2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Active Hotspot: An Issue-Oriented Model to Monitor Software Evolution and Degradation

Qiong Feng, Yuanfang Cai
Drexel University
Philadelphia, PA
qf28, yc349@drexel.edu

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

Di Cui, Ting Liu
Xi'an Jiaotong University
Xi'an, China
cuidi,tingliu@mail.xjtu.edu.cn

Hongzhou Fang
Drexel University
Philadelphia, PA
hf92@drexel.edu

***Abstract*—Architecture degradation has a strong negative impact on software quality and can result in significant losses. Severe software degradation does not happen overnight. Software evolves continuously, through numerous issues, fixing bugs and adding new features, and architecture flaws emerge quietly and largely unnoticed until they grow in scope and significance when the system becomes difficult to maintain. Developers are largely unaware of these flaws or the accumulating debt as they are focused on their immediate tasks of address individual issues. As a consequence, the cumulative impacts of their activities, as they affect the architecture, go unnoticed. To detect these problems early and prevent them from accumulating into severe ones we propose to monitor software evolution by tracking the interactions among files revised to address issues. In particular, we propose and show how we can automatically detect *active hotspots*, to reveal architecture problems. We have studied hundreds of hotspots along the evolution timelines of 21 open source projects and showed that there exist just a few dominating active hotspots per project at any given time. Moreover, these dominating active hotspots persist over long time periods, and thus deserve special attention. Compared with state-of-the-art design and code smell detection tools we report that, using active hotspots, it is possible to detect signs of software degradation both earlier and more precisely.**

***Index Terms*—software evolution, architecture debt**

I. Introduction

Software degradation has a strong negative impact on software quality and productivity, and may cause significant financial losses, e.g. [1]–[3]. Curtis et al. [4] reported that, on average, there is $360,000 of technical debt for every 100,000 LOC in complex software systems [5]. These severe debts do not happen overnight. Developers evolve software by continuously addressing issues, fixing bugs and adding new few features, and architecture flaws emerge quietly and largely unnoticed [6]–[8]. These problems may continuously "evolve"—grow in scope and significance—until the system becomes difficult to maintain. Developers are largely unaware of the accumulating debt; they are focused on their immediate tasks of adding features and fixing bugs. As a consequence, the cumulative impacts of their activities, as they affect the architecture, go unnoticed. When these problems are finally detected, the damage is already done and the problems may be very costly to fix. Identifying these debts early is thus crucial.

Existing tools detect technical or architectural debts [9], [10] either from one snapshot of the software, source or compiled (such as SonarQube [11] and Structure101 [12]), or using a combination of code snapshots and revision history, such as DV8 [13], [14]. Tools using snapshots only tend to report a large number of problems. For example, SonarQube detected 94 files with major or blocker smells in Tika 0.5 out of just 194 files—nearly 50%, making it difficult for the user to select and prioritize *true debts*. Tools that leverage revision histories are more likely to identify true debts since the project history records penalties, in terms of bugs and changes. But these tools usually have some user-settable thresholds. In order to detect severe problems, the thresholds are not small. For example, one anti-pattern detected by DV8 is called *Unstable Interface*. According to its default setting, a file will be detected as an unstable interface if it has 1% of all the project's files as dependents and has co-changed with at least 10 of them at least two times [14]. When this anti-pattern is detected, the problem has already had an impact. Similarly, for all these tools, how early and how accurate debts are detected depends on threshold settings. For example, a class won't be detected as a *God Class* until its size or complexity reaches specified thresholds.

In this paper, we propose a novel model, *Active Hotspot* (AH), that can be used to detect and monitor the emergence and evolution of software degradation by tracking how files and their relations are changed within each issue, such as adding a new feature or fixing a bug. In other words, we use *issues* as first-class entities of evolution, and data sources of our analysis [15]. Concretely, we first track and treat the source files that are modified to address multiple issues as *seed files*, calculate their *architectural* and *semantic* relations through four *propagation pattern* (which will be described in next paragraph) with other modified files, and form the minimal number of file groups, each of which is an *active hotspot* (or *hotspot*).

To study how changes propagate from/to *seed files*, we manually examined a large number of relations among files modified to address various issues. We have identified 4 recurring and repetitive patterns over many projects: 1) dissemination—changes to one file propagate to multiple dependent files (one-to-many); 2) concentration—changes to multiple files cause another file to change (many-to-one); 3) domino—changes to one file cause ripple effects to a sequence of dependent files; and 4) scattershot—changes that scatter in multiple files without syntactic dependencies.

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00095

Using active hotspots, we studied the evolution of 21 large-scale open source projects, and revealed the following results:

(1) During a period of evolution, measured as 100 issue fixes, there always exist a few *dominating hotspots*, usually 2 or 3, and rarely more than 5, that attract the majority of files that were modified to address issues.

(2) These dominating hotspots tend to be persistent and long-lasting: of 560 dominating hotspots we found in all 21 projects, their average life time is 24.6 months. Some of them persist through the entire life of the project. These two observations imply that these dominating hotspots are focal points of evolution that deserve special attention.

(3) The number of files involved in hotspots is not correlated with the size of the project, which contrasts with most tools whose reported problems and measures grow as the project grows. This means that even when a project grows, the development team won't necessarily be overwhelmed by increasing numbers of reported problematic files.

(4) Considering the identification of change-prone/error-prone files as an information retrieval problem, we used active hotspots to detect such files and revealed that, compared with state-of-the-arts techniques, hotspots can identify change-prone and error-prone files more precisely.

(5) Using hotspots as an architecture debt detection tool our empirical study shows that, compared with state-of-the-art techniques, hotspots can reveal major problems, especially architecturally important files that are root causes of architectural debt, much earlier.

This empirical study presents strong evidence that active hotpots can be leveraged to identify true architectural debts earlier and more effectively, so that they can be prevented from incurring severe losses.

II. Hotspot Definitions

In this section, we define our core concepts.

Seed file: a file revised by multiple issues in a given time period. The rationale is that if a file is repetitively changed by multiple issues for different reasons then this file may have violated the single responsibility principle [16]. The user can set a threshold to determine the number of commits for a file to be a seed. In the study reported in this paper we consider files changed for issue-fixing two or more times as seed files.

Propagation file: if a file changed together with a seed file in one of propagation patterns, which we will explain in II-A, then we call it a *propagation file*. For example, suppose f_a is a seed file, and changed together with f_b and f_c. If f_b inherits from f_a, and f_c inherits from f_b, we consider f_b and f_c as the *propagation files* of f_a. Moreover, in this case, we classify these three files as following the *domino* propagation pattern, one of the 4 types of *propagation patterns* among files.

Active Hotspot (or "Hotspot"): a set of files that were changed together during a given time period, including a set of seed files, V_{seed}, and a set of propagation files, V_{prop}. Formally, a hotspot can be represented as a directed connected graph: $Hotspot_{\Delta t} = (V, E)$, in which V is the union of seed files and propagation files, i.e., $V_{seed} \cup V_{prop}$, and E is a set of ordered pairs of files within V, and each pair of files has propagation pattern relations among them. There may be multiple mutually exclusive hotspots during a given time period.

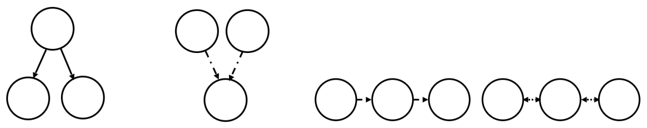
(a) Dissemination (b) Concentration (c) Domino (d) ScatterShot

Fig. 1: Change Propagation Patterns

In the study reported here we only include propagation files through one of the four propagation patterns into a hotspot because (1) these four patterns are widespread (as we will show), and (2) these patterns are the atomic relations forming higher level architecture problems.

