2014 6th IEEE International Workshop on Managing Technical Debt

Explicating, Understanding and Managing Technical Debt from Self-Driving Miniature Car Projects

Md Abdullah Al Mamun
Division of Software Engineering
Chalmers University of Technology
and University of Gothenburg
Gothenburg, Sweden
Email: abdullah.mamun@chalmers.se

Christian Berger
Division of Software Engineering
Chalmers University of Technology
and University of Gothenburg
Gothenburg, Sweden
Email: christian.berger@gu.se

Jörgen Hansson
School of Informatics
University of Skövde
Skövde, Sweden
Email: jorgen.hansson@his.se

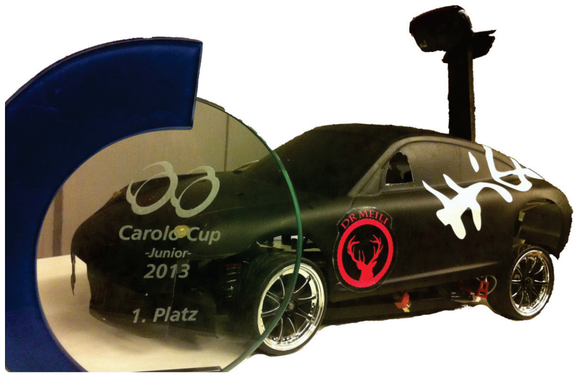
***Abstract*—Technical debt refers to various weaknesses in the design or implementation of a system resulting from trade-offs during software development usually for a quick release. Accumulating such debt over time without reducing it can seriously hamper the reusability and maintainability of the software. The aim of this study is to understand the state of the technical debt in the development of self-driving miniature cars so that proper actions can be planned to reduce the debt to have more reusable and maintainable software. A case study on a selected feature from two self-driving miniature car development projects is performed to assess the technical debt. Additionally, an interview study is conducted involving the developers to relate the findings of the case study with the possible root causes. The result of the study indicates that "the lack of knowledge" is not the primary reason for the accumulation of technical debt from the selected code smells. The root causes are rather in factors like time pressure followed by issues related to software/hardware integration and incomplete refactoring as well as reuse of legacy, third party, or open source code.**

I. Introduction

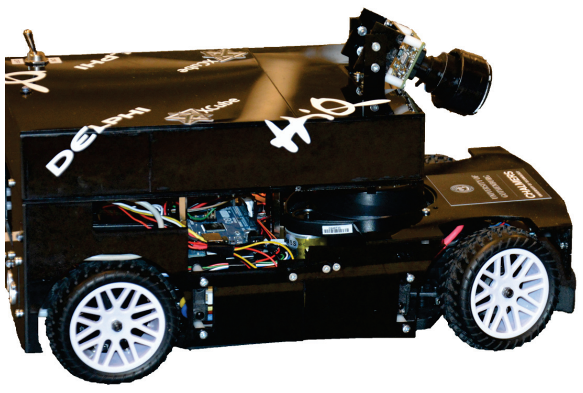
The automotive industry is going through a major transition where all important car manufacturers are seriously working towards the development of self-driving vehicles. Along with this trend, some of the Original Equipment Manufacturers (OEMs) have the vision to provide fully functional driver-less cars by 2020 [1].

The 2007 DARPA Urban Challenge facilitated the research and development of self-driving cars and safety systems in today's vehicles [2], [3]. Prototypical vehicles like Google Car have already shown the promise of self-driving vehicle technology [4]. However, the use of highly expensive sensors like a 64-layer laser scanner on the car's roof is not a commercially viable solution. Thus, researchers and industry are left with the challenge of realizing automated driving functionalities through general sensor types like radar, sonar, camera, etc. and operating them with reduced computational power.

The aforementioned competition also fostered the yearly competition "CaroloCup" [1] for universities. This competition was initiated at Technische Universität Braunschweig to promote research and education for the development of 1/10 scale self-driving vehicles. These vehicles need to demonstrate similar features as their real-scale counterparts on a small scale real world track. The capabilities range from following their own lane, overtaking stationary and/or moving vehicles blocking their own lane, handling intersections safely according to the traffic rules, and to park a vehicle on a sideway parking strip.



(a) Self-driving miniature vehicle developed in 2013 from team Meili.



(b) Self-driving miniature vehicle developed in 2014 from team Legendary.

Fig. 1. Self-driving miniature vehicles developed by Chalmers University of Technology and University of Gothenburg students for the Carolocup competition in the last two years.

The vehicle in Fig. 1(a) participated at the CaroloCup 2013 and won the junior-level competition. The vehicle in Fig. 1(b) participated at the advanced level of the 2014 Carolocup

[1]www.carolocup.de

competition and it was among the top three newcomer teams. Parts of the software from the 2013 project have been reused by team Legendary in 2014.

