Identifying and Quantifying Architectural Debt

Lu Xiao, Yuanfang Cai
Drexel University
Philadelphia, PA, USA
{lx52,yc349}@drexel.edu

Rick Kazman
University of Hawaii &
SEI/CMU
Honolulu, HI, USA
kazman@hawaii.edu

Ran Mo, Qiong Feng
Drexel University
Philadelphia, PA, USA
{rm859,qf28}@drexel.edu

ABSTRACT

Our prior work showed that the majority of error-prone source files in a software system are architecturally connected. Flawed architectural relations propagate defects among these files and accumulate high maintenance costs over time, just like debts accumulate interest. We model groups of architecturally connected files that accumulate high maintenance costs as *architectural debts*. To quantify such debts, we formally define *architectural debt*, and show how to automatically identify debts, quantify their maintenance costs, and model these costs over time. We describe a novel *history coupling probability* matrix for this purpose, and identify architecture debts using 4 patterns of architectural flaws shown to correlate with reduced software quality. We evaluate our approach on 7 large-scale open source projects, and show that a significant portion of total project maintenance effort is consumed by paying interest on architectural debts. The top 5 architectural debts, covering a small portion (8% to 25%) of each project's error-prone files, capture a significant portion (20% to 61%) of each project's maintenance effort. Finally, we show that our approach reveals how architectural issues evolve into debts over time.

CCS Concepts

•Software and its engineering → Software architectures;

Keywords

Software Architecture, Software Quality, Technical Debt

1. INTRODUCTION

Technical Debt (TD) is a metaphor to describe the long-term consequences of shortcuts taken in coding activities to achieve near-term goals [7]. Debts are introduced when developers opt for "quick and dirty" solutions, but postpone longer-term improvements. Our prior work [26] showed that most error-prone files in a project are architecturally connected through flawed relations. These flawed relations can propagate defects among large numbers of files, and incur increasing maintenance costs over time. A flawed architecture relation is similar to a *debt* in that it accumulates *penalty*, in terms of maintenance costs, the same way a debt accumulates interest. We call such flaws *architectural debts*.

Although the concept of TD has been influential, it has until now largely been a metaphor. The differences with real (financial) debt are crucial. A real debt always starts from a *principal*, and grows with a certain *interest rate*. How to quantify the principal and interest rate in software investments has been a challenge. Our goal is to advance the understanding and management of *architectural debt*, a type of technical debt, by quantifying it.

We define the concept of *architectural debt* (ArchDebt) as a tuple consisting of: 1) a group of architecturally connected files, and 2) a model of the maintenance cost growth for such files. Based on this definition, we contribute an approach to automatically locate architecture debts. Once we locate each debt we model its growth using regression models. Our approach to identify ArchDebt has two parts. We first create a novel *history coupling probability* (HCP) matrix to manifest the probability of changing one file when another file is changed. Then we index file groups through the lens of 4 patterns of prototypical architectural flaws that have been shown to correlate with reduced software quality [21], namely *hub*, *anchor-submissive*, *anchor-dominant*, and *modularity violation*.

Given an ArchDebt, we quantify the maintenance costs (approximated by bug-fixing churn) spent on the files involved in the debt. From the costs incurred in each release, we can model the growth trend using linear, logarithmic, exponential or polynomial regression models. These models represent coherent scenarios of stable, reducing, increasing, and fluctuating maintenance interest rates respectively. Finally, we rank the identified architectural debts according to the maintenance costs they have accumulated.

We have evaluated our approach using seven Apache open source projects, and identified many instances (between 74 and 204) of ArchDebts in each project. The results show that a significant portion (from 51% to 85%) of the maintenance effort in each project is consumed by paying interest on these debts, and that non-trivial portions (20% to 61%) of the maintenance effort is consumed by just five ArchDebts, which represent a small portion of all error-prone files. Our evaluation also revealed that about half of the identified debts fit linear regression models, indicating a

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884825

steady increase in the penalty that these debts incur. About 1/3, 1/10, and less than 1/10 of all debts fit polynomial, logarithmic, and exponential models respectively, indicating the interest rate trends of these debts can vary drastically. Finally, we qualitatively analyzed the design problems behind debts, and how these evolve over time.

This approach will not only enable an analyst to precisely locate architectural debts, but also rank and prioritize them, so that informed decisions can be made on if, where, and how to refactor. Although the ArchDebt detection and modeling approaches we propose only work retrospectively when the penalty has already accumulated, this approach can be used to monitor the formation of a debt [23], and thus prevent it from growing early in the software development process.

2. BACKGROUND

We now introduce the key concepts our work is based on.

Design Rule Space. In our prior work [26] we proposed a novel architectural model—*Design Rule Space*(DRSpace)—based on the Baldwin and Clark's *design rules* [2]. Building upon existing definitions of software architecture [3], we characterize a software architecture as a set of overlapping DRSpaces, each reflecting a unique aspect of the architecture. Each DRSpace is a subset of a system's source files and some kind of relationships (dependencies) among these files. Each DRSpace has one or more "leading file(s)", which all other files in the DRSpace depend on, directly or indirectly. The leading files are usually the files with architectural importance, such as interfaces or abstract classes, which we call *Design Rules*. The relations within a DRSpace may be structural—such as "`Implement`", "`Extend`", "`Call`"—or relations may be based on history coupling between source files—indicating the number of times two files changed together as recorded in the project's revision history.

There are numerous DRSpaces in any non-trivial software system, e.g., each dependency type forms a DRSpace: files connected by "`Extend`" and "`Inherit`" relationships form an *inheritance* DRSpace, and files that are coupled in the project's revision history form an *evolution* DRSpace. We created an *architecture root* detection algorithm that computes the intersection between DRSpaces and the project's "error space"—the set of error-prone files in a system [26]. We showed that the majority of the error-prone files are concentrated in just a few DRSpaces, suggesting that these error-prone files are not islands–they are architecturally connected [26]. Furthermore, we showed that these DRSpaces frequently contain architectural issues (flaws) that, we claim, are the root causes of error-proneness.

Design Structure Matrix (DSM). We use a DSM [2] to represent a DRSpace. Each element in the DSM is a source file, and each cell represents the relationships between the file on the row and the file on the column. For example, Figure 5 is a DRSpace with leading file `ColumnParent`. Each cell shows the structural dependencies — "`implement`", or "`dp`" — between the file on the row and the file on the column, followed by the conditional probability of change propagation. In the original DRSpace [26], we used the number of times two files changed together in the project's revision history to represent their history dependency. In this paper, we replace this count with a *probability*. For example, cell[6,2] contains "`Implement`", meaning that the file on row 6, `CassandraServer`, implements the interface on row 2, `Cassandra`; cell[2,6] contains "48%", meaning that when `Cassandra` changes, there is a 48% probability that `CassandraServer` will change with it.

Architecture Issues. Our recent work [21] defined, implemented, and validated an algorithm for detecting recurring architectural issues in software systems, which we call *hotspot patterns*, including: 1) *unstable interface*, where an influential file changes frequently with its dependents in the revision history; 2) *modularity violation*, where structurally decoupled files frequently change together in the project's revision history; 3) *unhealthy inheritance*, where a super-class depends on its sub-class or where a client class depends on both a super-class and its sub-class; 4) *cyclic dependency*, where a set of files forms a dependency cycle. In the 9 projects we examined, we observed a strong correlation between the number of flaws a file has and: 1) the number of bugs reported and fixed in it, 2) the number of changes made to it, and 3) the amount of effort spent on it (in terms of committed lines of code to fix bugs and to make changes).

3. DEFINITION AND IDENTIFICATION

In this section, we define *architectural debt* (ArchDebt) and present an ArchDebt identification approach.

