Examination of tools for managing different dimensions of Technical Debt
Dwarak Govind Parthiban¹, University of Ottawa

Abstract

With lots of freemium and premium, open and closed source software tools that are available in the market for dealing with different activities of Technical Debt management across different dimensions, identifying the right set of tools for a specific activity and dimension can be time consuming. The new age cloud-first tools can be easier to get onboard, whereas the traditional tools involve a considerable amount of time before letting the users know what it has to offer. Also, since many tools only deal with few dimensions of Technical Debt like Code and Test debts, identifying and choosing the right tool for other dimensions like Design, Architecture, Documentation, and Environment debts can be tiring. We have tried to reduce that tiring process by presenting our findings that could help others who are getting into the field of “Technical Debt in Software Development” and subsequently further into “Technical Debt Management”.

I. INTRODUCTION

TECHNICAL DEBT (TD) is a term that was coined by Ward Cunningham [1] in the year of 1992. It is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer. There are several dimensions of technical debt like code debt, test debt, documentation debt, environment Debt, design Debt, and architecture Debt. As with financial debt, technical debt must be paid back, and is comprised of two parts: principal and interest. In the software development metaphor, the interest is paid in the form of additional work required to maintain the software system given its sub-optimal code. Time spent improving the code, which isn’t directly adding customer value and which wouldn’t be necessary if the code were optimally designed currently, represents paying down the principal on the debt.

MANAGING the technical debt mostly consists in “repaying the principal” to achieve business value. Technical debt management is the set of activities that: prevent potential technical debt from being accrued, deal with the accumulated technical debt to make it visible and controllable, keep a balance between the cost and value of the software. Technical debt management includes the following eight activities: representation, prevention, communication, prioritization, monitoring, measurement, identification, and repayment of technical debt.

IN THIS PAPER, we first introduce a wide-array of software tools that are currently available in the market according to our knowledge. We then choose some among them which we believe to be complementary to each other, examine them further using some of the open source projects available in GitHub, and report our findings which is a subset of all the functionalities that those tools have. We present our findings in a such a way that the readers can associate them to the appropriate dimension of the technical debt and the corresponding activity within technical debt management. Finally, we propose a cost model for TD principal calculation and suggest a new tool that has recently hit the market by providing details about it.

II. TOOLS AND PROJECTS

Among the tools that are mentioned in table I, we chose Sonarcloud, Checkstyle, Jacoco, DesiginateJava, Codescene and Lattix. The reason for our selection is we wanted the tools to be mostly complementary to each other (in their default settings with no or very few customizations) so that we can cover many dimensions within technical debt than just focussing on code and test debts. Also, we wanted them to be appropriate for different activities within TD management.

We also chose four open source Java projects which could be built using Maven. The chosen projects were used for the examination of the abovementioned tools. Some details about those projects are mentioned in table II. The measures: lines of code and number of classes were retrieved from Sonarcloud. The github stars takes into account of the total number of stars till April 15, 2019 12:30 PM EDT.

¹dpart016@uottawa.ca
### TABLE I: Some of the tools available in the market

| Name            | Cloud | On-premise | Distribution | Dimensions            |
|-----------------|-------|------------|--------------|-----------------------|
| Sonarqube       | ✔     | ✔          | Open         | Code, Test            |
| Sonarcloud      | ✔     |            | Closed       | Code, Test            |
| Teamscale       | ✔     | ✔          | Closed       | Code, Test, Documentation, Architecture |
| Codescene       | ✔     |            | Closed       | Code, Architecture    |
| Codacy          | ✔     |            | Closed       | Code                  |
| DesigniteJava   | ✔     |            | Open         | Code, Design          |
| Scrutinizer     | ✔     |            | Closed       | Code                  |
| Lattix          | ✔     |            | Closed       | Design, Architecture  |
| Jacoco          | ✔ (Sonarcloud plugin) | ✔ (Standalone, Maven Plugin, Eclipse Plugin) | Open | Test |
| Checkstyle      | ✔     |            | Open         | Code                  |

### TABLE II: Java projects used for examination

| Name       | Description                                      | Lines of Code | Number of Classes | GitHub Stars |
|------------|--------------------------------------------------|---------------|-------------------|--------------|
| Java WebSocket | Barebones websocket client and server implementation | 5K            | 65                | 5197         |
| JDBM3      | Embedded Key Value Java Database                 | 9.1K          | 63                | 312          |
| Jedis      | A blazingly small and sane redis java client     | 20.8K         | 129               | 7933         |
| MyBatis    | SQL mapper framework for Java                    | 22.6K         | 397               | 10,360       |

### III. FINDINGS

#### A. Quality assessment

1) Among the chosen tools, Sonarcloud and Lattix allow assessing the following software quality attributes,
   - Sonarcloud - Reliability, Maintainability, Security [See figures 1, 2, and 4 respectively]
   - Lattix - Stability [See figures 3, 5, 6, and 7]

2) Below are the screenshots of quality attributes from the abovementioned tools,

![Fig. 1: Reliability](image-url)
• Reliability attribute helps in seeing bugs’ operational risks to the projects. The closer a bubble’s color is to red, the more severe the worst bugs in the project. Bubble size indicates bug volume in the project, and each bubble’s vertical position reflects the estimated time to address the bugs in the project. Small green bubbles on the bottom edge are best.

