# Integrating Reuse into the Rapid, Continuous Software Engineering Cycle through Test-Driven Search

Marcus Kessel
kessel@informatik.uni-mannheim.de

Colin Atkinson
atkinson@informatik.uni-mannheim.de

## ABSTRACT

Today's advanced agile practices such as Continuous Integration and Test-Driven Development support a wide range of software development activities to facilitate the rapid delivery of high-quality software. However, the reuse of pre-existing, third-party software components is not one of them. Software reuse is still primarily perceived as a time-consuming, unsystematic and ultimately, "discontinuous" activity even though it aims to deliver the same basic benefits as continuous software engineering – namely, a reduction in the time and effort taken to deliver quality software. However, the increasingly central role of testing in continuous software engineering offers a way of addressing this problem by exploiting the new generation of test-driven search engines that can harvest components based on tests. This search technology not only exploits artifacts that have already been created as part of the continuous testing process to harvest components, it returns results that have a high likelihood of being fit for purpose and thus of being worth reusing. In this paper, we propose to augment continuous software engineering with the *rapid, continuous reuse* of software code units by integrating the test-driven mining of software artifact repositories into the continuous integration process. More specifically, we propose to use tests written as part of the Test-First Development approach to perform test-driven searches for matching functionality while developers are working on their normal development activities. We discuss the idea of rapid, continuous code reuse based on recent advances in our test-driven search platform and elaborate on scenarios for its application in the future.

## CCS CONCEPTS

• **Software and its engineering → Reusability**; **Agile software development**; *Software verification and validation*;

## KEYWORDS

Rapid Continuous Integration, Rapid Continuous Code Reuse, Test-Driven Search, Test-Driven Development, Test-Driven Reuse

**ACM Reference Format:**
Marcus Kessel and Colin Atkinson. 2018. Integrating Reuse into the Rapid, Continuous Software Engineering Cycle through Test-Driven Search. In *RCoSE'18: RCoSE'18:IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering , May 29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3194760.3194761


*RCoSE'18, May 29, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5745-6/18/05…$15.00
https://doi.org/10.1145/3194760.3194761

## 1 INTRODUCTION

Ever since its inception, the basic objective of software engineering has been to support the delivery of high-quality software in short time frames with a minimum amount of effort. In the early days, software reuse [11] was regarded as a key way of achieving these objectives [9] by allowing the work invested in already existing high-quality software artifacts to be re-exploited. However, although preplanned, anticipated approaches to software reuse such as software product line engineering (SPLE) have received a lot of attention [2], unanticipated reuse approaches such as ad hoc [16] and/or pragmatic reuse are still perceived as being unsystematic [12] and time consuming [4] (i.e. leading to high risk and/or low return on investment).

In contrast to software reuse, continuous software engineering facilitates the efficient delivery of high-quality software through a variety of agile practices [6] that emphasize short, iterative and incremental development cycles. However, these practices primarily focus on developing software "from-scratch" rather than on attempting to exploit pre-existing components so all the knowledge and data wrapped up in them is lost for future projects. This is very unfortunate since the domain knowledge contained in privately and publicly owned software repositories, especially the data produced by agile practices such as continuous integration [10] and test-driven development [5], provides potential for promoting unanticipated, copy-and-modify reuse as an alternative to from-scratch development in rapid, continuous software engineering. Even if existing software components are not incorporated "as is" into a new application, the architectural "know how" that went into creating them can greatly improve the initial design of new components, and thus reduce the amount of refactoring work that has to be done to correct them. Any technique that can reduce the amount of refactoring performed in agile projects can significantly increase their efficiency.

We believe the increasing use of tests to automate continuous integration [5, 10] systems can be exploited by advances in test-driven search technology [13] to support the *rapid, continuous reuse of software components*. In its basic form, a test-driven search engine takes one or more tests as input and returns all the candidate components that pass all the given test cases. Since in general, tests serve as "executable specifications" of desired functional behavior, they can be used to retrieve existing, high-quality functionality from internal and/or external software repositories for inclusion in the system in the continuous integration process.

The purpose of this position paper is twofold. The first is to make the case that a key enabling technology for achieving the scalable, rapid, continuous reuse of software code units in agile development projects is now available (i.e. test-driven search engines). The second is to point out that reuse and continuous integration are synergistic and can be used to effectively complement each other.

To support this position, we first present some realistic scenarios for rapid reuse based on our past experience with our test-driven search platform MEROBASE [14]. Second, we explain some of the core challenges involved in making rapid, continuous software reuse practicable. The remainder of the paper is organised as follows. Section 2 introduces our terminology and assumptions and explains the notion of rapid, continuous reuse in the context of test-driven search. Then, Section 3 introduces and discusses some envisioned scenarios. Finally, Section 4 gives an overview of challenges involved in rapid continuous reuse, while Section 5 presents the conclusion and final remarks.

