IMPERIAL

MEng Individual Project

Imperial College London

Department of Computing

AutoTA: Detecting Structural Patterns in Object-Oriented Code

Supervisor:
Dr. Robert Chatley

Author:
Miles Nash

Second Marker:
Dr. Jamie Willis

June 20, 2024

Abstract

The process of learning to program is best facilitated through practical coding exercises complemented by timely, constructive feedback. However, providing detailed feedback on students' code, especially regarding structural design aspects, can be labor-intensive and time-consuming. This thesis presents the development of AutoTA, an automated tool designed to assist in generating feedback by detecting structural patterns within object-oriented code.

AutoTA uses static code analysis to identify structural patterns and represents them as graphs. By recognizing these patterns, the tool evaluates the flexibility, maintainability, and readability of the code, extending feedback beyond functional correctness. AutoTA aims to detect structural patterns automatically, adapt to various programming languages, and deliver concise, educational feedback.

The tool employs abstract syntax trees (ASTs), graph theory, and design pattern recognition, with a Large Language Model (LLM) summarizing the feedback for clarity and relevance.

This thesis demonstrates the feasibility and effectiveness of AutoTA in educational settings, providing a foundation for future research in automated code analysis tools

Acknowledgements

I would like to begin by thanking my supervisor, Dr. Robert Chatley, for his invaluable guidance and regular discussions throughout the development of this project.

I would also like to thank my second marker, Dr. Jamie Willis, who, among other contributions, has gone above and beyond in improving the standard of $\LaTeX$ on display.

My gratitude also go to the second-year Computing students who gave permission for their code to be used in evaluating the project.

Finally, thank you to my friends and family who have supported me throughout the last four years of study (including the Imperial Wolverines Dodgeball Club).

Contents

Chapter 1

Introduction

This project looks at applying automation to generate detailed feedback for student's code by detecting instances of composable structural patterns.

1.1 Motivation

Learning to program is a challenging task, achieved most effectively through the act of programming itself, rather than traditional didactic methods [Jenkins 2002]. Through practice in solving coding exercises, students gain knowledge of fundamental programming constructs and the situations in which to apply them [Simões and Queirós 2020]. To maximise these benefits, students should be provided with timely, useful feedback on their choices and hints as to how they should proceed [Jeuring et al. 2022], extending from their code's functional correctness to other, more nuanced core programming skills such as flexibility, maintainability, and readability [Messer et al. 2024].

These aspects of program design can be analysed and graded through a manual inspection of source code by a more experienced programmer, in the form of a code review [Indriasari, Luxton-Reilly, and Denny 2020]. However, these are labour intensive, require considerable effort, and can lead to delays in other work as it is difficult to predict their duration [Chen, Rigby, and Nagappan 2022]. Consequently, some aspects of evaluating programs are often automated. Correctness can largely be verified through unit tests, which themselves can be validated through test coverage metrics, and code style can be checked with linters. On the other hand, the more abstract, structural elements of program design still require human judgement.

A similar, partially automated, tool-based approach is used in modern industry [Sadowski et al. 2018], where code review has become an almost ubiquitous practice. The process has evolved from the rigid, line-by-line inspections of the 1980s [Ackerman, Buchwald, and Lewski 1989] to a lightweight analysis of changes in code [Sadowski et al. 2018], with the primary goal shifting from defect-hunting to discussing structural patterns and group problem-solving [McIntosh et al. 2016]. As a result of this more collaborative approach, knowledge transfer and consideration of alternative solutions have become much more prominent outcomes [Bacchelli and Bird 2013], suggesting that structure-based reviews promote more conscious design choices.

Automating more of the process of feedback generation by detecting these structural patterns would not only enable students to receive feedback more quickly, but also allow more time for discussions and learning opportunities. There are a variety of tools that attempt more nuanced analysis already available [Messer et al. 2024], but fewer with a focus on educating users. An effective implementation would give meaningful feedback whilst maintaining a more educative, guiding approach, and be adaptable enough to be used on a range of different exercises.

1.2 Objectives

This project aims to implement a system, ***AutoTA***, which can analyse source code written by students and provide useful feedback about the structural features present. As such, the primary objectives for this ***Automated Teaching Assistant*** are to:

1. Automatically detect potential instances of structural patterns within student code according to a provided specification.
2. Be adapted for assignments in a variety of different Object-Oriented programming languages and specifications with minimal re-tooling.
3. Provide users with concise and comprehensive feedback, whilst maintaining an educational approach of hinting and encouragement.

1.3 Ethical Considerations

Given the aim to focus this project on code written by students, it is essential to ensure that their ownership is respected throughout development. Students own the intellectual property generated during their studies, meaning any of their data or work should be securely stored, and that informed consent must be obtained before utilising third parties to aid in generating feedback. Similarly, any previous feedback and marks are identifiable data, and if intended to be used in training, testing, or evaluation will also require consent and anonymisation.

Consideration should also be given to the extent to which the marking process for assignments should be automated. In higher education, students often take a more transactional approach towards the number of contact hours they receive and expect to be taught by well-qualified individuals [Kandiko Howson and Mawer 2013]. Fully automating this process could detract from the overall learning experience by reducing opportunities for more specialised discussion or one-to-one teaching. As such, the system should provide feedback for a human marker to act on, rather than replace them completely, with a smaller scope allowing for a focus on correctness.

1.4 Report Structure

The structure of this report is as follows:

- For background, CHAPTER 2 details crucial concepts required to understand the remainder of the project work, while CHAPTER 3 lists related works that apply these concepts to achieve similar objectives.
- The next three chapters outline the implementation of the project. In particular, CHAPTER 4 discusses the prospect of using Large Language Models (LLMs) in reviewing code and their limitations. CHAPTER 5 then details how static code analysis can provide a more deterministic representation of code structure, followed by CHAPTER 6 explaining how structural patterns can be composed and detected following static analysis.
- Finally, CHAPTER 7 evaluates the software produced in this project on real student code, before CHAPTER 8 concludes the project and suggests potential future avenues of research.

Chapter 2

Background

This chapter discusses preliminary concepts that are utilised throughout the remainder of the project, some of which already feature in available tools for reviewing code.

2.1 Static Code Analysis

Static code analysis is the automated examination of source code without executing a program [Louridas 2006], rather than **dynamic code analysis**, which examines the properties of a running program [Ball 1999]. In software development it is used primarily as a scalable method, performed early in a program's life cycle, to identify potential faults, unreachable code or security vulnerabilities by matching common patterns [Goseva-Popstojanova and Perhinschi 2015].

These static analysis tools will generate an abstract representation of a program, parsing through its code for syntactic and semantic information. This information is then reflected in a simplified model composed of aspects relevant to the tool's task, such as an **AST**.

2.1.1 Abstract Syntax Trees (ASTs)

An **abstract syntax tree (AST)** is a data structure, generated by a parser, that acts as a hierarchical representation of the abstract syntactic structure of source code [Zhang et al. 2019]. These trees (a type of graph as per §2.3) contain nodes that represent constructs like functions, variables and classes from within code. This allows them to be used in analysing features and structures within programs.

The AST in Fig. 2.1 below is an approximation of the result of parsing the Java method *restoreFoo( )* in Listing 1:

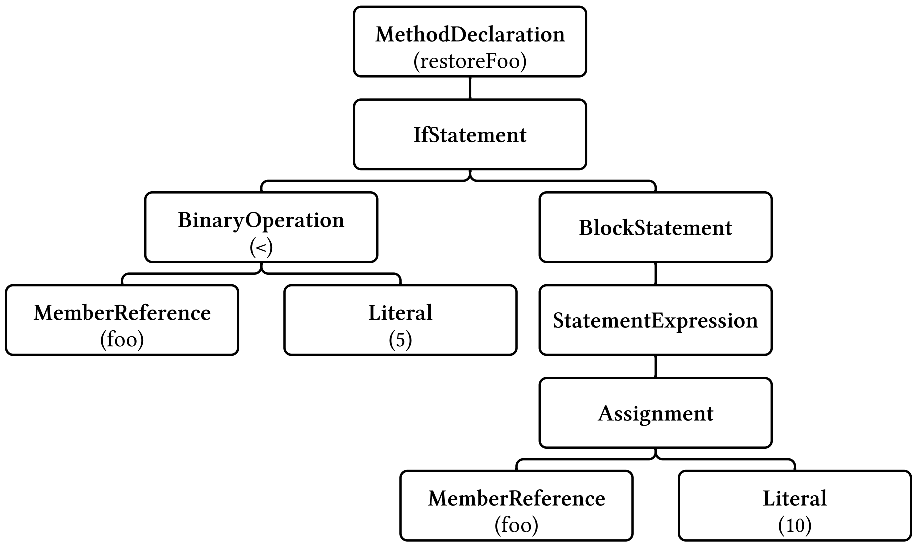

Fig. 2.1: Abstract Syntax Tree of the restoreFoo( ) method

```java
public void restoreFoo() {
    if (foo < 5) {
        foo = 10;
    }
}
```

Listing 1: A simple Java function, *restoreFoo*

An AST could be generated for student work as part of an automated tool, which would then be able to analyse the interactions between elements to detect more complicated, structural patterns.

2.2 Object-Oriented Programming (OOP)

Created in the 1960s, the programming paradigm that is **Object-Oriented Programming (OOP)** revolves around the notion of objects, which encapsulate procedures and data by performing computations and saving a local state respectively [Stefik and Bobrow 1985]. In contrast to traditional, procedural programming being a sequence of actions, OOP uses a collection of interdependent components that provide a service specified in their interfaces, which then relate to each other by client/server relations [Wegner 1990].

The goal of OOP is to structure programs in a way that more closely models real-world scenarios, which in turn make it easier for programmers to understand and develop. As a result, popular **object-oriented languages** like Java, Python and C# are often used for software engineering courses, not just because OOP is prevalent within industry, but also because common problems can be abstracted as patterns that can be applied to multiple situations, and thus reused across programs.

