Clean Code and Design Educational Tool

Simona Prokić*, Katarina-Glorija Grujić*, Nikola Luburić*, Jelena Slivka*, Aleksandar Kovačević*, Dragan Vidaković* and Goran Sladić*
* Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
{simona.prokic, katarina.glorija, nikola.luburic, slivkaje, kocha78, vdragan, sladicg}@uns.ac.rs

***Abstract* - Many different code snippets can implement the same software feature. However, a significant subset of these possible solutions contains difficult-to-understand code that harms the software's maintainability and evolution. Such low-quality code snippets directly harm profit, as frequent and fast code change enables businesses to seize new opportunities. Unfortunately, they are also prevalent in an industry that consists mostly of junior programmers.**

We developed a platform called Clean CaDET to tackle the prevalence of low-quality code from two angles. The Smell Detector module presents a framework for integrating AI-based code quality assessment algorithms to identify low-quality code as the programmer is writing it. The Smart Tutor module hosts a catalog of educational content that helps the programmer understand the identified issue and suggests possible solutions. By combining the quality assessment with the educational aspect, our integrated solution presents a novel approach for increasing the quality of code produced by our industry.

***Keywords* - clean code; code smells; maintainability; readability**

I. INTRODUCTION

Societies' ever-growing needs and desires generate a high demand for new software solutions. Rushing to fulfill the market's needs, software vendors sacrifice quality to develop software products faster and with more functionality. Developers prioritize getting the code to work and neglect writing readable and easy to change code [1]. Such low-quality code contains code smells, which represent structures in the code that decrease the software's ability to adapt and address new requirements [2][3].

A codebase with many code smells is challenging and costly to maintain and evolve [1][2]. Such code is error-prone and unreliable, resulting in failures that can harm the software's users and its vendor's brand [3][4]. Furthermore, they adversely affect the development process, reducing the developer's productivity [3].

Detecting harmful code smells is not simple. Notably, the definition of many code smells is ambiguous, and there is a high disagreement among developers when determining if some code is affected by a smell [5]. The existence of code smells can depend on the software semantics and architecture. Likewise, some code smells emerge when using design patterns and thus do not require refactoring [6]. Finally, it is not always easy or economically feasible to refactor code smells [2]. Therefore, determining the presence, harmfulness, and solution for a code smell requires significant developer expertise.

We developed the **Clean** **C**ode **a**nd **D**esign **E**ducational **T**ool (Clean CaDET) to help developers discover, understand, and resolve harmful code smells. Clean CaDET tackles this problem from two angles. The Smell Detector module presents a framework for integrating AI-based code quality assessment algorithms to identify low-quality code as the programmer is writing it. The Smart Tutor module hosts a catalog of educational content that helps the programmer understand the identified issue and suggests possible solutions. By combining the quality assessment with the educational aspect, our integrated solution presents a novel approach for increasing the quality of code produced by our industry and lowering the overall cost of software development.

Section 2 presents related work for code smell detection and educational tools that are already presented and used in software engineering. Section 3 presents a description of the primary use case and Clean CaDET Platform components. We conclude our work in Section 4, where we present directions for future work.

II. RELATED WORK

We divide this section into two subsections. First, we examine several systematic literature reviews of code smell detection approaches that influenced our Smell Detector module's design. Then, we explore educational tools used for teaching code quality, from which we modeled our Smart Tutor module.

A. Code smell detection

The authors of [7] presented a systematic mapping study on code smell detection, examining studies published between 2000 and 2017. Their first question examined which code smells were researched the most. They found that Long Method, Feature Envy, and God Class have been detected in 65% of the studies. The next question asks what approaches have been proposed to detect Design Smells in software. They concluded that the most frequently used approaches were Metric-based, followed by Logical/Rule-Based and Machine Learning-Based.

The authors of [8] presented a systematic literature review on machine learning techniques for code smell detection, considering studies published between 2000 and

2017. One of the questions was which code smells were considered in the studies. They found that God Class is the most frequently researched code smell, followed by Long Method, Feature Envy, Functional Decomposition, and Spaghetti Code. The next question examines the used machine learning algorithms. Decision Tree is the most frequent algorithm, followed by Support Vector Machine. Other frequently used algorithms are Naive Bayes and Random Forest. The question which was also researched is related to independent variables used for code smell detection. Most studies use structural metrics for the detection of code smells.