For future competitions, we have planned to further reuse and improve the existing algorithms. Thus, our goal is to avoid accumulating too much technical debt that might impose a heavy payback of interest in evolved versions of the algorithms in the future.

Technical debt is a metaphor that uses concepts from financial debt to describe the trend of increasing software development costs over time. The term technical debt was first introduced by Cunningham who described it as "*Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt*." [5]. A research from Gartner and CAST reported that even on a conservative estimate an average Fortune 2000 company has more than $200 million IT debt on average and the global IT debt in 2010 can be as high as $500 billion that can be doubled by 2015 [6].

Therefore, we want to identify and understand how much technical debt we have already acquired and how the 2013 project from team Meili influenced the 2014 project in team Legendary in terms of technical debt.

A. Research Goal and Research Questions

The overall goal of this study is to better understand where to focus in the next year's participation to ensure reusability and maintainability of the software for future student teams.

Therefore, we are concentrating on the following research questions:

RQ1: How did the technical debt evolve over time for a self-driving miniature car that competed twice in an international competition?

RQ2: How did the experience from a previous participation influence the technical debt of the recent software?

RQ3: What are the most important issues related to the development process that need to be adapted to reduce the technical debt for teams?

To answer research questions RQ1 and RQ2, we performed a case study on one of the most important features "lane-detection" from the 2013 and 2014 projects. The case study includes reviewing the software with SonarQube, a technical debt analysis tool. Subsequently, the developers of the features were interviewed regarding different factors related to the research questions RQ3.

B. Contributions of the Article

The data analysis indicates that the developers are mostly aware of the severity of the most occurring code smells in their own source code. However, they still took on the debt due to some other influencing factors. Time pressure followed by hardware/software integration and incomplete refactoring came up as the most important factors. Furthermore, the source code review revealed that a large portion of the accumulated technical debt in the Legendary-2014 project sourced from third party and freely available code collected from the Internet.

C. Structure of the Article

The rest of the paper is structured as follows: Firstly, related work followed by research methodologies are presented. Then, the results of the paper are outlined in Sec. IV. The analysis and discussion are described in Sec. V and the paper ends with a conclusion and outlook for future work.

II. Related work

Codabux and Williams performed an industrial case study consisting of observing and interviewing developers to understand how technical debt is characterized, addressed, prioritized, and how different factors relate to the decisions that led to technical debt [7]. They worked with a mid-sized agile development team.

Another industrial case study exploring the effect of technical debt on the software project is performed by Guo et al. [8]. They tracked a single delayed maintenance task throughout its lifecycle and simulated how managing technical debt can impact the project result. The study results also showed how and to what extent technical debt affects software projects. In a similar direction Zazworka et al. [9] worked towards identifying the impact of software quality with the technical debt.

Xuan et al. [10] proposed a concept of debt-prone bugs to model the technical debt in software maintenance, which is conceptually similar to SonarQube. They performed a case study on Mozilla to examine the impact of debt-prone bugs in software products.

SonarQube[2] (formerly Sonar) is a plugin-based open source web application for quality management. SonarQube manages results of various code analysis tools. It is used to continuously analyze and measure a project's technical quality. SonarQube computes technical debt based on the Software Quality Assessment based on Lifecycle Expectations methodology (SQALE). The SQALE [11] is a methodology that organizes non-functional requirements related to code quality. Non-functional requirements are realized in terms of coding rules and issues in the SonarQube implementation of the SQALE method.

A conceptual study of technical debt by Martini et al. [12] identifies factors that might favor the accumulation of technical debt. Based on our experience with the development of self-driving miniature cars [13] and the related projects [14], [15], we base our study on selected factors from their study that are relevant to the car projects. The list of factors is shown in Fig. 3.

III. Methodology

This section outlines the research methodology, data collection process, and the environment of the project considered in this study.

The goal of this study is to understand technical debt from a core feature of a self-driving miniature car that is participating in

[2]www.sonarqube.org/

an international competition. By the competition rules, a project group consists of bachelor-level and master-level students. Our student teams have participated twice in the past two years. Due to the fact that students finish their studies, we are faced with a team turnover each year. However, we want to avoid starting from scratch each year by reusing as much as possible to save time and effort and also to improve existing solutions.

In 2013, our student group competed at the junior-level of the competition and in 2014 at the advanced level. The team in 2014 started with the existing code for image processing and lane-detection from the 2013 team. Since it is planned to participate in this competition in the upcoming years as well, we are aiming for code reuse and maintenance for the future teams. Thus, it is important to understand the current status of the code and the root-causes in the development process that might favor accumulating technical debt.