A. Propagation Patterns

To better understand file relationships in co-commits, and how issues propagate among these files, we manually examined 100 bug patches from multiple systems that revised five files or more, and categorized the relations among the files involved in each bug fix. We observed that, overwhelmingly (96 out of 100 times), their relations follow four recurring patterns:

(1) Dissemination (Figure 1a). Methods or fields were modified/added/deleted from one file, and several other files using these changed methods or fields were updated accordingly. This pattern involves one file, such as a parent or utility class, and several dependent classes. The bug patch for JDT Bug 427072[1] is such an example. The purpose of this patch was to fix a compilation failure when the type of a method is ambiguous. The solution is to change the matching strategy in Type Binding. The change started by modifying the *sIsMoreSpecific* method in class *Expression.java*, then all its subclasses *ConditionalExpression.java*, *LambdaExpression.java*, *MessageSend.java* and *ReferenceExpression.java* were changed accordingly. In this patch, *ConditionalExpression.java* and *MessageSend.java* used the updated method *sIsMoreSpecific* and needed to change the way the method was called. The other two classes, *LambdaExpression.java* and *ReferenceExpresssion.java*, needed to replace the original matching method *TypeBinding.equalsEquals* with the newly updated *sIsMoreSpecific*. This patch represents the *Dissemination* pattern where changes to one file propagate to multiple files that depend on it.

```
Expression.java
public boolean sIsMoreSpecific(TypeBinding s, TypeBinding t)
{- TypeBinding expressionType = this.resolvedType;
- if (expressionType == null || !expressionType.isValidBinding())
-   return false;
- if (s.isBaseType() && t.isBaseType())
-   return s.isCompatibleWith(t);
- return s.findSuperTypeOriginatingFrom(t) != null;
+ return s.isCompatibleWith(t);}
```

[1]Due to the limitation of paper size, we only present patches directly related to the pattern.

```
ConditionalExpression.java
public boolean sIsMoreSpecific(TypeBinding s, TypeBinding t) {
+   if (super.sIsMoreSpecific(s, t))

MessageSend.java
public boolean sIsMoreSpecific(TypeBinding s, TypeBinding t) {
+  if (super.sIsMoreSpecific(s, t))

LambdaExpression.java
+     if (super.sIsMoreSpecific(s, t))

ReferenceExpression.java
+   if (super.sIsMoreSpecific(s, t))
```

(2) Concentration (Figure 1b). This pattern describes the case where one class changes due to changes to multiple other classes it depends on. That is, two or more files were modified to change/add/delete methods or fields, and another file which depends on them has to be updated. Take JDT Bug 404649 as an example, class *IProblem.java* adds an *int* field *SuperAccessCannotBypassDirectSuper*, and another class, *ProblemReasons.java*, also adds an *int* field *AttemptToBypassDirectSuper*. The third class *ProblemReporter.java* needs to use both of the newly added fields in its method *illegalSuperAccess*. This pattern describes the cases where one class is dependent on the changes from multiple other classes through structural dependencies.

```
core/compiler/IProblem.java
+ int SuperAccessCannotBypassDirectSuper = TypeRelated + 1054;

internal/compiler/lookup/ProblemReasons.java
+ final int AttemptToBypassDirectSuper = 21;

internal/compiler/problem/ProblemReporter.java
+public void illegalSuperAccess(TypeBinding superType, TypeBinding directSuperType,
      ASTNode location) {
+ if (directSuperType.problemId() != ProblemReasons.AttemptToBypassDirectSuper)
+   needImplementation(location);
+ handle(IProblem.SuperAccessCannotBypassDirectSuper ,
```

(3) Domino (Figure 1c). This pattern describes the case where changes originated in one file ripple through multiple other files, resulting in a cascade of consequent changes. The patch for JDT Bug 490657 is such an example: Class *ProblemReasons.java* was first modified by adding an *int* field *ServiceImplDefaultConstructorNotPublic*. After that, another class *ProblemReporter.java* was also changed to add a new method *invalidServiceImpl()* that used the newly added field in *ProblemReasons.java*. Finally, the third class, *ModuleDeclaration.java* also added a new method *validate()* which uses the method *invalidServiceImpl()* added in the second class, *ProblemReporter.java*. This pattern describes the well-known ripple effects caused by direct and indirect structural dependencies.

```
lookup/ProblemReasons.java
+ final int ServiceImplDefaultConstructorNotPublic = 31;

problem/ProblemReporter.java
+public void invalidServiceImpl(int problem, TypeReference impl) {
+ String[] args = new
      String[]{CharOperation.charToString(impl.resolvedType.readableName())};
+   case ProblemReasons.ServiceImplDefaultConstructorNotPublic:

ast/ModuleDeclaration.java
+ private void validate(TypeReference serviceInf, TypeReference serviceImpl) {
+   if (problemId != ProblemReasons.NoError) {
+     this.scope.problemReporter().invalidServiceImpl(problemId, serviceImpl);
```

(4) ScatterShot (Figure 1d). This pattern is similar to code clones, where similar patch logic is injected into multiple files. For example, JDT Bug 520795 was "*Private interface methods should not be visible outside*". The patch modified the same method(*isPrivate()*) in 3 classes: *MethodBinding.java*, *ReferenceBinding.java* and *Scope.java*. These 3 classes are responsible for verification logic and shared this crosscutting concern. If method accessibility needs to be changed in the future, it is possible that these three classes need to be changed together again. Files involved in the *ScatterShot* pattern may or may not have structural dependencies among them, but the methods that were changed in this pattern do not have structural relations.

```
MethodBinding.java
+ if (this.declaringClass.isInterface() && isStatic() && !isPrivate()) {

ReferenceBinding.java
+   if (method == null || method.isStatic() ||
      method.redeclaresPublicObjectMethod(scope) || method.isPrivate())

Scope.java
+         if (candidate.isStatic() && candidate.declaringClass.isInterface() &&
      !candidate.isPrivate()) {
```

These results are generalizable. We created a tool to identify these recurring patterns and analyzed 1,503 bug patches that modified 5 files or more. This analysis showed that 97.3% of them participated in at least one of these four patterns, suggesting that these patterns are widespread.

Besides, these patterns are the atomic relations forming higher level architecture problems. For instance, when multiple *disseminations* with the same top file are combined, this top file will have a large number of dependents and may be qualified as *Wide Hierarchy* design smell in Designite or *Unstable Interface* anti-pattern in DV8.

III. Illustrative Example

Figure 2 illustrates the evolution of a hotspot, simplified from the Apache Camel project. We split its evolution history into 315 sliding *evolution windows*, each consisting of 100 issues that modify at least one source file. Each window is 20 issues apart from the next: the first evolution window, $w1$, starts when the 1st issue was resolved and ends when the 100th issue was resolved, and $w2$ starts at the 21st issue and ends with the 120th issue, and so forth.

Figure 2 is labeled with 5 consecutive evolution windows. An oval denotes an issue, and the circles within it are the files added/deleted/modified to address the issue. A red circle denotes a file changed by more than one issue in the evolution window—a *seed file*. A yellow circle denotes a propagation file participating in one of the 4 propagation patterns with *seed files*, which we call *propagation files*. Other files, represented by small, unlabeled gray circles, are modified but have no relations with the seed files. An *active hotspot* (AH) consists of all the red and yellow circles and their relations. *Each hotspot is identified by their seeds files.* If two hotspots share any seed files, we consider them as the same hotspot.

