Chapter 1

Related work

In the previous chapter, we have stressed the paramount importance of frequently integrating one's changes into the upstream repository. This process can prove to be a complex and lengthy operation. As a result, software engineers have sought and found ways to automate this task. These solutions and practices embody Continuous Integration (CI). However, CI is not the golden bullet for software engineering, as there is a flip side to applying this practice. After every integration, we must execute the entire test suite to ensure that we have not introduced any regressions. As the project evolves and the size of the codebase increases, the number of test cases will increase accordingly to preserve a sufficiently high coverage level [6]. Walcott, Soffa and Kapfhammer illustrate the magnitude of this problem by providing an example of a project consisting of 20 000 lines of code, whose test suite requires up to seven weeks to complete [7].

Fortunately, developers and researchers have found multiple techniques to address the scalability issues of ever-growing test suites. We can classify the techniques currently known in literature into three categories [6]. These categories are Test Suite Minimisation (TSM), Test Case Selection (TCS) or Test Case Prioritisation (TCP). We can apply each technique to every test suite, but the outcome will be different. TSM and TCS will have an impact on the execution time of the test suite, at the cost of a reduced test coverage level. In contrast, TCP will have a weaker impact on the execution time but will not affect the test adequacy.

The following sections will discuss these three approaches in more detail and provide accompanying algorithms. Because the techniques are very similar, the corresponding algorithms can (albeit with minor modifications) be used interchangeably for every approach. The final section of this chapter will investigate the adoption and integration of these techniques in modern software testing frameworks.

1.1 Classification of approaches

1.1.1 Test Suite Minimisation

The first technique is called Test Suite Minimisation (TSM), also referred to as *Test Suite Reduction* in literature. This technique will try to reduce the size of the test suite by permanently removing redundant test cases. This problem has been formally defined by Rothermel [8] in definition 1 and illustrated in Figure 1.1.

Definition 1 (Test Suite Minimisation)**.**
Given:

- $T = \{t_1, \ldots, t_n\}$ *a test suite consisting of test cases* t_j.
- $R = \{r_1, \ldots, r_m\}$ *a set of requirements that must be satisfied in order to provide the desired "adequate" testing of the program.*
- $\{T_1, \ldots, T_m\}$ *subsets of test cases in* T*, one associated with each of the requirements* r_i*, such that any one of the test cases* $t_j \in T_i$ *can be used to satisfy requirement* r_i.

Subsequently, we can define Test Suite Minimisation as the task of finding a subset T' *of test cases* $t_j \in T$ *that satisfies every requirement* r_i.

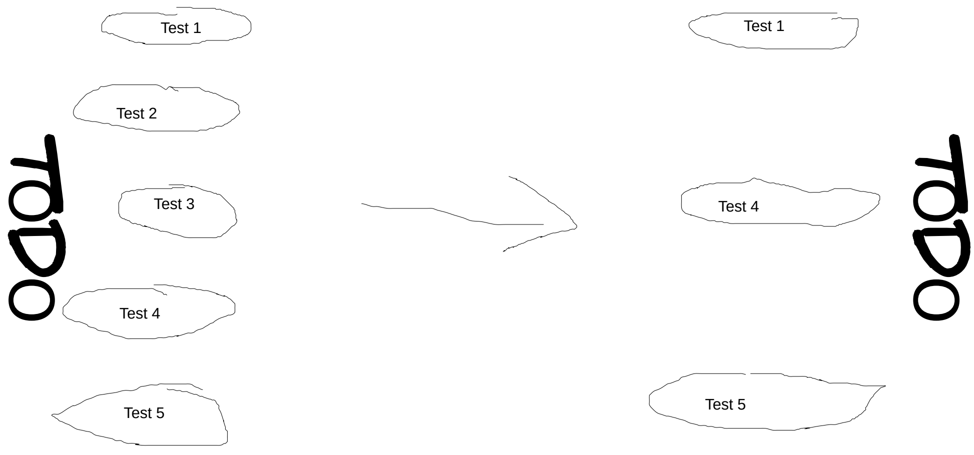

Figure 1.1: Test Suite Minimisation

If we apply the concepts of the previous chapter to the above definition, we can interpret the set of requirements R as source code lines that must be covered. A requirement r_i can subsequently be satisfied by any test case $t_j \in T$ that belongs to the subset T_i. Observe that the problem of finding T' is closely related to the *hitting set problem* (definition 2) [8].