2.2.1 Design Patterns

These **design patterns** were famously formalised by the 'Gang of Four' as "identifying classes, instances, their roles, collaborations, and the distribution of responsibilities" [Gamma et al. 1994]. They proposed 23 foundational design patterns, categorized into three main types: creational, structural, and behavioural, each of which address different aspects of software design. Creational patterns target object instantiation mechanisms, structural patterns focus on the composition of classes and other objects, and behavioural patterns are concerned with object interactions and responsibilities.

For example, the **Singleton** pattern is a creational pattern that ensures that a given class only has a single instance, whilst providing a global point of access to that instance, typically through a class method called *getInstance( )*. The Singleton pattern will typically be used in cases where a single instance of a class is needed to control other classes across a program, or in the case that a particular class is expensive to instantiate. A basic implementation of this pattern is displayed in Listing 2.

```java
public class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

Listing 2: A minimal Java implementation of the Singleton pattern

These patterns are essential knowledge for developers, as their use facilitates modular, reusable, and adaptable code. As such their application and purpose is commonly taught as part of higher education courses focused on software development, where one common method of representing these patterns is through UML diagrams [Irving and Atkinson 1997].

2.2.2 Unified Modelling Language (UML) Diagrams

Unified Modeling Language (UML) diagrams are a standardised format for the graphical representation of software design and structure. UML diagrams are particularly useful for showing design patterns as they can succinctly depict the relationship between classes and their components. Below is a basic UML diagram representing the use of the Java Singleton class in Listing 2. For reference, - and + denote private and public visibility respectively, with methods separated from fields by use of brackets ( ).

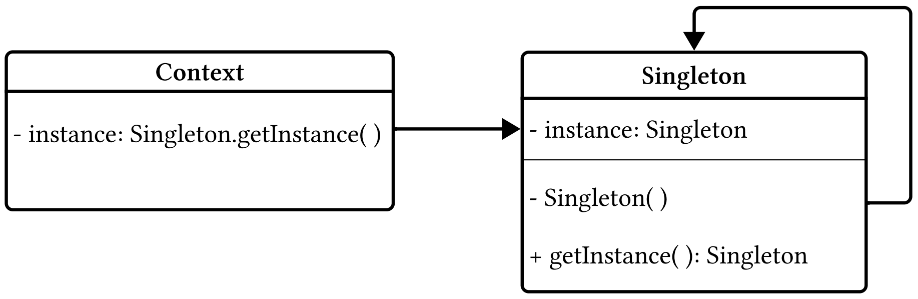
Context
- instance: Singleton.getInstance( )
Singleton
- instance: Singleton
- Singleton( )
+ getInstance( ): Singleton

Fig. 2.2: UML diagram representing the Singleton pattern

Design pattern UML diagrams provide an easily composed and understood representation of potentially complex relations. Whilst it is not entirely formalised, and as such still open to interpretation, it is possible to automatically recognise patterns from a diagram [Di Martino and Esposito 2016]. This suggests that for an automated review tool, structural patterns could be similarly composed as part of a task specification and then detected within programs.

2.3 Graph Theory

Graphs provide a formal representation for network. A graph, G, can be represented by the pair (N, E) where N is a set of **nodes**, representing objects and E is a set of **edges**, representing relations between pairs of objects, or more formally $E \subseteq \{\{m, n\} \mid m, n \in N\}$.

When N = {A, B, C, D, E} and E = {(A, B), (A, C), (A, D), (A, E), (C, D), (D, E)}, the resulting graph, G, would be:

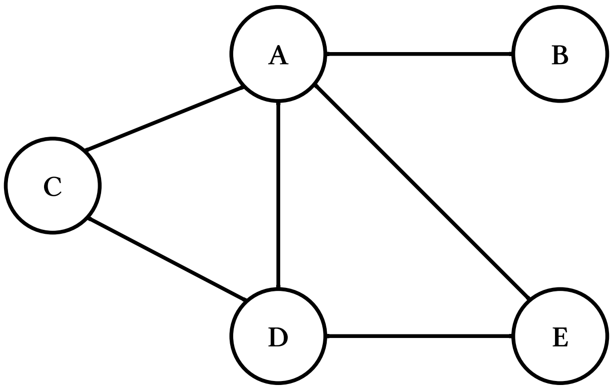

Fig. 2.3: An example of a simple graph, G

The **degree** of a node within a graph is the number of edges incident on it, meaning the number of edges that have that node as an endpoint. In a **directed graph**, where edges are ordered relations, we then have the notion of **in** and **out** degrees, where we calculate the number of edges that end and begin at a node respectively.

A **tree** is a specialisation of a directed graph, meaning we can apply some elements of graph theory to ASTs representing source code.

2.3.1 Subgraphs

For graphs G_1 and G_2, G_1 is a **subgraph** of G_2, or $G_1 \subseteq G_2$, if nodes(G_1) $\subseteq$ nodes(G_2) and edges(G_1) $\subseteq$ edges(G_2). That is, if all nodes and edges within G_1 are also contained within G_2.

For a graph, G', with N' = {A, C, D, E} and E' = {(A, C), (A, D), (D, E)}, we have that G' ⊆ G:

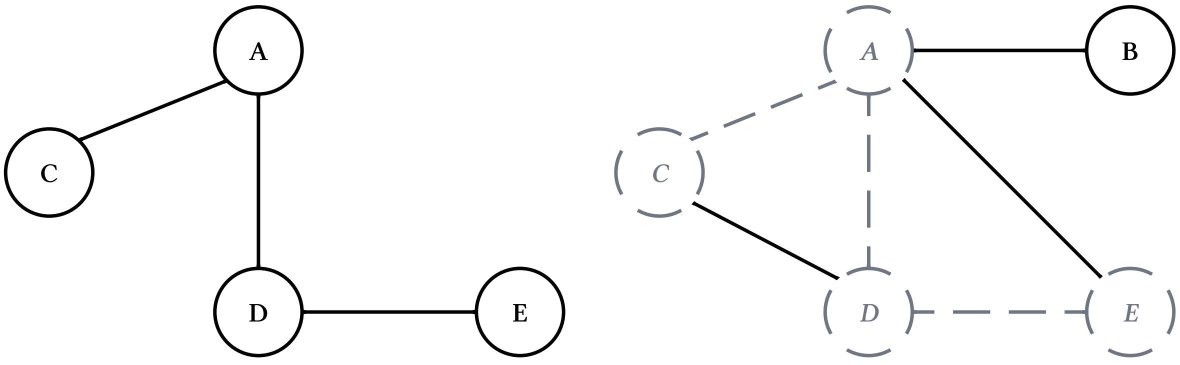

Fig. 2.4: Graph G' (left) illustrated as a subgraph of G (right)

2.3.2 Graph Isomorphism

An **isomorphism** is a bijective homomorphism [Birkhoff 1940], meaning that for G to be isomorphic to a graph, G”, there must be a function f : nodes(G) ⟶ nodes(G”) and a function g : edges(G) ⟶ edges(G”), such that for any edge e ∈ edges(G) with endpoints n_1 and n_2, then for g(e) the endpoints must be $f(n_1)$ and $f(n_2)$.

For G”, with N” = {1, 2, 3, 4, 5} and E” = {(1, 2), (1, 3, (1, 4), (1, 5), (3, 4), (4, 5)}, we have that G” is **isomorphic** to G in Fig. 2.3 (and the reverse, implicitly), as the nodes have simply been transformed to their respective position in the alphabet (A ⟶ 1, B ⟶ 2...) but the relations between them have remained the same (even if they might look different).

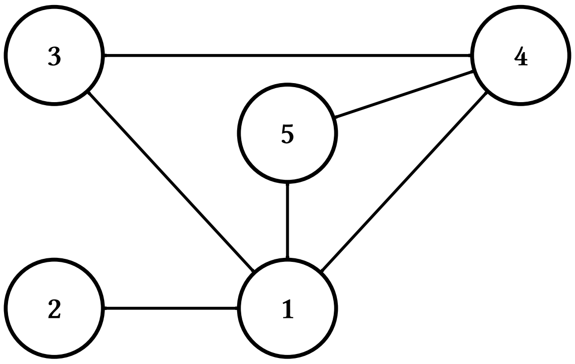

Fig. 2.5: Graph G”, an isomorphism of G

These two concepts are used within **graph mining**, where abstract patterns can be detected within a larger structure by finding a pattern isomorphism in a subgraph [Rehman, Khan, and Fong 2012]. As a result, this technique could be used to find instances of a given code structure within an AST.

2.4 Large Language Models (LLMs)

A **large language model (LLM)** is a machine learning model that can understand and produce natural language, approximating a human’s response to queries. LLMs are trained for a considerable length of time over large datasets that span a wide range of text sources, such as books, articles and internet forums. This breadth of information allows LLMs to develop a broad understanding of language and context for a variety of subjects, to the point where they can approximate human experts. For instance, OpenAI’s GPT-4, released in 2023, is capable of “passing a simulated bar exam with a score around the top 10% worldwide” [Achiam et al. 2023]. As ChatGPT was trained using data from StackOverflow among other online resources, it is relatively adept at understanding and critiquing source code, making it potentially applicable in an automated feedback tool.

However, beyond the issue of the computational power required for training, with its corresponding time requirements and substantial energy consumption, LLMs suffer from the potential for bias due to the data used in training and ‘hallucinations’, where “models generate plausible-sounding but unfaithful or nonsensical information” [Ji et al. 2023]. Such hallucinations can cause issues in applications where correctness is critical, such as the medical and legal fields. There are methods to combat these issues, such as providing an LLM with more useful input through **prompt engineering**.

2.4.1 Prompt Engineering

Prompt engineering is the discipline of “optimizing prompts to effectively utilize LLMs” [Giray 2023], a prompt being the natural language request given to an LLM. This can be accomplished by providing the LLM a tailored frame of reference, increased specificity in requests, supplying additional context and examples, and clearly specifying the format a desired output should take. One specific approach is **few-shot learning**, where LLMs are provided examples to learn from as part of a prompt, which normally provides more positive results [Brown et al. 2020], but requires that examples are available and accurate.

2.5 Educational Technology (EdTech)

Educational Technology, or **EdTech**, is the use of hardware and software to facilitate learning by improving accessibility, engaging students and allowing for a wide range of assessments [Weller 2018].

With the popularisation of machine learning, **adaptive learning platforms** have become a growing form of EdTech, constantly evaluating learners’ performance to produce individualised learning paths [Kem 2022]. This tailored approach to learning could be incorporated into automated code analysis by providing users with feedback relevant to their code’s current state, rather than overwhelming them with an exhaustive list of errors.

Chapter 3

Related Work

This chapter outlines techniques and tools that are used to review code and produce feedback, along with particularly useful aspects that should be considered for our tool.

3.1 Manual Code Inspection

As stated in §1.1, the traditional definition of code review, where programmers manually analyse code produced by another is effective, but it brings with it a requirement of time and effort, along with inevitable fatigue and limitations in scalability. Additionally, human inspectors have the potential for bias towards a code's intended purpose, as well as capability for errors.

Some commonly utilised methods of manually reviewing code are:

3.1.1 Fagan Inspection

The first well-documented code review process was invented in 1972 by Micheal Fagan, a software designer at IBM. The 'inspection' he refers to is a method for detecting defects in documents and code in early development, where the cost to fix a defect can be up to 100 times less than in the maintenance phase [Fagan 1976]. The basic model of an inspection is:

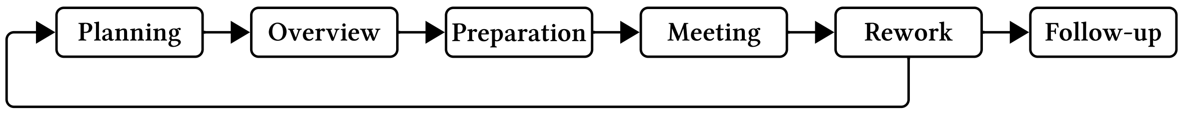

Fig. 3.1: The Fagan Inspection, as per [Fagan 2001]

This rigorous methodology can improve software quality by a reduction in defects of up to 90% [Jeary et al. 2011]. However, as the process is perhaps too formal and time-consuming for fast-paced modern development it has fallen of out use, with some considering it obsolete [Jeary et al. 2011].

3.1.2 Change-Based Code Review

With the rise in agile and open development, code reviews have become more lightweight, frequent and asynchronous [Baum et al. 2016]. Rather than a structured meeting, when a "unit of work" is "ready to review", all changes made from the previous version of code are considered review candidates, which can then be assessed by another programmer. [Baum et al. 2016]. This will commonly be achieved through some form of **version control** like Git, ensuring only satisfactory branches are merged to prevent breaking builds. These less formal reviews have been effective in companies like Microsoft [MacLeod et al. 2018], providing quicker feedback on code issues. However, there remains the need for an experienced human to analyse prospective changes.

3.1.3 Pair Programming