In $w229$, the file named *DefaultCamelContext.java* ($f1$) was modified by two different issues, which makes it a seed file. $f1$ and $f2-5$ depends on $f6$, *CamelContext.java*, and they form a *dissemination* pattern. Since $f2-6$ is only modified by one issue in this evolution window, they are colored yellow, denoting that even if they are not seed files, they participated in at least one propagation pattern, and all six files form an active hotspot in $w229$.

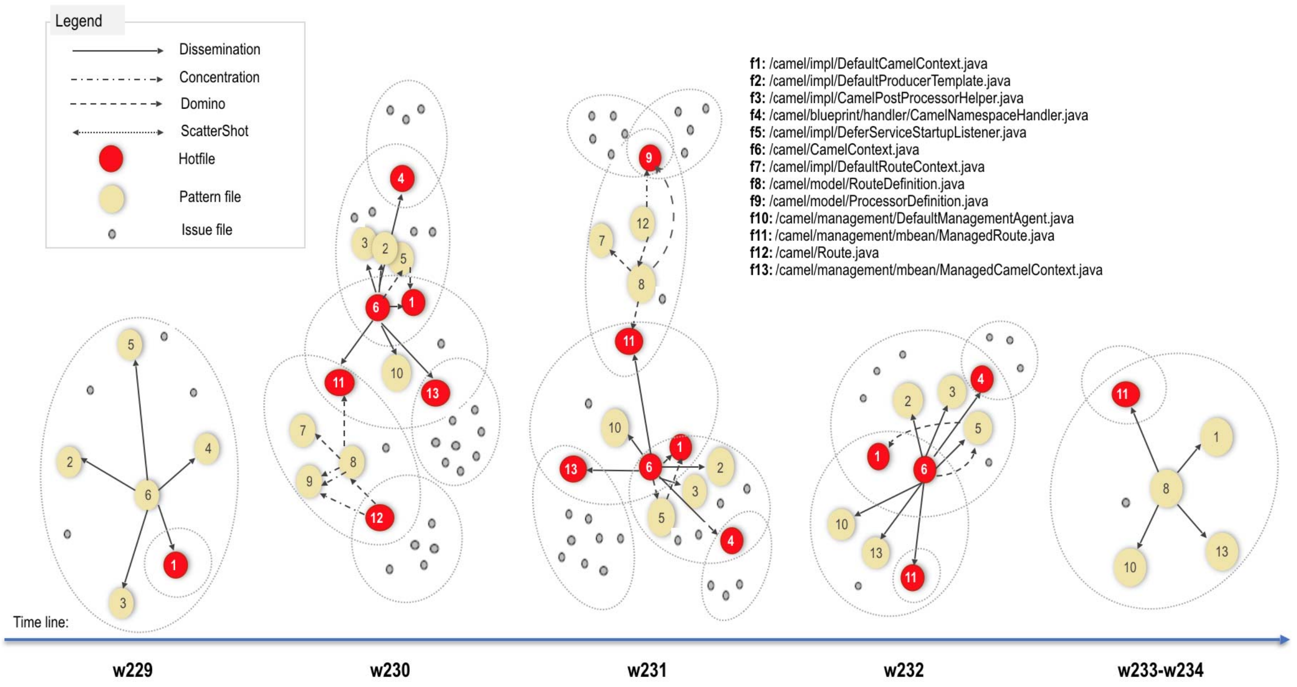
Legend
Dissemination
Concentration
Domino
ScatterShot
Hotfile
Pattern file
Issue file
f1: /camel/impl/DefaultCamelContext.java
f2: /camel/impl/DefaultProducerTemplate.java
f3: /camel/impl/CamelPostProcessorHelper.java
f4: /camel/blueprint/handler/CamelNamespaceHandler.java
f5: /camel/impl/DeferServiceStartupListener.java
f6: /camel/CamelContext.java
f7: /camel/impl/DefaultRouteContext.java
f8: /camel/model/RouteDefinition.java
f9: /camel/model/ProcessorDefinition.java
f10: /camel/management/DefaultManagementAgent.java
f11: /camel/management/mbean/ManagedRoute.java
f12: /camel/Route.java
f13: /camel/management/mbean/ManagedCamelContext.java
Time line:
w229
w230
w231
w232
w233-w234

Fig. 2: The Life Cycle of an Active Hotspot in Camel

In the next evolution window, $w230$, $f6$ became a seed file since it was revised by two different issues, and four more seed files were detected. As a result, these seeds files and the propagation files expanded the hotspot to 13 files. In $w231$, the hotspot still had 13 files, but $f9$ replaced $f12$ as a seed file. In $w232$, $f1$, $f6$, $f4$, and $f11$ remain to be seed files but the hotspot shrank to 9 files. In $w233$ and $w234$, the hotspot further shrunk to 5 files.

This example is extracted from nine years of Camel's evolution history that consists of 6390 issues. As we can see from the dataset, $f1$ appeared to be a seed file as early as in the first evolution window, and $f11$ became a seed file in $w93$. These two files remain to be actively leading hotspots all through the lifecyle of Camel. In window 235, $f11$ remained to be the seed of a hotspot with files. After that, this group became less active until window 273 when $f1$ and $f11$ started to lead a 5-file hotspot again. These files remains to be active till the very last of the evolution history, window 315, when the hotspot grew to 17 files.

IV. Hotspot Detection

Based on these definitions, we detect *hotspots* as follows: we first extract all the seed files within any a given time period, and then find all the files that participated with these seed files through any of the four propagation patterns. The seed files and all propagation files form an *active hotspot*. Figure 3 depicts the overall process of our approach with the following four steps:

Step 1: *Detecting seed files.* This step takes issue records and revision history records as inputs, as well as a specified period of time, and outputs a set of seed files. Specifically we extract issue IDs from a system's issue tracking system and match them with its commit history, with the assumption that if a system is well managed, a commit should be associated with an issue ID. We extract the source files committed to resolve each issue, and output files that were changed for at least two issues as a set of seed files. In this step, all test files are filtered out as we focus our investigate only on source files.

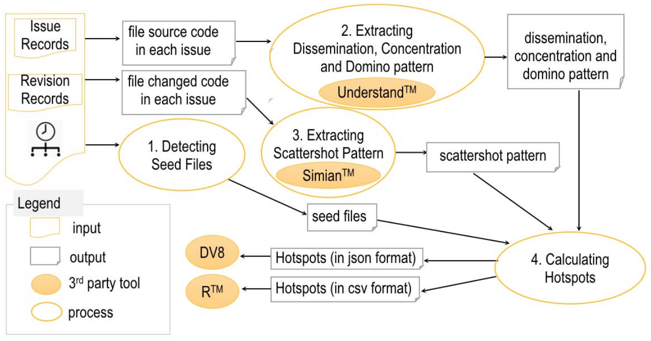
Issue Records
Revision Records
file source code in each issue
file changed code in each issue
2. Extracting Dissemination, Concentration and Domino pattern
Understand™
dissemination, concentration and domino pattern
1. Detecting Seed Files
3. Extracting Scattershot Pattern
Simian™
scattershot pattern
seed files
Legend
input
output
3rd party tool
process
DV8
R™
Hotspots (in json format)
Hotspots (in csv format)
4. Calculating Hotspots

Fig. 3: Overview of our Approach

Step 2: *Extracting dissemination, concentration and domino patterns.* In a given period of time, for each commit, we use the command *git show* to extract all the source files changed by that commit, and use a reverse engineering tool Understand [17] to extract syntactic dependencies among these source files, such as *inherit*, *call* and *use* etc. This dependency information can be used to detect *dissemination*, *concentration*