The authors of [9] also presented a systematic literature review on code smell detection. They researched studies published between the years of 1999 and 2015. One of their questions examined which code smells were detected in the reviewed studies. The most detected code smells included Feature Envy, Data Class, Long Method, Large Class, and Long Parameters List. They also examined which detection techniques were used. They concluded that the most widely used techniques were Search-based, while Metric-based follows behind. Most techniques in the Search-based category apply machine learning algorithms.

[10] compared various code smell detection tools. They found a set of 84 different tools used in reviewed studies, but only 29 of them were available online for download. One of their research questions examined the most frequently detected code smells. Duplicated Code and Large Class were the major targets of detection tools. They also compared the detection tools to determine the agreement among them and computed their precision and recall. The results showed that the evaluated tools provide high agreement regarding detecting code smells. Regarding the recall and precision, the results show a poor recall percentage, while the precision results vary depending on the tool.

The authors of [11] explored the impact of historical software version changes in predicting the future maintainability of the software. In the experiment, 40 real-life open-sourced Java-based software components published in the Maven Repository were included, for which the history of 19 successive releases was available. The results of the data analysis show that the gain of software maintainability prediction performance was the greatest when the prediction model was enriched by software metric measurements of versions closer to the latest release.

The listed literature reviews identified three important design considerations for our Smell Detector:

1. There are many types of code smells, where most have unique traits. The Smell Detector should have an abstract representation of a code issue and be extensible to support new types of code smells.
2. Smell detection algorithms vary as well and include different machine learning models and rule engines. The Smell Detector should have an abstract representation of a smell detection algorithm and be extensible to support new types of algorithms.
3. While some detection algorithms work on plain source code, most require some code processing to derive the abstract syntax tree or structural metrics. Another module is required to process the code and calculate the necessary metrics for the Smell Detector.

B. Educational tools

A program developed by [12] reads code and sends it to the rule engine whose purpose is to analyze the syntax tree to identify an opportunity for code refactoring. It has three types of interaction:

- Automatic refactoring - Applying the refactoring method automatically right after user confirmation.
- Refactoring proposal – Display detailed instructions explaining to the user where and how the particular refactoring method should be applied.
- Refactoring Questionnaire - Asking the user additional questions to clarify the conditions and define the appropriate refactoring method.

In [13], the authors present an Automated assessment management system - ArTEMiS for interactive learning. This system tests solutions to programming exercises automatically and provides instant feedback for students to solve the exercise iteratively. It offers an online code editor with interactive exercise instructions and is programming languages independent. They evaluated their solution on four courses and figured out that ArTEMiS is suitable for beginners and can handle 200 submissions per minute while providing feedback within 10 seconds, and therefore helps students realize their progress and gradually improve their solutions. It reduces the effort of instructors and enhances the learning experience of students.

In [14], the authors gave three smelly code snippets to 30 professors to monitor how those professors would help students refactor the code. They collected the hints which the professors gave to students. They also compared teachers' hints to professional tools' reports and concluded that tools do not give feedback with increasing detail as teachers would. Because tools do not know what the code they analyze should do, they do not point out issues related to control flow and algorithmic optimizations. Also, because of the tools' default high threshold, minor issues are usually not reported.

Educational tools were developed for learning about other code quality aspects, such as security. In our previous work [15], we examined how e-learning tools that rely on gamification effectively teach students how to write more secure code.

Finally, adaptive e-learning systems have been the focus of researchers in recent years [16][17]. Such systems consider the student's learning style to offer educational content and learning objects best suited for the student's cognition.

Based on the surveyed literature, we define the following design considerations for our Smart Tutor module:

1. Getting timely feedback on the progress of an assignment is an effective learning strategy. The Smart Tutor should examine issues discovered by the Smell Detector and offer suitable educational content, based on some recommender subsystem.
2. Gamification facilitates student engagement and learning. The Smart Tutor's Learner Model should integrate a gamified subsystem that tracks the student progress, offers challenges, and facilitates competition through leaderboards.
3. Adaptive e-learning enables a scalable personalized approach to teaching. The Smart Tutor requires a learner model to track the learning styles of the student, as well as an instructional model that maps the learning objects to these learning styles and effective learning strategies.