![Maintainability Figure](image)

**Fig. 2: Maintainability**

• Maintainability attribute helps in seeing code smells’ long-term risks to the projects. The closer a bubble’s color is to red, the higher the ratio of technical debt to project size. Bubble size indicates code smell volume in the project, and each bubble’s vertical position reflects the estimated time to address the code smells in the project. Small green bubbles on the bottom edge are best.

• Security attribute helps in seeing vulnerabilities’ operational risks to your projects. The closer a bubble’s color is to red, the more severe the worst vulnerabilities in the project. Bubble size indicates vulnerability volume in the project, and each bubble’s vertical position reflects the estimated time to address the vulnerabilities in the project. Small green bubbles on the bottom edge are best.

• Stability attribute of a system reports how sensitive the architecture is to the changes in atoms (say classes) within the subsystem. A higher stability value corresponds to less dependency on atoms within the selected system.

| Subsystem                      | System Stability |
|--------------------------------|------------------|
| root                          | 92.495%          |
| org.apache.jdbm               | 86.484%          |
| src                           | 98.506%          |
| src.main.java.org.apache.jdbm | 96.684%          |
| src.test.java.org.apache.jdbm | 99.684%          |

**Fig. 3: System stability (overall and top-level components) for JDBM3**
Fig. 4: Security

Fig. 5: System stability (overall and top-level components) for Java WebSocket
### System Stability (overall and top-level components) for Jedis

| Subsystem                        | System Stability |
|----------------------------------|------------------|
| $root                            | 78.151%          |
| redis.clients.jedis              | 78.151%          |
| redis.clients.jedis.*            | 68.615%          |
| redis.clients.jedis.commands     | 69.890%          |
| redis.clients.jedis.exceptions   | 55.841%          |
| redis.clients.jedis.params       | 53.907%          |
| redis.clients.jedis.tests        | 99.232%          |
| redis.clients.jedis.util         | 69.406%          |

**Fig. 6:** System stability (overall and top-level components) for Jedis

### System Stability (overall and top-level components) for MyBatis

| Subsystem                        | System Stability |
|----------------------------------|------------------|
| $root                            | 89.998%          |
| org.apache.ibatis                 | 89.998%          |
| org.apache.ibatis.*              | 84.439%          |
| org.apache.ibatis.annotations     | 64.377%          |
| org.apache.ibatis.autoconstructor | 99.867%          |
| org.apache.ibatis.binding        | 88.655%          |
| org.apache.ibatis.builder        | 80.880%          |
| org.apache.ibatis.cache          | 76.414%          |
| org.apache.ibatis.cursor         | 75.715%          |
| org.apache.ibatis.databases.blog | 100.000%         |
| org.apache.ibatis.datasource     | 69.938%          |
| org.apache.ibatis.domain         | 99.477%          |
| org.apache.ibatis.exceptions     | 66.897%          |
| org.apache.ibatis.executor       | 74.394%          |
| org.apache.ibatis.io             | 79.568%          |
| org.apache.ibatis.jdbc           | 97.737%          |
| org.apache.ibatis.lang           | 100.000%         |
| org.apache.ibatis.logging        | 70.582%          |
| org.apache.ibatis.mapping        | 70.187%          |
| org.apache.ibatis.parsing        | 79.644%          |
| org.apache.ibatis.plugin         | 86.074%          |
| org.apache.ibatis.reflection     | 74.361%          |
| org.apache.ibatis.scripting      | 66.467%          |
| org.apache.ibatis.session        | 74.322%          |
| org.apache.ibatis.submitted      | 99.873%          |
| org.apache.ibatis.transaction    | 68.081%          |
| org.apache.ibatis.type           | 81.296%          |

**Fig. 7:** System stability (overall and top-level components) for MyBatis
3) Based on multiple quality assessments,

| Project    | Reliability rank | Maintainability rank | Security rank | Stability rank | Sum | Overall quality rank |
|------------|------------------|----------------------|---------------|---------------|-----|----------------------|
| Java WebSocket | 1 | 1 | 1 | 4 | 7 | 1 |
| JDBM3       | 3 | 2 | 2 | 1 | 8 | 2 |
| Jedis       | 2 | 3 | 3 | 3 | 11 | 3 |
| MyBatis     | 3 | 4 | 4 | 2 | 13 | 4 |

TABLE III: Overall quality rank

- In the above table, the ranks for quality attributes are based on the rating given by the tools. For example, if project X and Y get security ratings of A and B, security rank of X is higher (1 is higher than 2) than that of Y. If two projects have same rating, then higher rank is given to the one who has less value in the Y axis, which is typically the time taken for remediation efforts.
- The overall quality rank [Column 6] is calculated by ranking the sums [Column 5] of individual ranks [Columns 1, 2, 3, 4].