3.1 ArchDebt Definition

We formally define the software architecture of a system, implemented at release r, as a set of overlapping DRSpaces:

$$SoftArch_r = \{DRSpace_1, DRSpace_2, ..., DRSpace_n\} \quad (1)$$

where n is the number of DRSpaces, each revealing a different aspect of the architecture. For example, each dependency type can form a distinct DRSpace [26]. We define an *Architectural Debt (ArchDebt)* as a group of *architecturally connected files* that incur high maintenance costs over time due to their flawed connections, as follows:

$$ArchDebt =< FileSetSequence, DebtModel > \quad (2)$$

The first element, *FileSetSequence*, is a sequence of file groups, each extracted from a different project release:

$$FileSetSequence = (FileSet_1, FileSet_2..., FileSet_m) \quad (3)$$

where m is the number of releases that *ArchDebt* impacts, $m \leq R$, the number of project releases. $FileSet_r$, $r = 1...m$ is a connected file group in release r. The number of files in each $FileSet$ may vary in different releases.

The second element, $DebtModel$ is a formula capturing the growth, i.e. interest rate, of the ArchDebt, in the form of maintenance costs for $FileSetSequence$.

3.2 ArchDebt Identification

Given this definition of *ArchDebt*, we first identify $FileSet-Sequence$, and then build a $DebtModel$ to capture the "interest rate" based on the costs $FileSetSequence$ has incurred. There are numerous DRSpaces in each release, and numerous debt candidates (file groups) in each DRSpace. We illustrate our process of searching for a $FileSetSequence$ on analogy with searching for web pages on the internet, consisting of the following steps as shown in Figure 1:

1) Crawling: this step collects a subset of DRSpaces from each $SoftArch_r$, r from 1 to R, similar to crawling and collecting web pages.

2) Indexing: this step identifies (indexes) a specific file group, $FileSet$, from each $DRSpace$ selected in the first

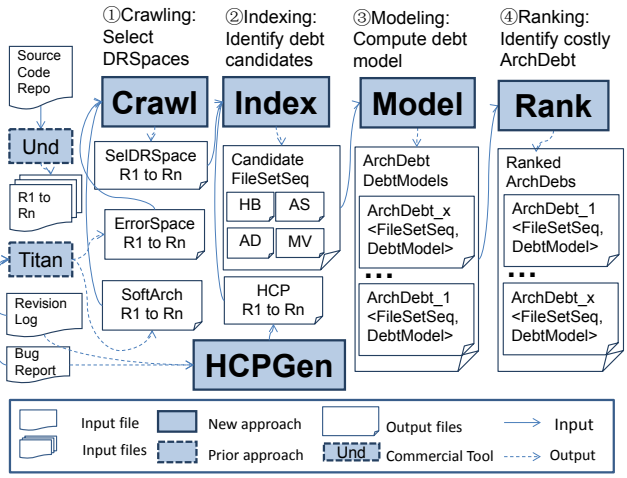


Figure 1: Approach Framework

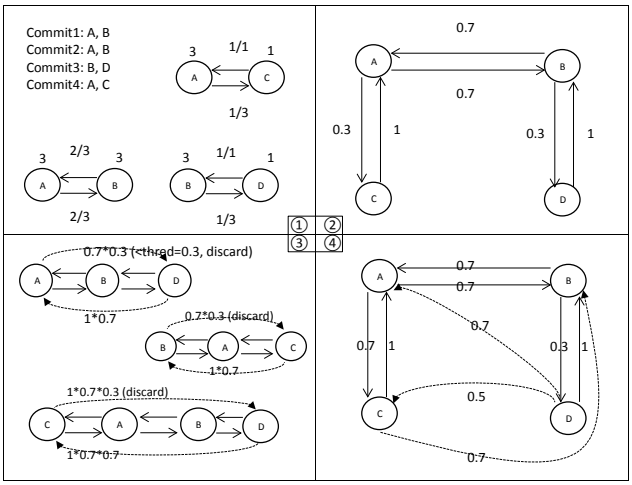


Figure 2: Generate HPC Matrix

step, then locates sequences of related $FileSets$ in different releases as a $FileSetSequence$.

3) Modeling: we measure the maintenance costs incurred by each sequence of $FileSet_r$. An $ArchDebt$ is defined as a $FileSetSequence$ whose costs increase over time.

4) Ranking: we rank the severity of each $ArchDebt$ according to the amount of maintenance costs they have accumulated in the project's evolution history.

3.2.1 Crawling: Selecting DRSpaces

We first define the set of error-prone files in a particular release r as an error space: $ErrorSpace_r$=$\{f_1, f_2, ..., f_n\}$, where file f_i, $i = 1...n$, was revised to fix bugs at least once from release 1 to release r. According to this definition: $ErrorSpace_r$ is a subset of $ErrorSpace_{r+1}$. For each release r, we select a set of $DRSpaces$ from $SoftArch_r$, each led by a file in $ErrorSpace_r$, and form a $SelectedDRSpace$ set as the output of $Crawling$:

$$SelectedDRSpace_r = Crawling(SoftArch_r, ErrorSpace_r) \quad (4)$$

Each DRSpace in $SelectedDRSpace_r$ is led by an error-prone file in $ErrorSpace_r$, and contains other files that depend on the leading error-prone file. If there are n files in $ErrorSpace_r$, there are n DRSpaces in $SelectedDRSpace_r$.

3.2.2 Indexing: Identify ArchDebt Candidates

Next we find the $FileSetSequences$ that are debt candidates. Files in such a sequence must have changed together in the project's revision history. We first calculate a history coupling model—HCP matrix—and then we filter file groups using 4 *indexing patterns*.

HCP Matrix.

In our prior work [26], we used a DSM to model *history coupling*: each cell in the DSM displays the number of times two files changed together. To manifest how a change to a file influences other files, we propose a new model: the **history coupling probability (HCP) matrix**. Although each column and row in a HCP still represents a file, we use each cell to record the *conditional probability* of changing the file on the column, if the file on the row has been changed, i.e., the odds of changes propagating from file to file.

Figure 2 shows an example of the creation of a HCP. Part 1 shows 4 files A, B, C, and D, that change in 4 commits: Commit1{A,B} (Commit1 changes A and B), Commit2{A,B}, Commit3{B,D}, and Commit4{A,C}. First, we compute the pair-wise conditional change probabilities for any pair of files. For example, the probability of changing file A, given that file C has changed, denoted by $Prob\{A|C\}$, is the number of times A and C change in the same commits divided by the total number of changes to C. Similarly, $Prob\{C|A\}$ is the number of times A and C change in the same commits divided by the total number of changes to A. Hence, $Prob\{A|C\}$ is 1/1, indicating that A *always* changes with C, and $Prob\{C|A\}$ is 1/3, indicating a probability of 1/3 that C changes with A. In this relation, we label C as *dominant* and A as *submissive* because $Prob\{A|C\} > Prob\{C|A\}$. We compute the probabilities for every pair of files and get the graph in part 2 of Figure 2.

As shown in part 3, we compute the N-Transitive-Closure of the graph in part 2 to identify history dependencies between files that change in distinct but potentially related commits. The conditional probabilities between files without direct connections are the multiplication of the probabilities on the transitive links. For example, files B and C never change in the same commits, but they change with A in Commit1 and Commit4. Hence, there are transitive history connections between B and C. $Prob\{B|C\}$ is $Prob\{B|A\}$* $Prob\{A|C\}$=0.7*0.3=0.21, and $Prob\{C|B\}$ is $Prob\{C|A\}$* $Prob\{A|B\}$=1*0.7=0.7. We only keep links with probabilities of at least 0.3 to avoid keeping weak connections. In case there are multiple paths between two files, we keep just the highest probability. Part 4 shows the N-Transitive-Closure, stored in an adjacency matrix called a HCP matrix. For each release r of a project, we compute a HPC matrix (HPC_r), consisting of files in $ErrorSpace_r$, from the bug-fixing revision history between release 1 to release r.

Indexing Patterns.