Algorithm 1: Generate Hotspots from Seedfiles and a list of Propagation Patterns

```
1  Function generateHotspot ($seedfiles, pps$);
   Input : $seedfiles$ and a list of propagation patterns $pps$
           with each pattern represented by $(V, E)$
   Output: A list of hotspots with each $hotspot = (V, E)$
2  Initialize $fileset$ as a Set;
3  for each $seedfile$ in $seedfiles$ do
4      for each pp in pps do
5          if $seedfile$ is one node in pp then
6              Put all nodes files in pp into $fileset$;
7          end
8      end
9  end
10 Put all $seedfiles$ into $fileset$;
11 Change $fileset$ to a list: filelist;
12 Initialize a UnionFind $uf$ with the size of $filelist$ ;
13 for each pp in pps do
14     if $fileset$ contains any two files(f1, f2) in pp then
15         $index1 \leftarrow filelist.index(f1)$;
16         $index2 \leftarrow filelist.index(f2)$;
17         $uf.union(index1, index2)$;
18     end
19 end
20 Initiate a list of hotspots with each hotspot = $(V, E)$;
21 Populate file names in the same group in UnionFind $uf$
       as the nodes of one hotspot into hotspots;
22 for each hotspot in hotspots do
23     for each pp in pps do
24         if pp contains a edge between any two files(f1,
             f2) in hotspot then
25             Populate this edge into this $hotspot$;
26         end
27     end
28 end
29 return $hotspots$;
```

and *domino* patterns among the changed source files. For example, if one changed file *call* multiple other changed files, then a *concentration* pattern will be detected.

Step 3: *Extracting scattershot pattern.* In a given period of time, for each commit, we use *git diff* to extract added, deleted, or changed lines in each source file. After that, we apply *Simian* [18] to detect similar code pieces (clones) among the changed code entities of all source files. If any two source files have at least one similar changed line, then we say these two files have a clone relation and three files with a clone relation form a *scattershot* pattern.

Step 4: *Calculating active hotspots.* Using the seed files output from step 1 and the four pattern information from steps 2 and 3 as inputs, this step generates a set of mutually exclusive *hotspots* in both csv and json formats, so that these hotspots can be visualized using other tools. Specifically, we bind files participated in the same patterns with a seed file with the seed file into a hotspot. If another hotspot is sharing one or more same files with this hotspot, then these two hotspots will be merged together into one single hotspot. The details are shown in Algorithm 1. Our R-script can be used to visualize issues' interaction shown in Figure 2 and the detailed pattern information can be shown in DV8.

V. Empirical Study

The objective of our empirical study is to understand the nature and potential of active hotspots—file groups generated from evolution history using issues as first-class analysis artifacts. For this purpose, we studied the evolution history from 21 large-scale open source projects, and their basic data are shown in Table I. As we introduced in Section III, we split the project's evolution history into multiple sliding *evolution windows*, each consisting of 100 issues that modify at least one source file, and each window is 20 issues apart from the next. We choose 100 issues as a window size as our previous approach [19] shows that 100 issues can provide valuable information with a reasonable size. We will discuss this threshold in Section VI.

As shown in Table I, the number of evolution windows differs for different projects, ranging from 29 to 317, depending on the length of the history and their activeness, i.e. the number of issues that are resolved (We do not consider open issues in this experiment as they are continue changing). The table also shows that the number of hotspots detected within each window ranges from 1 to 38. The average number of hotspots detected within a evolution window is 17.5.

We investigate 5 research questions. The first three help us understand active hotspots in terms of their life cycles and sizes. The last two test the potential of using active hotspots to detect problematic files and architecture debts, in terms of timeliness and efficiency.

RQ1: How files are distributed among the multiple hotspots detected within a period of time? That is, do files tend to aggregate into a few hotspots, or are they usually randomly distributed? If a small number of hotspots always attract a large number of files, then these hotspots are more likely to have architecture issues, and thus deserve special attention. We would like to understand if such *dominating* hotspots exist in most projects, and if so how many such dominating hotspots can be normally found.

RQ2: Do active hotspots tend to be long lasting—remaining active over a system's evolution—or are they transient? Since a given time period of evolution usually contains multiple hotspots it is possible that some hotspots are more long-lasting than others. Active hotspots that persist over a long period of time are change-prone by definition and should be a focal point of interest as these are potential architecture debts. If most projects have such long-lasting hotspots, this implies that architecture debts are ubiquitous. In this case it is important not only to track these change-prone file sets, but also to reveal their architectural relations and how changes/bugs propagate.

RQ3: Are the sizes of hotspots correlated with the sizes of projects? Research [20] has shown that the sizes of files

TABLE I: Subjects and Hotspots

	Start Time	#File Range	#Issues	#Windows	Min #AH	Max #AH	Avg #AH	#File_DominatingAH / #File_AllAH	LongestHotspotLife (in months)
Accumulo	2011-10	1087∼1656	1696	80	11	31	20.1	63.4%	66
Ambari	2011-08	518∼4200	4418	216	6	26	15.9	72.2%	74
Avro	2009-04	264∼521	677	29	9	22	16.1	74.5%	68
Calcite	2012-04	1622∼1746	992	45	6	21	12.5	81.7%	46
Camel	2007-03	6483∼16373	6390	315	11	31	19.7	47.8%	115
Cassandra	2009-03	611∼2010	5006	246	2	24	11.2	82%	94
Cxf	2008-04	4229∼6403	3321	162	13	33	20.4	40.7%	92
Derby	2004-08	2828∼2927	2043	98	11	29	19.1	63.7%	133
Hadoop	2009-05	2218∼7060	2407	116	8	29	17.7	60%	67
Hbase	2007-04	671∼1320	6422	317	1	28	13.1	78.1%	61
Kafka	2011-08	140∼1539	1427	67	9	23	15.7	74.8%	47
Kylin	2014-05	668∼1393	1211	56	9	35	18.8	73.6%	40
Mahout	2008-01	1200∼1220	683	30	16	30	23.9	57.3%	83
Maven	2003-09	830∼965	1238	57	6	27	16.8	67.7%	153
OpenJpa	2006-05	1312∼4582	1165	54	10	27	18.0	70.2%	111
Pdfbox	2008-02	585∼1007	1766	84	11	28	19.2	53.1%	77
Pig	2009-03	1527∼1766	1615	76	5	28	14.4	71.5%	91
Spark	2010-03	131∼893	986	45	5	34	22.0	56.8%	41
Tika	2007-03	194∼1040	1172	47	8	28	18.9	60.3%	103
Wicket	2004-09	2174∼2954	3196	150	10	38	19.6	30.6%	77
Zookeeper	2008-05	354∼474	721	32	10	23	13.9	74.7%	99
Min			677	29	1	21	11.2	30.6%	40
Max			6422	317	16	38	23.9	82.0%	153
Average			2312	111	8	28.3	17.5	64.5%	82.8

Min/Max/Avg #AH: mininum/maximum/average number of active hotspots among all evolution windows

and projects have strong correlations with several measures of interest; essentially, bigger projects and bigger files score worse for almost every measure of productivity and quality that a project manager cares about. For example, the larger the project, the more code smells can be found; the larger a file is, the more likely it is error-prone. Here we would like to understand if the size of active hotspots is also correlated with the size of the project. If hotspots are not correlated with size then we could feel greater confidence that they are truly measuring debt.

RQ4: If we consider the identification of problematic files as an information retrieval problem, and we use active hotspots to detect such problematic files, compared with state-of-the-art smell detection techniques, do active hotspots provide better precision and recall? Ideally, a tool should have a balanced precision and recall, to help a user pinpoint problematic files.