Definition 2 (Hitting Set Problem)**.**
Given:

- $S = \{s_1, \ldots, s_n\}$ *a finite set of elements.*
- $C = \{c_1, \ldots, c_n\}$ *a collection of sets, with* $\forall c_i \in C : c_i \subseteq S$.
- K *a positive integer,* $K \leq |S|$.

The hitting set is a subset $S' \subseteq S$ such that S' contains at least one element from each subset in C.

In the context of Test Suite Minimisation, T' corresponds to the hitting set of T_is. In order to effectively minimise the amount of tests in the test suite, T' should be the minimal hitting set [8]. Since we can reduce this problem to the NP-complete *Vertex Cover*-problem, we know that this problem is NP-complete as well [2].

1.1.2 Test Case Selection

The second approach closely resembles the previous one. However, instead of permanently removing redundant test cases, Test Case Selection (TCS) has a notion of context. In this algorithm, we will not calculate the minimal hitting set at runtime, but before executing the test suite, we will perform a *white-box static analysis* of the source code. This analysis identifies which parts of the source code have been changed and executes only the corresponding test cases. Subsequent executions of the test suite will require a new analysis, thus making the selection temporary (Figure 1.2) and modification-aware [8]. Rothermel and Harrold define this formally in definition 3.

Definition 3 (Test Case Selection)**.**
Given:

- P *the previous version of the codebase*
- P' *the current (modified) version of the codebase*
- T *the test suite*

Test Case Selection aims to find a subset $T' \subseteq T$ that is used to test P'.

1.1.3 Test Case Prioritisation

Both TSM and TCS attempt to execute as few tests as possible to reduce the execution time of the test suite. Nevertheless, in some cases, we may require to execute every test case to guarantee correctness. In this situation, we can still optimise the test suite.

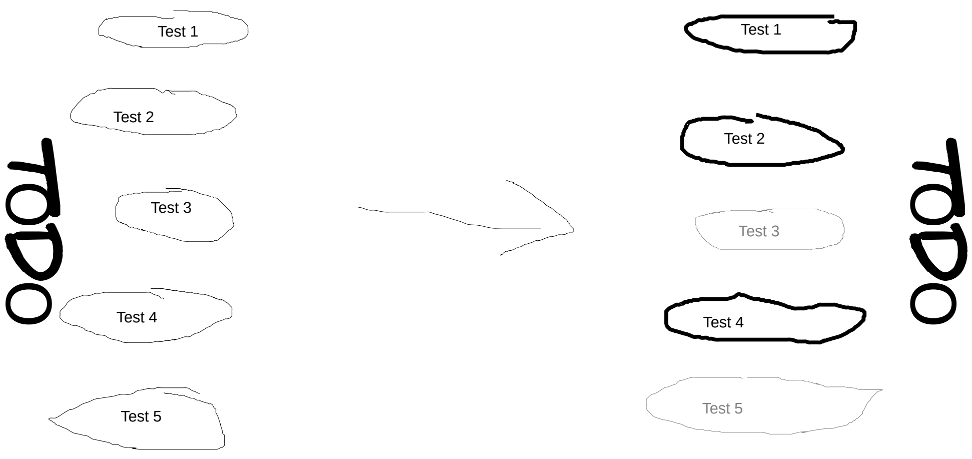

Figure 1.2: Test Case Selection

Test Case Prioritisation (TCP) aims to find a permutation of the sequence of test cases, rather than eliminating specific tests from being executed (Figure 1.3). We choose the order of the permutation in such a way that we can complete a predefined objective as soon as possible. Once we have achieved our objective, we can early terminate the execution of the test suite. In the worst-case scenario, we will still execute every test case. Some examples of objectives include covering as many lines of code as fast as possible or executing tests ordered on their probability of failure [8]. Definition 4 provides a formal definition of this approach.