B. Technical debt identification

| Dimension          | Some Items                                                      |
|--------------------|----------------------------------------------------------------|
| Code debt          | Coding guideline violations, Code smells, Inconsistent style    |
| Design debt        | Design rule violations, Design smells, Violation of design constraints |
| Test debt          | Lack of tests, Inadequate test coverage, Improper test design |
| Architecture debt  | Architecture rule violations, Modularity violations, Architecture smells |

TABLE IV: Categorization of some of the technical debt items associated with their technical debt dimension according to [2]

With the use of these many tools, it is obvious that combinedly the tools can identify a lot of TD types with a lot more instances for each of those types. To describe a high-level diversity, we have just presented a subset of them in table V. It could be seen that with current tools, the diversity of TD items in design and architecture dimensions seem to be lower than those present in code and test dimensions.

| TD type             | TD item            | TD dimension | Identified By         |
|---------------------|--------------------|--------------|-----------------------|
| Long method         | Code smells        | Code debt    | DesigniteJava, Sonarcloud |
| Long parameter list | Code smells        | Code debt    | DesigniteJava, Sonarcloud |
| Complex method      | Code smells        | Code debt    | DesigniteJava, Sonarcloud |
| Whitespace around   | Coding guideline violation | Code debt | Checkstyle |
| Missing javadoc comment | Coding guideline violation | Code debt | Checkstyle |
| Add at least one assertion to this test case | Improper test design | Test debt | Sonarcloud |
| Coverage below 90%  | Lack of tests      | Test debt    | Jacoco |
| Add some tests to this class | Lack of tests | Test debt | Sonarcloud, Jacoco |
| Deficient encapsulation | Design smells | Design debt | DesigniteJava |
| Hub-like modularization | Design smells | Design debt | DesigniteJava |
| Unutilized abstraction | Design smells | Design debt | DesigniteJava |
| Intercomponent cyclicity | Architecture smells | Architecture debt | Lattix |

TABLE V: Some of the identified TD types, items, and dimensions from the chosen projects

However, tools like Lattix can identify a few more items like Architecture rule violations which fall under architecture debt. But for the tool to identify such violations, the rules should have been enabled at first. A glimpse of it is shown in the Appendix A.

C. Technical debt representation

The TD instances are mentioned in a structured tabular format in this subsection. The values in those tables were retrieved from different tools. For example, Codescene became handy to find the author responsible for a TD instance as it provides a rich social analysis.
1) Java WebSocket: See tables VI to XII

| ID     | jws_cd_1               | ID     | jws_cd_2               |
|--------|------------------------|--------|------------------------|
| TD type name | Long method           | TD type name | Whitespace around     |
| TD item name | Code smells         | TD item name | Coding guideline violation |
| Location | Method decodeHandshake in class WebSocketImpl in package org.java_websocket | Location | Line 193 in class AbstractWebSocket in package org.java_websocket |
| Responsible/Author | Davidiusdadi   | Responsible/Author | marci4               |
| Dimension | Code debt            | Dimension | Code debt              |
| Date/Time | April 15, 2019     | Date/Time | April 15, 2019         |
| Context | A private method in a Java concrete class. | Context | A private method in a Java abstract class. |
| Propagation | Impacts other public methods in the same class that uses this method. | Propagation | Impacts other public methods in the same and derived classes that makes use of this method. |
| Intentionality | Unintentional | Intentionality | Unintentional |

**TABLE VI**

| ID     | jws_td_1               | ID     | jws_td_2               |
|--------|------------------------|--------|------------------------|
| TD type name | Coverage below 90%     | TD type name | Add at least one assertion to this case |
| TD item name | Inadequate test coverage | TD item name | Improper test design |
| Location | All source files (coverage is only 64.2%) | Location | Line 151 in class Issue256Test in package org.java_websocket.issues |
| Responsible/Author | marci4, Marcel P  | Responsible/Author | marci4               |
| Dimension | Test debt            | Dimension | Test debt              |
| Date/Time | April 15, 2019     | Date/Time | April 15, 2019         |
| Context | Jacoco coverage report for junit tests. | Context | A junit test. |
| Propagation | Impacts all source and test files. | Propagation | No impact to other classes. |
| Intentionality | Intentional | Intentionality | Unintentional |

**TABLE VII**

| ID     | jws_td_3               | ID     | jws_dd_1               |
|--------|------------------------|--------|------------------------|
| TD type name | Add some tests to this class | TD type name | Unutilized abstraction |
| TD item name | Lack of tests         | TD item name | Design smells          |
| Location | Class AutobahnClientTest in package org.java_websocket.example | Location | Class SSLSocketChannel in package org.java_websocket.issues |
| Responsible/Author | Davidiusdadi   | Responsible/Author | marci4               |
| Dimension | Test debt            | Dimension | Design debt            |
| Date/Time | April 15, 2019     | Date/Time | April 15, 2019         |
| Context | A Java concrete class. | Context | A Java concrete class. |
| Propagation | No impact to other classes. | Propagation | No impact to other classes. |
| Intentionality | Intentional | Intentionality | Unintentional |

