Software Architecture Measurement —Experiences from a Multinational Company

Wensheng Wu[1], Yuanfang Cai[2], Rick Kazman[3], Ran Mo[2], Zhipeng Liu[1], Rongbiao Chen[1], Yingan Ge[1], Weicai Liu[1], and Junhui Zhang[1]

[1] Huawei Technologies Co., Ltd., Shenzhen, Guangdong, China
{wuwensheng,zhipeng.liu,chenrongbiao,
geyingan,liuweicai,zhangjunhui3}@huawei.com
[2] Drexel University, Philadelphia, PA, USA
{yc349,rm859}@drexel.edu
[3] University of Hawaii, Honolulu, HI, USA
kazman@hawaii.edu

Abstract. In this paper, we present our 4-year experience of creating, evolving, and validating an automated software architecture measurement system within Huawei. This system is centered around a comprehensive scale called the *Standard Architecture Index* (SAI), which is composed of a number of measures, each reflecting a recurring architecture problem. Development teams use this as a guide to figure out how to achieve a better score by addressing the underlying problems. The measurement practice thus motivates desired behaviors and outcomes. In this paper, we present our experience of creating and validating SAI 1.0 and 2.0, which has been adopted as the enterprise-wide standard, and our directions towards SAI 3.0. We will describe how we got the development teams to accept and apply SAI through pilot studies, constantly adjusting the formula based on feedback, and correlating SAI scores with productivity measures. Our experience shows that it is critical to guide development teams to focus on the underlying problems behind each measure within SAI, rather than on the score itself. It is also critical to introduce state-of-the-art technologies to the development teams. In doing so they can leverage these technologies to pinpoint and quantify architecture problems so that better SAI scores can be achieved, along with better quality and productivity.

Keywords: Software Measurement, Software Architecture, Software Quality

1 Introduction

In this paper, we present our 4-year long experience within Huawei to evaluate, measure, and improve the architectures of their software products. As a multinational company, Huawei is constantly seeking to improve product quality and maintain rapid feature delivery. As with all software systems, as they age and evolve, more and more effort is spent on maintenance. Recognizing the profound influence of software architecture in accommodating a rapidly-changing market environment, improving productivity and efficiency, and shortening the Time-to-Market (TTM) cycle, in 2013, Huawei embarked on a software architecture improvement program.

Research on software measurement has a long history. From McCabe's canonical Cyclomatic complexity [15] to C&K [6], LK [13], and MOOD metrics suites [8] for object-oriented programs, research on code-level metrics continuously evolves. System and architectural level metrics have been proposed as well. For example, Coleman's maintainability index [7] combines multiple measures into a single scale. Quality assessment tools, such as SonarQube, also provide metrics and maintainability ratings[4].

Huawei practitioners, however, found that existing metrics and indexes could not provide sufficient insight into their architectures. First, there is insufficient evidence that these metrics can be used to compare and contrast projects so that management could reliably discern which projects were suffering maintenance problems. Nor could they rely on these metrics to monitor the evolution of a project so that architecture degradation could be detected early. Second, if a system obtained a sub-optimal score, it was not clear what the underlying problems were and hence what should be done to fix these problems so as to achieve a better score. Finally, our understanding of software architecture and evolution advances over time. New concepts, such as code smells and anti-patterns, have been widely accepted but were not reflected in legacy metrics.

To address these shortcomings, Huawei's Research and Development group proposed to institutionalize a standard software architecture measurement system, to uniformly monitor hundreds of products. For this purpose, we conducted extensive research on architecture design, architecture measurement theory, tooling and practice. Since several software architecture measurement tools were already in use by some product teams within Huawei, we also conducted internal research to collect their experiences, which showed that the most commonly applied metrics are at the source code level, and there are few widely used metrics at the design/architectural level. In addition, many of the adopted metrics were based on the experience and intuition of the development teams, without rigorous theoretical foundations.

Based on these early insights, we created the first version of our software architecture measurement standard, SAI 1.0, which adopted multiple software architecture measurement practices, and used the ISO/IEC 25010 software measurement model as a reference. We first used SAI 1.0 to measure open source projects to benchmark and adjust the model, and then conducted internal pilot studies to collect feedback. After two rounds of pilot studies, we evolved the standard to SAI 2.0. This version provided more precise guidance on where the architecture problems are, and the scope of their impact. During this process we were searching for and adopting state-of-the-art technologies to pinpoint, visualize, and quantify the underlying architectural problems. This way, the teams could actually achieve a better score by fixing these problems. Throughout this process we have validated SAI quantitatively by correlating its scores with productivity measures, and qualitatively by interviewing practitioners. SAI 2.0 is now widely accepted within Huawei and we are working towards the creation of SAI 3.0.