RQ5: If we use active hotspots to detect architecture debts, compared with state-of-the-art techniques, can they reveal the existence of architecture problems earlier? Since we are tracing software evolution by tracking issue interactions, we hypothesize that active hotspots can reveal architectural problems earlier in a project's timeline than existing approaches.

Next we introduce our empirical study of these questions in each of the subsections.

A. Dominating Hotspots

Table I shows that evolution windows usually contain multiple hotspots. We want to examine how files are distributed among these hotspots. Our first observation is that in any window there usually exist a few hotspots that are dominating, meaning that they attract the majority of files that were changed during that time period. Consider hotspots with 5 or more files (which we set as the minimum threshold for a file group to be considered to have structural impact [21]) as dominating hotspots. We observe that, out of all 2322 windows, only 69 windows in 3 out of 21 projects have no hotspots of 5 or more files. This means that during most of these projects' evolution, there existed dominating hotspots.

Our second observation is that the number of such dominating hotspots is small. Of all 2322 windows, only 79 of them have more than 5 dominating hotspots. Camel and Ambari have 24 (out of 315) and 12 (out of 216) such windows respectively; all other projects have 6 or fewer. In other words, in the vast majority of windows studied, the number of dominating hotspots ranges between 1 and 5.

Our third and most important observation, as shown in column *#Fl_DominatingAH/#Fl_AllAH* of Table I , is that 30.6% to 82% of files in hotspots are captured by dominating hotspots: on average, about 65% of files are aggregated into dominating hotspots. Of all 21 projects, only the dominating hotspots in Camel, Cxf and Wicket have fewer than 50% of all hotspot files (48%, 41% and 31% respectively) on average. The implication is that, during most of the evolution period, files in hotspots tend to aggregate into a few instances of dominating hotspots.

As an example, Figure 4 shows the hotspot statistics in Zookeeper from 2009 to 2017. The purple line shows the number of dominating hotspots in each window. The blue line shows the file count within them, and the red line shows the total file count of all hotspots. The figure shows that at each evolution window, there are at most 5 dominating hotspots, which, on average, captured 75% of all hotspot files; the red line and the blue lines are very close to each other.

Now we are ready to answer RQ1: indeed, during most evolution periods there usually exist just a handful of big hotspots, which can be called *dominating hotspots*, that link the majority of active files together. These dominating hotspots should be the focal points for analysis.

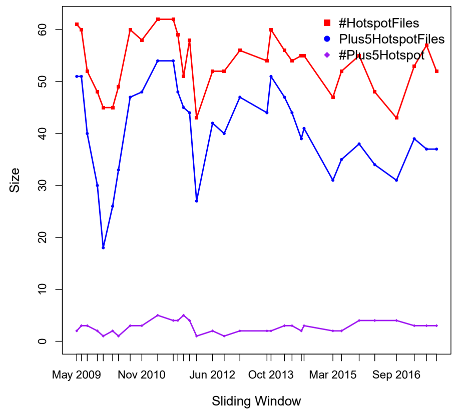

Fig. 4: Zookeeper Hotspot Statistics

B. Persistent Hotspots

Now that we have discovered that at any period of evolution there are just a few dominating hotspots we would like to examine if these dominating hotspots tend to be persistent and long-lasting. If the answer is yes, it mean that these files, especially the seed files, must be change-prone and/or error-prone, and thus have incurred the most maintenance costs.

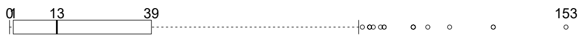

Fig. 5: Dominating Hotspot's Life in Months

We studied 560 dominating hotspots covering all 21 projects. The boxplot in Figure 5 shows that their lifespan ranges from 0 months to 153 months, with an average 24.6 months and a median 13.0 months. As shown in the last column of Table I, the most persistent hotspot within each project lasts between 40 (Kylin) to 153 months (Maven), and the average is 83 months.

As an example, Figure 6 shows the persistence of Zookeeper's dominating hotspots. In this figure, the first column shows the starting date of an evolution window. The numbers in the diagonal cell are the total number of dominating hotspots, and the number in $cell(i,j)$ $(i > j)$ refers to the number of hotspots that existed in earlier window j and remain in the current evolution window i. For example, $cell(4,4)$ shows that there are 3 hotspots in window 4: 2014-04-16. $Cell(14,1)$ shows that there are 2 hotspots that last from window 1: 2013-09-18 to window 14: 2017-08-10.

Now we are ready to answer RQ2: dominating hotspots can last a long time; they are persistent and long-living which, again, shows that they should be the focal points of analysis.

w1	**2013-09-18**	2													
w2	**2013-10-22**	2	2												
w3	**2014-02-12**	2	2	3											
w4	**2014-04-16**	2	2	3	3										
w5	**2014-07-03**	1	1	2	2	2									
w6	**2014-07-26**	2	2	3	3	2	3								
w7	**2015-03-25**	2	2	3	3	1	2	2							
w8	**2015-06-05**	2	2	3	3	1	2	2	2						
w9	**2015-10-31**	2	2	3	3	1	2	2	2	4					
w10	**2016-03-10**	1	1	2	2	2	2	1	1	2	4				
w11	**2016-09-08**	1	1	2	2	2	2	1	1	3	3	4			
w12	**2017-02-04**	2	2	3	3	2	3	2	2	4	3	4	3		
w13	**2017-05-18**	2	2	3	3	2	3	2	2	4	3	4	3	3	
w14	**2017-08-10**	2	2	3	3	2	3	2	2	4	3	4	3	3	3

Fig. 6: Zookeeper Dominating Hotspot Persistence

C. Correlation with Project Size

Given these results, we now hypothesize that hotspots can be used to detect problematic files and architecture problems during evolution. Prior research [20] has shown that the sizes of files and projects have strong correlations with several measures of interest. Essentially, bigger projects and bigger files score worse for almost every measure of productivity and quality that a project manager cares about: the larger the project, the more code smells can be found; the larger a file is, the more likely it is error-prone.

Here we investigate whether the number of files involved in hotspots is also correlated with project size. If the answer is yes, then it is possible that hotspots are not very different from other analysis techniques; if the answer is no, it means hotspots are independent and even when a project grows the development team won't be overwhelmed with increasing numbers of reported flaws. To assess this hypothesis, we compared the numbers of distinct problematic files reported by hotspots and by other tools, and calculated their correlations with project sizes, in terms of file counts and lines of code(LoC).

As an example, Table II shows the number of files reported as having code/design problems detected by all 6 tools from tika 0.5 to tika 1.19. The data reveal that, except for *hotspots*, when the Tika project increased from 194 files to 1040 files, the number of reported files by all these tools monotonically increased. For example, files with smells detected by Designite increased from 176 to 654 (91% to 63% of the project files). By contrast, the *hotspot* approach is the only one in which the file counts are not correlated with project sizes.