**TABLE VIII**

**TABLE IX**

| ID     | jws_ad_1               |
|--------|------------------------|
| TD type name | Intercomponent cyclicality (9.07%) |
| TD item name | Architecture smell |
| Location | Classes in packages org.java_websocket, org.java_websocket.drafts and org.java_websocket.server |
| Responsible/Author | marci4, Davidiusdadi |
| Dimension | Architecture debt |
| Date/Time | April 15, 2019 |
| Context | Java classes across different packages. |
| Propagation | Impacts the classes that branches from the existing dependency cycles. |
| Intentionality | Unintentional |

**TABLE X**

**TABLE XI**

2) JDBM3: See tables XIII to XVII.
### TABLE XIII

| ID    | jdb_cd_1       | ID    | jdb_cd_2       |
|-------|----------------|-------|----------------|
| TD type name | Complex method | TD type name | Switch without “default” clause |
| TD item name | Code smells | TD item name | Coding guideline violation |
| Location | Method equals in class SerialClassInfoTest in package org.apache.jdbm | Location | Line 633 in class Serialization in package org.apache.jdbm |
| Responsible/Author | Jan Kotek | Responsible/Author | Jan Kotek |
| Dimension | Code debt | Dimension | Code debt |
| Date/Time | April 15, 2019 | Date/Time | April 15, 2019 |
| Context | A public method in a nested, static Java class. | Context | A public method in a public Java concrete class. |
| Propagation | Impacts other methods that calls this method. | Propagation | Impacts other methods that calls this method. |
| Intentionality | Unintentional | Intentionality | Unintentional |

### TABLE XIV

| ID    | jdb_td_1       | ID    | jdb_td_2       |
|-------|----------------|-------|----------------|
| TD type name | Coverage below 90% | TD type name | Add at least one assertion to this case |
| TD item name | Inadequate test coverage | TD item name | Improper test design |
| Location | All source files (coverage is only 82.6%) | Location | Line 91 in class BTreeTest in package org.apache.jdbm |
| Responsible/Author | Jan Kotek | Responsible/Author | Jan Kotek |
| Dimension | Test debt | Dimension | Test debt |
| Date/Time | April 15, 2019 | Date/Time | April 15, 2019 |
| Context | Jacoco coverage report for junit tests. | Context | A junit test in a Java class. |
| Propagation | Impacts all source and test files. | Propagation | No impact to other classes. |
| Intentionality | Unintentional | Intentionality | Unintentional |

### TABLE XV

| ID    | jdb_dd_1       |
|-------|----------------|
| TD type name | Deficient encapsulation |
| TD item name | Design smells |
| Location | Class DBStore in package org.apache.jdbm |
| Responsible/Author | Jan Kotek |
| Dimension | Design debt |
| Date/Time | April 15, 2019 |
| Context | A Java concrete class. |
| Propagation | Impacts methods that calls or makes use of that method or attribute. |
| Intentionality | Intentional |

### TABLE XVI

| ID    | jed_cd_1       | ID    | jed_cd_2       |
|-------|----------------|-------|----------------|
| TD type name | Long parameter list | TD type name | Unicode escapes should be avoided |
| TD item name | Code smells | TD item name | Coding guideline violation |
| Location | Constructor in class JedisClusterInfoCache in package redis.clients.jedis | Location | Line 161 in class BitCommandsTest in package redis.clients.jedis.tests.commands |
| Responsible/Author | Jungtaek Lim | Responsible/Author | Marcos Nils |
| Dimension | Code debt | Dimension | Code debt |
| Date/Time | April 15, 2019 | Date/Time | April 15, 2019 |
| Context | A public method in a Java concrete class. | Context | A junit test in a Java class. |
| Propagation | Impacts methods that calls this method. | Propagation | No impact to other classes. |
| Intentionality | Unintentional | Intentionality | Intentional |

### 3) Jedis

See tables XVIII to XXII
### TABLE XX

| ID   | jed_td_1  |
|------|-----------|
| **TD type name** | Coverage below 90% |
| **TD item name** | Inadequate test coverage |
| **Location** | All source files (coverage is only 10.7%) |
| **Responsible/Author** | Jonathan Leibiusky |
| **Dimension** | Test debt |
| **Date/Time** | April 15, 2019 |
| **Context** | Jacoco coverage report for junit tests. |
| **Propagation** | Impacts all source and test files. |
| **Intentionality** | Intentional |

### TABLE XXI

| ID   | jed_td_2  |
|------|-----------|
| **TD type name** | Add at least one assertion to this case |
| **TD item name** | Improper test design |
| **Location** | Line 111 in class SSJedisClusterTest in package redis.clients.jedis.tests |
| **Responsible/Author** | M Sazzadul Hoque |
| **Dimension** | Test debt |
| **Date/Time** | April 15, 2019 |
| **Context** | A junit test in a Java class. |
| **Propagation** | No impact to other classes. |
| **Intentionality** | Unintentional |