Now we compute the interaction between $SelectedDRSpaces_r$ and HCP_r to find $FileSet_r$ from each release. We observe that, in most cases, even though the number of files in a $FileSet$ may vary in different releases, they are always con-

nected to at least one file over all releases. For example, if more child classes are defined to extend a parent class over time, the group of files connected to the parent class grows. We thus call this one special file the *Anchor file* of the group, denoted as file a. We thus define $FileSet_r$ as:

$$FileSet_r = \{a, M_r | M_r = \{m_i : i \text{ from 1 to } n\}| \\ \forall m_i \in M_r, m_i \text{ architecturally connected with } a \text{ in release } r\} \quad (5)$$

where $FileSect_r \in FileSetSequence$, a is the anchor file, and the files contained in M_r may change with release r. We call M_r the member files of a in release r.

We also define two boolean expressions to describe the relationships between two files (x and y) in release r: $S_r(x \to y)$ and $H_r(x \to y)$. $S_r(x \to y)$ means y structurally depends on x in release r. $H_r(x \to y)$ means x is dominant and y is submissive in their co-changes between release 1 to release r. In HCP_r, $HCP_r[x, y]$ is the probability of changing y, given x has changed. If $HCP[x, y] > HCP_r[y, x]$, then x is dominant and y is submissive. $HCP[x, y] = HCP_r[y, x]$ means x and y are equally dominant. Formally:

$$\text{In release } r, \\ S_r(x \to y) \text{ is true if } y \in DRSpace_r\_x, \text{otherwise it is false} \\ H_r(x \to y) \text{ is true if } HCP[x, y] >= HCP_r[y, x] \\ \wedge HCP[x, y] \neq 0, \text{otherwise it is false} \quad (6)$$

For any pair of a and m in a $FileSet_r$, we identify 4 relationships: $S_r(a \to m)$, $S_r(m \to a)$, $H_r(a \to m)$, and $H_r(m \to a)$. Each relationship could be either true or false. We enumerated all 16 combinations of these 4 relationships. The 4 combinations with $H_r(a \to m)$ and $H_r(a \to m)$ $false$ are irrelevant to our analysis (as we need history to measure debt). From the remaining 12 possible combinations, we defined 4 indexing patterns—*Hub, Anchor Submissive, Anchor Dominant, Modularity Violation.* Each pattern corresponds to prototypical architectural issues that proved to correlate with reduced software quality [21].

Given any anchor file $a \in ErrorSpace_r$, we can calculate its $FileSet_{r\_a}$ using $SelectedDRSpace_r$ and HCP_r through the lens of the 4 indexing patterns:

Hub—the anchor file and each member have structural dependencies in both directions and history dominance in at least one direction. The anchor is an architectural hub for its members. This pattern corresponds to cyclic dependency, unhealthy inheritance (if the anchor file is a super-class or interface class), and unstable interface (if the anchor file has many dependents). Informally such structures are referred to as "spaghetti code", or "big ball of mud". A $FileSet_{r\_a}$ with anchor file a in release r that matches a *hub* pattern is denoted by $HBFileSet_{r\_a}$ and is calculated as:

$$HBFileSet_{r\_a} = Index_{HB}(a, SelectedDRSpace_r, HCP_r) \\ = \{a, M_r | \forall m \in M_r, S_r(a \to m) \wedge S_r(m \to a) \\ \wedge (H_r(a \to m) \vee H_r(m \to a))\} \quad (7)$$

Figure 3 is a Hub *FileSet* for the PDFBox project, anchored by `PDAnnotation`. The dark grey cell represents the anchor file (cell[4,4] for `PDAnnotation`). The cells showing the historical and structural relationships between member files and the anchor file are in lighter grey. In this *HBFileSet*, the anchor file structurally depends on each member file, and each member file also structurally depends on the anchor file. When the anchor file changes, each member file has a 50% probability of changing as well. When a member file changes, the anchor file always changes with it. A *HBFileSet* is potentially problematic because the anchor file, like a hub, is strongly coupled with every member file both structurally and historically.

	1	2	3	4	5	6	7
1 PDA*Line	(1)	,100%	,100%	dp,100%	,100%	,100%	,100%
2 PDA*SquareCircle	,100%	(2)	,100%	dp,100%	,100%	,100%	,100%
3 PDA*FileAtt*	,100%	,100%	(3)	dp,100%	,100%	,100%	,100%
4 PDA*	dp,50%	dp,50%	dp,50%	(4)	dp,50%	dp,50%	dp,50%
5 PDA*Text	,100%	,100%	,100%	dp,100%	(5)	,100%	,100%
6 PDA*Link	,100%	,100%	,100%	Extend,dp,100%	,100%	(6)	,100%
7 PDA*Widget	,100%	,100%	,100%	Extend,dp,100%	,100%	,100%	(7)

A* stands for Annotation

Figure 3: Hub

	1	2	3	4	5	6	7	8
1 AbstractType	(1)							
2 UUIDSerializer	,100%	(2)	,50%			,100%		,50%
3 UUIDType	ext,dp,33%	dp,	(3)				,33%	,50%
4 AbstractCell	dp,50%			(4)				
5 TypeCast	dp,33%		,33%		(5)		,33%	,33%
6 IntegerSerializer	,100%	,100%	,50%			(6)		,50%
7 LongType	ext,dp,67%		,67%		,33%		(7)	dp,67%
8 DateType	ext,dp,40%		,60%				dp,40%	(8)

Figure 4: Anchor Submissive

Anchor Submissive—each member file structurally depends on the anchor file, but each member historically dominates the anchor. This pattern corresponds to an unstable interface, where the interface is submissive in changes. An *Anchor Submissive FileSet* with anchor a in release rt is:

$$ASFileSet_{r\_a} = Index_{AS}(a, SelectedDRSpace_r, HCP_r) \\ = \{a, M_r | \forall m \in M_r, S_r(a \to m) \wedge \\ \rightharpoonup S_r(m \to a) \wedge H_r(m \to a) \quad (8)$$

Figure 4 shows an *ASFileSet* with anchor `AbstractType` in Cassandra. Each member file directly or indirectly depends on the anchor file, but when the member files change, the anchor file changes with each of them, with historical probabilities of 33% to 100%. A *ASFileSet* is problematic because history dominance is in the opposite direction to the structural influences: the anchor should influence the member files, not the other way around.

Anchor Dominant—each member file structurally depends on the anchor file and the anchor file historically dominates each member file. This pattern corresponds to the other type of unstable interface, where the interface is dominant in changes. An *Anchor Dominant FileSet* with anchor a in release rt can be calculated as:

$$ADFileSet_{r\_a} = Index_{AD}(a, SelectedDRSpace_r, HCP_r) \\ = \{a, M_r | \forall m \in M_r, S_r(a \to m) \wedge \\ \rightharpoonup S_r(m \to a) \wedge H_r(a \to m)\} \quad (9)$$

Figure 5 shows an *ADFileSet* calculated using anchor `ColumnParent` in Cassandra. Each member file (from row 2 to row 6) structurally depends on (cell[2 to 6:1]) the anchor file (row 1), and when the anchor file changes, the member files change as well with probabilities from 41% to 100% (cell[1:2 to 6]). A *ADFileSet* presents potential problems where the anchor file is unstable and propagates changes to

	1	2	3	4	5	6
1 ColumnParent	(1)	,100%	,50%	,41%	,50%	,100%
2 Cassandra	dp,	(2)				,48%
3 CliClient	dp,	dp,	(3)			
4 Column*Reader	dp,	dp,		(4)		
5 ThriftValidation	dp,				(5)	
6 CassandraServer	dp,	Implement,			dp,	(6)

Figure 5: Anchor Dominant

	1	2	3	4	5	6	7	8
1 JMXETPEMBean	(1)	,100%	,44%	,50%		,100%	,100%	,50%
2 DebuggableTPExecutor		(2)	,31%					
3 StorageService			(3)	dp,	dp,Use,			
4 ColumnFamilyStore			dp,	(4)				
5 MessagingService			dp,		(5)		dp,	
6 NodeProbe			,44%		dp,	(6)		
7 StatusLogger		,50%		dp,50%	dp,	,50%	(7)	
8 JMXCTPExecutor	,50%	,100%	,31%	,100%	,50%	,50%	,50%	(8)

Figure 6: Modularity Violation

member files that structurally depend on it.