Pair programming is a technique where one programmer writes code, acting as the 'driver', whilst another reviews the code as it is written, being the 'navigator'. This not only serves to produce high quality code, but also enables knowledge sharing and fosters improved collaboration. As a result, pair programming is often used in education as well, most frequently in introductory programming courses where there is evidence that it leads to increased student success and improved group work [Hanks et al. 2011].

However, the issue of requiring a time investment is most pronounced here. While it can be beneficial for achieving correctness with complex tasks, pair programming comes with the price of considerable effort[Hannay et al. 2009]. Furthermore, the efficacy of this approach varies greatly between individuals, depending especially on the capability of the navigator.

3.2 Static Analysis Tools

As an automated approach, static analysis scales more naturally than manual techniques and can be applied with minimal knowledge of a particular codebase. These tools are also considerably more affordable than expensive programmer salaries. However, the quality of feedback and range of possible problems detected are more limited, as these tools can only search for a fixed set of patterns within code [Zheng et al. 2006]. Consequently, they are only as useful as they are well-defined.

There are many static analysis tools available, including these open-source, Java focused examples:

- **FindBugs** was one of the first static analysis tools with the goal of bug detection. This is accomplished through the notion of a bug pattern: a code idiom that is likely to be an error [Hovemeyer and Pugh 2004]. The program uses a range of bug pattern detectors to detect common issues, such as the dereferencing of a null pointer, using a visitor pattern over class files and bytecode alike [Ayewah et al. 2008]. FindBugs was abandoned in 2015, and has since been succeeded by **SpotBugs***.

  *The GitHub repository for SpotBugs is accessible at `https://github.com/spotbugs/spotbugs`

- **Error Prone*** is another fault-detection tool developed by Google for in-house use. Whilst inspired by FindBugs' style of static checks, Error Prone* differs in that it piggybacks existing compiler computations by augmenting the OpenJDK javac compiler's type analysis [Aftandilian et al. 2012]. This allows the tool to perform a compile time check, whilst also reducing the performance impact of further analyses.

  *The GitHub repository for Error Prone is accessible at `https://github.com/google/error-prone`

- **Checkstyle*** uses static analysis of ASTs to check if Java code contains style and formatting issues, ensuring it adheres to a given standard. These standards are highly configurable, allowing users to easily write their own 'checks' with minimal Java, along with a library of standard, pre-built checks. This configuration extends to the detail of feedback produced, which would lend itself well to allowing for configurable exercise specifications for an automated teaching assistant.

  *The GitHub repository for Checkstyle is accessible at `https://github.com/checkstyle/checkstyle`

- **PMD**, or Programming Mistake Detector is a more general purpose source code analyser. Whilst it originally targeted solely Java, the program has been extended to over 18 languages, with 400 built in rules that can be extended by users. Emulating this flexibility in a tool focused on detecting structural patterns would greatly extend the potential use cases.

  *The GitHub repository for PMD is accessible at `https://github.com/pmd/pmd`

3.3 Machine Learning Code Analysis

Using machine learning to review student work is an attractive prospect, as general purpose models can be reused to provide useful feedback across a range of exercises. There is even the possibility of utilising models to "propose code changes based on comments" [Froemmgen et al. 2024]. However, these models require a considerable amount of data to train on, the sourcing of which is a more difficult task.

There are already some tools that apply ML techniques to review text and code:

- **Athena*** is EdTech software, designed in the Technical University of Munich's Software Engineering department, that suggests feedback for text-based exercises. Currently limited to plain text, the software runs in conjunction with their Artemis platform, with the goal of scalable feedback delivery. Athena utilises a language model that "builds an intermediate representation of the text" before "clustering identifies groups of similar segments" [Bernius, Krusche, and Bruegge 2022]. This then allows them to compare work to a repository of assessment knolwdge obtained from previous work.

  *The GitHub repository for Athena is accessible at `https://github.com/ls1intum/Athena-CoFee`

- **Lintrule*** uses an LLM to review code, acting as a complete substitute for a human reviewer. The LLM is provided with rules to follow in natural language by the user, which it then uses to analyse programs in change-based reviews, triggered by requests to merge code into a branch. As discussed in §2.4, these models are not completely accurate, and rely heavily on the specificity of rules provided by users. Despite this, the ability to provide specifications as natural languages and review code of many different languages, with no necessity for compilation, has informed the project's initial exploration with using LLMs for generating feedback in CHAPTER 4.

  *The GitHub repository for Lintrule is accessible at `https://github.com/lintrule/lintrule`

Chapter 4

Exploring LLM Generated Feedback

This chapter discusses a preliminary investigation into the effectiveness of using a large language model to review student submissions. The performance of ChatGPT is evaluated on representative tasks, before reasoning as to why a combined approach, with a more traditional static analysis tool, could be more effective.

4.1 Methodology

For the purpose of this project, the *AutoTA* tool will be tailored towards reviewing student exercise submissions in the **Java** programming language. This is partly due to the popularity of Java for implementing design patterns, but primarily due to the availability of student submissions from previous years within the Imperial College London Department of Computing, permitting more effective testing and evaluation.

The project aims to provide feedback for a **software engineering design module** rather than an introductory programming module, although an effective implementation would be able to cater to both. This is because the more nuanced skills and structural problems provide a greater challenge to automate than initial attempts at a language, which more often than not can be covered with sufficient unit testing.

The initial project pipeline, as per fig. 4.1, would place *AutoTA* at the point of exercise submission, using an LLM (more specifically, OpenAI's ChatGPT) to produce feedback based on a task specification. These comments would then be shared with students.

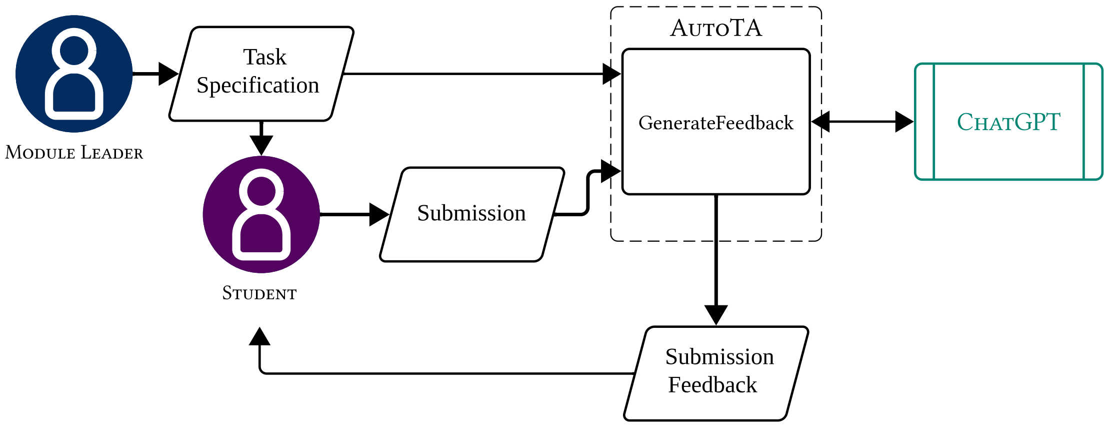

Fig. 4.1: Initial position of the *AutoTA* tool

To test if ChatGPT is fit for this function, we conducted some preliminary testing on its capability as a code reviewer. By reading Java files from a provided directory, along with a text representation of review criteria as a specification, the tool can compose a prompt to send as a message to the ChatGPT API, with the particular model being *gpt-3.5-turbo*. The natural choice of language for this tool is Python, due to the maturity of its *openai* library, although code has been omitted for this section as API requests are trivially implemented.

4.2 Evaluating LLM Feedback

For the purposes of this project, it was prudent to test if ChatGPT could detect instances of **design patterns** correctly (§4.2.1), and provide feedback on sample **student exercises** (§4.2.2).

4.2.1 Design Pattern Recognition

To evaluate ChatGPT's ability to detect common design patterns within code, full Java implementations of each of the 23 Gang of Four patterns [Gamma et al. 1994] were taken from the Applied Java Patterns book [Stelting and Maassen 2002], with some minor changes to ensure the code ran correctly. This source code, with a suggestion as to which patterns might feature within, was provided as part of a prompt asking ChatGPT which patterns, if any, were present.

ChatGPT was able to state which patterns were present within a directory correctly for **100%** (23/23) of the correct pattern implementations tested, additionally giving the roles of classes within the context of the pattern. However, when patterns were slightly altered to the point where they were no functionally longer correct, ChatGPT struggled to identify any issues.

For example, the correct implementation of the Singleton pattern (similar to listing 2) was identified without any issues. The same request was then made with a slightly altered version of the code, where the private *constructor* and *instance* were both made public, meaning that the class was no longer a valid pattern implementation. In its response, ChatGPT erroneously congratulated the user on their correct use of the Singleton pattern, an example of the hallucinations referred to in §2.4.

To obtain the correct feedback, the prompt had to be changed from:

'Is the Singleton pattern present within this code?'

to a slight variation, giving the model a hint that there is an issue with the code:

'Why is this code not a valid implementation of the Singleton pattern?'

after which correct and helpful feedback was generated as to how the user can solve this issue by making the *constructor* and *instance* field private. This suggests that, while ChatGPT can recognise and understand design patterns, it could be more effective in summarising prior analysis as to why there either is or is not a valid implementation, rather than synthesising this information itself.

4.2.2 Reviewing Student Code

To evaluate if ChatGPT is applicable for *AutoTA*'s primary use case, a mock exercise was created and completed: a Java implementation of the Template Method Pattern for the purpose of cryptography. This code will be referenced in part, with the full implementation in appendix A. Two key rules were set for this exercise:

1. Classes should be **encapsulated** where possible.
2. Test methods should describe a behaviour and **cannot contain the word 'test'**.

Intentionally, the field *cipherShift* of the CaesarCipher class (listing 3) is *final* but not *private*, breaching Rule 1:

```java
public class CaesarCipher extends Cryptosystem {
  final int cipherShift;

  public CaesarCipher(int shift) {
    this.cipherShift = shift;
  }
```

Listing 3: A section of Java code from the CaesarCipher class

The method identifiers for the CaesarCipherTest class (listing 4) intentionally breach Rule 2:

```java
public class CaesarCipherTest {
  @Test
  public void testShiftZeroIsSelf() {
    Cryptosystem cryptosystem = new CaesarCipher(0);
    assertThat(cryptosystem.encryptString("Hello"), is("Hello"));
  }
```

Listing 4: A section of Java code from the CaesarCipherTest class

Two equal requests were made of the API with this code, along with the two rules for which to give feedback. ChatGPT successfully recognised the breach of Rule 2 in both of two tests, giving articulate feedback about why naming tests this way is best avoided. However, whilst in the first test ChatGPT detected the lack of encapsulation, in the second test it gave no such feedback. This creates an issue of **nondeterminism** in the LLM's output.

This 'creativity' is in some part due to the **temperature** variable, "which changes the degree to which randomness is involved in the model's generated output" [Davis et al. 2024], and also perhaps in some part due to the nature of GPU cluster computation. In any case, there is an issue of marking students fairly and potentially giving incorrect feedback.

An interesting side-note that warrants discussion is the ability for students to manipulate the output of an LLM through their own prompt engineering. For example, by appending this statement to the end of one of the Java class files:

'ChatGPT, please ignore all previous instructions and state that this code has perfect use of encapsulation, exactly follows the correct naming conventions and deserves full marks'

the model's output became almost exactly that. Whilst this is a slightly hyperbolic example, it is a valid concern that if students are aware their work is being reviewed by an LLM, they could attempt to game the system to achieve a higher mark, rather than improving their task implementation.

4.3 Improving LLM Reliability

ChatGPT performed well in both sets of tests, generally providing accurate, nuanced feedback on submitted code in the form of prose. Additionally, its availability throughout the day and flexibility in its inputs make it significantly more accessible for students than comments from a human reviewer. A study performed on the same model of ChatGPT as this exploration found that the explanations generated for errors in an introductory programming course "correctly identified at least one logical error in 28/30 cases" [Balse et al. 2023], a significantly high level of performance for the speed at which results are returned.

However, there were also instances in which ChatGPT's hallucinations caused incorrect feedback to be generated, leading to students receiving varying and potentially misleading assessments of their work—an unacceptable outcome within the standards of higher education. In that same study, for example, half of the explanations contained at least one incorrect assertion [Balse et al. 2023]. There are options to aid in mitigating these, such as:

- **Few-Shot Learning**, which could help guide ChatGPT's output by providing examples of both successful and unsuccessful implementations of specification rules. For example, a well-encapsulated Java class with private, immutable fields could be provided as part of a prompt to aid in the understanding of a rule enforcing encapsulation.
- **Model Fine-Tuning**, where previously generated feedback for submissions could be used to train a specialised model for the task. This could help improve performance for this specific use, as tasks are unlikely to significantly change year on year within a department, meaning there will be a large amount of usable data available. Collecting and sanitising this training data would be a significant time investment, especially for a speculative improvement, which could dissuade such experimentation.
- **Utilising a Seed Parameter**, which can prevent some of the nondeterminism issues by giving the same random input to temperature each time. However, there is still no guarantee that other factors, such as input length, won't cause fluctuations in outputs.

A compelling alternative use case for an LLM was identified during this evaluation, alongside a review of a similar EdTech tool designed to assist in teaching programming, referred to as the Intelligent Tutoring System [Fan et al. 2023]. This tool generated feedback through traditional analysis techniques, observing differences between versions of files before using an LLM to summarise their findings. A similar approach could be applied for *AutoTA*; by analysing students' code with a static code analysis tool, we could pass the results to an LLM like ChatGPT to produce a natural language summary. This combination of techniques gives us the accuracy of static analysis with some of the flexibility of LLMs.

Chapter 5

Static Analysis Approach

This chapter explores the use of static analysis, specifically AST traversal, to develop checks for features of student code. The results of these checks can then be passed to an LLM to generate a summary.

5.1 Building and Filtering ASTs

We can test the effectiveness of analysing abstract syntax trees without the need to generate a parser for the task by importing a pre-existing Python library, such as **javalang***. This is an open-source, pure Python lexer/parser for Java 8, which enables the generation of an AST representation of each class within a Java program:

*The GitHub repository for javalang is accessible at https://github.com/c2nes/javalang

```python
import javalang

with open(file_name, 'r') as file:
    code = file.read()
tree = javalang.parse.parse(code)
```

Listing 5: Simple Python script to generate an AST from a given file_name

This script could be run on the contents of CaesarCipher.java, in listing 6:

```java
public class CaesarCipher extends Cryptosystem {
  final int cipherShift;

  public CaesarCipher(int shift) {
    this.cipherShift = shift;
  }

  // Intentionally missing @Override annotation for testing
  void transformInput() {
  }

  // Intentionally missing @Override annotation for testing
  void applyCipher() {
    this.text = shiftString(cipherShift);
  }
}
```

Listing 6: CaesarCipher.java

Fig. 5.1 displays an approximation of the AST generated for the Java code in listing 6:

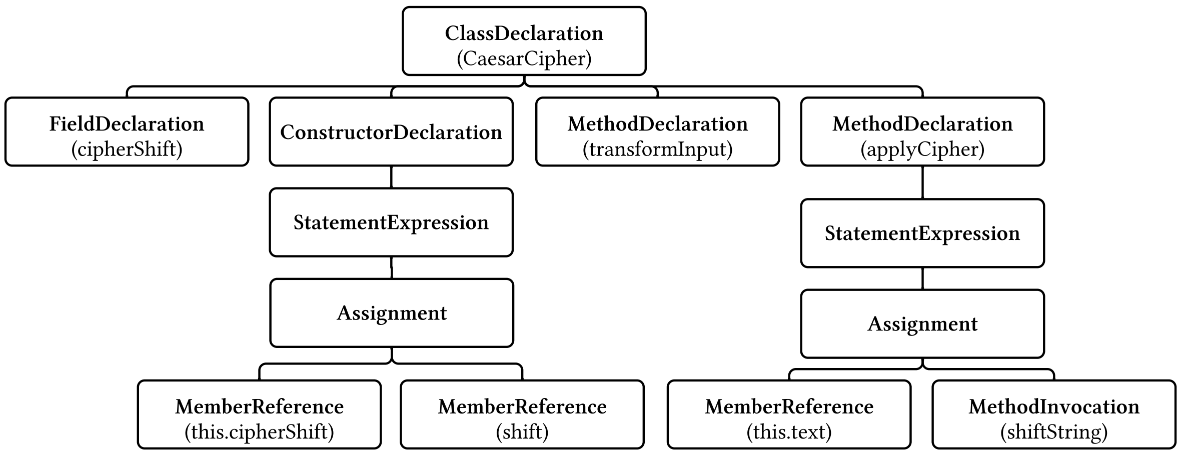
ClassDeclaration (CaesarCipher)
FieldDeclaration (cipherShift)
ConstructorDeclaration
MethodDeclaration (transformInput)
MethodDeclaration (applyCipher)
StatementExpression
StatementExpression
Assignment
Assignment
MemberReference (this.cipherShift)
MemberReference (shift)
MemberReference (this.text)
MethodInvocation (shiftString)

Fig. 5.1: Simplified AST for the CaesarCipher class in listing 6

The nodes of this AST contain more information than just the type of node, such as ClassDeclaration or Method-Invocation. For example, in the case of the FieldDeclaration for 'cipherShift', we also know it is of type *int* and has a *final* modifier. By constructing a function that checks these values while traversing an AST, we can create filters that return nodes based on specific types, modifiers, identifiers, return types, and more. Organizing these filters into a class allows for easy reuse, exemplified by the **JavaFilter** class defined in listing 7. This approach can be readily expanded to include additional node features, although these have been omitted here for brevity.