Since Structure101, SonarGraph and SonarQube require successful builds of a project (by Maven or Gradle) as input, Tika is the only project where we could build a series of snapshots from its source code in its Git repository. As a result, to test if this observation is generalizable, for other projects we only compared the number of files with problems reported by Designite and DV8 with *Hotspots* because these tools only require source code as input. Using 123 snapshots from our subjects, we first did a Shaprio-Wilk normality test to see if the samples reported by these tools follow a normal

TABLE II: Smells Detected in Tika

Version	#File_Designite	#File_DV8	#File_SonarQube	#File_SonarGraph	#File_Structure101	#File_Hotspot	#File_Proj	LoC_Proj
tika-0.5	176	102	94	9	110	34	194	26160
tika-1.7	371	227	224	42	335	40	552	97143
tika-1.10	465	308	318	62	573	73	739	122868
tika-1.14-rc1	519	389	353	74	646	84	835	138167
tika-1.16	599	399	420	80	757	54	975	163515
tika-1.19	654	453	464	87	837	52	1040	178078

TABLE III: Shapiro-Wilk Normality Test

	#File in Designite	#File in DV8	#File in Hotspot	Proj	
				#File	Loc
W	0.87	0.88	0.87	0.77	0.83
P-Value	8.4E-09	2.3E-08	5.9E-09	1.9E-12	9.7E-11

TABLE IV: Pearson Correlation

	#File_Designite |		#File_DV8 |		#File_Hotspot |	
	#File_P	LoC_P	#File_P	LoC_P	#File_P	LoC_P
Coefficient	0.80	0.83	0.83	0.85	-0.21	-0.10
P-Value	2.2E-16	2.2E-16	2.2E-16	2.2E-16	0.02	0.27

distribution, which is confirmed as shown in Table III. Then we did a Pearson correlation test between the reported file counts, the total file counts, and total lines of code in each snapshot. Table IV shows that files with smells detected by Designite are highly correlated with project file counts and lines of code with a coefficient 0.80 and 0.83. DV8 produces similar results. Only the number of files captured by *Hotspots* are not correlated with size, with Pearson coefficients of -0.21 and -0.10.

Now we are ready to answer RQ3: the number of files that should be the focal points of analysis is solely related to the intensity of issue interactions and the architectural relations among files, and not to project or file sizes.

D. Capturing Bug/Change-Prone Files

TABLE V: Predicting Change-Prone and Bug-Prone files

	Change2			Change3		
	Precision	Recall	F1	Precision	Recall	F1
Designite	3.1%	49.2%	5.9%	1.8%	56.4%	3.4%
Sonarqube	7.5%	52.8%	13.1%	4.0%	59.8%	7.5%
Sonargraph	9.9%	29.0%	14.7%	6.0%	33.6%	10.2%
Structure101	12.6%	9.5%	10.8%	12.1%	17.4%	14.3%
DV8	5.7%	49.7%	10.3%	2.9%	55.6%	5.6%
Hotspot	23.2%	21.7%	22.4%	13.7%	27.8%	18.4%

	Bug2			Bug3		
	Precision	Recall	F1	Precision	Recall	F1
Designite	5.1%	61.1%	9.3%	3.7%	66.4%	6.9%
Sonarqube	11.6%	61.9%	19.5%	9.0%	74.6%	16.1%
Sonargraph	13.1%	34.7%	19.0%	11.7%	38.0%	17.9%
Structure101	18.8%	13.3%	15.6%	19.3%	16.6%	17.8%
DV8	9.3%	59.4%	16.1%	6.9%	74.0%	12.7%
Hotspot	44.1%	30.4%	36.0%	38.6%	39.2%	38.9%

Files in hotspots, by definition, are change-prone and/or error-prone since they are constructed by *seed files* and propagation files that changed together with them. For this reason we hypothesize that hotspots could be leveraged for change/bug prediction. Here we compare hotspots and files with smells reported by other tools in a given snapshot, in terms of their ability to capture bug-prone and change-prone files in the next snapshot.

To investigate this question, we chose 15 out of the 21 projects from which we could successfully create a build for a release so that we could compare all 6 tools including SonarQube, SonarGraph and Structure101. For each of the 15 projects, we chose a release in 2018 so that we could use the most recent evolution period as an "oracle". Since hotspots and DV8 both require revision history as input, we used the revision log that includes 100 bugs before the selected releases for both tools, to make a fair comparison.

To form an oracle data set for error-proneness and change-prone prediction, we included 100 bug-fixes and 100 general fixes after the selected release date. All the files that had been modified by 2 or more bug-fixes form a Bug2 file set; Similarly, all the files that had been modified by 2 or more general fixes (both bug and non-bug fixes) form a Change2 file set. Bug3 (including files modified by 3 or more bug-fixes) is a subset of Bug2, Change3 is a subset of Change2, so on and so forth.

Table V shows the results for all 6 tools. We excluded Change1 and Bug1 since files belonging to Change 1 but not Change2 (i.e., files that are only changed once) should not be counted as change-prone or error-prone. The table shows that hotspots and Structure101 have relatively low recall and high precision because they reported smaller numbers of files. The other 4 tools have high recall at the expense of low precision because they report a large number of files. The F1-score provides a more balanced measure: the grey cells indicate that hotspots always have the best F1-score. Take Bug2 for example: on average, 44.1% of all the files in dominating hotspots have bugs at least two times in the next evolution period, and they cover about 30.4% of all the error-prone files.

Besides, though other tools report a large number of problematic files, hotspots detected 11 files which can not be detected by other tools. We manually inspected these files' source code and their relations with other files and identified these files as flawed files. And these files continuously incur bugs or changes after the selected releases.

These data illuminate RQ4: compared with other tools, hotspots can capture error-prone and change-prone files more effectively, and thus have the potential to be used for bug/change prediction and localization. Since dominating hotspots—a big directed graph with propagation relations—are the majority of hotspot files, it means that these files contributed significantly to error and change-proneness, which we propose is through their propagation patterns.

E. Early Detection of Architectural Problems

TABLE VI: Hotspot can Reveal Architecture Flaw Early

	Detected in DV8	Detected in Earlier Hotspot	Average Months Advanced Detected in Hotspot
Unstable Interface	128	111 (86.7%)	25
Crossing	264	210 (79.5%)	18

Our main objective, by focusing on dominating hotspots, is to detect architectural problems early so that they can be prevented from growing too big or too complex. We would like to examine if hotspots can reveal severe architectural problems earlier, compared with other architectural debt detection tools.

This comparison is challenging. Consider dependency cycles, which are checked by most tools, as an example. Even though an active cycle can be observed immediately if we extract changed files' relations during one fix, we cannot compare it with the cycles detected by other tools systematically because the members of a cycle change over time and it is impossible to uniquely trace a cycle especially when different tools define cycles slightly differently. Similarly, for *Fat files*, *God classes*, etc., it is intuitive that if a file is repeatedly changed then it may be a hotspot and may become overly complex. To check how early these problems can be detected, we would need to create multiple builds to run Sonarqube, Structure101, etc., which is not practical.

Since hotspots focus on files repeatedly changed by multiple issues, we wanted to investigate if and to what extent these files were architecturally important. The earlier such files can be identified, the more likely their problems can be promptly detected and fixed. Among the 6 types of anti-patterns reported by DV8, two of them also detect files with architectural consequences and contribute most of the maintenance cost. One is *Unstable Interface*—a file that has many dependents and co-changed with them often, and the other is *Crossing*—a file with high fan-in and fan-out that co-changes frequently with both its dependents and the files it depends on. Since both anti-patterns detect architecturally important files and their impact scope, we examine if hotspots can identify these key files early.

To conduct this experiment, we first detected unstable interfaces and crossings from a series of snapshots in a project, and noted the snapshot time when they were first detected. We then scanned our *hotspots* starting from the first snapshot's release time to the last snapshot's release time to examine: 1) whether the unstable interface and crossing files identified in DV8 can be found in a hotspot; 2) if and how much earlier these files can be detected than when they are first detected by DV8.

The results are as follows: DV8 detected 128 unstable interfaces and 264 crossing files from all the projects. Of these files, 111 out of 128 unstable interfaces are detected within hotspots, and 210 out of 264 crossing files are detected within hotspots. We manually examined the 71 files which we did not find in our *hotspots* and found that 49 of these files are in the Hadoop project, which is an ecosystem and use different issue ids (such as HDFS-[0-9]+ or YARN-[0-9]+) to mark commits in other modules. In our experiment we used HADOOP-[0-9]+ to extract issues. These 49 files can be found in our *hotspots* if we modify our issue id system. The other 22 files are neither seed files nor files connecting to seed files, and can not be detected within hotspots. These 22 files are all structurally connected to other files, but their co-changes are not captured by multiple issue fixes. Thus we know that 94.4%$((111+210+49)/(128+264))$ of these key files can be identified through hotspots.