Modularity Violation—there are no structure dependencies between the anchor and any member, however they historically couple with each other. In a *modularity violation* the anchor and member files share assumptions ("secrets") that are not represented in any structural connection. A $MVFileSet$ with anchor a in release r is calculated as:

$$MVFileSet_{r\_a} = Index_{MV}(a, SelectedDRSpace_r, HCP_r) = \{a, M_r | \forall m \in M_r, \nrightarrow S_r(a \rightarrow m) \wedge \nrightarrow S_r(m \rightarrow a) \wedge (H_r(m \rightarrow a) \vee H_r(a \rightarrow m))\} \quad (10)$$

Figure 6 is a $MVFileSet$ with anchor `JMXCTPExecutor` (row 8) in Cassandra. The anchor file, on the bottom of the matrix, is structurally isolated from the member files. However, when the anchor file changes, there are historically 31% to 100% probabilities that the member files change as well, and when the member file `JMXETPEMBean` (on row 1) changes, the anchor file has a 50% chance to change with it. This pattern identifies potential problems where the anchor file and the member files share common assumptions, without explicit structural connections, and these assumptions are manifested by historical co-change relationships.

Identify ArchDebtCandidates.

For each release r, we use each a in $ErrorSpace_r$ as the anchor file to calculate a $FileSet$ for each of the 4 patterns: HB, AS, AD, and $MV\ FileSet_{r\_a}$. The *FileSetSequence* in the *Hub* pattern with anchor file a is denoted by $HBFileSetSequence_a$. Similarly, for anchor a, we can identify AS, AD, and $MV\ FileSetSequence_a$. Using any error-prone file as the anchor, we can identify 4 *FileSetSequences*, each of which is an *ArchDebtCandidate*.

As a result, for each $a \in ErrorSpace_r$ and for each release r , we can exhaustively detect $4^*|\cup_{r=1}^{n} ErrorSpace_r|$ candidates, which equals $4^*|ErrorSpace_n|$ because $ErrorSpace_n$ is a super set of all $ErrorSpaces$ in earlier releases.

3.2.3 Modeling: Build Regression Model

Now that we have identified the $FileSetSequences$, the candidates of ArchDebt, we: (1) measure maintenance costs incurred by each $FileSet$ within a *FileSetSequence*, and (2) formulate a *DebtModel* to capture cost variation.

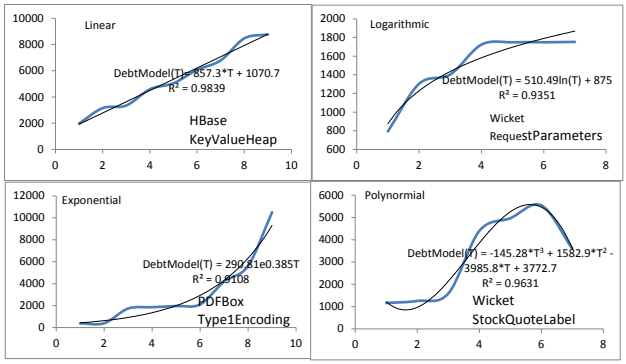


Figure 7: 4 Types of Regression Model

Measure ArchDebtCandidates.

From each *FileSetSequence*, we first exclude any singleton $FileSet_r$, that is, a set containing just 1 file since this can not involve architecture problems. After that, we define the **age** of a *FileSetSequence* as the number of $FileSets$ in it after singleton $FileSets$ are filtered out. Then, for each $FileSet_r$, we measure the **maintenance effort**, denoted by $Effort\_FileSet_r$, that it consumes by the end of release r. For any file $f \in FileSet_r$, we approximate its maintenance costs as the amount of bug-fixing churn expended on it by the end of release r. We denote the maintenance cost for file f by release r as $ErrorChurn_{r\_f}$. $Effort\_FileSet_r$ is the sum of maintenance costs spent on each file in the set:

$$Effort\_FileSet_r = \sum\nolimits_{\forall f \in FileSet_r} ErrorChurn_{r\_f} \quad (11)$$

To qualify as a debt, first a *FileSetSequence* should have long-lasting impacts. This can be evaluated using the age of *FileSetSequence*. Second, *FileSetSequence* should require ever-increasing maintenance effort. Suppose a software system has n releases. Let $FileSet_f$ and $FileSet_l$ be the first and last element in *FileSetSequence*. A *FileSetSequence* is identified as a real debt if it satisfies the following conditions:

$$\begin{cases} age >= n/c; \\ Effort\_FileSet_l > Effort\_FileSet_f. \end{cases}$$

where c is a tunable parameter. Here, we use c=2, meaning that *FileSetSequence* influences at least half of the releases. Otherwise, the candidate is not a meaningful debt, at least not yet. The second condition requires that the maintenance costs on *FileSetSequence* increase over time.

Formulate DebtModel.

For each *FileSetSequence* identified as a real debt, we select a regression model as its *DebtModel* to describe the growing trend (the interest rate) of maintenance costs over time. We use four types of regression models: linear, logarithmic, exponential, and polynomial (up to degree 10). Figure 7 shows typical examples of these 4 models. Each model represents a coherent scenario. In a linear model (part 1 of Figure 7), the penalties of a debt increase at a stable rate in each version. In a logarithmic model (part 2), the penalties of a debt increase more slowly over time (e.g., when devel-

opers refactor a group of files, they become easier change, so the interest rate decreases over time). In an exponential model (part 3), the penalties of a debt increase at ever-faster rates over time (e.g., the structure of a tangled group of files worsens, often in the early stages of a project, before anyone worries about *TD*). In a polynomial model (part 4), the penalties of a debt fluctuate over the releases.

We calculate the maintenance costs—$Effort\_FileSet_r$ for each $FileSet_r$ in a *FileSetSequence* using equation 11. The $Effort\_FileSet_r$ of all $FileSet_r$ in a *FileSetSequence* form an array that we call $Effort\_Array$. $Effort\_Array[i] = Effort\_FileSet_r$, where $FileSet_r$ is the ith element of *FileSetSequence*. We define an integer array $T[i] = r$, where r is the release number of the ith element in *FileSetSequence*. Each release r is numbered by its order in the release in history. In the $DebtModel$ of a *FileSetSequence*, $Effort\_Array$ is the independent value and T is the dependent value. We created a *ModelSelector* algorithm to select a regression model for the relationship between T and $Effort\_Array$. The formula and R^2 of the regression model are returned as $DebtModel$:

$$DebtModel = ModelSelector(EffortArray, T) \quad (12)$$

We define a global parameter R^2_{thresh} (R^2 threshold) for *ModelSelector*. R^2_{thresh} ranges from 0 to 1; the higher the value, the stricter $Effort\_Array$ and T fit the model. Our *ModelSelector* algorithm first tries to fit the $Effort\_Array$ and T to a linear regression model. If the R^2_{Lin} of the linear model reaches the threshold R^2_{thresh}, it returns the linear model. If not, it builds both logarithmic and exponential models, and computes their R^2 values. If the R^2 values of both models reach R^2_{thresh}, *ModelSelector* returns the model that gives a higher R^2. Otherwise, it returns the model that reaches the threshold. If the debt fits neither with $R^2 >= R^2_{thresh}$, it tries polynomial models of degrees up to 10. A polynomial model where $R^2_{poly} >= R^2_{thresh}$ or the degree reaches 10, whichever is satisfied first, is returned.