A. *Design of the study*

We planned and conducted a case study to assess how technical debt evolved in the implementation of the lane detection algorithm of a self-driving miniature vehicle. To further investigate the reasons of the sources of technical debt, the developers were interviewed by using an online questionnaire.

We used SonarQube v4.3, Sonar-Runner v2.4 and the SonarWay[3] plugin for C/C++ v2.3 to review the C++-source code and to collect code metrics.

A quality profile of a SonarQube language plugin has language specific rules in addition to the SQALE specific rules such as duplicate blocks, failed unit tests, etc. that are common to every language. We used the default quality profile of the SonarWay plugin for this study because we wanted to know which are the most frequently occurring sources of technical debt in the projects that are also broadly recognized as the causes thereof.

A code smell is a source code symptom and a possible indication of a deeper problem e.g., weakness in the design that can potentially slow down the evolution of the software or increase the risk of bugs in the future. Code smells are not bugs and they do not prevent the current version of the software from functioning. The term "code smell" was used more frequently after it had been focused in the book by Fowler et al. [16].

From our source code review with the SonarQube tool, we found that the most frequently occurring issues are related to code smells. Based on this input, we designed the questionnaire for a subsequent interview study with the developers. We decided to focus on the most occurring code smells than the less frequently existent design-related rule violations as the overall system design of the self-driving miniature car is following a widely adopted design paradigm (cf. [17], [18]). The motivation thereof is that if we find that the developers are aware of the severity of the code smells but they still ignore them in their coding, we expect that there were other process-related or environment-related factors like *time pressure, software/hardware integration, incomplete refactoring*, etc. that influenced them to take such shortcuts.

[3]www.sonarsource.com/products/plugins/languages/cpp/

We selected developers as participants in our study who contributed on a similar and comparable level to the core feature of the self-driving car that we used for this study. Based on these criteria, three developers were selected, where two of them were selected from Meili-2013 project and the other developer from the Legendary-2014 project. The developers were interviewed using an online questionnaire based on the output of the code review from the SonarQube tool.

B. *Data Collection*

For this study, we collected data in two steps. Firstly, we reviewed the code followed by conducting an interview to collect the feedback and impressions from the developers of the selected feature through an online questionnaire.

1) Reviewing Source Code: From the 2013 project, we used the final snapshot of the development for the lane detection algorithm; for the 2014 project that used the 2013 project's latest version to begin with, we have used 33 snapshots of the lane-detection feature dating back from October 2013 to February 2014 in total.

The selected quality profile of the SonarWay v2.3 C/C++ plugin consists of 84 coding rules, which were checked on the selected implementations. From the code review, we found that the most frequently occurring rules are related to code smells. It should be noted that the quality profile includes rules related to design, for example, coding rules related to duplication, complexity, etc. which usually take more time to fix than simple code smells. However, those design-related rule violations were greatly outnumbered by the simple code smells that are quick to fix.

The collected code metrics for a specific snapshot of a selected feature are code size (*LOC, lines, statements, files, functions*), duplications (*% of duplication, by lines, by blocks, by files*), complexity (*by functions, by files, total complexity*), *technical debt* (in person hour), and *number of identified issues* (in the categories *blocker, critical, major, minor, info*).

The collected metrics from the code review is used to answer *RQ1* and *RQ2*. It also helps partially to answer *RQ3* focusing on the development process. The duplication, complexity, and the issues found in the developed code vs. the reused code from other sources also answer whether the factor *"reuse of legacy/open source/third party"* code is a contributing factor to technical debt. The questionnaires are finally used to answer research question *RQ3*.

2) Questionnaire: The questionnaire is designed based on the result of the code review by SonarQube. Among the identified issues, we took the most frequently occurring issues, which are types of code smells. The study was performed after the project was finished, and the students successfully passed their examination. During the project, we never introduced the concept of technical debt.

The developers were asked to categorize the selected code smells according to their respective severity without knowing that the code smells originated from the miniature vehicle's source code. They categorized the code smells listed in Fig. 2 in the scale *critical, major,* and *minor*. Detailed descriptions together with examples were provided for each of these code smells.

- CS1: *"new" and "delete" should be used*
- CS2: *The right hand operand of a logical && or ||operator shall not contain side effects*
- CS3: *Sections of code should not be "commented out"*
- CS4: *If-else statements must use braces*
- CS5: *The statement forming the body of a switch, while, do . . . while, and for-statement shall be a compound statement*
- CS6: *Nested code blocks should not be used*
- CS7: *Insecure functions "strcpy", "strcat", and "sprintf" should not be used*
- CS8: *Magic numbers should not be used*
- CS9: *If statements should not be nested too deeply*
- CS10: *An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator, respectively*
- CS11: *Switch statements should have at least three case clauses*

Fig. 2. List of the most frequently occurring issues (code smells in this case) from the source code review by SonarQube.