Although our objective is to comprehensively measure software architecture, including both code and models, in reality, source files are the only reliable artifacts available. In this paper, we focus on *architecture* as module structures that can be inferred from source files [2], and our *architecture assessment* focuses on relations among files. Our contributions from this experience are as follows:

[4] https://docs.sonarqube.org/display/SONAR/Metric+Definitions

1) First, a successful architecture measurement system should be able to support quantitative comparison of different projects, and to monitor architecture degradation over time. The quantitative score should be supported by quality and productivity data continuously collected through the development process. Most importantly, the components of the score should indicate the existence of flaws in code, design or architecture, so that development teams can understand where improvement is needed.

2) Second, product teams should be equipped with state-of-the-art tools and conduct key activities to pinpoint, quantify, and visualize architectural debts. Using the overall architecture score as the guidance, the teams could use these tools to identify and remove the underlying flaws, and improve quality and productivity, which in turn, will improve the score. These tools and activities are critical for the teams to take actions based on architecture measurement.

3) Finally, an effectively way to convince practitioners to adopt such a measurement system is to demonstrate concrete improvement in terms of quality and productivity through pilot studies, and closely work with practitioners, taking their opinions into consideration and adjust the model accordingly. Once the practitioners experienced the benefits of applying this framework, they are not only willing to measure their systems on a daily basis, but also to leverage and improve these technologies in innovative ways.

2 Objective, Challenges, and Strategies

Our objective in creating a standard architecture measurement system is to improve the quality and productivity of Huawei's software products. The challenge is to have this measure accepted by both management and development teams, and to demonstrate the benefits to both parties. To achieve these objectives, the designers of SAI needed to answer the following questions:

Q1. How to create a software measurement system that adds value and provides practical benefits to development teams?

The measurement system must be based on solid theoretical foundations, and provide clear guidance and benchmarks. An improved score should be associated with improved quality and productivity, and vice versa.

Q2. Is it possible to accurately measure software architecture quantitatively?

The measurement system must be quantitative so that the users can understand to what extent a system has been improved or degraded. If a system is known to have a healthy architecture and has been successful, or if the development team is suffering from maintenance difficulties, then the score should faithfully reflect these conditions.

Q3. How to convince developers to accept and use these measures? If architecture measurement is just an extra task that cannot be integrated into the regular development process, then it will become a burden to the development team. Thus we not only needed to answer the first question satisfactorily, but we also had to convince the development teams to integrate the standard into their development process. If the standard is to be accepted in the long run, developers must see that the application of the standard leads to improved productivity and efficiency.

Q4. How to make these measures actionable? That is, can the development team figure out what to do to get a better score?

Since the ultimate purpose of the measurement system is to improve software quality and productivity, the development team should be able to figure out where and how the software should be improved.

To address these challenges, the designers of SAI, under the leadership of the first author, have applied the following strategies, not only to create a theoretically sound and practically beneficial index, but also to prove its benefits to development teams, thus promoting its acceptance:

1. *Conducting internal and external research.* Since 2013, Huawei has established collaborations with researchers around the world, and conducted extensive research on architecture design, measurement theory, technology, tooling, and practice. We also conducted research internally to understand the different emphases and priorities of teams, and the advantages and limitations of SAI. SAI is thus the result of internal and external research, based on both theory and practice.
2. *Conducting pilot studies and rigorous evaluation.* After each version of SAI was designed, and before it was applied to Huawei projects, we first used the index to measure open source systems to conduct initial evaluations and make adjustments. After that, we solicited Huawei development teams to conduct pilot studies to evaluate the measurement system. The evaluation process integrated interviews to assess if the measures faithfully reflect the practitioners' intuitions, and quantitative analysis to see if the variation of measures had a strong correlation with productivity.
3. *Augmenting measurement with tools and key activities.* To help the developers understand where and how to improve their architecture quality, and hence their measurement score, we provided several tools, including a graphical view of the components of their SAI scores, an architecture guarding tool called UADP, design structure matrix (DSM) [1,23,24] tools, and the associated architectural debt quantification methods [25,12], which we will elaborate in the next few sections.

3 The First Version of SAI

Figure 1 depicts the theoretical structure of our architecture measurement system. As shown in the figure, the ultimate objective of SAI is to improve a suite of quality attributes, including maintainability, reliability, security, performance, etc. [2]. For each quality attribute, we measure the software from three dimensions: 1) structural measures that assess the structural relations among files, modules, and components; 2) class/function measures that assess implementation quality and styles within each source file; and 3) global measures that focus on design and implementation choices with global impacts, such as the density of global variables and the rate of unused APIs. Based on these measures, we defined a comprehensive index, which we call *Product SAI* or *Project SAI*. The higher the SAI, the lower the quality of the product architecture; the lower the SAI, the better the quality. In this section, we present the components of the first version of SAI, its supporting tools, and our pilot study results.