### TABLE XXII

| ID   | jed_dd_1  |
|------|-----------|
| **TD type name** | Insufficient modularization |
| **TD item name** | Design smells |
| **Location** | Interface JedisClusterCommands in package redis.clients.jedis.commands |
| **Responsible/Author** | phufool |
| **Dimension** | Design debt |
| **Date/Time** | April 15, 2019 |
| **Context** | A Java interface. |
| **Propagation** | Impacts all the clients of this interface. |
| **Intentionality** | Intentional |

### TABLE XXIII

4) **MyBatis**: See tables XXIII to XXVIII.

| ID   | myb_cd_1  |
|------|-----------|
| **TD type name** | Magic number |
| **TD item name** | Code smells |
| **Location** | Line 77 in class CacheTest in package org.apache.ibatis.submitted.cache |
| **Responsible/Author** | Kazuki Shimizu |
| **Dimension** | Code debt |
| **Date/Time** | April 15, 2019 |
| **Context** | A junit test in a Java class. |
| **Propagation** | No impact to other classes. |
| **Intentionality** | Unintentional |

### TABLE XXIV

| ID   | myb_cd_2  |
|------|-----------|
| **TD type name** | Line is longer than 100 characters |
| **TD item name** | Coding guideline violation |
| **Location** | Line 47 in class DeleteProvider in package org.apache.ibatis.annotations |
| **Responsible/Author** | Kazuki Shimizu |
| **Dimension** | Code debt |
| **Date/Time** | April 15, 2019 |
| **Context** | Docstring for a Java interface. |
| **Propagation** | No impact to other classes. |
| **Intentionality** | Unintentional |

### TABLE XXV

| ID   | myb_td_1  |
|------|-----------|
| **TD type name** | Coverage below 90% |
| **TD item name** | Inadequate test coverage |
| **Location** | All source files (coverage is only 84.3%) |
| **Responsible/Author** | Eduardo Macarron, Nathan Maves, Iwao Ave |
| **Dimension** | Test debt |
| **Date/Time** | April 15, 2019 |
| **Context** | Jacoco coverage report for junit tests. |
| **Propagation** | Impacts all source and test files. |
| **Intentionality** | Unintentional |

### TABLE XXVI

| ID   | myb_td_2  |
|------|-----------|
| **TD type name** | Add at least one assertion to this case |
| **TD item name** | Improper test design |
| **Location** | Line 390 in class BindingTest in package org.apache.ibatis.binding |
| **Responsible/Author** | Eduardo Macarron |
| **Dimension** | Test debt |
| **Date/Time** | April 15, 2019 |
| **Context** | A junit test in a Java class. |
| **Propagation** | No impact to other classes. |
| **Intentionality** | Unintentional |
### D. Technical debt estimation

Out of all the used tools, only Sonarcloud gives an estimate of TD principal by default. However, the estimate doesn’t take into account all of the TD items across all the TD dimensions. The estimate is calculated only based on the following TD items: code smells, lack of tests, and improper test design.

| Project       | TD principal estimate |
|---------------|-----------------------|
| Java WebSocket| 3                     |
| JDBM3         | 9                     |
| Jedis         | 10                    |
| MyBatis       | 20                    |

| Project |
|---------|

TABLE XXIX: TD principal estimates from Sonarcloud in terms of person-days

### E. Technical debt monitoring

Dashboards [See figures 9 and 8] and warnings/alerts [See figure 8] can be enabled by integrating some of the tools with the IDE or with the continuous integration (CI) servers. Let’s say Sonarqube is integrated with Jenkins, then developers and product owners can be kept informed about the TD instances that has happened or might soon happen because of the recent commits. Tools like Codescene can directly look for the commits that is been made to a repository and can warn the stakeholders by re-running the analysis and producing the reports.

Fig. 8: Dashboard from Codescene for one of the project “Jedis”
Fig. 9: Dashboard from Sonarcloud for all of the chosen projects

F. Technical debt repayment

In this subsection, we propose techniques to repay the principal of three TD instances for every chosen project.

| ID  | Proposed techniques for repayment                                           |
|-----|---------------------------------------------------------------------------|
| jws_cd_1 | Split the method decodeHandshake into multiple methods by extracting code from the branch statements i.e., make the body of branch statements as individual methods. |
| jws_cd_2 | Add whitespace around all the symbols in line 193 as per Google style guide for Java. |
| jws_ad_1 | Many classes in org.java.websocket depend on classes from other packages. To reduce the percent of intercomponent cyclicality, either move the coupled classes into same package if possible or introduce a bridge class in current package and make it to talk to classes in other packages. |

| ID  | Proposed techniques for repayment                                           |
|-----|---------------------------------------------------------------------------|
| jdb_cd_1 | Move some of the conditional statements into a separate method or try making use of polymorphism. |
| jdb_cd_2 | Add a default case in the switch block. |
| jdb_dd_1 | In line 115, change the public modifier to private or protected. |