We additionally interviewed the developers using the following questions to understand the current drawbacks and bottlenecks in the source code of the lane detection algorithm that can be troublesome to refactor the source code in the future.

1) What are the most urgent parts in your source code that are required to be fixed/changed so that others can make proper use of it in the future?
2) How many days does a developer need to fix the issues that you have mentioned in Question-1?
3) What are the most influential factors that caused these parts that need to be fixed?

For the last question, the students were asked to rank each of following factors in Fig. 3 in the Likert scale of 4. An even number of choice in the Likert scale is used to avoid a neutral middle-point which might be selected by developers as a default answer when, for example, they are indifferent in their opinion.

- *Lack of domain experience*
- *Lack of experience with the middleware*
- *Lack of experience with the tools*
- *Issues related to software/hardware integration*
- *Malfunctioning hardware*
- *Uncertainty of use cases in the beginning*
- *Time pressure: Deadlines with penalties*
- *Priority of features over product*
- *Reuse of Legacy / third party / open source*
- *Parallel development*
- *Incomplete refactoring*
- *Technology evolution*
- *Human factors*

Fig. 3. List of possible factors contributing to technical debt.

After the first questionnaire had been completed, a second questionnaire was used to ask the developers regarding their agreement about the severity levels of the code smells as detected by SonarQube. In this questionnaire, we informed the developers about the severity of the code smells ranked by SonarQube. Afterwards, we asked them about how much they agree with the ranking of the code smells by SonarQube in the Likert scale of 4 with the options *strongly agree*, *agree*, *disagree*, and *strongly disagree*. The result of this questionnaire is shown in Tab. IV. An even number of options was used with the same motivations as mentioned before for the previous questionnaire.

The purpose of the questions regarding code smells in the first questionnaire is to understand the knowledge of the developers about the code smells. The purpose of the second questionnaire is to cross-check their responses from the first questionnaire. We sent the second questionnaire about a week after getting the responses of the first questionnaire.

IV. Results

This section presents the results from the case study and the interviews. Tab. I shows five different categories of issues. "blocking" is the most crucial issue category, which we did not identify in any of the selected versions. The "info" category represents very low priority issues, for instance, *"Comments should not be located at the end of lines of code,"* which suggests that comments should be placed somewhere before a line rather than at the end of a line. The fixing time for this least severe issue is one minute per instance. Due to the absence of the "blocking" category and the low importance of the "info" category, we excluded both of them and considered the remaining three issue categories in this study.

Among the code smells in Fig. 2 selected from all the reported issues as mentioned in Table I, critical code smells take about 15-20 minutes, major code smells take about 10-15 minutes, and minor code smells take about 5-10 minutes for each instance. Even though the fixing time for a single code smell seems small, considering the large number of occurrences, they accumulated a considerable amount of technical debt. In fact, the selected eleven code smells accumulate almost 50% of all the accumulated technical debt calculated in the code bases.

TABLE I. Issues from the projects as identified by SonarQube.

Severity	2013 project # of detected issues	2014 project # of detected issues
Blocking	-	-
Critical	-	26
Major	37	451
Minor	29	208
Info	6	43
Total	72	728

Tab. II shows a part of the interview study results from the first questionnaire. The issues are ranked according to their overall score rated by the developers. The "*overall rating*" in Tab. II is calculated averaging the responses of all three developers.

Tab. III shows the aggregated result of the case study. The first and third columns are representing the same information but are repeated for the sake of clarity for the developers' response vs. SonarQube rating of the code smells. The descriptions of the code smells are available in the Sec. III-B2. The color code in this table is used to graphically depict the distance between

TABLE II. FACTORS HAVING AN IMPACT ON THE DEVELOPMENT THAT IN CONSEQUENCE MIGHT NEGATIVELY IMPACT THE SOFTWARE REUSE.

Factors	2013 project		2014 project	
	Developer I	Developer II	Developer III	Overall rating
Time pressure: Deadlines with penalties	●●●○	●●●●	●●●●	●●●◕
Issues related to software/hardware integration	●●●●	●●●○	●●●○	●●●◔
Incomplete refactoring	●●●○	●●●●	●●●○	●●●◔
Human factors	●●●●	●●●○	●●○○	●●●⊖
Lack of experience with the middleware	●●●●	●●○○	●●●○	●●●⊖
Lack of domain experience	●●●●	●●○○	●●○○	●●◕⊖
Uncertainty of use cases in the beginning	●●●○	●●●○	●●○○	●●◕⊖
Reuse of Legacy / third party / open source	●○○○	●●●○	●●●○	●●◔⊖
Priority of features over product	●○○○	●●●●	●●○○	●●◔⊖
Lack of experience with the tools	●●●○	●●○○	●●○○	●●◔⊖
Technology evolution	●●○○	●●○○	●●●○	●●◔⊖
Malfunctioning hardware	●●●○	●●●○	●○○○	●●◔⊖
Parallel development	●●○○	●●●●	●○○○	●●◔⊖