Of these files that are captured by hotspots, they are all detected earlier than by DV8, as shown in Table VI. On average, hotspots can detect unstable interfaces and crossings 25 months and 18 months earlier respectively.

Using Figure 2 as an example. $f1$ and $f6$ signal the most severe architectural problem: many files depend on them, and they change together frequently with many other files; this is a typical *unstable interface* flaw [22]. Out of the 315 evolution windows examined, these two files are part of a hotspot 257 and 121 times respectively. Our tool detected these two files in hotspots during the first window. DV8, by contrast, will not flag them as problematic until enough revision history has been accumulated to indicate that other files are regularly changing together with $f1$ and $f6$. These problems could easily go unnoticed until they incur severe costs. Actually, given the default settings of DV8, using the same revision history as hotspots, DV8 wouldn't have reported them at all, unless we changed the settings, or provided longer revision history.

As another example—a hotspot found in window 70 (2010-06-17) in Derby—the file *Connection.java* has 4 fan-outs and 2 fan-ins, but it is not until this grows to 7 fan-ins and 5 fan-outs (in version 10.7.1.1 2010-12-14) that DV8 reported it as a Crossing since its default setting are 4 fan-ins and 4 fan-outs. Using hotspots, the user does not need to consider how to change the tool settings. Instead, they just need to select evolution periods of interest, such as a sprint, a release, or the time after refactoring.

This then answers RQ5: by closely monitoring software evolution through issues, it is possible to identify architectural problems earlier in the lifecycle.

VI. Discussion

Hotspots and Architectural Anti-patterns. In section V-E, we prove that hotspots can detect architectural important files: unstable interface and crossing earlier than DV8. The rational behind this is that hotspots are constructed through issue interactions (e.g. files addressing two issues are overlapped), which shows early sign of architecture problems. If two issues are addressing different problems, then the shared files violate the single responsibility principle. If two issues are addressing the same problems, then the design of shared files also deserve further investigations. In addition, hotspots also incorporate a small architecture unit which shows changes propagate among multiple files in a real system. These propagation patterns and their combinations can indicate architecture flaws. For example, we observed that the last file in one domino instance is also the first file of another domino instance and, though these two domino chains, changes propagate not only among multiple files, but also across multiple packages/subsystems. If the chain is long and files in this chain are in different packages, files may be forgotten during bug fixing. And this is similar to an anti-pattern "Message Chain" [23] defined by Fowler et al. We also have similar observation with other patterns and their combinations. So we can say each propagation pattern is an atomic relation forming higher level and more severe architecture problems. Combining issue interactions with these propagation patterns, hotspots can represent the most active architectural unit and indicate possible architectural problems to avoid further architectural decay.

Next we discuss the limitations of our proposed models and the threats to validity of our study.

Limitations. First the accuracy of our study depends on the quality of project revision histories. Recent studies [24], [25] state that it is possible for one commit to fix multiple issues, or, for a commit to not be explicitly linked to an issue. Our analyses assume that files fixing an issue are related to each other, though we do remove issues which modified more than 10 source files, to reduce noise. We intend to further assess this threat via qualitative analysis.

Second, our definition of *active hotspot* uses *seed* files and includes files with propagation relations with them. There may exist more effective ways to find more precise hotspot, e.g. at method-level instead of file-level. Improving and further evaluating the hotspot detection algorithm is future work.

Finally, the activeness of a system will significantly influence the detected hotspots. Consider Avro: this system has not been very active in recent years. As a result, few hotfiles can be found and the detected hotspots are small. It is possible that there are many files in the system still have code, design, or architecture problems. However, if the system remains to be inactive, these problems may not generate extra maintenance costs, and hence shouldn't be counted as technical debts. Investigating how to assess the activeness of a project, and how the activeness influences our models, are parts of our future work.

Threats to Validity. One major threat to validity of our empirical study is the choice of using 100 issues as the unit of evolution window. It is not clear if the results will differ if we chose a different length, or how to optimally choose the length of an evolution window. Conducting sensibility analysis on the sizes of evolution windows is our future work. In practice, an evolution window could be a sprint, a release, a period after refactoring, etc.

Another threat to validity is choosing 5 files as the threshold of being a dominating hotspot. We will also test the sensibility of this threshold in the future. Similar to the choice of 100 issues as the unit of evolution windows, in practice, the existence of dominating hotspots will be prominent, and there is no need to set such a threshold.

An external validity is that we only studied 21 open source projects, all of which were implemented in Java and all of which are from the Apache ecosystem. While we have no reason to assume that the effects that we have studied would be different with different programming languages, we can not currently claim that our results can be generalized to projects implemented using other languages. Similarly we have no expectation that our results would differ in closed-source contexts, or projects from other OSS ecosystems, but this remains a potential threat.

VII. Related Work

Software evolution. Chapin et al. [26] proposed a method to classify evolution types based on a semi-hierarchical manner of the change in four aspects from documentation to functionality. Godfrey et al. [27] compared software evolution with biological evolution, and examined how software evolves in response to environmental pressure and emergent design. Tu et al. [28] studied the evolution of the Linux kernel and discussed why Linux continues to exhibit such strong growth. Robles at al. [29] studied the evolution of a Linux distribution, Debian, with respect to its overall size, use of programming languages, maintenance of packages, and file sizes over seven years.

Architecture evolution has also been explored [30]–[32]. Chaikalis et al. [30] incorporated structural and domain information, such as the creation of relations among existing and new classes and the removal of edges, into a network-based prediction model. They used 10 open-source projects to evaluate this model and showed that their derived models can provide insight into future trends. Zimmermann [31] examines architectural refactoring as an evolution technique that revisits decisions and identifies related design, implementation, and documentation tasks. Xiao et al. [32] also studied the evolution of architectural debt and showed how such debts grow. The above studies on software evolution focus on the evolution of software architecture, knowledge bases, and so forth. Our study is complementary to these studies, by investigating software evolution through one of the smallest architectural unit:*active hotspot* and studying *active hotspot*'s evolution continuously by sliding issue's winodws instead of focusing on separate releases.

Architecture problem identification. The following work is the state-of-art in terms of identifying architectural problems underlying high-maintenance. Xiao et al. [32], [33] revealed that most error-prone files often connected into just a few file groups, and proposed 4 types of architectural debts. Mo et al. [22] proposed five file-level *architectural flaws* and proved that these flaws are highly correlated with error-proneness and change-proneness. Mondal et al. [34] investigated bug propagation through code clones and conclude that up to 33% of code clones in a bug fix can be related to bug propagation. Our study identifies three other common ways for a bug to propagate, and use them to construct *active hotspot*.

Different from this prior work, our approach is the first attempt to monitoring software evolution by tracking how issues interact. The concept of *active hotspot* takes into consideration the temporal natural of architecture connections, and, as we will explain later, enables more precise and timely identification of architecture problem that may grow into severe debts. Hence designers can treat these problems early before incurring significant loss.