In the *ModelSelector* algorithm, we give higher priority to linear, logarithmic, and exponential models over polynomial models. We do not simply pick the best fit (i.e., the model with highest R^2). The reason is that the linear, logarithmic, and exponential models present three general types of penalty interest rate: stable, decreasing, and increasing. The polynomial model, however, catches minor fluctuations of the penalty trend, most likely a result of noise due to extraneous factors. For example, the debt in part 1 of Figure 7, intuitively a linear model ($DebtModel(r) = 857 * r + 1070$ with R^2 of 0.98), can fit into a polynomial model $DebtModel(r) = -2 * r^6 + 59 * r^5 - 680 * r^4 + 3874 * r^3 - 11342 * r^2 + 16538 * r - 6466$, with a higher R^2 (0.99). The polynomial model fits better (higher R^2), but the linear model is preferred. As long as a debt penalty generally ($R^2 >= R^2_{thresh}$, where e.g. R^2_{thresh} is 0.8) fits into a linear, logarithmic or exponential model, we choose those models.

For each *FileSetSequence*, we identify its *DebtModel*. This completes our *ArchDebt* identification.

3.2.4 Ranking: Identify High-maintenance ArchDebt

Not all architectural debts have the same severity—the maintenance costs they incur. Debts with higher maintenance costs deserve more attention. We rank all the identified architectural debts according to their cumulative maintenance cost as follows. We define a pair $p_f = <f, ErrorChurn_f>$, where f is an error-prone file, $ErrorChurn_f$ is the maintenance costs for f, approximated by bug-fixing churn on f. Let *EffortMap* be the set of p_f, such that $\forall f \in ErrorSpace_n$ (n is the latest release), there exists a $p_f \in EffortMap$. *EffortMap* is one of the inputs to the *ranking* algorithm. The other input is the identified $ArchDebts$.

$$RankedDebts = ranking(ArchDebts, EffortMap) \quad (13)$$

In the ranking algorithm, we rank the importance of each *ArchDebt* according to *EffortMap* iteratively. In each iteration, we select *maxArchDebt* that consumes the largest portion of effort for files in *EffortMap* from *ArchDebts*. The effort for duplicate files is excluded, and the iteration terminates when all *ArchDebts* are ranked.[1] The top debts returned consume the largest maintenance effort, and deserve more attention and higher priority for refactoring.

4. EVALUATION

To evaluate the effectiveness of our approach, we investigate the following research question:

RQ: Whether the file groups identified in ArchDebts generate and grow significant amount of maintenance costs? That is, are they true and significant *debts?*

If the identified file groups only consume a small portion of overall maintenance effort, then they do not deserve much attention. Similarly, if the identified file groups cover a large portion of the system itself, it is not surprising if they also consume the majority of maintenance effort. In both cases, we cannot claim that they are debts worthy of attention.

4.1 Subjects

We chose 7 Apache open source projects as our evaluation subjects. These projects differ in scale, application domain, length of history, and many other project characteristics. They are: Camel—a integration framework based on Enterprise Integration Patterns; Cassandra—a distributed DBMS; CXF—a Web services framework; Hadoop—a framework for reliable, scalable, distributed computing; HBase—the Hadoop distributed, scalable, big data store; PDFBox—a library for working with PDF documents; and Wicket—a component-based web application framework. A summary of these projects is given in Table 1. The second column is the start to end time and the total number of months (in parentheses) for each project. The third column "#R" shows the number of releases selected per project. We selected releases to ensure that the time interval between two releases is approximately 6 months. The column "#Cmt" is the number of commits made over the selected history. The column "#Iss" is the number of bug reports, extracted from the project's bug-tracking system. The last column shows the size range, measured as the number of files in the first and the last selected release.

4.2 Analysis Results

To answer our research question, we measured the amount of maintenance effort spent on the ArchDebts we identified. Since we can not directly measure the amount of effort in working hours or budgets, we use error-fixing churn as an approximation: the number of lines of code modified and committed to fix bugs.

[1]For pseudo code of all algorithms, see: https://www.cs.drexel.edu/~lx52/ArchDebt.html

Table 1: Subject Projects

Subject	Length of history (#Mon)	#R	#Cmt	#Iss	#Files
Camel	7/2008 to 7/2014 (72)	12	14563	2790	1838 to 9866
Cassandra	9/2009 to 11/2014 (62)	10	14673	4731	311 to 1337
CXF	12/2007 to 5/2014 (77)	13	8937	3854	2861 to 5509
Hadoop	8/2009 to 8/2014 (60)	9	8253	5443	1307 to 5488
HBase	12/2009 to 5/2014 (53)	9	6718	6280	560 to 2055
PDFBox	8/2009 to 9/2014 (62)	12	2005	1857	447 to 791
Wicket	6/2007 to 1/2015 (92)	15	8309	3557	1879 to 3081

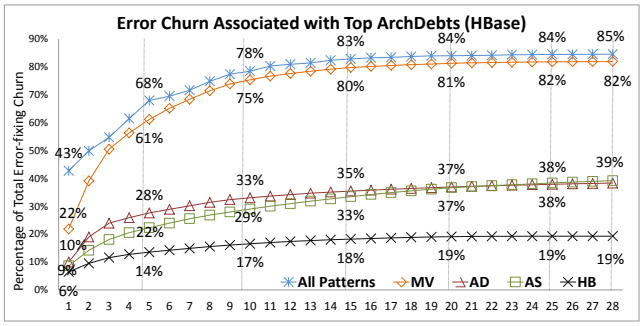


Figure 8: Debt Churn Consumption (HBase)

We use HBase as an example to illustrate our observations. Figure 8 shows the percentage of maintenance effort associated with the files in *FileSets* of all identified *ArchDebts* in HBase. The x-axis is the number (from 1 to 28) of identified architectural debts. The y-axis is the accumulated percentage of maintenance effort associated with the top x *ArchDebts*. Each line represents the percentage of each type of debt. This figure depicts, from bottom to top, you can see: *Hub*, *Anchor-Submissive*, *Anchor-Dominant*, and *Modularity Violation* debts respectively. The line on the top is the total percentage of the 4 types of debts. The values of the top line are not simply the sum of the values of the 4 types because different types of debts may share some files. Thus we make the following observations in HBase.

(1) Architectural debts consume a significant percentage (85%) of the total project maintenance effort. A significant portion of the maintenance effort is spent on paying interest on related groups of files. If they can identify such debts early, a project can save significant effort by paying down the debts via refactoring [15]. As the number of debts increases, the total does not reach 100% because not all errors are architecturally connected. Occasionally, developers introduce errors that can be fixed in isolation.

(2) The top few architectural debts consume a large percentage of maintenance effort. The top **5** *Modularity Violation* debts in HBase consume **61%** of total effort, wherease **all** *Modularity Violation* debts consume **82%** of total effort. Similar observations hold for *Anchor-Submissive*, *Anchor-Dominant*, and *Hub* debts. The lines flatten as the number of debts increases, indicating that most of the effort concentrates in the top few debts. This means that instead of reviewing all identified debts, project leaders only need to focus on the top few.

(3) *Modularity Violation* debt is the most common and expensive debt. *Hub* debts consume the least percentage of effort, while *Anchor-Dominant* and *Anchor-Submissive* take similar percentages. We can see that the line for *Modularity Violation* is close to the line for the sum of all types. This is because *Modularity Violation* debts involve the files in other debts as well.

We made consistent observations from all 7 projects, as summarized in Table 2. Column "All Debts Ch%" shows that, for all 7 projects, from 51% to 85% of the total maintenance effort is consumed by architectural debts. And, a large percentage (31% to 50%) of the effort is consumed by the top 5 *Modularity Violation* debts (shown in sub column "Ch%" under "Modularity Vio"). *Modularity Violation* debts impact the largest number of files and consume the greatest effort, *Hub* debts consume the least, while *Anchor-Submissive* and *Anchor-Dominant* rotate their orders.