III. CLEAN CADET PLATFORM

Clean CaDET is an artificial intelligence (AI) digital assistant that serves the developer and helps them write higher quality code. It analyzes the developer's code in search of quality issues. For each discovered issue, it selects personalized educational materials that help the programmer understand the identified issues and guide them to write higher quality code.

The Platform[1], or back-end part of the system, consists of three high-level modules:

- **The Repository Compiler**: processes the code and compiles it into an abstract, language-agnostic representation called the CaDET Model which contains calculated metrics and features.
- **The Smell Detector**: AI-based module analyses and processes the CaDET Model to determine the presence of code smells.
- **The Smart Tutor**: AI-based module maps the detected issues to suitable educational content based on the developer's experience with the Platform.

The Platform supports several use cases. The primary use case includes code processing, detecting quality issues in the code, and selecting the educational materials for improving the code quality, as highlighted in Figure 1. The developer writes the code within the Integrated Development Environment (IDE) (1). Currently supported IDEs are Visual Studio and Visual Studio Code. The Clean CaDET Plugin sends the written source code to the quality analysis upon request (2). Within the Repository Compiler, the source code is transformed into a CaDET model and sent to the Smell Detector (3). The Smell Detector processes the CaDET model and detects quality issues, sending it to the Smart Tutor (4). Finally, the Smart Tutor selects educational materials for detected issues and sends them back to Clean CaDET Plugin (5). The developer receives educational materials from which he learns how to write better quality code (6).

A. *The Repository Compiler*

Figure 2. represents the Repository Compiler components. This module receives code from the CaDET Plugin within the Code Repository Service component, which forwards it to the Code Model Factory (1a). Within the Code Model Factory component, the source code is processed and transformed into an abstract representation called the CaDET Model. The Platform's current implementation allows the conversion of C# source code into abstract representation but is designed to be extensible to support similar languages, such as Java.

In addition to transforming the source code into a CaDET Model, the Code Model Factory sends the abstract model to the Feature Extractors component (2), which calculates the features such as code metrics. Since the metrics in our approach can be easily added or removed, details of implementation are not given in the paper. Currently implemented metrics can be found in the Platform repository[1]. AI Models within the Smell Detector

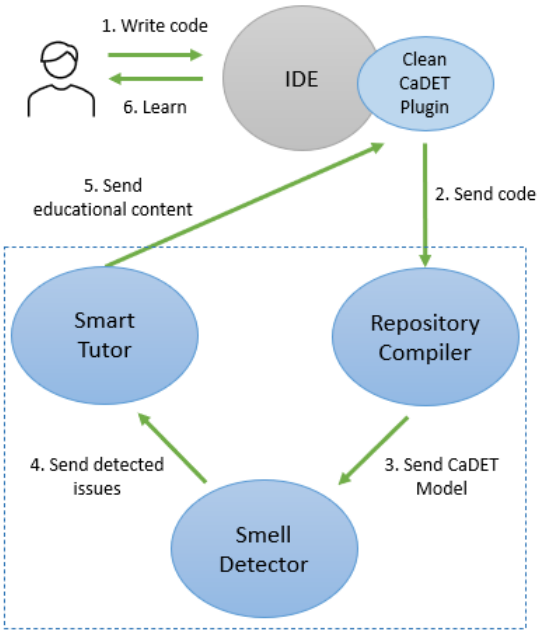

Figure 1. Interaction diagram for the primary use case

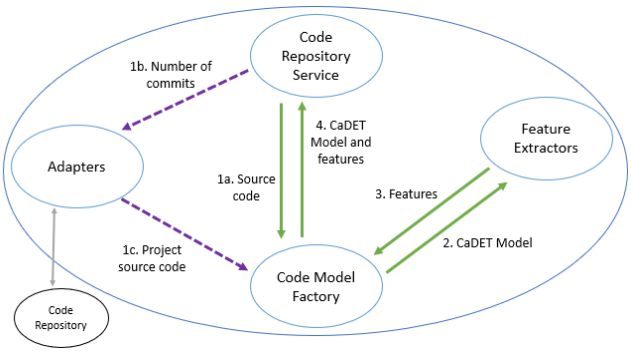

Figure 2. The Repository Compiler components

[1] The source code of the Platform is available on the https://github.com/Clean-CaDET/platform.

module can be trained using these features, and could detect code smells based on code metrics [7][8], as well as the text used in the code [9], variable names, and other features. The Feature Extractors component sends the calculated features to the Code Model Factory (3), which forwards them and the abstract model to the Code Repository Service (4).