Legend:
●●●●: Large Extent, ●●●○: Some Extent, ●●○○: Small Extent, ●○○○: None al all

the developers' response and the actual value defined by the tool. For example, the background color "*green*" represents that there is no difference between the response of the respective developer with the SonarQube rating. The meaning of the other background colors is provided in the legend of the table. In addition, the colored cells in Tab. III also include short textual codes in parenthesis representing the corresponding cell colors, so that the color codes are readable even in the black and white printouts. The last column in this table shows how the selected code smells evolved over time for the project from 2014 to have an indicator about the urgency of the identified code smell.

Tab. IV shows the result of the second part of the questionnaire. It shows how much the developers agree or disagree with the code smells' severity ranked by the SonarQube. The color codes in this table are used to intuitively depict the responses of the developers, which can also be read from the dots in the cells.

TABLE IV. DEVELOPERS AGREEMENT WITH THE SEVERITY OF THE CODE SMELLS RANKED BY SONARQUBE.

Severity	Code Smells	2013 project		2014 project
		Developer I	Developer II	Developer III
Critical	CS1	●●●●	●●●○	●●●○
	CS2	●●●●	●●●○	●●○○
Major	CS3	●●●○	●●○○	●●○○
	CS4	●●●●	●●○○	●●●●
	CS5	●●●○	●●○○	●●●●
	CS6	●●●○	●●●○	●●○○
	CS7	●●○○	●●●○	●●●●
Minor	CS8	●●●●	●●●○	●●●○
	CS9	●●●○	●●●○	●●●○
	CS10	●●●○	●●●○	●●●○
	CS11	●●●○	●●●○	●●○○

Legend:
●●●●:Strongly Agree (Dark Green), ●●●○:Agree (Light Green),
●●○○:Disagree (Light Red), ●○○○:Strongly Disagree (Dark Red)

V. ANALYSIS AND DISCUSSION

Misunderstanding or not being aware of a code smell can influence the technical debt in a project. When programmers are unaware of the risk of a certain code smell, they can ignore them and in consequence, the technical debt will grow. In our case, since the developers are aware of the majority of the code smells, the reason for the technical debt accumulation must be motivated by other factors that have influenced the developers to ignore them.

Considering Tab. II, the most important factors that contributed to the technical debt were time pressure, incomplete refactoring, malfunctioning hardware, lack of experience with the middleware, parallel development, and human factors. From our observations as being supervisors of these projects, we know that malfunctioning hardware can severely delay the project. As the project members are also responsible for the hardware and software integration for their vehicle, they have to learn and work with different middleware stacks. Learning, configuring, and recompiling can be highly time-consuming. The incomplete refactoring can also be related to time pressure.

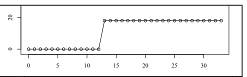
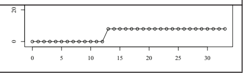
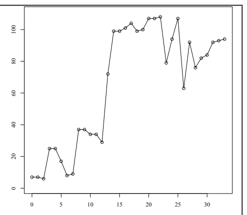
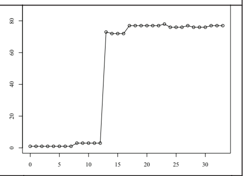
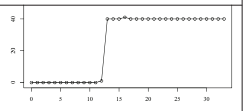
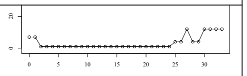
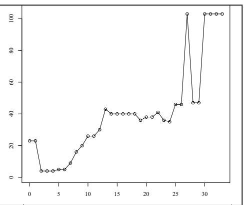
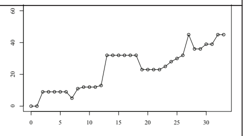
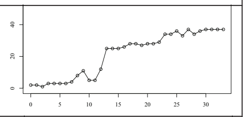
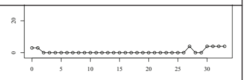
As we have studied the evolution of the code base, we have observed that a big portion of the accumulated technical debt in the 2014 project came from third party/freely available code reused from the Internet. Hence, the team "imported" technical debt that was introduced by people outside their own project. For these parts of the code, which are not entirely written by the developers, the status of the code smells remained more unchanged throughout the project compared to the code that is entirely written by the developers.

Relating back to the research questions, for *RQ1*, the technical debt for the lane detection algorithm grows drastically from two days in the 2013 project to 23 days in the 2014 project.

To answer *RQ2*, we observe that even though the 2014 project reused the solution of the 2013 lane detection implementation, the additions extended the feature from 420 LOC in 2013 to 2,826 LOC in 2014 by using a considerable amount of third party code. Thus, we have observed that the selected feature in the 2014 project has accumulated a lot more technical debt compared to the 2013 project.