Definition 4 (Test Case Prioritisation)**.**
Given:

- T *the test suite*
- PT *the set of permutations of* T
- $f : PT \mapsto \mathbb{R}$ *a function from a subset to a real number, this function is used to compare sequences of test cases to find the optimal permutation.*

Test Case Prioritisation finds a permutation $T' \in PT$ *such that* $\forall T'' \in PT : f(T') \geq f(T'') \Rightarrow (T'' \neq T')$

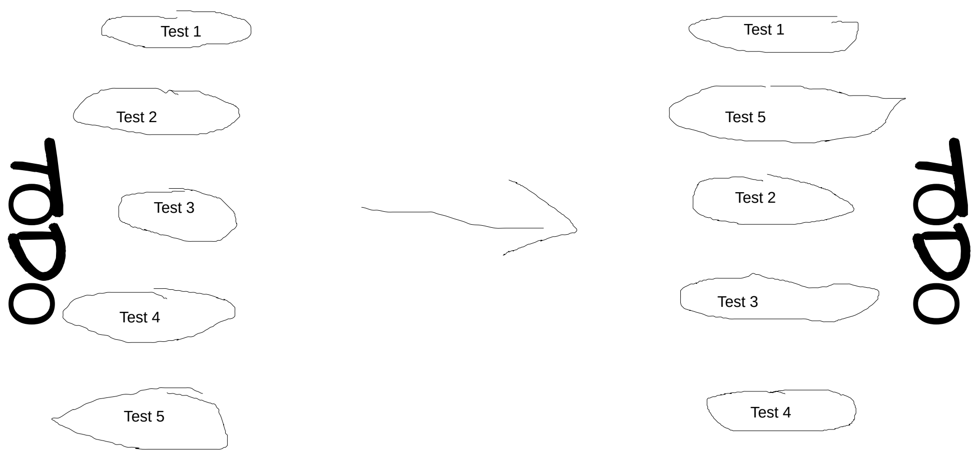

Figure 1.3: Test Case Prioritisation

1.2 Algorithms

TCP is essentially an extended version of TSM since we can first execute the minimised test suite and afterwards the remaining test cases. Additionally, section 1.1.1 has explained that TSM is an instance of the minimal hitting set problem, which is an NP-complete problem. Consequently, we know that both TSM and TCP are NP-complete problems as well and therefore, we require the use of *heuristics*. A heuristic is an experience-based method that can be applied to solve a hard to compute problem by finding a fast approximation. However, the found solution will mostly be suboptimal, or sometimes the algorithm might even fail to find any solution at all. Given the relation between TSM and the minimal hitting set problem, we can implement an optimisation algorithm by modifying any known heuristic that finds the minimal hitting set. This paper will now proceed by discussing a selection of these heuristics. The used terminology and the names of the variables have been changed to ensure mutual consistency between the algorithms. Every algorithm has been adapted to adhere to the conventions provided in definitions 5 and 6.

Definition 5 (Naming convention)**.**

- $TS = \{T_1, \dots, T_n\}$*: the set of all test cases* t *in the test suite.*
- $RS = \{T_1, \dots, T_n\} \subseteq TS$*: the representative set of test cases* t *that have been selected by the algorithm.*
- $C = \{c_1, \dots, c_m\}$*: the set of all source code lines in the application, that are covered by at least one test case* $T \in TS$*.*
- $CT = \begin{bmatrix} CT_1 & \dots & CT_m \end{bmatrix}$*: the set of test groups.*
  - $CT_c = \{T_1, \dots, T_n\} \subseteq TS$*: the test group* c*, which corresponds to the set of all test cases* $T \in TS$ *that cover the source code line* $c \in C$*.*

- $TL = \begin{bmatrix} TL_1 & \dots & TL_n \end{bmatrix}$: *the set of coverage groups.*
  - $TL_t = \{c_1, \dots, c_m\} \subseteq C$: *the set of all source code lines* $c \in C$ *that are covered by test case* $t \in TS$.