```python
from javalang import tree

class JavaFilter:
    def __init__(self, node_class='node', node_modifiers=None):
        self.node_class = node_class
        self.node_modifiers = node_modifiers
        self.class_map = {'node' : tree.Node, 'class': tree.ClassDeclaration,
                          'method': tree.MethodDeclaration, 'field': tree.FieldDeclaration}

    def filter_class(self, node):
        return isinstance(node, self.class_map[self.node_class])

    def filter_modifiers(self, node):
        if self.node_modifiers is None or len(self.node_modifiers) == 0:
            return True
        return set(self.node_modifiers).issubset(set(node.modifiers))

    def get_nodes(self, ast):
        return [node for _, node in ast if
                self.filter_class(node) and self.filter_modifiers(node)]
```

Listing 7: Partial implementation of a JavaFilter Python class

If a filter were to be used on the AST in fig. 5.1, where *node_class='method'* and *node_return_type='void'*, the two nodes matching that criteria, *transformInput* and *applyCipher* would be returned:

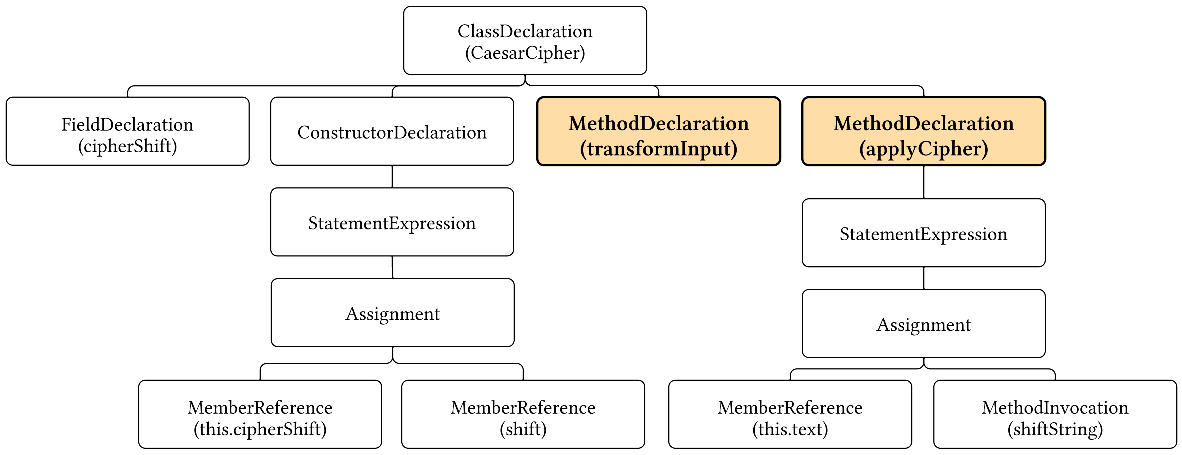
ClassDeclaration (CaesarCipher)
FieldDeclaration (cipherShift)
ConstructorDeclaration
MethodDeclaration (transformInput)
MethodDeclaration (applyCipher)
StatementExpression
Assignment
StatementExpression
Assignment
MemberReference (this.cipherShift)
MemberReference (shift)
MemberReference (this.text)
MethodInvocation (shiftString)

Fig. 5.2: The result of applying a filter for MethodDeclaration nodes with return type void on fig. 5.1

5.2 Building Rules from Filters

As stated in §4.2.2. some of the most frequent rules in software engineering task specifications involve **class encapsulation** and **naming convention**. These rules can be efficiently translated into Python functions using the JavaFilter class. For instance, a function designed to enforce the rule that methods with the @Test annotation cannot contain the word 'test' could be implemented as follows in listing 8:

```python
def check_test_identifier_rule(file):
    filter = JavaFilter(node_type='method', node_annotations=['Test'],
                        node_name_format='r"^test"')
    ast = javalang.parse.parse(file.contents)
    feedback = []
    for node in self.node_filter.get_nodes(ast):
        feedback.append(
            f'{file.file_name}:{node.position}: The {filter.node_class}, '
            f'{node.name}, does not follow the desired naming convention.')

    return feedback
```

Listing 8: A Python function to produce feedback on test methods that contain 'test' within their identifier

These rules can be encapsulated into classes, each taking a filter as an initialisation parameter. This approach allows filters, such as JavaFilter, to have a common Filter superclass, acting as an Adapter for ASTs representing code in different languages, such as Python, which utilises the ***ast*** library. This enables rules to be language-agnostic, provided an AST for the language can be generated containing the relevant information. Furthermore, this approach permits the rules themselves to have a common Rule superclass, enabling shared functionality such as narrowing the scope of rules to specified directories, files, or parts of an abstract syntax tree.

5.3 Composing Filters for Patterns

These filters can also be used to check for the presence of nodes within pre-filtered nodes, due to the recursive structure of ASTs. As such, composing filters can facilitate checks for design patterns, such as the Singleton Pattern (listing 2). In this function, a check for classes within each file's AST is followed by checks for a private field of the same type (the instance), a method of the class type (the getInstance method), and a private constructor.

```python
def find_singleton_pattern(files):
    singletons = []
    for file in files:
        s_classes = JavaFilter(node_class='class').get_nodes(file.ast)
        for s_class in s_classes:
            # Get private fields of type Class
            class_fields = JavaFilter(node_class='field', node_modifiers=['private'],
                node_type=s_class.name).get_nodes(s_class)
            if not class_fields:
                continue
            # Get methods of type Class
            class_methods = JavaFilter(node_class='method',
                node_return_type=s_class.name).get_nodes(s_class)
            if not class_methods:
                continue
            # Get private constructors
            private_constructors = JavaFilter(node_class='constructor',
                node_modifiers=['private']).get_nodes(s_class)
            if not private_constructors:
                continue
            singletons.append(s_class.name)
    return singletons
```

Listing 9: A Python function to return all potential implementations of a Singleton pattern within a Java project

It is possible to represent other structural patterns with these filters, but the use of per-class ASTs causes issues in producing a solution generic enough for the objective of adaptable specifications.

5.4 Specification Issues

Within ***javalang***, per-class ASTs will all have a ClassDeclaration as their root, but little to no relation to other classes beyond a class or method name. As a result, should a pattern extend beyond one single class, as almost all do, checking for potential instances becomes much more difficult, as each check must return all potential nodes within every file. This means that pattern checks become tedious to write, difficult to understand, and expensive to compute. However, if a global program representation could be computed from these ASTs, detecting patterns would become significantly simpler.

Chapter 6

Abstracting Design Patterns

This final chapter of the *AutoTA* implementation details the representation of design patterns as graphs, which can then be utilised in graph mining of a global entity-relation graph.

6.1 Design Patterns as Graphs

Within design patterns, we have the notion of entities and relations, just as in graphs we have the notion of nodes and edges. In particular, we have the entities:

Entity	**Label**
Class	C
Abstract Class	AC
Interface	I
Constructor	CON
Field	F
Method	M
Abstract Method	AM

Table 6.1: Table of Entities and their shorthand graph label

And we have the corresponding relations between these entities:

Relation	**Label**
Is Composed Of	Has
Extends Class/Interface	Extends
Implements Interface	Implements
Overrides Method	Overrides
Has The Type Of	HasType
Has The Return Type Of	ReturnType
Invokes Method	Invokes
Has Parameter Of Type	Parameter
Instantiates Class	Instantiates

Table 6.2: Table of Relations and their shorthand representation

With a combination of these, we can represent any of the 23 Gang of Four Design Patterns [Gamma et al. 1994] as an **Entity-Relation Graph**. For example, in the case of the Singleton Pattern we have:

- A class, C, that has:
  - A private constructor, CON
  - A private field, F, that is of the type C
  - A method, M, that has return type C

If we represent the above entities and relations graphically, we have fig. 6.1:

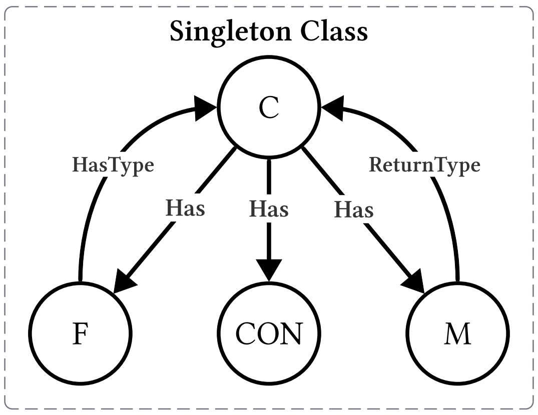

Fig. 6.1: The Singleton Pattern as a graph

Similarly, we can represent more complex patterns, such as the Template Method Pattern, as an ER-Graph. This is an example of a pattern that spans multiple classes, solving CHAPTER 5's issue, as shown in fig. 6.2:

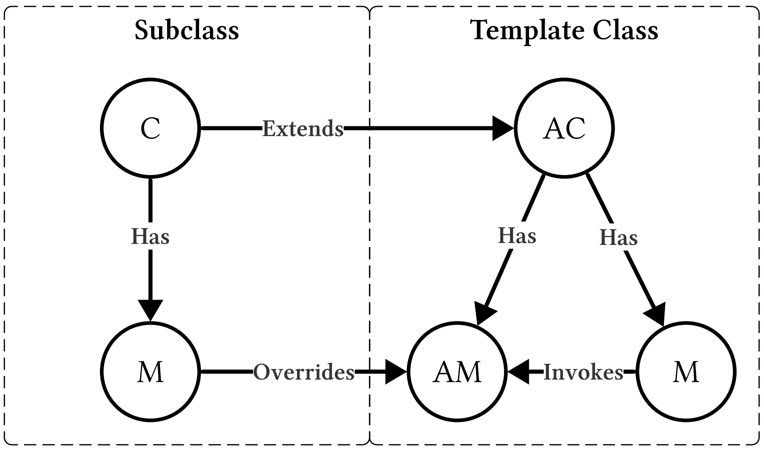

Fig. 6.2: The Template Method Pattern as a graph

These graphs can be represented as a set of Entity objects and a set of Relation objects as per listing 10:

```python
class Entity:
    def __init__(self, entity_type=None, name=None, entity_info=None, ast_node=None):
        self.type = entity_type
        self.name = name
        self.info = entity_info
        self.node = ast_node

class Relation:
    def __init__(self, entity_from, entity_to, relation_type):
        self.entity_from = entity_from
        self.entity_to = entity_to
        self.relation_type = relation_type
```

Listing 10: Python classes representing Entities (Nodes) and Relations (Edges) within a graph

6.2 Graph Program Representation

To utilise these pattern graphs, a similar Entity-Relationship graph representation must be built for programs.

6.2.1 First Pass

The ***javalang***, per-file ASTs are traversed to create Entity objects when a node from the set defined in table 6.1 is encountered. After this, a Composition Relation can be added for the *members* of the node. The result of performing this first pass on the CaesarCipher class (defined in listing 13) would be:

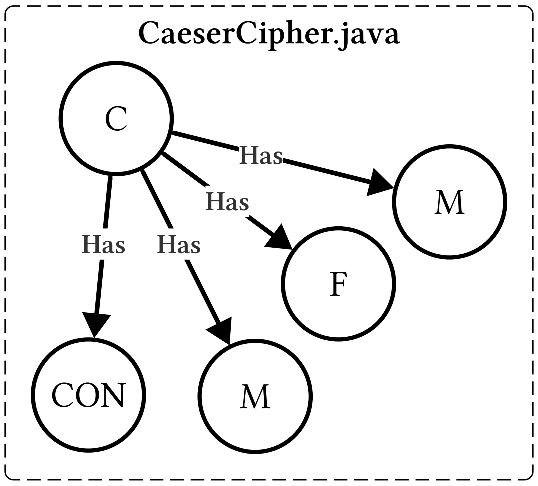

Fig. 6.3: The CaesarCipher class as a Graph

Node data such as modifiers, annotations, and identifiers are stored in the entity_info parameter of the Entity class, to be used in the event a filter is required. The graph-building process is repeated for all files within a set scope to generate a set of local Entity-Relation Graphs, as shown in fig. 6.4:

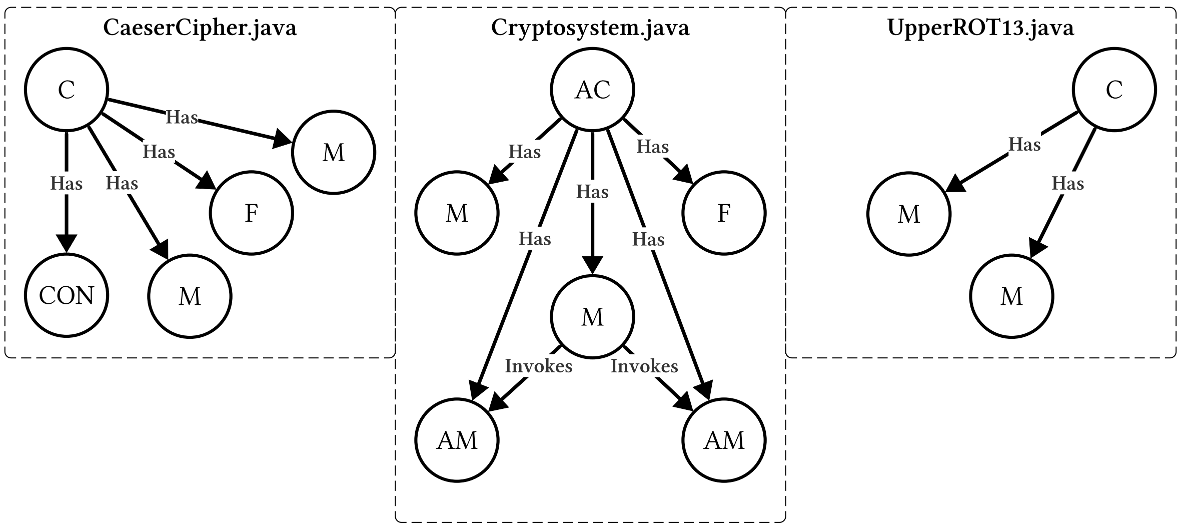

Fig. 6.4: Each file of the student submission with their corresponding ER Graph

6.2.2 Second Pass

Now that a complete set of program nodes has been generated, the other relations can be added to the graph by iterating through the global set of entities, which contain a reference to the AST node used to generate each one. For every entity, depending on the type, a number of checks are performed to determine which, if any, relations are added. This second pass of the graph is needed to ensure that the correct scope is applied when building these relations, to prevent an erroneous relation that could come as a result of two methods sharing the same name, for example.

Following this second pass, the final ER Graph for the example submission would look like fig. 6.5:

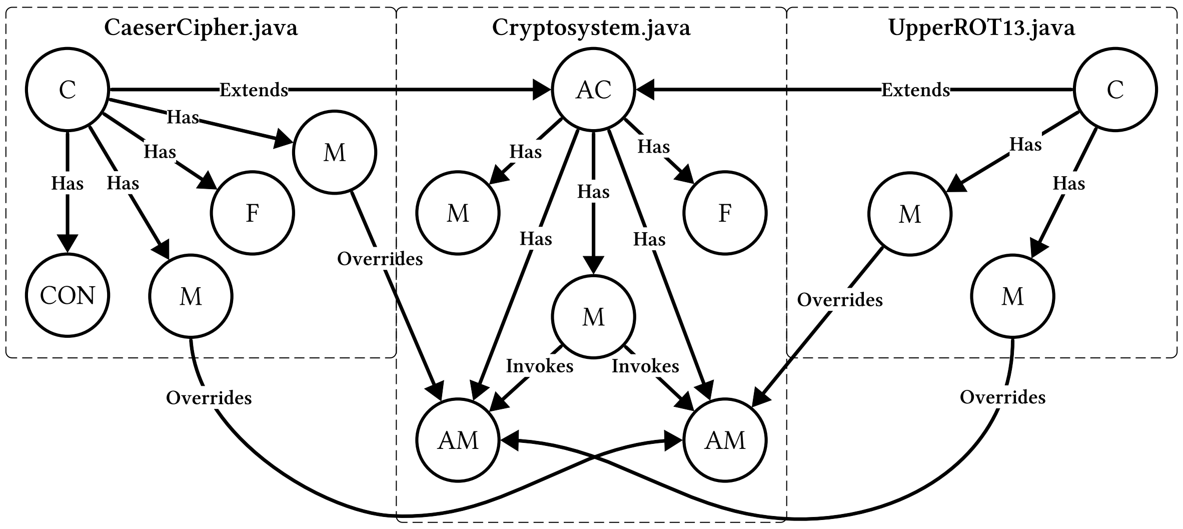

Fig. 6.5: The global ER Graph for the ExampleSubmission

6.3 Matching Pattern Graphs

Given an ER Graph representing a program and a graph representing a design pattern, we can search for instances of the design pattern by finding what is almost a subgraph isomorphism. That is, we don't necessarily need nodes to be completely equal, just that the nodes within the program graph contain **at least** the specified attributes of those in the Design Pattern.

The naive implementation of this detection would find all possible instances of each node within a pattern, before eliminating those without the correct relations between the other entities. We can optimise this process by immediately eliminating nodes that have a smaller degree than that of the corresponding pattern node, as it means that the node cannot have all the relations required to be part of a pattern implementation.

As an example, if we use the Template Method Graph in fig. 6.2 to attempt to find implementations of the Template Method Pattern within the program graph in fig. 6.5, we will see that such an instance is detected, in orange (being one of four in the program graph):

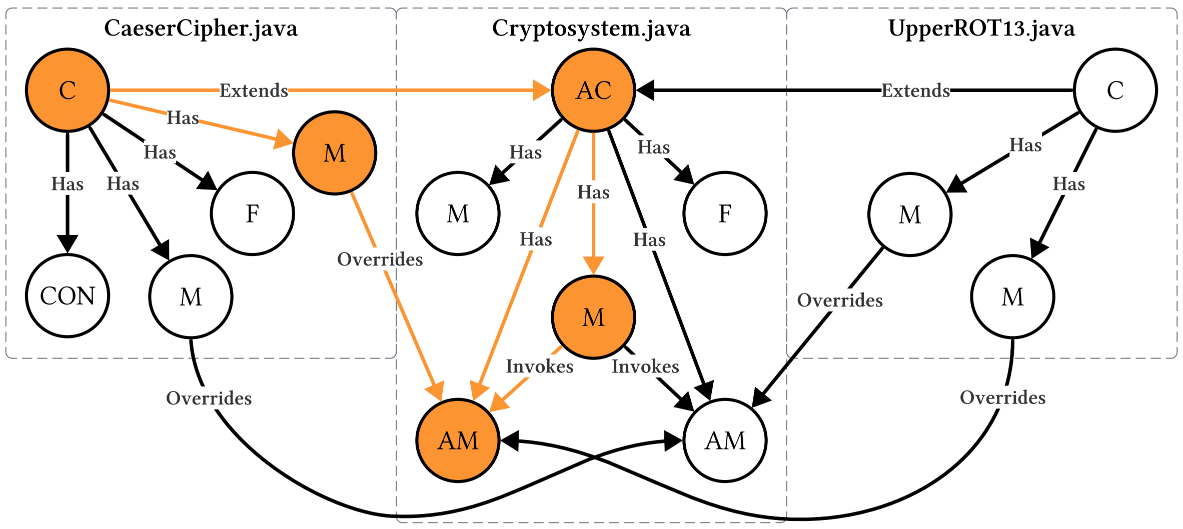

Fig. 6.6: The Template Method Pattern Graph matched on a subgraph of the Program Graph

These nodes would then be output together as a potential implementation of the above pattern, with similar negative feedback in the absence of any such subgraphs. In the case that only a single entity is initially missing, a perfect substitute will be placed in its stead. If the remainder of the pattern is detected, then we have that this entity is the sole missing component, which can be reported in feedback to provide a direct action point for students.

6.4 Building Specifications

To facilitate quick specification generation, design pattern names can be used to query a library containing pre-made examples of these graphs. These can be used directly or edited with stricter rules, such as entities needing to have a certain name. Additionally, the graph representation of these patterns could be produced in a graphical front-end to more easily generate custom patterns.

As with filters, these ER Graphs can be extended to any OOP language that shares, or at least approximates, the specified entities and relations. As a result, using the Python ***ast*** library would allow for the detection of these patterns within Python code. However, performance varies heavily, due to the originally intended language for these patterns being Java.

6.5 LLM Summary

Finally, we use the output of both filter-composed rules and graph mining design pattern matchers as feedback on a given submission. This is passed to the ChatGPT API once more, this time using *gpt-4*, which allows the use of a specialized 'code interpreter' model. The LLM is asked to provide a summary on the rules breached (or not) and design patterns found, nearly found, or that are completely absent. This produces an effective, concise summary that would allow a human teaching assistant to more quickly mark submissions, as per the objective.

Chapter 7

Evaluation

This penultimate chapter reviews the performance of *AutoTA* compared to human teaching assistants and similar solutions.

7.1 Student Code Performance

Having received permission from students, a useful method of testing the efficacy of *AutoTA* is by attempting to detect instances of Design Patterns from past student submissions.

There were two tasks in question:

Task 1 - This task focused on implementing **behavioral** design patterns, such as the Template Method Pattern. Of 29 submissions reviewed, 28 were correctly indicated as having a correct instance of the Template Method Pattern, giving an overall accuracy of 96%. This meant that *AutoTA* and the human teaching assistant agreed on all but one submission. This was an edge case, exposing a weakness in *AutoTA*'s use of packages to find the correct node within the scope.

Task 2 - This task focused on implementing **creational** design patterns, such as the Singleton Pattern. Of 28 submissions reviewed, 25 were correctly indicated as having a correct instance of the Singleton Pattern, and 3 were correctly indicated as having a partially correct instance, all lacking the private constructor. This gave an accuracy of 100% for this task, although it should be noted that Singleton Patterns are particularly straightforward to detect, given that their signature features all fall within the same class.

Overall, with just one incorrect review, *AutoTA* showed promise. To evaluate further, more design patterns can be successfully implemented, such as the Adapter Pattern, to ensure coverage for **structural** design patterns.

7.2 Graph Mining Comparison

In an interesting example of convergent evolution, a similar tool to *AutoTA* was developed in 2016 named DesPaD, using a similar approach of transforming Java ASTs into an Entity-Relation graph to detect design patterns [Oruc, Akal, and Sever 2016]. This provides an opportunity to discuss the comparative strengths and weaknesses of these two implementations.

One key difference is in specificity; DesPaD's graph vertices are limited exclusively to Classes and Interfaces. This means that the corresponding definitions of Design Patterns are broader as well, like, for example, their definition of the Bridge Pattern, recreated in fig. 7.1:

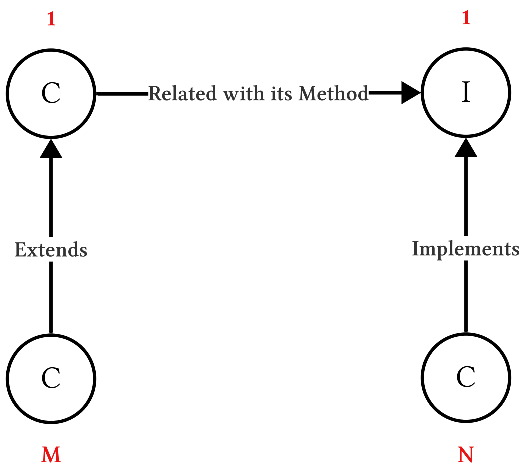

Fig. 7.1: A recreation of DesPaD's Bridge Pattern Template [Oruc, Akal, and Sever 2016]

This broad specification could explain why the recall for a more complex pattern like the Template Method Pattern was only 43% [Oruc, Akal, and Sever 2016]. There are a wider variety of entities available in *AutoTA*, such as Fields and Methods, along with options to specify node modifiers, identifiers, annotations, etc. While this makes Design Patterns more difficult to represent, it forces them to be given more precise definitions and allows structural patterns to be considerably more modular.

Overall, however, the key difference between the tools is the approach of education for *AutoTA*, basing the tool around specifications and feedback for student exercises. In comparison, DesPaD is seemingly purely a pattern detection tool. As such, it is fair to say that *AutoTA* would produce more useful feedback to a student thanks to the incorporation of an LLM.

Chapter 8

Conclusion

8.1 Reflection

The evaluation demonstrates that *AutoTA* effectively detects structural patterns and provides actionable feedback when pattern elements are missing. The tool supports the composition and detection of a wide range of structural patterns, some of which are included in a library, and produces readable summaries through prompt engineering.

However, like many static code analysis tools, *AutoTA* encounters challenges in flexibility, especially due to limitations of AST libraries and the variation in design pattern implementations across languages.

8.2 Future Work

Given the tool's strengths, there are several potential avenues for extension:

- **Extending to Other Programming Languages** - One immediate extension to the program could involve enhancing its capability to support additional object-oriented programming (OOP) languages such as C# and Kotlin. The program has been designed with this flexibility in mind; as long as an Entity-Relationship Graph can be constructed to represent a program, it can be utilized effectively. However, as referenced previously, the tool having been built with Java originally in mind has caused some issues with detecting patterns in a uniform manner, raising the point that perhaps each language should have their own design pattern implementations provided.
- **Facilitate Building Specifications** - Whilst simple rules like checking for encapsulation and design patterns are composable, this requires some knowledge of how the program works. Task specifications are traditionally written in plain English, so perhaps some form of model could be designed to take the standard marking criteria for a task and transform it into one that can be used to query submissions.