To address *RQ3*, time pressure appears to be the most urgent issue during the development process. Among the other factors, incomplete refactoring, malfunctioning hardware, middleware experience, parallel development, and human factors are rated higher on average. Since, extending the competition's deadline is not possible, internal processes, environment, and support needs to be improved. For example, more assistance with the middleware technology and the hardware integration can be provided to the students.

TABLE III. Perceptions from the developers on the severity of the code smells compared to the SonarQube tool. Most occurring code smells from the selected categories are presented with their amount of occurrence and time to fix. Graphs are showing how the code smells evolved over time in the 2014 project. The background colors indicate the difference of the choices by the developers compared to the severity of the code smells determined by SonarQube.

Severity	Top 3 code smells from the selected categories	SonarQube	2013 project		2014 project	Code review by SonarQube				Code smells over time for the 2014 project (X-axis: revision number, Y-axis: number of code smell occurrence)
						Latest code from 2013 project		Latest code from 2014 project		
			Developer I	Developer II	Developer III	Number of occur-rence	Fixing time (hour)	Number of occur-rence	Fixing time (hour)	
Critical	CS1	●●●●	●●●○ (lr)	●●●○ (lr)	●●○○ (dr)	-	-	18	6	
	CS2	●●●●	●●●○ (lr)	●●●○ (lr)	●●●● (gr)	-	-	8	2	
Major	CS3	●●●○	●●○○ (lr)	●●●● (lb)	●●○○ (lr)	7	2.33	94	31.33	
	CS4	●●●○	●●●○ (gr)	●●○○ (lr)	●●●○ (gr)	-	-	77	12.83	
	CS5	●●●○	●●●○ (gr)	●●○○ (lr)	●●●○ (gr)	-	-	40	6.67	
	CS6	●●●○	●●●● (lb)	●●●● (lb)	●●●○ (gr)	7	1.17	12	2	
	CS7	●●●○	●●●● (lb)	●●●○ (gr)	●●●○ (gr)	5	1.67	-	-	
Minor	CS8	●●○○	●●●● (db)	●●●● (db)	●●○○ (gr)	23	1.92	103	8.58	
	CS9	●●○○	●●●● (db)	●●●○ (lb)	●●○○ (gr)	-	-	45	15	
	CS10	●●○○	●●●○ (lb)	●●●○ (lb)	●●●○ (lb)	2	0.17	37	3.08	
	CS11	●●○○	●●○○ (gr)	●●○○ (gr)	●●●○ (lb)	3	0.25	4	0.33	
Total hours to fix the selected code smells						7.5 hours		87.83 hours		
Total days to fix all issues			10 days	7 days	3 days	2 days		23 days		

Legend:
●●●●:Critical, ●●●○: Major, ●●○○: Minor, ●○○○: None
Indication of Color: *Green* (gr) - perfect match,
Light Blue (lb) - overrated by one notch, *Dark Blue* (db) - overrated by two notches,
Light Red (lr) - underrated by one notch, *Dark Red* (dr) - underrated by two notches.

Even though the "*reuse of legacy/third party/open source*" aspect scored between major and minor on average; the result of the source code review reveals that this is a highly influencing factor accumulating a high amount of technical debt in the 2014 project. In the light of this factor, a required measure for the future projects would be to increase the awareness of the existing technical debt in the third party code base. Furthermore, introducing a technical debt estimation tool to detect violations of rules early during the development would additionally help to increase the awareness.

Kruchten et al. [19] associated technical debt with dimensions "invisible" and "negative value" in their four color backlog model. They suggested the use of a dedicated backlog for technical debt so that it becomes explicit and visible. From the perspective of explicitness and visibility, automated tools like SonarQube can be used. In addition, such a tool helps to increase the awareness as also identified by Kruchten et al. as the most important aspect to implement at the first place.

The color coded cells in Tab. III show how the developers understand the severity of the code smells compared to the severity rated by the SonarQube tool. The *green (gr)* color means that the developers rated the code smells with the equal severity as rated by the tool. The *dark red (dr)* and *light red (lr)* colors indicate that the developers underrate the severity of the relevant code smells. Underrating indicates challenges with a lack of knowledge about the code smells that need to be tackled with knowledge sharing/distribution and training. On the other hand, *dark blue (db)* and *light blue (lb)* colors indicate an overrating of the code smells. We do not consider overrating as a negative indication because if a developer considers a certain code smell as severe he would like to tackle it which in return will reduce the overall technical debt. Thus, we can conclude that only *red* colors represent negative indications. The *red* colored cells represent 27% of the whole response space resulting in less than one third. Since code smell rating represents knowledge of the developers, we can conclude that the *lack of knowledge* is not the most urgent key factor that contributes to accumulating technical debt from the most frequently occurring code smells.