TABLE XXXI: TD repayment for JDBM3

| ID  | Proposed techniques for repayment                                           |
|-----|---------------------------------------------------------------------------|
| jed_cd_1 | Group the parameters into some collection data structure. |
| jed_td_2 | Add an assertion statement either by comparing to connection status or to the value retrieved. |
| jed_dd_1 | The interface seem to have lot of methods. It can be broken down into many interfaces by grouping similar client-specific methods together. |

TABLE XXXII: TD repayment for Jedis

| ID  | Proposed techniques for repayment                                           |
|-----|---------------------------------------------------------------------------|
| myb_cd_1 | Introduce a variable and assign it to an integer with value 1 and use that variable instead. |
| myb_cd_2 | Break the long line within `<li>` tag by introducing a `<br>` tag to keep the number of characters less than 100. |
| myb_td_1 | Add more tests to source files which are far below the set threshold till the project coverage crosses the threshold. |

TABLE XXXIII: TD repayment for MyBatis

However, in reality, acting upon immediately on all the TD instances is not worthy. There are tools like Codescene which helps in prioritizing the refactoring targets. It prioritizes TD instances based on their technical debt interest rate. Look at the screenshot [figure 10] from Codescene for one of the projects.
Refactoring Targets
Prioritize improvements to these files since they have the highest technical debt interest rate.

Fig. 10: Refactoring targets for MyBatis

**G. Technical debt prevention**

There are no tools out there than can automatically prevent the occurrence of a TD. Because, it happens mostly due to human choices and mistakes. However, with rich information that could be exposed from the source code repositories [See III-E], we can prevent a TD instance from getting deployed into production systems. Also, once the developers get to know their mistakes with the help of such tools, the frequency of the same TD type getting introduced in the future can gradually get decreased.

**H. Challenges faced**

It was never a straightforward process of selecting the projects, feeding them into multiple tools, and observing the results. We overcame several limitations to present our empirical observations in a coherent manner. Here is a glimpse on some of the challenges which were worth mentioning,

- As many tools were very much similar to each other in terms of their functionality, choosing a diverse set of tools to cover many TD dimensions was the first and foremost challenge. Few tools were not free. So it took a couple of email conversations to get a limited time access.
- DesigniteJava quickly runs out of memory on a 12GB machine for projects involving $>100K$ LOC. As we wanted results for a chosen project from all the chosen tools, we had to choose projects which were not huge yet not small.
- For few tools, integration with Maven and Gradle were not consistent as both of those build tools behave differently. So we decided to stick just to Maven projects. But searching for Maven Java projects in GitHub was slightly time consuming as it seemed to have been outnumbered by Gradle projects.
- Another factor in the abovementioned time consuming search process was Jacoco. It’s out-of-the-box support for multi-module Maven projects was not simple. Because of that, for such projects, coverage was reported as 0% even when they had tests. So we had to limit ourselves to single-module Maven projects. But we are confident that by spending a little more time on configurations and customizations, multi-module projects can be made to work.

**IV. PROPOSAL**

**A. Cost model**

Here we propose a simple cost model for estimating the TD principal similar to the one present in Sonarqube TD plugin. But here we consider only till the level of TD item and not the TD type [See V].
| Cost                                      | Default value (in person-hours) |
|-------------------------------------------|---------------------------------|
| cost_to_fix_a_code_smell                  | 5                               |
| cost_to_fix_a_coding_guideline_violation  | 1                               |
| cost_to_fix_an_improper_test_design       | 4                               |
| cost_to_fix_a_lack_of_test               | 2                               |
| cost_to_fix_inadequate_test_coverage      | (difference between set threshold and current coverage) × Project’s LOC 1000 |
| cost_to_fix_a_design_smell               | 15                              |
| cost_to_fix_an_architecture_smell        | 25                              |

**TABLE XXXIV: A cost model for estimating TD principal**

So, the general simple formula would be,

\[
\text{TD principal estimate} = \text{cost to fix a code smell} \times \# \{\text{code smells}\} \\
+ \text{cost to fix a coding guideline violation} \times \# \{\text{coding guideline violations}\} \\
+ \text{cost to fix an improper test design} \times \# \{\text{improper test designs}\} \\
+ \text{cost to fix a lack of test} \times \# \{\text{lack of tests}\} \\
+ (\text{expected coverage} - \text{current coverage}) \times \frac{\text{Project’s LOC}}{1000} \\
+ \text{cost to fix a design smell} \times \# \{\text{design smells}\} \\
+ \text{cost to fix an architecture smell} \times \# \{\text{architecture smells}\}
\]

Now, with the cost model as mentioned in the above table **XXXIV**, we estimate the TD principal for the chosen projects but by only considering the instances that were represented as multiple tables within the subsection III-C.