If a debt contains a large number of files, it is not surprising that they take a large percentage of effort. We observed, however, that **(4) the top 5 architectural debts contain only a small number of files, but consume a large amount of the total project effort.** We compare the number of files in the top 5 architectural debts versus the percentage of effort they take. For example, in table 2, column "Modularity Vio" under "Top 5 Debts" shows that, in Camel, there are 206 files (13% of all the error-prone files) in the top 5 *ModularityViolation* debts, and these 206 files consume 32% of the total project bug-fixing effort. Similarly, in Camel, the top 5 *Anchor Submissive*, *Anchor Dominant*, and *Hub* debts contain only 1%, 4%, and 2% of the error-prone files, but consume 7%, 16%, and 5% of the total effort respectively. From the column "All 4 types" under "Top 5 Debts", we can observe that, for all the projects, the top 5 architectural debts contain from only 11% to 32% of the error-prone files, but consume 27% to 49% of the total effort. The average ratio of percentage of effort to the percentage of files in the top 5 debts is 2.

Finally, we analyze the file size (in lines of code) of the debts we identified. Much research has shown that file size correlates with error rates and churn. We would like to know that the debts identified by our approach are not just a set of large files. To show this we counted the LOC of the files in the top 5 debts, and observed that the sizes of these files are randomly distributed. Figure 9, for example, shows the file size distribution of the top 5 *Modularity Violation* debts in Cassandra. The x-axis is the range of file size: 10% means the top 10% largest files, 10-20% means files in the 10-20% range in LOC, and so forth. The y-axis is the percentage of files in the top 5 debts that belong to each size range. For example, 22% of the files in top 5 debts are in the top 10% largest files, and 11% of the files are in the range of 90-100% range (that is, the smallest files). The top 5 debts do contain a non-trivial number of large files (22% from the top 10% size range), consistent with other studies showing that large files tend to be problematic. But Figure 9 shows that the top 5 debts contain files in *all* size ranges.

In summary, we can claim that the architectural debts identified by our approach are truly debts that account for a large amount (from 51% to 85%) of maintenance effort. Most (31% to 61%) of the maintenance effort concentrates in the top 5 architectural debts, which contain only a small percentage (13% to 25%) of the project's files.

5. DISCUSSION

We now discuss which model best describes the *interest rate* of an *ArchDebt* and illustrate how our approach helps to understand and monitor the evolution of ArchDebts.

Table 2: Top 5 Debt:#Files vs Churn

Projects	All Debts Ch%	Top 5 Debts									
		All 4 types		Modularity Vio		Anchor Sub.		Anchor Dom.		Hub	
		Fls	Ch%	Fls	Ch%	Fls	Ch%	Fls	Ch%	Fls	Ch%
Camel	59%	230(15%)	35%	206(13%)	32%	20(1%)	7%	60(4%)	16%	40(2%)	5%
Cassandra	72%	273(28%)	57%	196(20%)	50%	72(7%)	28%	33(3%)	32%	26(3%)	16%
CXF	56%	200(11%)	27%	136(8%)	20%	70(4%)	6%	22(1%)	10%	12(1%)	3%
Hadoop	51%	145(25%)	44%	118(20%)	42%	45(8%)	22%	10(2%)	16%	10(2%)	6%
HBase	85%	349(30%)	67%	290(25%)	61%	87(7%)	15%	36(3%)	27%	23(2%)	13%
PDFBox	67%	133(32%)	49%	107(25%)	45%	35(8%)	12%	30(7%)	26%	17(4%)	10%
Wicket	62%	295(22%)	38%	214(16%)	31%	130(10%)	11%	35(3%)	13%	14(1%)	7%

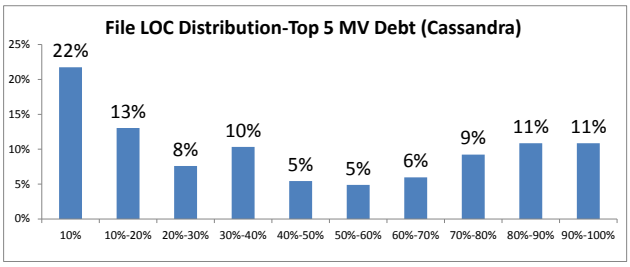


Figure 9: Top 5 Debts File Size Distribution (Cassandra)

Table 3: Debt Costs Model Distribution

Project	#Ds	$R^2_{threshold} = 0.75$				$R^2_{threshold} = 0.8$			
		Lin	Log	Exp	Poly	Lin	Log	Exp	Poly
Camel	199	52%	19%	0%	30%	39%	20%	2%	39%
Cassandra	180	61%	7%	2%	30%	53%	6%	3%	39%
CXF	189	56%	12%	1%	32%	45%	10%	4%	41%
Hadoop	74	46%	7%	7%	41%	36%	8%	3%	53%
Hbase	204	65%	7%	2%	25%	62%	4%	2%	31%
PDFBox	85	59%	4%	5%	33%	39%	1%	9%	51%
Wicket	153	46%	22%	1%	30%	38%	17%	1%	44%

5.1 The Interest Rate of ArchDebt

For each *ArchDebt*, we search for a suitable regression model to capture its interest rate, as introduced in 3.2.3, using R^2_{thresh} of 0.75 and 0.8 respectively. The results are reported in Table 3. The first column is project name. The second column is the number of instances of *ArchDebt* identified in a project. The third and forth columns are model distributions for R^2_{thresh} of 0.75 and 0.8 respectively.

When R^2_{thresh}=0.75, in all the projects, about half (46% to 65%) of the debts fit a linear regression model (with R^2 >= 0.75). For other debts where a linear model doesn't fit, a small percentage fits either a logarithmic (4% to 22%) or exponential (0% to 7%) model (with R^2 >= 0.75), and a polynomial model fits 25% to 41% of the identified debts.

When R^2_{thresh}=0.8, the models are less noise-tolerant. We can see that linear model is still common (36% to 62%) for all projects. But a small portion of debts, from 6% (HBase, 31% minus 25%) to 18% (PDFBox, 51% minus 33%), can no longer fit into linear, logarithmic, or exponential models, but fit a polynomial model.