Tab. III also shows the count of code smells identified in the projects. Some of the code smells occur in both projects while others are related to a specific project. This table also shows the amount of technical debt in hours for the selected code smells in both projects. The fixing time for a single instance of a code smell can be found from this table, which is between 5-20 minutes. The second last row of Tab. III shows that the selected 11 code smells accumulates 7.5 hours of technical debt for the 2013 project and 87.83 hours of technical debt for the 2014 project where the total calculated technical debt for these two projects are two days and 23 days, respectively. Thus, the selected eleven code smells accumulate about 50% of the whole technical debt calculated in the code basis for both projects.

The last column of Tab. III depicts the occurrence of the code smells over time for the 2014 project. A sudden boost can be noticed in the code smells CS1-5, CS9, and CS10. Interestingly, all of the graphs corresponding to these code smells show that the code smell counts increase suddenly at the same point of time. This indicates that something major had happened at that specific point in time. Analyzing the code base, we found that this happened due to the integration of a large amount of legacy/third party/open source code into the 2014 project. Among these, some parts of the software, which are related to CS1, CS2, CS4, and CS5, were not refactored and thus, we see the graphs continue almost constantly from the point of introduction to the end of the project. On the other hand, source code related to CS3, CS9, and CS10 were partially refactored.

The result of the second questionnaire in Tab. IV shows that the developers strongly agree with the severity rating of the SonarQube by 21.21%, agree by 54.55%, and disagree by 24.24%. It is interesting to see that the 24.24% disagreement level in the second questionnaire is very close to the result of 27% underrating of the code smells from the first questionnaire as depicted in Tab. III. From these results, we can see that about 25% of the code smells would remain in the system even if the developers are given enough time to avoid bad coding practices. This also gives us an indication how much we can gain by training the developers about code smells. However, the result of this study also reflects that if there are severe factors like time pressure and the like, developers might ignore correcting coding styles that do not raise a compilation or runtime error. On the other hand, the results also tell us that the lack of knowledge is not the most crucial factor for the accumulation of technical debt in a project.

The threats to validity to our work are discussed according to Runeson and Höst [20]. Regarding construct validity, we used a well-established tool to run the code analysis for the students' code. Thus, the code smells that have been identified were based on widely recognized issues. Furthermore, the interview study was conducted with the original authors of the specific feature and thus, their knowledge, impressions, and motivation could be included. To avoid any bias from a pending examination situation, this study was carried out after the involved students had passed their examination related to the respective projects.

Regarding internal validity, the small sample size can be considered an issue as we have focused on two projects with a total number of three developers. However, as both projects were addressing the same competition with the same rules and conditions, the setting for both were the same. Therefore, the evolution of technical debt could be studied for our specific setting.

Concerning external validity, our specific setting can be considered as a potential issue because we cannot easily generalize from our findings. However, as the identified root causes, presented in Tab. II might be present in other student robotic competition as well, our results are useful for such settings as well.

VI. Conclusion and Future Work

This paper reports a case study on the lane detection algorithm from two self-driving miniature vehicle projects that participated successfully in an international competition. The case study reviewed the software with the technical debt analysis tool SonarQube. An interview study with the developers of the features was carried out to understand the reasons of the existence of the code smells identified by the SonarQube tool. The interview study also focused on finding root causes of the technical debt in the project.

The analysis of data indicates that the developers are aware of the severity of the most occurring code smells in their own source code. However, they still took on the debt and the most causing reasons were time pressure, issues related to software-hardware integration and incomplete refactoring. In addition, the source code review reveals that a big portion of the accumulated technical debt in the 2014 project originated from using third party/freely available code collected from the Internet.

Thus, future projects that continue the work from the 2013 and 2014 projects need to adjust the development process to address the identified root causes and to increase the level of awareness for technical debt and its consequences. This is particularly important because the source code review revealed that the reuse of legacy, third party, or open source code significantly contributed to the accumulation of technical debt. As *time pressure, software/hardware integration*, and *incomplete refactoring* came up to be the most-urgent factors, improving the test and hardware/software integration process towards a continuous deployment on the resulting hardware platform will be the focus in future projects, which we believe should increase the early feedback and the overall quality of the system.

Acknowledgment

The authors would like to thank all of the students who contributed significantly to realize the Meili and the Legendary self-driving miniature vehicles.