Definition 6 (Cardinality)**.** *For a finite set* S, *the cardinality* $|S|$ *is defined as the number of elements in* S. *In case of potential confusion, we can use* $Card(S)$ *to denote the cardinality of* S.

1.2.1 Greedy algorithm

The first algorithm is a *greedy* heuristic, which was initially designed by Chvatal to find an approximation for the set-covering problem [6]. A greedy algorithm always makes a locally optimal choice, assuming that this will eventually lead to a globally optimal solution [1]. Algorithm 1 presents the Greedy algorithm for Test Suite Minimisation. The objective of the algorithm is to construct a set of test cases that cover every line in the code, by requiring as few test cases as possible.

Initially, the algorithm starts with an empty representative set RS, the set TS of all test cases and the set C of all coverable source code lines. Furthermore, TL denotes the set of coverage groups as specified in the definition. In essence, the algorithm will iteratively select test cases from TS and add them to RS. The locally optimal choice is always to select the test case that will contribute the most still uncovered lines, ergo the test case T for which the cardinality of the intersection between C and TL_t is maximal. After every iteration, we remove the code lines TL_t from C, since these are now covered. We repeat this selection process until C is empty, which indicates that we have covered every source code line. Afterwards, when we execute the test suite, we only need to execute test cases in RS. We can apply this algorithm to Test Case Prioritisation as well, by changing the type of RS to a list instead. We require a list to maintain the insertion order since this is equivalent to the ideal order of execution.

Algorithm 1 Greedy algorithm for Test Suite Minimisation

```
1:  Input: Set $TS$ of all test cases,
        Set $C$ of all source code lines that are covered by any $t \in TS$,
        $TL_t$ the set of all lines are covered by test case $t \in TS$.
2:  Output: Subset $RS \subseteq TS$ of tests to execute.
3:  $RS \leftarrow \emptyset$
4:  while $C \neq \emptyset$ do
5:      $t\_max \leftarrow 0$
6:      $tl\_max \leftarrow \emptyset$
7:      for all $t \in TS$ do
8:          $tl\_current \leftarrow C \cap TL_t$
9:          if $|tl\_current| > |tl\_max|$ then
10:             $t\_max \leftarrow t$
11:             $tl\_max \leftarrow tl\_current$
12:     $RS \leftarrow RS \cup \{t\_max\}$
13:     $C \leftarrow C \setminus tl\_max$
```

1.2.2 HGS

The second algorithm is the HGS algorithm. The algorithm was named after its creators Harrold, Gupta and Soffa [3]. Similar to the Greedy algorithm (section 1.2.1), this algorithm will also iteratively construct the minimal hitting set. However, instead of considering the coverage groups TL, the algorithm uses the test groups CT. More specifically, we will use the distinct test groups, denoted as CTD. We consider two test groups CT_i and CT_j as distinct if they differ in at least one test case. The pseudocode for this algorithm is provided in Algorithm 2.

The algorithm consists of two main phases. The first phase begins by constructing an empty representative set RS in which we will store the selected test cases. Subsequently, we iterate over every source code line $c \in C$ to create the corresponding test groups CT. As mentioned before, we will reduce this set to CTD for performance reasons and as such, only retain the distinct test groups. Next, we select every test group of which the cardinality is equal to 1 and add these to RS. The representative set will now contain every test case that covers precisely one line of code, which is exclusively covered by that single test case. Afterwards, we remove every covered line from C. The next phase consists of repeating this process for increasing cardinalities until C is empty. However, since the test groups will now contain more than one test case, we need to make a choice on which test case to select. The authors prefer the test case that covers the most remaining lines. In the event of a tie, we defer the choice until the next iteration.