For certain types of code smells, it may be essential to look at the history of code changes [18] (e.g., if a class changes very often, it is a signal that the class has too many responsibilities and represents the God class). In this case, the Platform needs historical data (commits) from the Version Control Systems (VCS). The Code Repository Service sends to the Adapters the length of history required to analyze the code (number of commits) (1b). The Adapters component interacts with the VCS repository, which delivers the project's source code from a specific commit. For each historical version of the project, the source code is sent to the Code Model Factory (1c), following the previously described flow.

The class diagram in Figure 3. shows the CaDET Model. The CaDETModel class refers to one software project and encapsulates it throughout its history. CaDETProject represents a snapshot of a project. Each project must specify the programming language it is written in and has a list of classes. Each object of the CaDETClass contains attributes such as class name, source code, and metrics. Besides, we have information about its parent and inner classes, members, fields, and modifiers. Each object of the CaDETMember class is described by name, source code, and metrics, as well as a list of parameters, fields, modifiers, accessed accessors and invoked methods. CaDETClassMetrics and CaDETMemberMetrics represent code metrics that are calculated for classes and members. The solution's design is flexible, and it is possible to make changes that will extend the model (e.g., adding new code metrics).

B. *The Smell Detector*

The Smell Detector module receives the CaDET Model from the Repository Compiler. This module contains a collection of detectors, i.e., algorithms from the artificial intelligence domain whose task is to process the CaDET Model. By processing code metrics and other model features, the Smell Detector finds code smells that diminish the code quality.

During the Smell Detector design, the focus was on the expandability and variability of the module. We defined an IDetector interface that models the detection strategy, which is then implemented by the concrete Detector classes. These specific detectors' implementations will take the appropriate data from the Repository Compiler and pass it to the AI models. This way, it is possible to plug in multiple code smell detectors. Currently implemented detectors can be found in the Platform repository[2]. Specific types of detectors are not given in the paper because, like metrics, they represent a dynamic part of the system and we will expand them over time.

Also, this leaves space for defining hybrid detectors that will combine several different models. Hybrid detectors can be useful when detecting several different types of code smells. Due to the lack of labeled data, ML-based approaches do not consider various code smell types. For this reason, a combination of ML-based and heuristic-based strategies (which apply predefined thresholds to calculated metric values) could cover a broader scope of code smell types and overcome the problem of limited training data.

After the code quality assessment, i.e., the detection of code smells, the Smell Detector generates a report on the detected code smells. The final report consists of several partial reports created by individual detectors. Each detector creates a report that includes information about all the code smells it has identified.

C. *The Smart Tutor*

After the Smell Detector's analysis, the main task of Smart Tutor is to select the appropriate educational content for the detected quality issues. Selected educational materials are sent to the Clean CaDET Plugin and displayed to the user.