Change Propagation As defined in [35], to guarantee the proper funtionality of a software system, when a particular entity is changed, it is required to change other entities of the software system. Researchers [35]–[37] have constructed a change propagation model from historical co-change, code structure, name similarity, etc., and measure the model's precision and recall. Furthermore, researchers have been analyzing more fine-grained change impact [38], [39]. For instance, Chianti first decomposes differences between two programs into atomic changes such as Changed Methods(CM) and Added Field (AF) [38]. Wang et al. [39] did an empirical study of multi-entity changes in real bug fixes and they discovered six recurring patterns. Similar to their techniques, we also studied propagation patterns. However, to construct one of the smallest architectural unit at the file level, we studied files' propagation patterns. We discovered 4 recurring propagation patterns at the file level and files involves these patterns tend to propagate changes to other files.

Code Smells/Anti-patterns. Numerous studies [40]–[45] claimed that code smells and anti-patterns are the causes of significant maintenance costs. Many tools [46]–[54] have been developed to reveal code smells and anti-patterns. However, recent studies [20], [55] suggest that code smells, which indicate bad design, are *not* the root causes of maintenance costs. Instead, file size and the number of revisions are more strongly correlated with maintenance costs. Using popular code smell detection tools, developers can detect numerous code smells, but not all of them are real issues or incur high maintenance costs. Our study focuses on bug propagation patterns that have already shown they incur high maintenance costs. Using these an analyst can focus on the true maintenance difficulty in a project, and pinpoint the underlying design problems at the finest granularity. As discussed in V-D, we compared our *active hotspot* with 5 other state-of-the-art smell detection tools and proved *active hotspot* can use less files to capture more maintenance files in the future.

VIII. Conclusion

In this paper, we proposed a model called *active hotspots* that can be used to monitor software evolution and detect potential degradation, using issues as first-class entities. An active hotspot is formed by seed files that are changed by multiple issues, and the files that are connected with them through one of the 4 propagation patterns. Using active hotspots as lenses, we studied the evolution history of 21 open source projects. The data revealed that within any evolution period, the majority of files revised for issue fixing are just aggregated into a few dominating hotspots, and most of these dominating hotspots are persistent and remain active for a long time, implying that these dominating hotspots should be the focal points of activity and deserve special attention.

Different from most code, design, or architecture smell detection tools, the number of files within hotpots do not increase with the size of the project, meaning that developers won't be overwhelmed by increasing numbers of reported flaws as the project grows. We also showed that hotspots have the potential to be leveraged for bug and change localization. Most importantly, by monitoring the emergence and evolution of hotspots, it is possible to detect architectural problems early so that these problems won't accumulate into severe maintenance costs.

References

[1] M. Feathers, *Working effectively with legacy code*. Prentice Hall Professional, 2004.

[2] J. Kerievsky, *Refactoring to Patterns*. Addison-Wesley, 2004.

[3] J. Carriere, R. Kazman, and I. Ozkaya, "A cost-benefit framework for making architectural decisions in a business context," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 2. IEEE, 2010, pp. 149–157.

[4] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the principal of an application's technical debt," *IEEE Software*, vol. 29, no. 6, pp. 34–42, 2012.

[5] ——, "Estimating the principal of an application's technical debt," *IEEE software*, vol. 29, no. 6, pp. 34–42, 2012.

[6] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, P. Abrahamsson, A. Martini, U. Zdun, and K. Systa, "The perception of technical debt in the embedded systems domain: An industrial case study," in *Managing Technical Debt (MTD), 2016 IEEE 8th International Workshop on*. IEEE, 2016, pp. 9–16.

[7] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman, "Are developers aware of the architectural impact of their changes?" in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 95–105.

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

[9] 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.

[10] A. Martini and J. Bosch, "On the interest of architectural technical debt: Uncovering the contagious debt phenomenon," *Journal of Software: Evolution and Process*, vol. 29, no. 10, p. e1877, 2017.

[11] S. SonarSource, "Sonarqube," *Capturado em: http://www. sonarqube. org*, 2013.

[12] "Structure101." [Online]. Available: https://structure101.com/

[13] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, 2019.

[14] R. Mo, W. Snipes, Y. Cai, S. Ramaswamy, R. Kazman, and M. Naedele, "Experiences applying automated architecture analysis tool suites," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 779–789.

[15] M. DAmbros, H. Gall, M. Lanza, and M. Pinzger, "Analysing software repositories to understand software evolution," in *Software evolution*. Springer, 2008, pp. 37–67.

[16] R. C. Martin, "The single responsibility principle," *The principles, patterns, and practices of Agile Software Development*, vol. 149, p. 154, 2002.

[17] "Understand." [Online]. Available: https://scitools.com/

[18] "Simian." [Online]. Available: https://www.harukizaemon.com/simian/

[19] Q. Feng, Y. Cai, R. Kazman, and R. Mo, "The birth, growth, death and rejuvenation of software maintenance communities," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2018, p. 5.

[20] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013.

[21] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: a new metric for architectural maintenance complexity," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 499–510.

[22] 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.

[23] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Jul. 1999.

[24] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 2013, pp. 121–130.

[25] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 134–144.

[26] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of Software: Evolution and Process*, vol. 13, no. 1, pp. 3–30, 2001.

[27] M. W. Godfrey and D. M. German, "The past, present, and future of software evolution," in *Frontiers of Software Maintenance, 2008. FoSM 2008.* IEEE, 2008, pp. 129–138.

[28] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Software Maintenance, 2000. Proceedings. International Conference on*. IEEE, 2000, pp. 131–142.

[29] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, and J. J. Amor, "Mining large software compilations over time: another perspective of software evolution," in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 3–9.

[30] T. Chaikalis and A. Chatzigeorgiou, "Forecasting java software evolution trends employing network models," *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 582–602, 2015.

[31] O. Zimmermann, "Architectural refactoring: A task-centric view on software evolution," *IEEE Software*, vol. 32, no. 2, pp. 26–29, 2015.

[32] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proc. 38th International Conference on Software Engineering*, 2016.

[33] L. Xiao, Y. Cai, and R. Kazman, "Design rule spaces: A new form of architecture insight," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 967–977.

[34] M. Mondal, C. K. Roy, and K. A. Schneider, "Bug propagation through code cloning: An empirical study," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 227–237.

[35] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *Proc. 20th IEEE International Conference on Software Maintenance*, Sep. 2004, pp. 284–293.

[36] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[37] H. Kagdi, J. I. Maletic, and B. Sharif, "Mining software repositories for traceability links," in *15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE, 2007, pp. 145–154.

[38] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *ACM Sigplan Notices*, vol. 39, no. 10. ACM, 2004, pp. 432–448.

[39] Y. Wang, N. Meng, and H. Zhong, "An empirical study of multi-entity changes in real bug fixes," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 287–298.

[40] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.

[41] D. Siewiorek and R. Swarz, *Reliable Computer Systems: Design and Evaluatuion*. Digital Press, 2017.

[42] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, pp. 129–143, 2004.

[43] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 227–236.

[44] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software maintenance and reengineering (CSMR), 2011 15th European conference on*. IEEE, 2011, pp. 181–190.

[45] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 2009, pp. 75–84.

[46] I. M. Bertran, "Detecting architecturally-relevant code smells in evolving software systems," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1090–1093.

[47] R. M. de Mello, R. F. Oliveira, and A. F. Garcia, "On the influence of human factors for identifying code smells: A multi-trial empirical study," in *Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on*. IEEE, 2017, pp. 68–77.

[48] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. B. Chikha, "Competitive coevolutionary code-smells detection," in *International Symposium on Search Based Software Engineering*. Springer, 2013, pp. 50–65.

[49] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 2004, pp. 350–359.

[50] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[51] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 268–278.

[52] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 403–414.

[53] A. Koenig, "Patterns and antipatterns," *The patterns handbook: techniques, strategies, and applications*, vol. 13, p. 383, 1998.

[54] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[55] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 682–691.