3.1 Computational Model and Benchmarks

Considering the availability of architecture measurement technology and tools, as well as the potential applicability and benefits to software development practice, SAI 1.0 was created to focus on maintainability measurement. Table 1 lists all the measures integrated into SAI 1.0, which shows that we intend to measure both source code and software models. In reality, however, high-level models are usually not available. Even if they are available, their accuracy is questionable. Although we have not been able to quantify all dimensions of all quality attributes today, SAI was designed to be extensible to accommodate additional dimensions and metrics in the future.

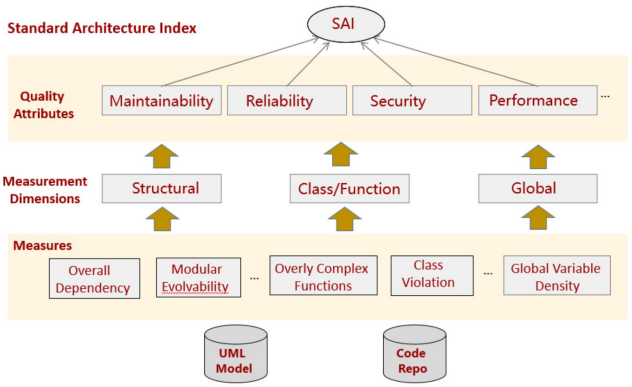

Fig. 1: SAI Overview

Table 1: Metrics in SAI 1.0

	Structure Measures	Class/Func. Measures	Global Measures
Model	Overall dependency level	Average class inheritance depth	Global Variable Density
	Module independent evolvability	Average class inheritance width	
	Logical architecture coupling		
	Logical architecture cohesion		
	Activity diagram complexity		
	Sequence diagram complexity		
	Module cyclic dependency rate		
Code	Overall dependency level	Average class inheritance depth	Global Variable Density
	Module independent evolvability	Average class inheritance width	Code duplication rate
	Module disorder rate	Class disorder rate	Design pattern defects
	Module cyclic dependency rate	Class violation rate	Redefined symbol rate
	Unstable dependency module	Overly complex class	Unused API rate
	Header file cyclical dependency	Overly complex function	Extra header file rate
	Modularity violation rate	Overly deep function	
	Overly complex module	Overly long function	

The metrics listed in Table 1 are either derived from existing metrics, or proposed by our collaborators or senior architects. Following are a few examples:

- *Class disorder rate*: measures if a class takes too many responsibilities, or accesses other class's data, derived from code smells such as Feature Envy [10], and anti-patterns, such as divergent changes [9].
- *Class violation rate*: measures how often classes are changed for similar reasons, such as code clone [10].
- *Average class inheritance depth*: derived from the well-known C&K metrics[6] for object-oriented code.

– *Modularity violation rate*: measures how often files belonging to structurally independent modules change together frequently, as recorded in the revision history, following the work of Wong et al. [21].

Most of these measures can be extracted automatically from a project's source code and revision history, but for some of them, we are still exploring practical methods to quantify them, such as *Module Independent Evolvability* that aims to measure the extent to which modules can evolve independently from each other [16]. We keep these measures in our framework so that they can be quantified in the future.

To promote the comprehensive architecture quality index, we must address the following two challenges: (1) how to ensure that the measurement is consistent with the developers' intuition and faithfully reflects architecture quality? (2) How to set thresholds of each measure for projects with different programming languages, ages, scales, markets etc., so that their intrinsic characteristics can be revealed? For example, what should be the threshold to determine if a function is "*overly complex*"? Shall we provide different thresholds for Java and C?

To address the first challenge, we proposed the concept of *Architectural Bad Smells*, which extends the original definition of *code smells* [10], which has been widely accepted in Huawei. Defining the architectural counterpart of *smells* makes SAI more convincing and acceptable to developers. We thus associated each architecture measure with a type of architecture smell (or *defect*), so that each measure is backed up by a group of specific architecture defects. For example, the *Module Disorder Rate* is associated with the *module disorder* architectural smell, that is, modules that take multiple responsibilities and cause divergent changes. The more modules are detected as being disordered, the higher the measure. This mapping has effectively reduced suspicions within development teams towards SAI. After trade-off analysis and careful comparison, the first version of SAI was created as *a sum of weighted architecture smells*:

$SAI = \Sigma(weight_i * \#ArchitectureSmell_i)/KLOC$
where $i = 0..n$, and n is the total number of architectural smells.