- **Java WebSocket**

  \[
  \text{TD principal estimate} = 5 \times 1(\# \{\text{jws_cd}\}) + 1 \times 1(\# \{\text{jws_cd}\}) + 4 \times 1(\# \{\text{jws_td}\}) + 2 \times 1(\# \{\text{jws_td}\}) \\
  + (90 - 64.2) \times \frac{5000}{1000} + 15 \times 1(\# \{\text{jws_dd}\}) + 25 \times 1(\# \{\text{jws_ad}\}) \\
  = 5 + 1 + 4 + 2 + 129 + 15 + 25 \\
  = 181 \text{ person-hours}
  \]

- **JDBM3**

  \[
  \text{TD principal estimate} = 5 \times 1(\# \{\text{jdb_cd}\}) + 1 \times 1(\# \{\text{jdb_cd}\}) + 4 \times 1(\# \{\text{jdb_td}\}) \\
  + (90 - 82.6) \times \frac{9100}{1000} + 15 \times 1(\# \{\text{jdb_dd}\}) \\
  = 5 + 1 + 4 + 67.34 + 15 \\
  = 92.34 \text{ person-hours}
  \]

- **Jedis**

  \[
  \text{TD principal estimate} = 5 \times 1(\# \{\text{jed_cd}\}) + 1 \times 1(\# \{\text{jed_cd}\}) + 4 \times 1(\# \{\text{jed_td}\}) \\
  + (90 - 10.7) \times \frac{20800}{1000} + 15 \times 1(\# \{\text{jed_dd}\}) \\
  = 5 + 1 + 4 + 1649.44 + 15 \\
  = 1674.44 \text{ person-hours}
  \]

- **MyBatis**

  \[
  \text{TD principal estimate} = 5 \times 1(\# \{\text{myb_cd}\}) + 1 \times 1(\# \{\text{myb_cd}\}) + 4 \times 1(\# \{\text{myb_td}\}) \\
  + (90 - 84.3) \times \frac{22600}{1000} + 15 \times 1(\# \{\text{myb_dd}\}) + 25 \times 1(\# \{\text{myb_ad}\}) \\
  = 178.82 \text{ person-hours}
  \]
B. More tools to manage TD

DeepSource, a tool which is relatively new and got released for users during the month of November 2018, is something practitioners should keep an eye on. The team behind it seem to rapidly expand the feature set and support for multiple languages. The important thing is that the tool has a very neat and an elegant UI, a clear documentation of what it has to offer, and a responsive support. Also, to run the initial analysis, DeepSource is similar to Codacy, Codescene and dissimilar to Sonarcloud (without a Continuous Integration setup). It is integrated directly to the GitHub accounts and runs the initial analysis by cloning the repositories directly to their servers. Below is a screenshot [figure 17] from its website that mentions about some of their fully-available and preview features,

![Get complete visibility of your code's health. All in one place.](image)

Source Code
Continuously analyze your source code and spot critical issues before they make their way into production. Catch bug risks, anti-patterns and style violations with smart prioritization.

Security
Discover security flaws and weaknesses with comprehensive source code vulnerability analyzers.

Documentation
Documentation plays a huge part in making software accessible and maintainable. Keep a check on your documentation coverage on every change to source code.

Dependencies
Choose your dependencies carefully. Avoid getting into “too many dependency problem” by keeping track of the rate of increase of external dependencies used.

Test Coverage
Deliver software with confidence with tests. Determine the quality and coverage of your test suite.

Debt Index
Get bird’s eye view of the potential technical debt areas in your code. Make sure every part of your source code receives the highest level of care.

Looking at figure 17, there are two out-of-the box feature that easily makes DeepSource standout among its peers. Firstly, its ability to address “Documentation Debt”. Secondly, the tool’s ability to find issues with dependencies which quickly reminds us of the “Build Dependency Debt” introduced by Google [3].

Figures 12 to 18 gives a walk-through of the steps involved(in DeepSource) to run an analysis on a source code repository. The captions of those figures aid the screenshots with a description of what next to do.
Get started with DeepSource

- Secure by design
- Start analyzing code in under 5 minutes
- Seamless integration with your development workflow
- Optimized for less than 5% false-positives
- No credit card required — upgrade whenever you want

Fig. 12: DeepSource - Sign up page. Click on Sign up with GitHub

Welcome to DeepSource! 👋

We've created a new account for you with the email address yottabytt@gmail.com.

Next: Choose a provider with which you store your code. You can use DeepSource on a personal account or on a team account that you are a part of. You'll be able to add more integrations anytime from your dashboard later.

Fig. 13: DeepSource - Sign up process.
Fig. 14: DeepSource - Sign up process. Grant appropriate permissions to the tool before installing.

Fig. 15: DeepSource - Choose repository. Search and select the repository. We chose the numpy repository (forked from the popular scientific computing package’s repository).
Add this `.deepsourceto.ml` in your repository

To enable analysis, copy this generated configuration and paste it in a new file named `.deepsourceto.ml` in your repository's root.

After you've done that activate analysis using the button below, and... done!