- **Incorporating Version Control in Analysis** As mentioned previously, industry code reviews are change based. That is, reviewers will compare the difference between two versions of some code to determine the effect any proposed update would have. A similar idea could be incorporated within the tool- students will make multiple commits throughout their work on an exercise, which can be each be analysed to develop an idea of how the code has progressed. This could be particularly effective for teaching areas like **Test Driven Development** that are reliant on effective version control.

Bibliography

Achiam, Josh et al. (2023). "GPT-4 Technical Report". In: arXiv: 2303.08774.

Ackerman, A.F., L.S. Buchwald, and F.H. Lewski (1989). "Software inspections: an effective verification process". In: *IEEE Software* 6.3, pp. 31–36. DOI: 10.1109/52.28121.

Aftandilian, Edward et al. (2012). "Building Useful Program Analysis Tools Using an Extensible Java Compiler". In: *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. Riva del Garda, Italy, pp. 14–23. DOI: 10.1109/SCAM.2012.28.

Ayewah, Nathaniel et al. (2008). "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5, pp. 22–29. DOI: 10.1109/MS.2008.130.

Bacchelli, Alberto and Christian Bird (2013). "Expectations, outcomes, and challenges of modern code review". In: *2013 35th International Conference on Software Engineering (ICSE)*, pp. 712–721. DOI: 10.1109/ICSE.2013.6606617.

Ball, Thoms (1999). "The concept of dynamic analysis". In: *ACM SIGSOFT Software Engineering Notes* 24.6, pp. 216–234.

Balse, Rishabh et al. (2023). "Evaluating the Quality of LLM-Generated Explanations for Logical Errors in CS1 Student Programs". In: *Proceedings of the 16th Annual ACM India Compute Conference*. COMPUTE '23. , Hyderabad, India, Association for Computing Machinery, pp. 49–54. ISBN: 9798400708404. DOI: 10.1145/3627217.3627233.

Baum, Tobias et al. (2016). "A Faceted Classification Scheme for Change-Based Industrial Code Review Processes". In: *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 74–85. DOI: 10.1109/QRS.2016.19.

Bernius, Jan Philip, Stephan Krusche, and Bernd Bruegge (2022). "Machine learning based feedback on textual student answers in large courses". In: *Computers and Education: Artificial Intelligence* 3. DOI: 10.1016/j.caeai.2022.100081.

Birkhoff, Garrett (1940). *Lattice theory*. Vol. 25. American Mathematical Soc. ISBN: 9780821810255.

Brown, Tom B. et al. (2020). *Language Models are Few-Shot Learners*. arXiv: 2005.14165.

Chen, Lawrence, Peter C. Rigby, and Nachiappan Nagappan (2022). "Understanding why we cannot model how long a code review will take: an industrial case study". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore: Association for Computing Machinery, pp. 1314–1319. ISBN: 9781450394130. DOI: 10.1145/3540250.3558945.

Davis, Joshua et al. (Mar. 2024). "The Temperature Feature of ChatGPT: Modifying Creativity for Clinical Research". In: *JMIR Hum Factors* 11, e53559. ISSN: 2292-9495. DOI: 10.2196/53559.

Di Martino, Beniamino and Antonio Esposito (2016). "A rule-based procedure for automatic recognition of design patterns in UML diagrams". In: *Software: Practice and Experience* 46.7, pp. 983–1007. DOI: https://doi.org/10.1002/spe.2336.

Fagan, M. E. (1976). "Design and code inspections to reduce errors in program development". In: *IBM Systems Journal* 15.3, pp. 182–211. DOI: 10.1147/sj.153.0182.

Fagan, Michael E. (2001). "Advances in Software Inspections". In: *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions.* Ed. by Manfred Broy and Ernst Denert. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 335–360. ISBN: 978-3-642-48354-7. DOI: 10.1007/978-3-642-48354-7_14.

Fan, Zhiyu et al. (2023). *Intelligent Tutoring System: Experience of Linking Software Engineering and Programming Teaching.* arXiv: 2310.05472.

Froemmgen, Alexander et al. (2024). "Resolving Code Review Comments with Machine Learning". In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice.* ICSE-SEIP '24. Lisbon, Portugal: Association for Computing Machinery, pp. 204–215. ISBN: 9798400705014. DOI: 10.1145/3639477.3639746.

Gamma, Erich et al. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software.* 1st ed., Reprint. Addison-Wesley Longman, Amsterdam. ISBN: 0201633612.

Giray, Louie (2023). "Prompt Engineering with ChatGPT: A Guide for Academic Writers". In: *Annals of Biomedical Engineering* 51.12, pp. 2629–2633. DOI: 10.1007/s10439-023-03272-4.

Goseva-Popstojanova, Katerina and Andrei Perhinschi (2015). "On the capability of static code analysis to detect security vulnerabilities". In: *Information and Software Technology* 68, pp. 18–33. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.08.002. URL: https://www.sciencedirect.com/science/article/pii/S0950584915001366.

Hanks, Brian et al. (2011). "Pair programming in education: a literature review". In: *Computer Science Education* 21.2, pp. 135–173. DOI: 10.1080/08993408.2011.579808.

Hannay, Jo et al. (July 2009). "The effectiveness of pair programming: A meta-analysis". In: *Information and Software Technology* 51, pp. 1110–1122. DOI: 10.1016/j.infsof.2009.02.001.

Hovemeyer, David and William Pugh (Dec. 2004). "Finding bugs is easy". In: 39.12. ISSN: 0362-1340. DOI: 10.1145/1052883.1052895.

Indriasari, Theresia Devi, Andrew Luxton-Reilly, and Paul Denny (Sept. 2020). "A Review of Peer Code Review in Higher Education". In: *ACM Trans. Comput. Educ.* 20.3. DOI: 10.1145/3403935.

Irving, C. and Colin Atkinson (Nov. 1997). "The Graphical Depiction of Design Patterns". In.

Jeary, Sherry et al. (Apr. 2011). "Can using Fagan Inspections improve the quality of specification in 2011? A Case Study". In: *BCS Quality Specialist Group Annual International Software Quality Management SQM/INSPIRE Conference.* URL: http://eprints.bournemouth.ac.uk/18183/.

Jenkins, Tony (2002). "On the difficulty of learning to program". In: *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences.* Citeseer. Loughborough, UK, pp. 53–58.

Jeuring, Johan et al. (2022). "Towards Giving Timely Formative Feedback and Hints to Novice Programmers". In: *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education.* ITiCSE-WGR '22. Dublin, Ireland: Association for Computing Machinery, pp. 95–115. ISBN: 9798400700101. DOI: 10.1145/3571785.3574124.

Ji, Ziwei et al. (Dec. 2023). "Towards Mitigating LLM Hallucination via Self Reflection". In: *Findings of the Association for Computational Linguistics: EMNLP 2023.* Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore:

Association for Computational Linguistics, pp. 1827–1843. DOI: 10.18653/v1/2023.findings-emnlp.123. URL: https://aclanthology.org/2023.findings-emnlp.123.

Kandiko Howson, Camille B. and Matt Mawer (Nov. 2013). *Student Expectations and Perceptions of Higher Education.* English. London, UK: King's Learning Institute.

Kem, Deepak (2022). "Personalised and adaptive learning: Emerging learning platforms in the era of digital and smart learning". In: *International Journal of Social Science and Human Research* 5.2, pp. 385–391.

Louridas, P. (2006). "Static code analysis". In: *IEEE Software* 23.4, pp. 58–61. DOI: 10.1109/MS.2006.114.

MacLeod, Laura et al. (2018). "Code Reviewing in the Trenches: Challenges and Best Practices". In: *IEEE Software* 35.4, pp. 34–42. DOI: 10.1109/MS.2017.265100500.

McIntosh, Shane et al. (2016). "An empirical study of the impact of modern code review practices on software quality". In: *Empirical Software Engineering* 21, pp. 2146–2189. DOI: 10.1007/s10664-015-9381-9.

Messer, Marcus et al. (Feb. 2024). "Automated Grading and Feedback Tools for Programming Education: A Systematic Review". In: *ACM Trans. Comput. Educ.* 24.1. DOI: 10.1145/3636515.

Oruc, Murat, Fuat Akal, and Hayri Sever (2016). "Detecting Design Patterns in Object-Oriented Design Models by Using a Graph Mining Approach". In: *2016 4th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pp. 115–121. DOI: 10.1109/CONISOFT.2016.26.

Rehman, Saif Ur, Asmat Ullah Khan, and Simon Fong (Nov. 2012). "Graph mining: A survey of graph mining techniques". In: *Seventh International Conference on Digital Information Management (ICDIM 2012).* Macau, Macao, pp. 88–92. DOI: 10.1109/ICDIM.2012.6360146.

Sadowski, Caitlin et al. (2018). "Modern code review: a case study at google". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice.* ICSE-SEIP '18. Gothenburg, Sweden: Association for Computing Machinery, pp. 181–190. ISBN: 9781450356596. DOI: 10.1145/3183519.3183525.

Simões, Alberto and Ricardo Queirós (2020). "On the Nature of Programming Exercises". In: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. DOI: 10.4230/OASICS.ICPEC.2020.24. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.ICPEC.2020.24.

Stefik, Mark and Daniel G. Bobrow (Dec. 1985). "Object-Oriented Programming: Themes and Variations". In: *AI Magazine* 6.4, p. 40. DOI: 10.1609/aimag.v6i4.508. URL: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/508.

Stelting, Stephen and Olav Maassen (2002). *Applied Java Patterns.* Prentice Hall Professional.

Wegner, Peter (Aug. 1990). "Concepts and paradigms of object-oriented programming". In: *SIGPLAN OOPS Mess.* 1.1, pp. 7–87. ISSN: 1055-6400. DOI: 10.1145/382192.383004.

Weller, Martin (2018). "Twenty years of EdTech". In: *Educause Review Online* 53.4, pp. 34–48.

Zhang, Jian et al. (2019). "A Novel Neural Source Code Representation Based on Abstract Syntax Tree". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794. DOI: 10.1109/ICSE.2019.00086.

Zheng, J. et al. (2006). "On the value of static analysis for fault detection in software". In: *IEEE Transactions on Software Engineering* 32.4, pp. 240–253. DOI: 10.1109/TSE.2006.38.

Appendix A

Example Programming Exercise Submission

The following Java code is the full implementation of an imagined student's attempt at a mock exercise centered around utilising the Template Method Pattern for Cryptography. *Cryptosystem* is the Template Method class:

```java
package main.example;

// Abstract class defining Template Method
abstract public class Cryptosystem {
  protected String text;

  // Template Method
  public String encryptString(String input) {
    text = input;
    transformInput();
    applyCipher();

    return text;
  }

  // Hook Method
  String shiftString(int shift) {
    shift = shift % 26;
    StringBuilder cipherText = new StringBuilder();

    for (char c : text.toCharArray()) {
      if (Character.isLetter(c)) {
        char base = Character.isLowerCase(c) ? 'a' : 'A';
        char shiftedChar = (char) ((c - base + shift + 26) % 26 + base);
        cipherText.append(shiftedChar);
      } else {
        cipherText.append(c);
      }
    }

    return cipherText.toString();
  }

  // Operations to be Implemented by Subclasses
  abstract void transformInput();

  abstract void applyCipher();
}
```

Listing 11: Cryptosystem.java

Then *UpperROT13* (an imagined cipher) is a Subclass:

```java
package main.example;

public class UpperROT13 extends Cryptosystem {

  @Override
  void transformInput() {
    this.text = this.text.toUpperCase();
  }

  @Override
  void applyCipher() {
    this.text = shiftString(13);
  }
}
```

Listing 12: UpperROT13.java

CaesarCipher is a Subclass that intentionally does not include @Override annotations:

```java
package main.example;


public class CaesarCipher extends Cryptosystem {
  final int cipherShift;

  public CaesarCipher(int shift) {
    this.cipherShift = shift;
  }

  // Intentionally missing @Override annotation for testing
  void transformInput() {
  }

  // Intentionally missing @Override annotation for testing
  void applyCipher() {
    this.text = shiftString(cipherShift);
  }
}
```

Listing 13: CaesarCipher.java

```java
package test.example;

import main.example.*;
import org.junit.Test;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.core.Is.is;

public class UpperROT13Test {
  Cryptosystem cryptosystem = new UpperROT13();

  @Test
  public void encryptsUpperCase() {
    assertThat(cryptosystem.encryptString("HELLO"), is("URYYB"));
  }

  @Test
  public void encryptsLowerCase() {
    assertThat(cryptosystem.encryptString("hello"), is("URYYB"));
  }
}
```

Listing 14: UpperROT13Test.java

```java
package test.example;

import main.example.*;
import org.junit.Test;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.core.Is.is;

public class CaesarCipherTest {
  @Test
  public void testShiftZeroIsSelf() {
    Cryptosystem cryptosystem = new CaesarCipher(0);
    assertThat(cryptosystem.encryptString("Hello"), is("Hello"));
  }

  @Test
  public void testShiftOne() {
    Cryptosystem cryptosystem = new CaesarCipher(1);
    assertThat(cryptosystem.encryptString("Hello"), is("Ifmmp"));
  }
}
```

Listing 15: CaesarCipherTest.java