References

[1] J. Hirsch, "Self-driving cars inch closer to mainstream availability," Oct. 2013. [Online]. Available: http://www.latimes.com/business/autos/la-fi-adv-hy-self-driving-cars-20131013,0,5094627.story

[2] F. W. Rauskolb, K. Berger, C. Lipski, M. Magnor, K. Cornelsen, J. Effertz, T. Form, F. Graefe, S. Ohl, W. Schumacher, J.-M. Wille, P. Hecker, T. Nothdurft, M. Doering, K. Homeier, J. Morgenroth, L. Wolf, C. Basarke, C. Berger, T. Gülke, F. Klose, and B. Rumpe, "Caroline: An Autonomously Driving Vehicle for Urban Environments," *Journal of Field Robotics*, vol. 25, no. 9, pp. 674–724, Sep. 2008. [Online]. Available: http://dx.doi.org/10.1002/rob.20254

[3] C. Berger and B. Rumpe, "Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System," in *Proceedings of the INFORMATIK 2012*, U. Goltz, M. Magnor, H.-J. Appelrath, H. K. Matthies, W.-T. Balke, and L. Wolf, Eds., Braunschweig, Germany, Sep. 2012, pp. 789–798.

[4] S. Thrun, "What we're driving at," p. 1, Apr. 2010. [Online]. Available: http://googleblog.blogspot.se/2010/10/what-were-driving-at.html

[5] W. Cunningham, "The WyCash portfolio management system," in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, ser. OOPSLA '92. New York, NY, USA: ACM, 1992, p. 2930. [Online]. Available: http://doi.acm.org/10.1145/157709.157715

[6] A. Kyte, "Measure and manage your IT debt," Gartner and CAST, Research Report, 2010. [Online]. Available: http://imagesrv.gartner.com/media-products/pdf/cast_software/gartner2.pdf

[7] Z. Codabux and B. Williams, "Managing technical debt: An industrial case study," in *Proceedings of the 4th International Workshop on Managing Technical Debt (MTD)*, May 2013, pp. 8–15.

[8] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. da Silva, A. L. M. Santos, and C. Siebra, "Tracking technical debt - an exploratory case study," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2011, pp. 528–531.

[9] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD)*. New York, NY, USA: ACM, 2011, pp. 17–23. [Online]. Available: http://doi.acm.org/10.1145/1985362.1985366

[10] J. Xuan, Y. Hu, and H. Jiang, "Debt-prone bugs: technical debt in software maintenance," *International Journal of Advancements in Computing Technology 2012a*, vol. 4, no. 19, pp. 453–461, 2012.

[11] J. Letouzey and M. Ilkiewicz, "Managing technical debt with the sqale method," *Software, IEEE*, vol. 29, no. 6, pp. 44–51, Nov 2012.

[12] A. Martini, J. Bosch, and M. Chaudron, "Architecture technical debt: Understanding causes and a qualitative model," in *(Accepted for publication) Proceedings of the 40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Verona, Italy, 2014.

[13] C. Berger, "From a Competition for Self-Driving Miniature Cars to a Standardized Experimental Platform: Concept, Models, Architecture, and Evaluation," *Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 63–79, Jun. 2014. [Online]. Available: http://arxiv.org/abs/1406.7768

[14] C. Berger, M. A. Al Mamun, and J. Hansson, "COTS-Architecture with a Real-Time OS for a Self-Driving Miniature Vehicle," in *Proceedings of the 2nd Workshop on Architecting Safety in Collaborative Mobile Systems (ASCoMS)*, E. Schiller and H. Lönn, Eds., Toulouse, France, Sep. 2013.

[15] C. Berger, E. Dahlgren, J. Grunden, D. Gunnarsson, N. Holtryd, A. Khazal, M. Mustafa, M. Papatriantafilou, E. M. Schiller, C. Steup, V. Swantesson, and P. Tsigas, "Bridging Physical and Digital Traffic System Simulations with the Gulliver Test-bed," in *Proceedings of 5th International Workshop on Communication Technologies for Vehicles 2013*, Lille, France, May 2013, pp. 169–184. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37974-1_14

[16] M. Fowler, K. Beck, J. Brant, and W. Opdyke, *Refactoring: Improving the Design of Existing Code*, ser. Object Technology Series. Addison-Wesley, 1999. [Online]. Available: http://books.google.se/books?id=1MsETFPD3I0C

[17] M. Buehler, K. Iagnemma, and S. Singh, Eds., *The 2005 DARPA Grand Challenge: The Great Robot Race*. Berlin Heidelberg: Springer Verlag, 2007.

[18] C. Berger and M. Dukaczewski, "Comparison of Architectural Design Decisions for Resource-Constrained Self-Driving Cars - A Multiple Case-Study," in *Proceedings of the INFORMATIK 2014*, Stuttgart, Germany, Sep. 2014, p. 12.

[19] P. Kruchten, R. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, Nov. 2012.

[20] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, Dec. 2008. [Online]. Available: http://link.springer.com/10.1007/s10664-008-9102-8