To address the second challenge of setting reasonable and justifiable thresholds and weights for each architectural smell, we took the following steps:

First, referring to existing architecture measurement practices, we assigned the initial thresholds and weights so that the development teams that were already practicing their own measurement could continue with little disruption. For example, for *Overly Complex function*, the threshold is set as "$CyclomaticComplexity > 15$"; *Duplicated Code* is defined as two units of code with more than 10 lines of similar code, etc. These thresholds were already being used by some product lines for a long time. Furthermore, we assigned slightly different thresholds and weights for C, C++ and Java.

Second, we assigned thresholds and weights for each measure according to the severity of the associated architectural issues. For example, the thresholds and weights of *Underused API* were determined by our architects and measurement experts based on their experiences. Third, we collected opinions from our development teams, and assigned more weight to architectural smells with significant impact on development efficiency, such as *Divergent Changes*. Finally, we collected subjective judgement and rankings of the impact of each architectural smell on development productivity and efficiency. We ranked the results accordingly and adjusted the weight of each architecture

measure and smell, so that it was consistent with the intuition of the architects and development teams.

Different teams may have different opinions about how the thresholds and weights should be assigned. To avoid potential arguments and disagreement, we decided not to publicize the weight of each architectural smells. For the three mainstream programming languages used in Huawei—C, C++ and Java—we selected a set of *successful* open source projects and obtained their average SAI scores. Here we consider an open source project to be *successful* if it has evolved for a very long period of time, supports distributed development by a large number of contributors, and has a large number of uses. We used their scores as the benchmark for our software products using the same languages, which we called SAI-benchmark.

These benchmarks, on one hand, enabled the development teams to understand their architecture quality in a straightforward way, and to assess the need to improve their architectures. On the other hand, it conveyed a clear message: if the SAI of a software product is higher than that of successful open source software systems, it means that product may have room for improvement.

3.2 Supporting Tools

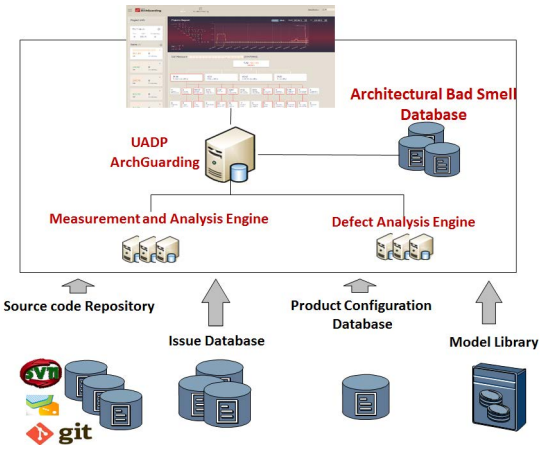

Fig. 2: UADP Overview

To improve the application of SAI, we provided a suite of tools and recommended to development teams that they improve their SAI scores, such as guarding their architecture in real time. We created a tool called *UADP ArchGuarding* ("UADP" for short), that comprehensively supported the measurement of all the components of SAI, and could be rapidly deployed in the development environment of our products. Figure 2 depicts the structure of UADP, which collects information from the code repository, issue database, product configuration database, and model library, measures and detects architectural smells, saves the defects into an architectural smell database, and presents the results in a graphical user interface as shown in Figure 3. Using this interface, a product team can conveniently monitor not only their SAI scores, but also the architectural smells associated with the scores. If the score is not satisfactory, they can figure out which architecture smells are causing the problems, and how to improve the score by fixing the underlying architecture problems.

3.3 Pilot Studies and Results

To test the effectiveness of SAI 1.0, we first collected data from six products, covering the major product domains within Huawei. After that, we reported the architecture measuring and guarding solutions (SAI and UADP), as well as the pilot study plan to management and obtained their approval. Six months later, we organized a teleconference, when all product teams participated in the pilot study presented their progress. The midterm reports were widely circulated within Huawei. At that time we also started the 2nd round of the pilot study, extending the scope to more than 30 products.

After one year of pilot study, by the end of 2014, the data shows that the architecture quality of the pilot products, as measured by SAI 1.0, improved 23.51% on average. As an example, guided by the architectural assessment score, Product A fixed 500+ architecture smells, reduced coupling among subsystems by 30%, increased modularity within subsystems, improved SAI score by 29.71%, and improved development efficiency, measured by the number of person-month per 1000 LOC, by 30%. One subsystem was refactored and reduced LOC from 471k to 199k, by removing a large amount of duplicated code. In this case, the *Divergent Change* architecture smell was reduced substantially. As another example, the SAI of product B improved 64.2%, coupling among modules was reduced by 20%, 2K LOC were removed, and maintenance effort decreased by 20%. Correspondingly, the ability to conduct parallel development improved significantly, the efficiency of feature delivery improved 18.8%, and efficiency of validation and verification also improved. A manager of the pilot product commented: "*Architecture guardian is an innovative and excellent practice within Huawei, and has produced profound and long-lasting impacts with excellent results*".