The authors have provided an accompanying calculation of the computational time complexity of this algorithm [3]. In addition to the naming convention introduced in definition 5, let n denote the number of distinct test groups CTD, nt the number of test cases $t \in TS$ and MAX_CARD the cardinality of the test group with the most test cases. In the HGS algorithm we need to perform two steps repeatedly. The first step involves computing the number of occurrences of every test case t in each test group. Given that there are n distinct test groups and, in the worst-case scenario, each test group can contain MAX_CARD test cases which we all need to examine once, the computational cost of this step is equal to $O(n * MAX\_CARD)$. For the next step, in order to determine which test case we should include in the representative set RS, we need to find all test cases for which the number of occurrences in all test groups is maximal, which requires at most $O(nt * MAX\_CARD)$. Since every repetition of these two steps adds a test case that belongs to at least one out of n test groups to the representative set, the overall runtime of the algorithm is $O(n*(n+nt)*MAX\_CARD)$.

Algorithm 2 HGS algorithm ([3])

```
1: **Input:** Distinct test groups $T_1, \dots T_n \in CDT$, containing test cases from $TS$.
2: **Output:** Subset $RS \subseteq TS$ of tests to execute.
3: $marked \leftarrow array[1 \dots n]$                                ▷ initially $false$
4: $MAX\_CARD \leftarrow max\{Card(T_i)|T_i \in CDT\}$
5: $RS \leftarrow \bigcup\{T_i|Card(T_i) = 1\}$
6: for all $T_i \in CDT$ do
7:     if $T_i \cap RS \neq \emptyset$ then $marked[i] \leftarrow true$
8: $current \leftarrow 1$
9: while $current < MAX\_CARD$ do
10:     $current \leftarrow current + 1$
11:     while $\exists T_i : Card(T_i) = current, marked[i] = false$ do
12:         $list \leftarrow \{t|t \in T_i : Card(T_i) = current, marked[i] = false\}$
13:         $next \leftarrow SelectTest(current, list)$
14:         $reduce \leftarrow false$
15:         for all $T_i \in CDT$ do
16:             if $next \in T_i$ then
17:                 $marked[i] = true$
18:                 if $Card(T_1) = MAX\_CARD$ then $reduce \leftarrow true$
19:         if $reduce$ then
20:             $MAX\_CARD \leftarrow max\{Card(T_i)|marked[i] = false\}$
21:         $RS \leftarrow RS \cup \{next\}$
22: function SelectTest($size$, $list$)
23:     $count \leftarrow array[1 \dots nt]$
24:     for all $t \in list$ do
25:         $count[t] \leftarrow |\{T_j|t \in T_j, marked[T_j] = false, Card(T_j) = size\}|$
26:     $tests \leftarrow \{t|t \in list, count[t] = max(count)\}$
27:     if $|tests| = 1$ then return $tests[0]$
28:     else if $|tests| = MAX\_CARD$ then return $tests[0]$
29:     else return $SelectTest(size + 1, tests)$
```

1.2.3 ROCKET algorithm

The third and final algorithm is the ROCKET algorithm. This algorithm has been presented by Marijan, Gotlieb and Sen [5] as part of a case study to improve the testing efficiency of industrial video conferencing software. Contrarily to the previous algorithms, which attempted to execute as few test cases as possible, this algorithm does execute the entire test suite. Unlike the previous algorithms that only take code coverage into account, this algorithm also considers historical failure data and test execution time. The objective of this algorithm is twofold: select the test cases with the highest successive failure rate, while also maximising the number of executed test cases in a limited time frame. In the implementation below, we will consider an infinite time frame as this is a domain-specific constraint and irrelevant for this thesis. This algorithm will yield a total ordering of all the test cases in the test suite, ordered using a weighted function.

The modified version of the algorithm (of which the pseudocode is provided in Algorithm 3) takes three inputs:

- $TS = \{T_1, \ldots, T_n\}$: the set of test cases to prioritise.
- $E = \begin{bmatrix} E_1 & \ldots & E_n \end{bmatrix}$: the execution time of each test case.
- $F = \begin{bmatrix} F_1 & \ldots & F_n \end{bmatrix}$: the failure statuses of each test case.
  - $F_t = \begin{bmatrix} f_1 & \ldots & f_m \end{bmatrix}$: the failure status of test case t over the previous m successive executions. $F_{ij} = 1$ if test case i has failed in execution $(current - j)$, 0 if it has passed.

