Enhancement work
- New file vs. Legacy work
- Work in files with High/Untestable McCabe complexity

Logistic model used because dependent variable is binary outcome

Regression Models

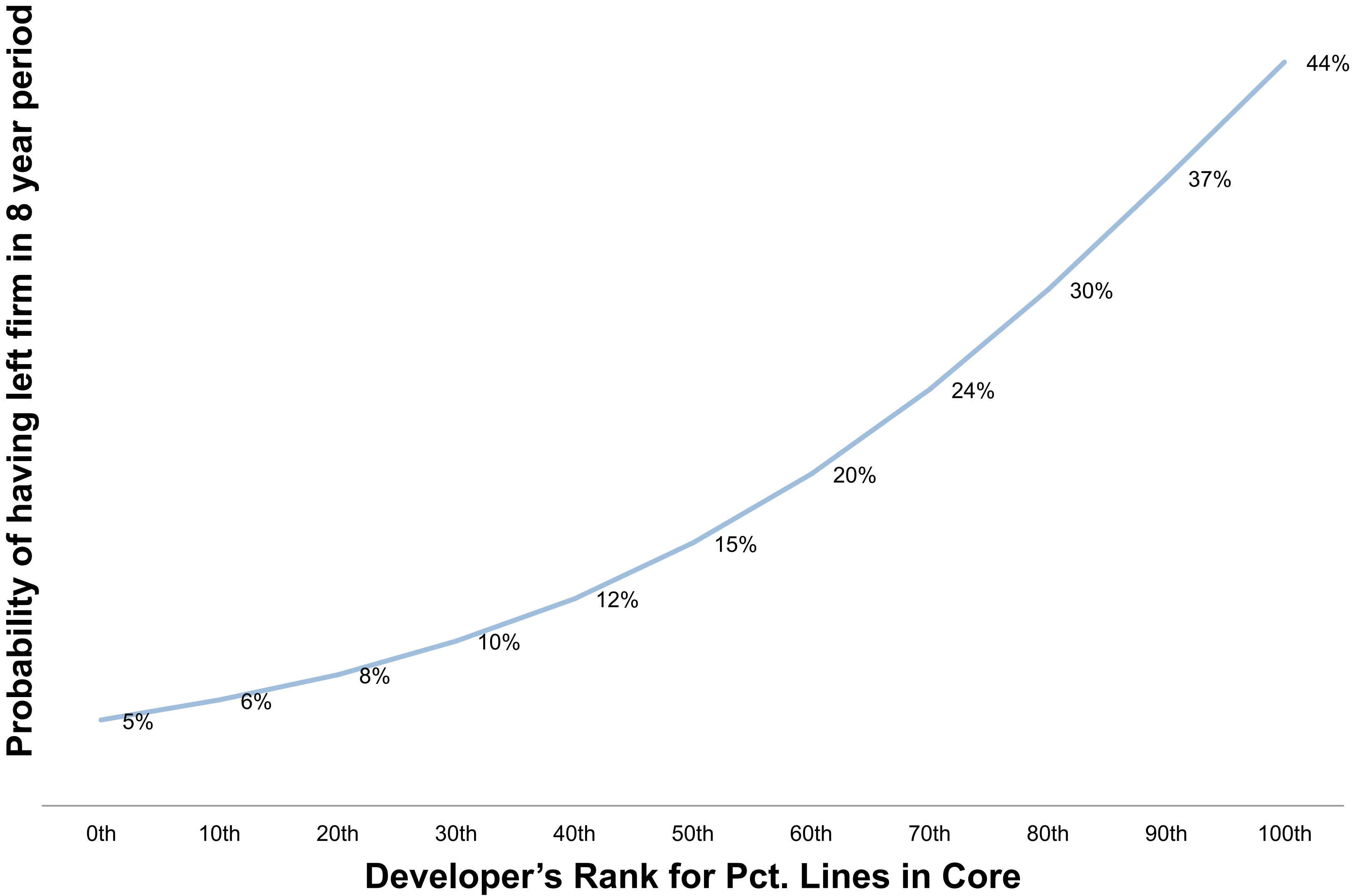
Predicting turnover among developers (Logistic model)

Parameter	**Model 1: developer attributes**	**Model 2: developer productivity**	**Model 3: type of work**	**Model 4: cyclomatic complexity**	**Model 5: all controls**	**Model 6: architectural complexity**	**Model 7: full**
Years employed	-0.0535				-0.0784		-0.0786
Is manager?	-0.8123				-1.0545		-1.1398
Lines produced per release		-0.0002 .			-0.0002		-0.0003 .
Fraction of lines to fix bugs			1.0526		0.6694		0.0579
Fraction of lines in new files			-0.1638		-0.6652		-1.3219
Fraction lines in high McCabe files				-0.0954	-0.2562		-1.4194
Fraction of lines in core files						3.5440 *	4.1114 *
Residual Deviance	91.525	90.884	93.112	94.03	86.656	87.181	78.632
Degrees of Freedom	105	106	105	106	101	106	100
AIC	97.525	94.884	99.112	98.03	100.66	91.181	94.632

N = 108 software developers

*Significance codes: .<0.1, *<0.05, **<0.01, ***<0.001*

Interpreting Development Staff Turnover Results via Simulation



Summary of Research Conclusions

Results

Architectual complexity is expensive

A firm can think about ways to estimate the savings that would result from successful redesign efforts by translating cost-driver information into dollar figures.