At the beginning of 2015, we summarized our application of SAI to Huawei R&D management, and decided to promote SAI to all products within Huawei. As the number of projects increased, issues concerning its design were reported. The following are some representative examples: *"Should* Overly Complex Class *be considered an architecture smell?"*, "Unstable Dependency Module *lacks theoretical support and can be ambiguous"*, etc. We improved the model and UADP tool accordingly.

Fig. 3: UADP GUI

4 The Second Version of SAI

Based on the feedback we received, we improved the model and released SAI 2.0 in the middle of 2015. In SAI 2.0, we improved the computational model and incorporated new technologies to facilitate architecture quality improvement.

4.1 Improved Computational Model

The most prominent changes of SAI 2.0 include the explicit mapping of architectural smells to quality attributes, and the categorization of architectural smells based on their scope: global, system, component, module, file, and function. The broader the scope of a smell, the more weight assigned to the corresponding measures. The overall SAI score becomes the sum of SAI scores for all quality attributions being considered. Each quality attribute, in turn, is mapped to multiple *architecture factors*, such as repeatability, system coupling, module balance, etc. These factors were either from ISO 25010, or proposed by Huawei senior architects. Each architecture factor is further mapped to a number of architecture smells that may negatively affect the factor. Each architecture smell, in turn, is weighted to reflect its impact scope, e.g., a clique involving 3 modules has a smaller impact scope factor than a clique covering 10 modules.

Our principles for setting the weight of each measure are as follows: 1) the broader the impact scope of an architectural smell, the higher the weight; 2) the higher level of abstraction of an architectural smell, the higher the weight; 3) the more contribution to architecture complexity of a smell, the higher the weight, and 4) the more impact on development and maintenance efficiency, the higher the weight. We conducted interviews with key architects and refined the weights based on the above principles.

The adjustment of SAI 2.0 was non-trivial. To minimize the impact to software products caused by architecture measurement standard changes, we experimented with the weight and threshold assignment so that the scores and trends obtained from SAI 2.0 were largely consistent with those of SAI 1.0. The benefit of using SAI 2.0 was to provide development teams with more fine-grained information about which aspects of the architecture needed improvement and which architecture attributes could be affected.

4.2 Integrating New Techniques

During the development, evolution, and application of SAI, we also established collaborations with universities and research institutes, to learn of state-of-the-art concepts and technologies. We explored a suite of new metrics and measurement techniques, including *independence level* [19], *propagation cost*(PC) [14], *decoupling Level* (DL) [16], *module balance* [3], architectural debt [25,12], etc., and a suite of new technologies that can be used by the development teams to enrich the automatic architecture measuring and guarding solutions. Next we introduce 3 technologies that have been adopted by Huawei development teams.

1. Design Structure Matrix and DSM-based Metrics. Collaborating with Drexel University and the University of Hawaii, in 2013, we introduced the Design Structure Matrix (DSM) [1] representation to Huawei development teams to model the dependency structures of their products. A DSM is square matrix in which the columns and

rows are labeled with the same set of elements in the same order, and a marked cell indicates that the elements on the row depends on the elements on the column. We found that DSM modeling is useful in two ways. First, we can model design elements at any level of granularity: files, classes, packages, or the self-defined modules. Figure 4 shows relations among modules in a Huawei product. This DSM shows that the first 4 modules form a strongly connected graph, and none of them depend on Module E. It also reveals the number of dependencies between each pair of the modules.

Second and more importantly, a DSM can model history and structural relations simultaneously. For example, in Figure 5, each cell is marked with the number of structural dependences between each module pair, and the number of times files within these modules are changed together. For example, the cell in the first row and second column is labeled "$68, 992$", meaning that there are 68 structural dependencies from Module A to Module B, and the files within these modules were changed together 992 times. This DSM shows that not only do these modules form cyclical dependencies, but that these cycles are expensive to maintain because files within them have been changed together hundreds of times, presenting compelling evidence that this product needs to be refactored. The DSM-based analysis was enthusiastically adopted by development teams.

	A	B	C	D	E
Module A	(1)	18	52	34	
Module B	144	(2)	68	132	
Module C	576	32	(3)	417	
Module D	283	82	27	(4)	
Module E	36	19	42	3	(5)

Fig. 4: Design Structure Matrix

Along with DSM modeling, we also adopted a suite of associated metrics, including *independence level* (IL) [19], *decoupling level*(DL) [16], and *propagation cost*(PC) [14]. These metrics complement with one another. PC was designed to measure how tightly source files within a system are *coupled* with each other. The idea is to calculate the transitive closure of a DSM until no more dependencies can be added, and then calculate percentage of non-empty cells to the total number of cells. The higher the PC, the tighter the system is coupled.