The algorithm starts by creating an array P of length n, which contains the priority of each test case. The priority of each test case is initialised at zero. Next, we construct an $m \times n$ failure matrix MF and fill it using the following formula.

$$MF[i,j] = \begin{cases} 1 & \text{if } F_{ji} = 1 \\ -1 & \text{otherwise} \end{cases}$$

Table 1.1 contains an example of this matrix MF. In this table, we consider the hypothetical failure rates of the last three executions of six test cases.

run	T_1	T_2	T_3	T_4	T_5	T_6
$current - 1$	1	1	1	1	-1	-1
$current - 2$	-1	1	-1	-1	1	-1
$current - 3$	1	1	-1	1	1	-1

Table 1.1: Example of the failure matrix MF.

Afterwards, we fill P with the cumulative priority of each test case. We can calculate the priority of a test case by multiplying its failure rate with a domain-specific weight heuristic ω. This heuristic reflects the probability of repeated failures of a test case, given earlier failures. In their paper [5], the authors apply the following weights:

$$\omega_i = \begin{cases} 0.7 & \text{if } i = 1 \\ 0.2 & \text{if } i = 2 \\ 0.1 & \text{if } i >= 3 \end{cases}$$

$$P_j = \sum_{i=1 \ldots m} MF[i,j] * \omega_i$$

Finally, the algorithm groups test cases based on their calculated priority in P. Every test case that belongs to the same group is equally relevant for execution in the current test run. However, within every test group, the test cases will differ in execution time E. The final step is to reorder test cases that belong to the same group in such a way that test cases with a shorter duration are executed earlier in the group.

Algorithm 3 ROCKET algorithm

```
 1: Input: Set $TS = \{T_1, \dots, T_n\}$ of all test cases,
        Execution time $E_t$ of every test case,
        Failure status $F$ for each test case over the previous $m$ successive iterations.
 2: Output: Priority of test cases $P$.
 3: $P \leftarrow array[1 \dots n]$                                  ▷ initially $0$
 4: $MF \leftarrow array[1 \dots m]$
 5: for all $i \in 1 \dots m$ do
 6:     $MF[i] \leftarrow array[1 \dots n]$
 7:     for all $j \in 1 \dots n$ do
 8:         if $F[j][i] = 1$ then $MF[i][j] \leftarrow -1$
 9:         else $MF[i][j] \leftarrow 1$
10: for all $j \in 1 \dots n$ do
11:     for all $i \in 1 \dots m$ do
12:         if $i = 1$ then $P[j] \leftarrow P[j] + (MF[i][j] * 0.7)$
13:         else if $i = 2$ then $P[j] \leftarrow P[j] + (MF[i][j] * 0.2)$
14:         else $P[j] + (MF[i][j] * 0.1)$
15: $Q \leftarrow \{P[j] | j \in 1 \dots n\}$                          ▷ distinct priorities
16: $G \leftarrow array[1 \dots Card(Q)]$                             ▷ initially empty sets
17: for all $j \in 1 \dots n$ do
18:     $p \leftarrow P[j]$
19:     $G[p] \leftarrow G[p] \cup \{j\}$
20: Sort every group in $G$ based on ascending execution time in $E$.
21: Sort $P$ according to which group it belongs and its position within that group.
```

1.3 Adoption in testing frameworks

In the final section of this chapter, we will investigate how existing software testing frameworks have implemented these and other optimisation techniques.

1.3.1 Gradle and JUnit

Gradle[1] is a dependency manager and development suite for Java, Groovy and Kotlin projects. It supports multiple plugins to automate tedious tasks, such as configuration management, testing and deploying. One of the supported testing integrations is JUnit[2], which is the most widely used testing framework by Java developers. JUnit 5 is the newest version which is still under active development as of today. Several prominent Java libraries and frameworks, such as Android and Spring have integrated JUnit as the preferred testing framework. The testing framework offers mediocre support for features that optimise the execution of the test suite, primarily when used in conjunction with Gradle. The following three key elements are available:

1. **Parallel test execution:** The Gradle implementation of JUnit features multiple *test class processors*. A test class processor is a component which processes Java classes to find all the test cases, and eventually to execute them. One of these processors is the `MaxNParallelTestClassProcessor`, which is capable of running a configurable amount of test cases in parallel. Concurrently executing the test cases results in a significant speed-up of the overall test suite execution.