Activate analysis

Fig. 16: DeepSource - Activate analysis. Copy the .toml file as per the instructions above to make the analysis work.

Fig. 17: DeepSource - History tab that gives information about the current and previous analysis.
We have thus presented our empirical observations which we hope to be beneficial to both practitioners and researchers. We believe this work can serve as a bridge connecting the concepts that are popular in literature with the real world software tools which are both old and new. Above all, we suspect this work can give a quick and easy end-to-end understanding even for an absolute beginner in the field of “Technical Debt in Software Development”. However, we may have not addressed many of the tools which might be actually be more popular and useful. But still, we believe this can be a starter for works that includes them.

ACKNOWLEDGMENT

We would like to thank Sean Barow (Director of Sales, Lattix), for quickly accepting our request and granting us a limited time access to “Lattix”.

REFERENCES

[1] Wikipedia contributors. Technical debt — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Technical_debt&oldid=887718884, 2019. [Online; accessed 18-April-2019].
[2] Ganesh Samarthyam, Mahesh Muralidharan, and Raghu Kalyan Anna. Understanding Test Debt, pages 1–17. Springer Singapore, Singapore, 2017.
[3] J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. Searching for build debt: Experiences managing technical debt at google. In Proceedings of the Third International Workshop on Managing Technical Debt, pages 1–6, 2012.

APPENDIX A

Architecture rule violations - A demonstration using Lattix

Lattix provides options to create multiple views that gives information about the project at different levels. Some of them are views for Dependency Structure Matrix(DSM) and Conceptual Architecture. DSM’s can be very helpful in identifying the cross-cut communication between classes and methods belonging to different components (say packages). Also, one can view architecture rule violations right within the DSM. As none of the chosen projects had specified such rules, we were not able to witness it. However, here were try to witness it.

• Look at figure 19, when there were no rules enabled and thus no violations for the project “MyBatis”
• Lattix gives an option of enabling/disabling “Can-Use” and “Cannot-Use” rules between components.

Fig. 18: DeepSource - Overview tab that gives high-level information about the analysis.

V. CONCLUSION

We have thus presented our empirical observations which we hope to be beneficial to both practitioners and researchers. We believe this work can serve as a bridge connecting the concepts that are popular in literature with the real world software tools which are both old and new. Above all, we suspect this work can give a quick and easy end-to-end understanding even for an absolute beginner in the field of “Technical Debt in Software Development”. However, we may have not addressed many of the tools which might be actually be more popular and useful. But still, we believe this can be a starter for works that includes them.

ACKNOWLEDGMENT

We would like to thank Sean Barow (Director of Sales, Lattix), for quickly accepting our request and granting us a limited time access to “Lattix”.

REFERENCES

[1] Wikipedia contributors. Technical debt — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Technical_debt&oldid=887718884, 2019. [Online; accessed 18-April-2019].
[2] Ganesh Samarthyam, Mahesh Muralidharan, and Raghu Kalyan Anna. Understanding Test Debt, pages 1–17. Springer Singapore, Singapore, 2017.
[3] J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. Searching for build debt: Experiences managing technical debt at google. In Proceedings of the Third International Workshop on Managing Technical Debt, pages 1–6, 2012.

APPENDIX A

Architecture rule violations - A demonstration using Lattix

Lattix provides options to create multiple views that gives information about the project at different levels. Some of them are views for Dependency Structure Matrix(DSM) and Conceptual Architecture. DSM’s can be very helpful in identifying the cross-cut communication between classes and methods belonging to different components (say packages). Also, one can view architecture rule violations right within the DSM. As none of the chosen projects had specified such rules, we were not able to witness it. However, here were try to witness it.

• Look at figure 19, when there were no rules enabled and thus no violations for the project “MyBatis”
• Lattix gives an option of enabling/disabling “Can-Use” and “Cannot-Use” rules between components.
Right Click on the cell which is highlighted in dark blue as shown in figure 19 which shows the dependency between the components org.apache.ibatis.binding and org.apache.ibatis.builder.

Select “Modify Rule” → Select “Cannot-Use”

Now look at figure 20, where you can see a small yellow triangle at bottom left of the cell indicating a violation.

More information about the violation is shown in a separate view as depicted in figure 21.
Fig. 20: DSM when rules were enabled. Note the small yellow triangle at bottom left of the cell indicating a violation.

| Rule Violations                                                                                                      |
|--------------------------------------------------------------------------------------------------------------------|
| org.apache.ibatis.builder.xml.XmlConfigBuilderTest                                                                 |
| @CANNOT-USE org.apache.ibatis.builder.MapperRegistry                                                                |
| org.apache.ibatis.builder.annotation.MapperAnnotationBuilder                                                         |
| @CANNOT-USE org.apache.ibatis.builder.BinderException                                                             |
| org.apache.ibatis.builder.xml.XmlConfigBuilderTest                                                                 |
| @CANNOT-USE org.apache.ibatis.builder.MapperRegistry                                                                |

Fig. 21: Rule violations view that gives further information about the violation that happens because of the dependencies.