	A	B	C	D	E
Module A	(1)	68,992	132,585	0,141	144,222
Module B	32,992	(2)	417,1915	0,246	576,378
Module C	82,858	270,1915	(3)	0,130	283,240
Module D	19,141	42,246	30,130	(4)	36,53
Module E	18,202	52,378	34,240	0,53	(5)

Fig. 5: DSM with History

Both DL and IL are based on the *Design Rule Hierarchy* (DRH) [23] algorithm that clusters a DSM into a hierarchical structure in which the most influential files are at the top of the hierarchy, and files in the lower layers depend on files in higher layers. Most importantly, the files within each layer are clustered into mutually independent modules. Accordingly, the modules at the bottom layer of a DRH are truly independent since they can be revised or replaced without influencing other parts of the system. The metric Independence Level (IL) [19] measures the portion of files that are clustered into the lowest layer of a DRH, that is, the portion of files that can be implemented and changed independently. The larger the IL, the more modules in the system can be changed independently. A new metric, Decoupling Level (DL) [16], evolved from IL to

take into account the size of each modules, the number of independent modules in all layers, and the dependency relations among modules. The more independent modules, the smaller each module, and the less coupled they are, the higher the DL.

One of the product teams leveraged DRH clustering in an innovative way to guide their refactoring activities, and obtained impressive results. The DSM in Figure 5 depicts the initial structure of their product, which made it clear that the original structure of the system was poorly modularized. The product teams thus used the modules clustered by the DRH algorithm as the *de facto* modular structure and refactored the system accordingly. By doing so, they increased the DL of the system by 250%. As a result, the SAI of the subsystem improved by 40%, much better than the open source benchmark. The DSM after refactoring is depicted in Figure 6. This figure shows that, even though the cycles were not completely removed, the number of dependencies among modules was significantly reduced. Given the excellent results of DSM analysis and the associated metrics, we have integrated them into architectural measurement and guarding solution, SAI standard, and recommended key activities.

	A	B	C	D	E	F	G	H	I
Module A	(1)								
Module B	13	(2)	2	2	1	1	15		
Module C	11	6	(3)		4		9		
Module D	5		1	(4)			3		
Module E	9	1	4		(5)	4	19		
Module F	6	2	2	2	2	(6)	39		
Module G	216	129	57	16	18	31	(7)		
Module H	3	4	32	417				(8)	
Module I	3	3	2	2	2	1	26		(9)

Fig. 6: DSM After refactoring

2. Architectural Debt Quantification. We explored architectural debt analysis techniques in 2015 and 2016, completed an architectural debt quantification prototype, and conducted a pilot study using Project M, which had experienced difficulty maintaining their product for a long time. Table 2. The table shows that, of the 166.71 person-months spent in Module A, 120.45 (72%) person-month were extra maintenance costs caused by the severe architecture debt. The percentage of debt in Module B is 75%. These numbers were acknowledged by the development team, and provided a foundation for their decision to conduct refactoring. We plan to further integrate this technology into our architecture automatic measuring and guarding solutions, and make it a key activity to motivate architectural refactoring. This will allow our development teams to quantify their debts and visualize architecture problems using DSMs, so that they can justify refactoring and pinpoint the focus of refactoring.

Table 2: Product M Architectural Debt Analysis

	Person Month	Debt Ratio (%)
Module A	166.71	120.45 (72%)
Module B	375.55	281.65 (75%)

4.3 Pilot Study and Results

In late 2015, we measured 20 products using SAI 1.0. After that we deployed SAI 2.0 and started a trial assessment in 2016. After 9 months of trial and adjustment, Huawei products switched to SAI 2.0. Since our ultimate purpose is to use SAI as a

measure to improve software productivity and efficiency, we collected data from multiple versions of 29 products that have applied SAI 2.0. For each project, we collected both the SAI measures for each release, and productivity measures during the process. In Huawei, we measure productivity using person-months per 1000 LOC. These data were collected continuously over the past two years. As shown in Figure 7, for 24 out of 29 projects, SAI measures are shown to have positive correlation with productivity. For about 1/3 of the projects, their productively is strongly correlated with their SAI scores.

For some of the projects, the number of data points is not enough to obtain statistically meaningful results. For projects whose SAI scores are shown to be negatively correlated with their productivity, we are investigating the reasons and will continue collecting the data and monitoring their variation over time. For the second project whose productivity appeared to be opposite to its SAI score, we found out that it is because the system was undergoing major repository merging and the productivity data was not correctly collected. We will filter out such data in the future.