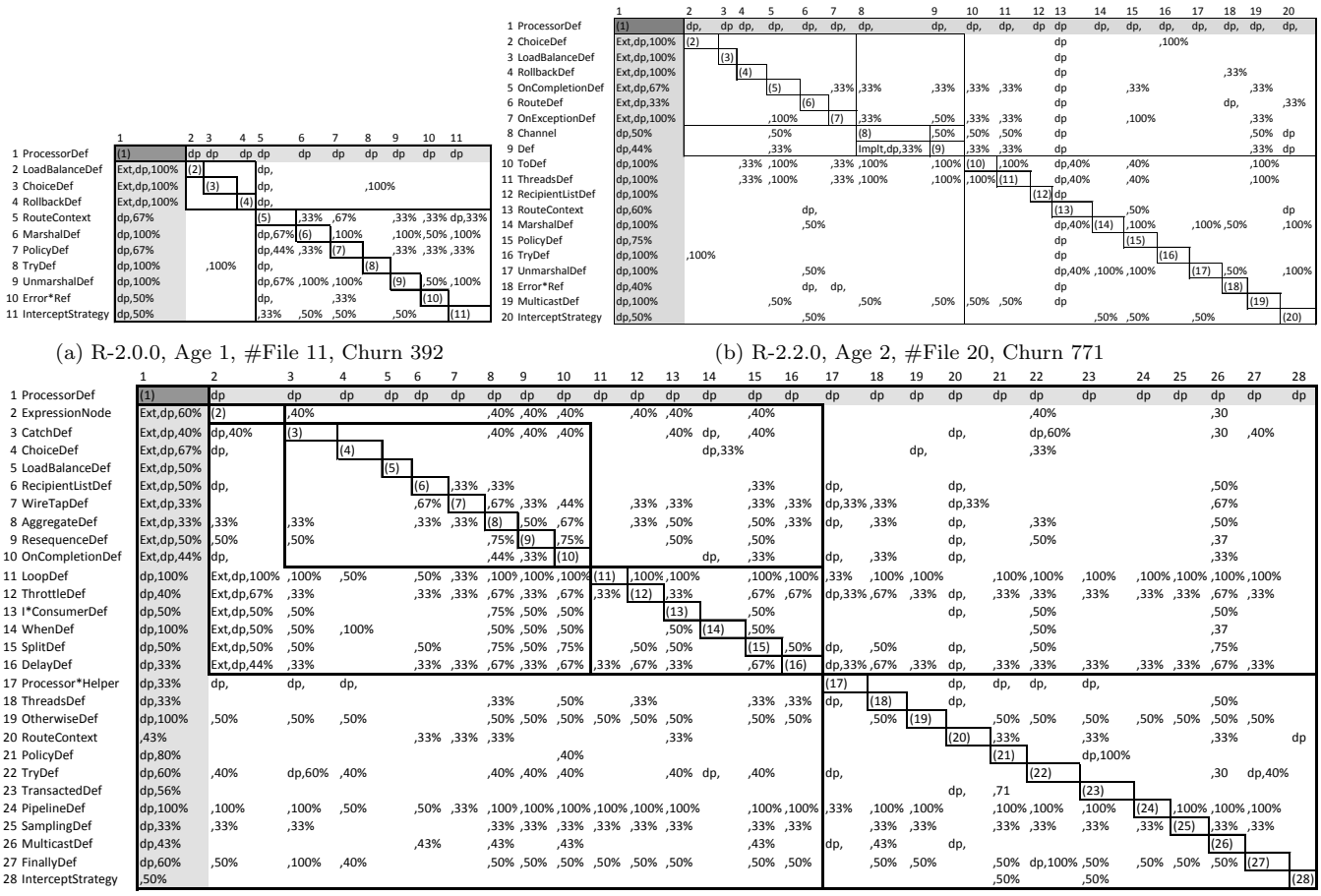
In summary, when R^2_{thresh} is 0.75, the linear model is most common—about half of the debts fit into it. This indicates that half of *ArchDebts* accumulate maintenance interest at a constant rate. Only a small portion of debts accumulate interest at a faster (less than 7% in exponential) or slower (less than 22% in logarithmic) rate. About 1/3 of the identified debts accumulate costs with a more fluctuating rate, which is captured by a polynomial model. More *ArchDebts* fit into a polynomial model as R^2_{thresh} increases.

5.2 Architectural Debt Evolution

We showed, in section 4, that the top 5 debts consume a large amount of effort. We manually inspected the evolution of these debts, and now illustrate how architectural flaws evolve into debts over time. As an example, consider the top *Hub* debt with anchor file `ProcessorDef` (referred to as $PDef$ in the following) in Camel (Figure 10). We have provided 3 snapshots of this debt—in release 2.0.0 (age 1), release 2.2.0 (age 2), and release 2.12.4 (age 11)—to show its evolution. Snapshots from age 3 to 10 are similar to age 11. "Ext" and "Impl" stand for "extend" and "implement", "dp" denotes all other types of structural dependencies.

In release 2.0.0, $PDef$ forms a hub with 10 member files: 3 files are its subclasses, 7 files are its general dependents, and $PDef$ structurally depends on all of them. Note that in this snapshot, all files, except `InterceptStrategy`, depend on `RouteContext` (column 5). The 11 files in this hub structurally form a strongly connected graph. According to the revision history, $PDef$ changes with all member files with probabilities from 50% to 100% (column 1). The dependents (on rows 5 to 11) of $PDef$ are highly coupled with each other. This is problematic in 3 ways: 1) the parent class $PDef$ depends on each subclass and each dependent class (unhealthy inheritance [21]); 2) the parent class is unstable and often changes with its subclasses and dependent classes (unstable interface [21]). 3) `RouteContext` forms cyclic dependencies with 9 files (cycles). Without fixing these flaws, we expect the maintenance costs of this group to grow.

In release 2.2.0, the impacts of this hub have enlarged—$PDef$ has 3 more subclasses and 6 more general dependents, and it depends on each of them as well. Each newly involved file also depends on `RouteContext` (column 13). The revision history shows that $PDef$ changes with its subclasses and dependents with probabilities of 33% to 100%. Also, the subclasses and dependents (rows 5 to 11) of $PDef$ are highly coupled with each other—changing any of them is likely to trigger changes to the rest. In following releases, the hub grows further. Up to release 2.12.4, $PDef$ has 9 subclasses and 18 general dependents—the size of the hub tripled compared to the start, and, as always, $PDef$ depends on each of them. In addition, 6 of the 18 general dependents (rows 11 to 16) of $PDef$ also become its grandchildren. The inheritance tree has increased in width and depth. The revision history shows $PDef$ still changes with its dependents with probabilities from 33% to 100%. The files in this snapshot are tightly coupled with each other, and so changing any file is likely to trigger changes to others.



(a) R-2.0.0, Age 1, #File 11, Churn 392

(b) R-2.2.0, Age 2, #File 20, Churn 771

(c) R-2.12.4, Age 11, #File 28, Churn 2134

Figure 10: Camel Hub Debt Evolution—Anchor ProcessorDefinition

The maintenance costs spent on this debt fit a linear regression model: $DebtModel(rt) = 158.75 * rt + 509.35$ with $R^2 = 0.89$. This means that, in every release, developers contribute 158.75 more lines of code to fix errors in the hub anchored by $PDef$. Although this model can only be obtained *after* the costs and penalty have accumulated, one could use our approach to detect architecture flaw patterns at any point (as described in [21]), monitor how file groups grow, monitor the formation of debts, and prevent significant costs by investing in proper refactorings [15].

6. RELATED WORK

Technical Debt. Since Cunninghan [7] coined the term, *TD* has referred to the consequences of short-cuts taken in software projects to achieve near-term goals. During the past decade, *TD* has drawn increasing attention [6, 10, 25].

Li et al. [19] conducted a mapping study on different categories of *TD* based on related literature published between 1992 and 2013. They classified ten coarse-grained *TD* types according to the phases of the software development life-cycle, such as requirements, architectural, and code. They found that Code *TD* is the most well-studied type, and Architectural *TD* has also received significant attention. They further categorized Architectural *TD* into seven sub-categories, including architectural smells [22], architectural anti-patterns [13][24], complex architectural behavioral dependencies [5], violations of good architectural practices [8], architectural compliance issues [16], system-level structural quality issues, and all others. *TD* can compromise both functional and quality requirements, such as performance, security, usability, and modifiability.

Alves et al. [1] organized 13 types of *TD* and their key indicators, including Architectural *TD*. They described Architectural *TD* as "problems encountered in software architecture", and referred to issues in software architecture, structure dependencies/analysis, and modularity violations as indicators of Architectural *TD*. Their work focused on building an ontology of *TD* rather than focusing on resolving a specific type of *TD*.

Everton et al. [9] proposed an approach to identify different types of "self-admitted" *TD* in software projects, by reviewing the comments left by developers. They identified five types of self-admitted *TD*: design, requirement, defect, test, and documentation *TD*. According to their study, the most common types of *TD* are design and requirement. But as the name "self-admitted" suggests, the *TD* identified in their work was limited to ones that the developers are aware of. There are forms of *TD* introduced unwittingly by developers, such as the architecture debts we have identified.

Martini et al. [20] conceptualized two patterns of Archi-

tectural *TD*: contagious debt and vicious circle. Contagious debt leads to ripple effects in projects. Vicious circle refers to a more severe contagious debt where the ripple effects form a loop. Their work has two limitations. First, it intensively relies on interviewing developers to identify these problems. As stated above, it is possible the developers are not aware of all the *TD* existing in their project. Furthermore, this approach is labor-intensive and relies highly on the expertise of the analyst. Second, this only identifies two anti-patterns, and these overlap with each other.