More defects

- 3.1X increase between periphery and core
- 2.6X for McCabe, combined effect 8.3X

Lower productivity

- 50% decline as developer moves from periphery to core (conservatively)

Higher staff turnover

- 10x increase in voluntary and involuntary terminations

Contributions

Academic literature:

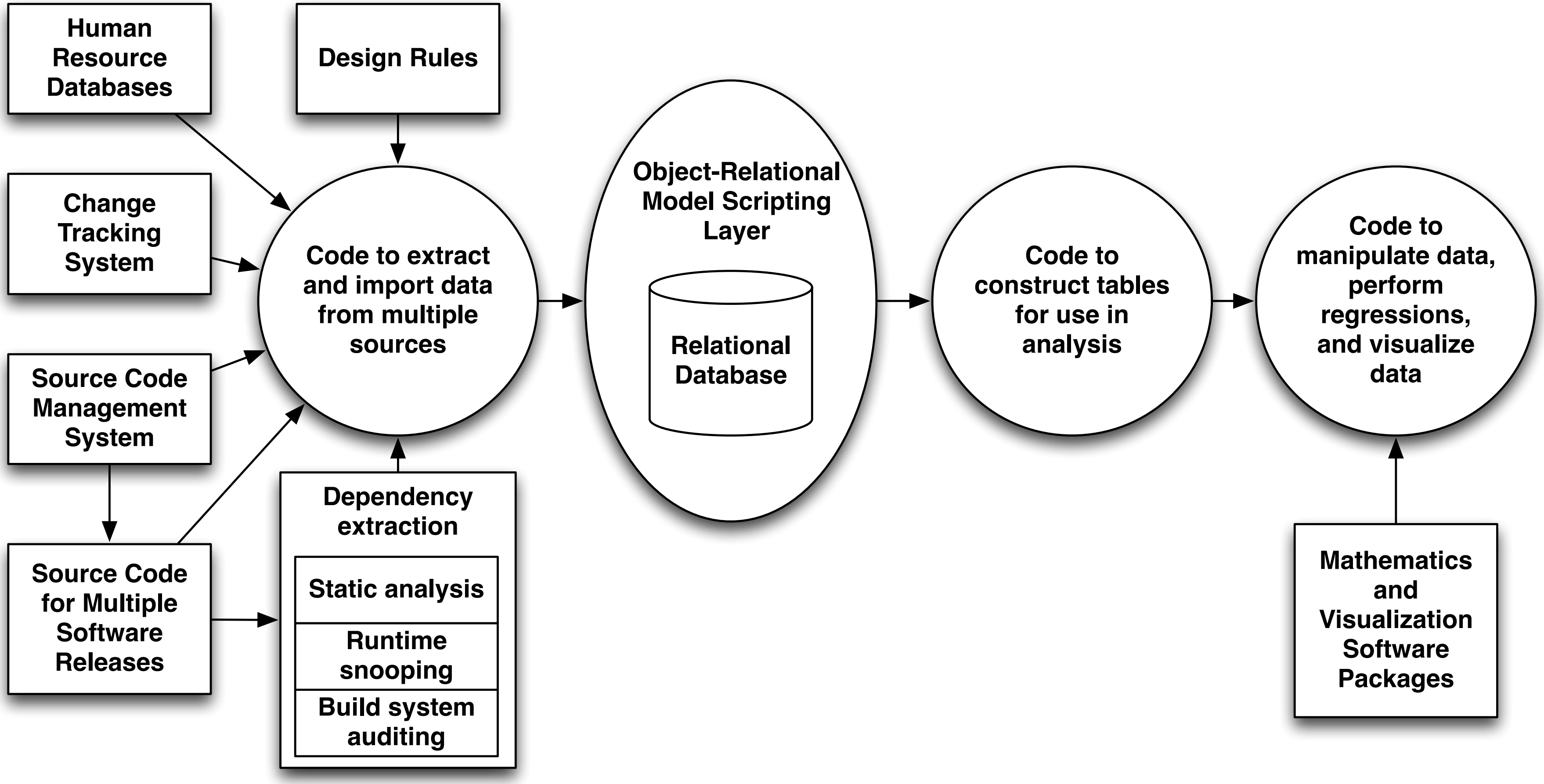
- Demonstration that architecture strongly impacts defect density. MacCormack metrics are as good as (or better than) the popular McCabe cyclomatic complexity metric at predicting bugs.
- Empirical evidence that architecture matters a lot.
- First study to link architecture to individual productivity.
- First study to link architecture to staff morale and turnover.

Managerial practice

- Demonstration that architecture impacts financial performance.
- Points towards method of estimating financial value of redesign.
- Identifies a good predictor of developer productivity; helps to address a fundamental weakness of commonly used software estimation models such as COCOMO
- *Suggests means of managing redesign efforts and evaluating their effectiveness.*

How Do I Improve My System?

Using a data management and analysis system similar to the one developed for this research, an organization would have a better ability to visualize software structure, track complexity and its costs, and attack root causes behind defects and project failures.


Human Resource Databases
Design Rules
Change Tracking System
Source Code Management System
Source Code for Multiple Software Releases
Code to extract and import data from multiple sources
Dependency extraction
Static analysis
Runtime snooping
Build system auditing
Object-Relational Model Scripting Layer
Relational Database
Code to construct tables for use in analysis
Code to manipulate data, perform regressions, and visualize data
Mathematics and Visualization Software Packages

Thank you

If you have any questions or comments, please contact Dan Sturtevant at dan.sturtevant@sloan.mit.edu

To get a copy of the dissertation, go here: https://wikis.mit.edu/confluence/display/ESDRATA/Dan+STURTEVANT