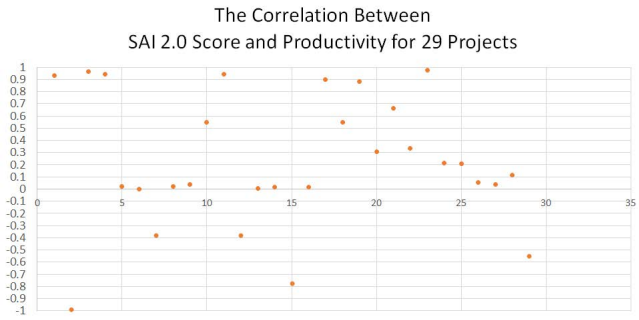

Fig. 7: SAI and Productivity

5 Lessons Learned and Results

Results. SAI has now been widely deployed and accepted within Huawei: it has now been used in more than 100 products. Thus we can now answer the research questions proposed in Section 2:

1. By creating a software measurement system with a score that can be traced to concrete architectural flaws, supported by key activities and tools such as the quantification of architecture debt, we helped development teams realize the need to conduct large scale architecture refactoring. The value of architecture measurement has been validated by widespread adoption in our development teams. Thus we now believe that architecture measurement should be integrated into the core daily activities of the software development process.
2. It is possible to measure software architecture quantitatively. We learned that we could obtain measures that reveal the level of severity of architecture smells, and that are consistent with the intuition of architecture experts, development teams, and management. In this way we can provide support for them to do the necessary refactoring.

3. By working with developers and being flexible, not expecting 100% compliance immediately, we found that our teams accepted these measures as meaningful.
4. Finally, our developers determined that the measures that we produced were indeed actionable. They used these measures to plan refactoring activities and those refactorings resulted in substantial improvements to our systems in terms of bug rates, change rates, and effort.

Overall we consider the design and deployment of SAI within Huawei to have been a great success. The process required patience and flexibility, it required as much listening as talking, but in the end produced meaningful results.

Lessons learned. Through the creation and evolution of SAI we have learned a number of lessons that, we believe, strongly affected our success.

First, at the beginning of the process, we were met with suspicion from development teams. On the surface, their doubts were directed towards the standard. But in reality, their doubts stemmed from the implicit pressure of ongoing performance reviews. At this point, we had to compromise. For teams to accept and adopt the standard smoothly, it was necessary to not only convince them in theory, but also to accept some compromises in practice, as we were proving the value of our approach.

Second, we needed to not only provide supporting tools, but also keep improving its usability, which is a key for development teams to accept the standard and apply it widely.

Third, the evolution of the standard is necessary, but it must be incremental. It was also critical to invite architects from a wide range of teams to participate in the construction of instruments from the beginning.

Fourth, we learned that we need to participate in conferences and learn from the research community. We must continuously assess and adopt new theories and methods from the field of architecture measurement. The process, from learning a new theory or method, to its complete integration into the practice of architecture measurement, is long and challenging.

Fifth, once the development teams were inspired to participate in architecture measurement, and once they gained sufficient confidence in the instrument, they brought unexpected innovations to the architecture measurement standard and solution. The product team who used DRH clustering to guide their refactoring process is an example. Another team proposed and employed the concept of *architecture hotspot* in a simple and practical way.

6 Related Work

In this section, we review related work on (architectural) complexity measures and change measures.

Software metrics. Various metrics have been used to measure software code complexity in past decades. Cyclomatic complexity [15] calculates the number of linearly independent paths through a program's source code to measure complexity. Various object-oriented metrics, such as the well-known CK metrics [6], LK Metrics [13] and MOOD Metrics [8], were proposed to measure object-oriented programs. Based on

software metrics, software measurement has been widely studied. For example, Coleman [7] applied two models for measuring software maintainability. Sjøberg et. al. [20] conducted a case study to examine the relations among code metrics and the correlations between these metrics and software maintainability. Instead of focusing on a specific class of metrics, our SAI integrates three types of metrics—structure, class/function and global metrics—to measure software maintainability. This helps development teams to monitor their software in different perspectives.

Change Measures. Change measures can be calculated from a project's version control system and its issue tracking system. Various change measures extracted from history, such as change frequency and code churn, have been used for software measurement. For example, Schulte [17] measure software based on the activeness of each file in the project's revision history. Wong et. al. [22] calculated co-changes as the number of times two files have been changed (committed) together in the project's revision history. They also showed that co-change measures contain important information which reveals improper relations—modularity violations—among files. Following Wong et. al's work [22], Schwanke et. at. [18] used structure dependency and change measures to predict defects and showed that modularity violations can arise as the result of implicit assumptions between structurally unrelated files.

Architecture Metrics. Several architecture-level metrics have been proposed to measure architecture quality. Mo et. al. [16] introduced Decoupling Level to measure how well a software system is decoupled into small and independent modules. MacCormack et al. [14] defined the Propagation Cost metric to measure how tightly coupled a system is. Bouwers et al. presented two architecture-level metrics, Component Balance [3] and Dependency Profiles [4]. They also investigated the usefulness of these metrics [5], and demonstrated that measurement results matched practitioner intuitions. Our SAI has integrated multiple *architectural* measures. To properly assess and understand a software architecture we used a specific architecture measure for each architecture smell.