Given the substantial research literature, it is surprising that definitions of the types of *TD* are still largely informal. In fact, the identification of *TD* relies heavily on interviews or reviewing developers' revision comments, and these are only problems that the developers are aware of. Many questions in *TD* research remain open. For example, how to precisely define the forms of *TD*, how to automatically identify these forms of *TD*, and how to measure *TD*: its costs and consequences.

Co-change Analysis An analysis of co-changes in software projects at the package, class, method, and statement level has been used to gain insight into problems in software development. Zimmermann et al. [28] applied data mining on revision histories to predict likely changes given a change that has already occurred. Kagdi et al. [14] proposed an approach to calculate the change impact scope of a software entity by combining structural coupling, reflected in source code, and change coupling, recorded in the project's revision history. Their approach improved the accuracy of change impact analysis, compared with either technique used independently. Gethers et al. [11] proposed an integrated approach to identify the impact set of a change request (e.g. a bug ticket in bug-tracking database), based on data mining of past source code commits and run-time traces.

Analysis of co-changes has also been used in reverse-engineering. Beck et al. [4] used co-change analysis to compute clusterings. They used an *Evolutionary Class Dependency Graph* to represent co-change coupling. They calculated three types of clusterings using (1) only co-change coupling, (2) only structure dependencies, and (3) a combination of the two. They found that clustering based on the combined approach yielded the best results.

Co-change analysis has also been applied to investigate problems in software projects, such as bugs and code smells. Kouroshfar et al. [17, 18] investigated how co-changes impact bugs. They found that co-changes dispersed across different sub-systems are more likely to result in bugs than localized co-changes. Girba et al. [12] used co-change patterns to identify hidden dependencies between different parts of software system that reveal bad smells. They defined history patterns in three granularity levels: method level, class level, and package level. These patterns can reveal code smells, such as similar code, cloned code, and shotgun surgery. Code smells have also been used as a heuristic for approximating *TD*. Zazworka et al. [27] reported that not all *TD* approximated by code smells will lead to high maintenance costs, and not all *TD* have code smells.

7. LIMITATIONS AND THREATS

We now briefly discuss our limitations and threats to validity. First, since we have only examined 7 projects and all of these are Apache projects, we can not guarantee that our results will generalize to other projects with different cultures and organizational policies. Second, our approach has the limitation that it relies on revision history to identify architectural debt. For projects without enough history data, our approach can still identify groups of files with the potential to become architectural debt. The building of a $DebtModel$ relies on having adequate history data. But our pattern matching approach is still feasible for projects with short history. We plan to evaluate the effectiveness of our approach on projects without enough history in our future work. Third, our approach relies on mining error-prone files from a project's revision history and bug tracking data. We use the bug report ID that developers enter into commits to locate error-prone files. The availability and accuracy of such information heavily depend on the project's protocols. This is both a limitation and threat to validity to our approach. Finally, we can't guarantee that error-fixing churn is the best proxy measure for effort. In our future work, we plan to explore more effort proxies, and we are collaborating with an industry project that records actual effort data, and we plan to compare this with our proxy measures of effort.

8. CONCLUSION

To quantify and manage architectural *TD*, we formally defined the concept of *architectural debt*, and then described an approach to automatically identify such debts, to measure their maintenance consequences, and to model their growth. We proposed a novel history model—the HCP matrix—to approximate the probabilities of change propagation among files, and defined 4 patterns based on the HCP matrix to capture problematic architectural connections among files.

We evaluated our approach on 7 large-scale Apache open source projects and the results showed that a significant portion (51% to 85%) of overall maintenance effort was consumed by paying interest on architectural debts. This suggests that projects could save a significant amount of maintenance costs if they can discover these debts *early*, and pay them down by refactoring. The top 5 architectural debts in each of the 7 projects consume a non-trivial portion (20% to 61%) of each project's maintenance effort, but they only contain a small portion of each project's error-prone files (8% to 25%). Thus investing in refactoring small groups of files could reap large benefits. Finally, we quantified the growing trend of maintenance costs for each debt. About half of the debts grow linearly, meaning that developers pay a consistently increasing penalty on these debts in every release. And using DSMs, we qualitatively illustrated how architectural issues connect more files, incur more maintenance costs, and evolve into debts over time.

Acknowledgments

This work was supported in part by the National Science Foundation under grants CCF-1065189, CCF-1514315, and CCF-1514561.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0003317.

9. REFERENCES

[1] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spinola. Towards an ontology of terms on technical debt. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 1–7. IEEE, 2014.

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

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 3rd edition, 2012.

[4] F. Beck and S. Diehl. Evaluating the impact of software evolution on software clustering. In *Proc. 17th Working Conference on Reverse Engineering*, pages 99–108, Oct. 2010.

[5] J. Brondum and L. Zhu. Visualising architectural dependencies. In *Proceedings of the Third International Workshop on Managing Technical Debt*, MTD '12, pages 7–14, Piscataway, NJ, USA, 2012. IEEE Press.

[6] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing technical debt in software-reliant systems. pages 47–52, 2010.

[7] W. Cunningham. The WyCash portfolio management system. In *Addendum to Proc. 7th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 29–30, Oct. 1992.

[8] B. Curtis, J. Sappidi, and A. Szynkarski. Estimating the principal of an application's technical debt. *IEEE Software*, 29(6):34–42, 2012.

[9] E. da S. Maldonado and E. Shihab. Detecting and quantifying different types of self-admitted technical debt. *SIGSOFT Softw. Eng. Notes*, Apr. 2015.

[10] D. Falessi, P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt at the crossroads of research and practice: report on the fifth international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes*, 39(2):31–33, 2014.

[11] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 430–440, Piscataway, NJ, USA, 2012. IEEE Press.

[12] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel. Using concept analysis to detect co-change patterns. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 83–89, New York, NY, USA, 2007. ACM.

[13] I. Griffith and C. Izurieta. Design pattern decay: The case for class grime. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 39:1–39:4, New York, NY, USA, 2014. ACM.

[14] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Proc. 17th Working Conference on Reverse Engineering*, pages 119–128, Oct. 2010.

[15] 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.

[16] R. Kazman and S. J. Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, Apr. 1999.

[17] E. Kouroshfar. Studying the effect of co-change dispersion on software quality. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1450–1452, May 2013.

[18] E. Kouroshfar, M. Mirakhorli, H. Bagheri, L. Xiao, S. Malek, and Y. Cai. A study on the role of software architecture in the evolution and quality of software. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 246–257, Piscataway, NJ, USA, 2015. IEEE Press.

[19] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *J. Syst. Softw.*, 101(C):193–220, Mar. 2015.

[20] A. Martini and J. Bosch. The danger of architectural technical debt: Contagious debt and vicious circles. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 1–10, May 2015.

[21] 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.

[22] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic. Mapping architectural decay instances to dependency models, 2013.

[23] M. Naedele, H.-M. Chen, R. Kazman, Y. Cai, L. Xiao, and C. A. Silva. Manufacturing execution systems: A vision for managing software development. *Journal of Systems and Software*, 101:59–68, Mar. 2015.

[24] L. Peters. Technical debt: The ultimate antipattern - the biggest costs may be hidden, widespread, and long term, 2014.

[25] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman. Technical debt: Showing the way for better transfer of empirical results. In J. Münch and K. Schmid, editors, *Perspectives on the Future of Software Engineering*, pages 179–190. Springer Berlin Heidelberg, 2013.

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

[27] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull. Comparing four approaches for technical debt identification. *Software Quality Journal*, pages 1–24, 2013.

[28] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. 26th International Conference on Software Engineering*, pages 563–572, May 2004.