The Smart Tutor follows the design of an adaptive e-

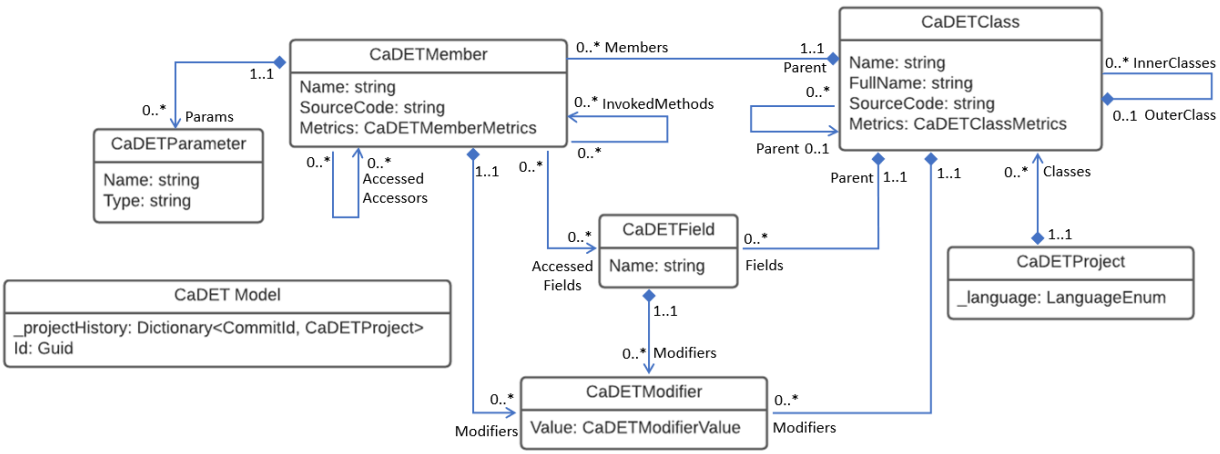

Figure 3. CaDET Model

[2] The source code of the Platform is available on the https://github.com/Clean-CaDET/platform.

learning system [16][17] and consists of three components, represented in Figure 4:

- The Learner Model
- The Instructional Model
- The Content Model

1) The Content Model

The Content Model houses domain-related knowledge organized in Knowledge Nodes. Each Knowledge Node contains Learning Objects sequenced in a specific order. Learning objects are small, reusable components that facilitate learning of the domain concept. Each is defined by a type (e.g., text, videos), the role it has for the subject matter (e.g., definition, example, formula).

2) The Learner Model

The Learner Model represents the learner's personal knowledge and progress in learning. It may also include other characteristics of the learner to adapt the educational content served to him. To offer personalized instructions, Smart Tutor will perform assessment measures that determine essential aspects of a learner.

There are two aspects to be assessed:

1. Personal traits: refers to information about the learner, including learning style [19] (e.g., whether the learner better masters the knowledge presented by text or images), cognitive abilities, and personality. This aspect allows the system to select the optimal educational content for the learner. To identify personal traits, the Smart Tutor will use a pretest (in the form of a questionnaire that the learner fills out) and the computer-based tracking of the learner's behavior.
2. Current knowledge: this information is important for the initialization of the Learner Model in terms of the content offered to the learner (e.g., educational content would focus on those areas where the learner has gaps in knowledge). The Smart Tutor will assess the current knowledge through tests and monitoring of the lessons the learner has completed.

3) The Instructional Model

The Instructional Model manages the presentation and selection of educational materials. As in the case of the Smell Detector component, the Instructional Model is designed to be pluggable. We defined an IRecommender interface that models the recommender strategy, which is then implemented by the concrete Recommender classes (e.g., knowledge-based, collaborative-based, or other types of RS, as well as hybrid solutions).

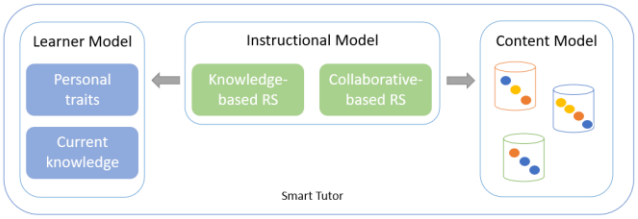

Figure 4. The Smart Tutor components

At the time of writing, the Instructional Model has a hybrid educational recommender system (RS) consisting of a knowledge-based and collaborative-based component.

Knowledge-based RS combines the learner model with the content model and defines rules for supplying the most appropriate learning objects, following effective learning strategies from [20]. Through the knowledge-based RS, educators introduce their expertise to the Smart Tutor and guide this module to construct effective knowledge nodes.

Collaborative-based RS relies on learner's feedback. During the use of the Platform, students will be able to give feedback on delivered educational materials. As more feedback gets collected, our knowledge-based component shall be supplemented with collaborative-based filtering, thus enabling our hybrid approach. This way, it will be possible to evaluate the educational RS and continuously improve it following the collected feedback.

IV. Conclusion

Clean CaDET is a multifaceted AI-powered solution that tackles the problem of software maintainability and code readability. On the one hand, its Smell Detector analyzes code and discovers readability issues and code smells. On the other hand, it offers educational content for training the developer to resolve the current problem and avoid introducing such issues in further development.