2. **Prioritise failed test cases:** Gradle provides a second useful test class processor: the `RunPreviousFailedFirstTestClassProcessor`. This processor will prioritise test cases that have failed in the previous run. This practice is similar to the ROCKET-algorithm (section 1.2.3), but the processor does not take into account the duration of the test cases.

3. **Test order specification:** JUnit allows us to specify the sequence in which it will execute the test cases. By default, it uses a random yet deterministic order[3]. The order can be manipulated by annotating the test class with the `@TestMethodOrder`-annotation, or by applying the `@Order(int)`-annotation to an individual test case. However, we can only use this feature to alter the order of test cases within the same test class. JUnit does not support inter-test class reordering. We could use this feature to (locally) sort test cases based on their execution time.

[1]`https://gradle.org`
[2]`https://junit.org`
[3]`https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-execution-order`

Figure 1.4: Logo of Gradle

Figure 1.5: Logo of JUnit 5

1.3.2 Maven Surefire

A commonly used alternative to Gradle is Apache Maven[4]. This framework also supports executing JUnit test cases using the Surefire plugin. As opposed to Gradle, Surefire does offer multiple options to specify the order in which the test cases will be executed using the `runOrder` property. Without any configuration, Maven will run the test cases in alphabetical order. By switching the `runOrder` property to `failedFirst`, we can tell Maven to prioritise the previously failed test cases. Another supported value is `balanced`, which orders test cases based on their duration. Finally, we can choose to implement a custom ordering scheme for absolute control.

Figure 1.6: Logo of Maven

1.3.3 OpenClover

OpenClover[5] is a code coverage tool for Java and Groovy projects. It was created by Atlassian and open-sourced in 2017. OpenClover profiles itself as "the most sophisticated code coverage tool", by extracting useful metrics from the coverage results and by providing features that can optimise the test suite. These features include powerful integrations with development software and prominent Continuous Integration systems. Furthermore, OpenClover can automatically analyse the coverage results to detect relations between the application source code and the test cases. This feature allows OpenClover to predict which test cases will have been affected, given a set of modifications to the source code. Subsequently, we can interpret these predictions to implement Test Case Selection and therefore reduce the test suite execution time.

Figure 1.7: Logo of Atlassian Clover

[4] `http://maven.apache.org/`
[5] `https://openclover.org`

Bibliography

[1] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[3] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. "A Methodology for Controlling the Size of a Test Suite". In: *ACM Trans. Softw. Eng. Methodol.* 2.3 (July 1993), pp. 270–285. ISSN: 1049-331X. DOI: 10.1145/152388.152391. URL: https://doi.org/10.1145/152388.152391.

[4] "ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:Concepts and definitions". In: *ISO/IEC/IEEE 29119-1:2013(E)* (Sept. 2013), pp. 1–64. DOI: 10.1109/IEEESTD.2013.6588537.

[5] D. Marijan, A. Gotlieb, and S. Sen. "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study". In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 540–543.

[6] Raphael Noemmer and Roman Haas. "An Evaluation of Test Suite Minimization Techniques". In: Dec. 2019, pp. 51–66. ISBN: 978-3-030-35509-8. DOI: 10.1007/978-3-030-35510-4_4.

[7] Kristen R. Walcott et al. "TimeAware Test Suite Prioritization". In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ISSTA '06. Portland, Maine, USA: Association for Computing Machinery, 2006, pp. 1–12. ISBN: 1595932631. DOI: 10.1145/1146238.1146240. URL: https://doi.org/10.1145/1146238.1146240.

[8] S. Yoo and M. Harman. "Regression Testing Minimization, Selection and Prioritization: A Survey". In: *Softw. Test. Verif. Reliab.* 22.2 (Mar. 2012), pp. 67–120. ISSN: 0960-0833. DOI: 10.1002/stv.430. URL: https://doi.org/10.1002/stv.430.