Architecture Debt Analysis. Various approaches have been proposed to detect and analyze architecture debt. In an industrial case study [11], the authors presented a cost-benefit model, which was used to estimate the effort that could be saved by a refactoring to decouple the architecture. In the case study of Kazman et. al. [12], they used an economic model to assess the effort that could be saved after refactoring the identified architecture debts. Our SAI not only integrates architecture debt analyses, but also integrates other state-of-the art techniques to assist a development team in detecting architecture problems and assessing the technical debt caused by these problems.

7 Conclusions and Future Work

In this paper, we have presented our 4-year experience of implementing a software architecture measurement system within Huawei. Our study has revealed the significant and positive impact of architecture measurement, in terms of ensuring software quality and productivity, and supporting continuous evolution and improvement of architecture. Our experience showed that formulas alone are not enough; it is important to back up formulas with concrete architecture smells. The development teams use these identified smells to determine how to improve their scores, and hence to improve their

architecture. We also confirmed that it is important to adopt state-of-the-art analysis approaches—such as DSM analysis and architecture debt quantification—as key activities to better understand the consequences of degrading the architecture and to quantitatively support refactoring decisions. The SAI scores have shown positive correlations with productivity measures for most of our pilot studies. And we have shown that, with some coaching, most product teams eventually adopt and even embrace software architecture measurements. The automatic architecture measurement and guarding solutions, as well as the SAI formula itself, are still being refined to address various issues and challenges, including:

- How to further increase the accuracy of our architecture measures, so that they can be fully consistent with the intuitions of architects and product teams, as well as the market performance of their products?
- How to measure service-oriented architectures and other similar architectures that consist of collections of interacting components, each of which has its own architecture and its own code base?
- Currently our architecture measurement tool is an independent web-based application. We are working to integrate it within an IDE, so that software developers can be immediately alerted to potential software decay. This way, our tools can be fully integrated into development activities, and architecture measurements will become part of the daily routine of software development.
- We also intend to explore AI-based architecture refactoring and evolution, employing machine-learning to make refactoring recommendations.

We believe these explorations will drive the progress, evolution and development of automatic architecture measurement and guarding solutions, as well as SAI itself.

References

1. C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.
2. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 3rd edition, 2012.
3. E. Bouwers, J. P. Correia, A. v. Deursen, and J. Visser. Quantifying the analyzability of software architectures. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pages 83–92, June 2011.
4. E. Bouwers, A. van Deursen, and J. Visser. Dependency profiles for software architecture evaluations, 09 2011.
5. E. Bouwers, A. van Deursen, and J. Visser. Evaluating usefulness of software metrics: An industrial experience report. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 921–930, May 2013.
6. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
7. D. Coleman, P. Oman, D. Ash, and B. Lowther. Using metrics to evaluate software system maintainability. *Computer*, 27:44–49, 08 1994.
8. F. B. e Abreu. The mood metrics set. In *Proc. ECOOP'95 Workshop on Metrics*, 1995.
9. M. Fowler. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Addison-Wesley, Apr. 1989.

10. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, July 1999.
11. C. Jeromy, K. Rick, and O. Ipek. A cost-benefit framework for making architectural decisions in a business context. In *Proc. 32ndInternational Conference on Software Engineering*, pages 149–157, 2010.
12. 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.
13. M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994.
14. A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, July 2006.
15. T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.
16. 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*, pages 499–510. ACM, 2016.
17. L. Schulte, H. Sajnani, and J. Czerwonka. Active files as a measure of software maintainability. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 34–43, New York, NY, USA, 2014. ACM.
18. R. Schwanke, L. Xiao, and Y. Cai. Measuring architecture quality by structure plus history analysis. In *Proc. 35rd International Conference on Software Engineering*, pages 891–900, May 2013.
19. K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna. From retrospect to prospect: Assessing modularity and stability from software architecture. In *Proc. Joint 8th Working IEEE/IFIP International Conference on Software Architecture and 3rd European Conference on Software Architecture*, pages 269–272, Sept. 2009.
20. D. I. Sjøberg, B. Anda, and A. Mockus. Questioning software maintenance metrics: A comparative case study. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 107–110, New York, NY, USA, 2012. ACM.
21. S. Wong and Y. Cai. Improving the efficiency of dependency analysis in logical models. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 173–184, Nov. 2009.
22. S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proc. 33rd International Conference on Software Engineering*, pages 411–420, May 2011.
23. S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 197–208, Nov. 2009.
24. 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.
25. 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.