We designed the solution with extensibility in mind. Its current infrastructure enables the relatively simple introduction of new:

- Algorithms for smell detection, Smell Detector's IDetector interface,
- Educational content, through the Smart Tutor's data model,
- Instructional models, through the Smart Tutor's IRecommender interface,
- Metrics and features, through the Repository Compiler's data model,
- Language support, through the Repository Compiler's Code Model Factory.

The last point, while possible, is the most difficult. Each language rests on different conventions and best practices and suffers from different forms of code smells. A future research direction is to map a language such as Python or C++ to the abstract CaDETModel while preserving some of the smell detectors' capabilities trained on the CaDETModel made from C# and Java.

As an educational tool, we plan to use Clean CaDET in the classroom as part of our further work. This research direction can explore effective learning strategies in the context of clean code and good software design. We will expand the Smart Tutor to examine how collaboration, gamification, and interactive media can facilitate active learning at scale [21] in this context.

Finally, clean code analysis and learning object recommendation are stable problems that support ongoing research, where Clean CaDET acts as a suitable platform for algorithm development and experimentation.

ACKNOWLEDGMENT

This research was supported by the Science Fund of the Republic of Serbia, Grant No 6521051, AI-Clean CaDET.

REFERENCES

[1] Martin, Robert C. Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009.

[2] Fowler, Martin. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.

[3] Sharma, Tushar, and Diomidis Spinellis. "A survey on software smells." Journal of Systems and Software 138 (2018): 158-173.

[4] Kaur, Amandeep. "A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes." Archives of Computational Methods in Engineering 27.4 (2020): 1267-1296.

[5] Hozano, Mário, et al. "Are you smelling it? Investigating how similar developers detect code smells." Information and Software Technology 93 (2018): 130-146.

[6] Palomba, Fabio, et al. "Do they really smell bad? a study on developers' perception of bad code smells." 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, 2014.

[7] Alkharabsheh, Khalid, et al. "Software Design Smell Detection: a systematic mapping study." Software Quality Journal 27.3 (2019): 1069-1148.

[8] Azeem, Muhammad Ilyas, et al. "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis." Information and Software Technology 108 (2019): 115-138.

[9] Rasool, Ghulam, and Zeeshan Arshad. "A review of code smell mining techniques." Journal of Software: Evolution and Process 27.11 (2015): 867-895.

[10] Fernandes, Eduardo, et al. "A review-based comparative study of bad smell detection tools." Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering. 2016.

[11] Mitja Gradišnik, Tina Beranič and Sašo Karakatič. "Impact of Historical Software Metric Changes in Predicting Future Maintainability Trends in Open-Source Software Development" Faculty of Electrical Engineering and Computer Science, University of Maribor. 2020.

[12] Sandalski, Mincho, et al. "Development of a refactoring learning environment." Cybernetics and Information Technologies (CIT) 11.2 (2011).

[13] Krusche, Stephan, and Andreas Seitz. "Artemis: An automatic assessment management system for interactive learning." Proceedings of the 49th ACM Technical Symposium on Computer Science Education. 2018.

[14] Keuning, Hieke, Bastiaan Heeren, and Johan Jeuring. "How teachers would help students to improve their code." Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education. 2019.

[15] Luburić, Nikola, Goran Sladić, and Branko Milosavljević. "Utilizing a vulnerable software package to teach software security design analysis." 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE, 2019.

[16] Truong, Huong May. "Integrating learning styles and adaptive e-learning system: Current developments, problems and opportunities." Computers in human behavior 55 (2016): 1185-1193.

[17] Normadhi, Nur Baiti Afini, et al. "Identification of personal traits in adaptive learning environment: Systematic literature review." Computers & Education 130 (2019): 168-190.

[18] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D. and De Lucia, A., 2014. Mining version histories for detecting code smells. IEEE Transactions on Software Engineering, 41(5), pp.462-489.

[19] Felder, R.M. and Silverman, L.K., 1988. Learning and teaching styles in engineering education. Engineering education, 78(7), pp.674-681.

[20] Brown, P.C., Roediger III, H.L. and McDaniel, M.A., 2014. Make it stick. Harvard University Press.

[21] Davis, D., Chen, G., Hauff, C. and Houben, G.J., 2018. Activating learning at scale: A review of innovations in online learning strategies. Computers & Education, 125, pp.327-344.