## 2 RAPID, CONTINUOUS REUSE

In the following we refer to testable software code units as software components. Since our proposal leverages the rich data and code artifacts generated by the software testing activities in agile practices (i.e test-driven development [5]), our definition of a "software component" draws upon the notion of testability (i.e., executability). More specifically, we define a software component in the context of software testing according to Ammann and Offutt [1] as follows. A software component is "[...] a piece of a program that can be tested independently of the complete program or system. Thus classes, modules, methods, packages, and even code fragments can be considered to be components".

### 2.1 Definition

Informally, we define the rapid, continuous reuse of software components as follows –

*Definition 2.1. Rapid, continuous reuse of software components* exploits test cases, written as part of a typical test-driven, agile development process, to serve as queries to test-driven search engines (i.e. definitions of required functionality) for the purpose of recommending reusable candidates.

We use the term "executable specification" to refer to the desired functional requirements, i.e desired functional behavior encoded in test cases[1]. Additional constraints that are put on the functional behavior such as non-functional requirements (i.e non-functional qualities such as security, performance etc.), can also be assessed by the testing activity. Since there is no hard and fast rule for distinguishing functional and non-functional properties of software, we assume that both are verifiable in the testing process.

Today, many software testing activities revolve around the idea of test-driven development [5] which is usually practiced as part of continuous integration. Developers write (unit) test cases in order to verify a piece of the software's functional behavior. Since developers are supposed to integrate their code and tests frequently into the shared code base [10] (i.e a software repository under version control), tests are executed in a repeated, usually automated manner, using such things as (reusable) regression test suites. Regression testing not only supports the continuous functional verification of written code, but also "guides" the verification of code refactoring steps to improve quality.

Test-first development practices [6], in particular, go beyond test automation and "force" developers to break down the system's functional requirements into testable pieces even before the code is written. Since agile projects already encourage developers to encode functional requirements into (unit) tests (i.e executable specifications), and also to employ test-first development where tests are written before application code exists, the ingredients for powerful and efficient (semi-)automated test-driven component reuse scenarios already exist. Moreover, it means that the upfront costs of realizing test-driven reuse scenarios are very low.

This is where test-driven search technology [13] such as Reiss' S6 [18] and MEROBASE [14] play a key role. These search engines were specifically developed to support "semantic" component searches by using tests to characterize the desired functionality for component discrimination rather than defect detection. These engines take a test suite as their input and return semantically (i.e. functionally) matching software components mined from software repositories. Since the tests are readily available and continuously executed in test-driven development, test-driven search engines can be integrated into the continuous integration chain with little effort.

Moreover, thanks to recent advances in signature matching [20] and automated adaptation strategies [7] test-driven search engines are now also able to perform efficient semantic searches over large scale component repositories. If supplied with suitable tests they can offer much higher precision than "classic" code search engines which are primarily based on (static) textual similarity matching (i.e natural language processing) [19].

Considerable effort has been invested in the past to build efficient and scalable build systems/platforms that provide a high degree of automation in the continuous integration and software testing processes. These systems not only continually build and test new software versions, they also offer a form of quality assurance reporting features (e.g., static code analyses like code smell detection or dynamic code analyses like code coverage). This is a rich source of data [8] that can be exploited with no further costs to support a developer's decision-making process when reusing components (i.e judging the quality of the recommended components based on the provided quality reports). As well as supporting functional discrimination, test-driven search engines can also take advantage of such quality reports to discriminate between software components based on non-functional criteria as well. In our past work [15], we have demonstrated that the complex requirements to be satisfied by reuse candidates can be represented using a preference-based ranking schema based on measurable objectives. Such an approach allows developers to encode their desired functional and non-functional properties in a goal-oriented way [3]. For instance, measurable objectives can be inferred from sources like code guidelines, requirements etc.

Finally, the suitability of a reused component can be continuously evaluated using the rapid feedback obtained from the short, continuous integration and test-driven development cycles. This ability can help correct suboptimal reuse decisions, on the one hand, and facilitate the accommodation of future (i.e. unforeseen) changes to the system's requirements, on the other hand.

## 3 CONTINUOUS REUSE SCENARIOS

In this section we discuss some of the main ways in which test-driven search engine technologies could be integrated into continuous integration platforms to promote rapid, continuous reuse in agile development projects.

[1] note that we treat "test" and "test case" as synonyms (cf. [1])

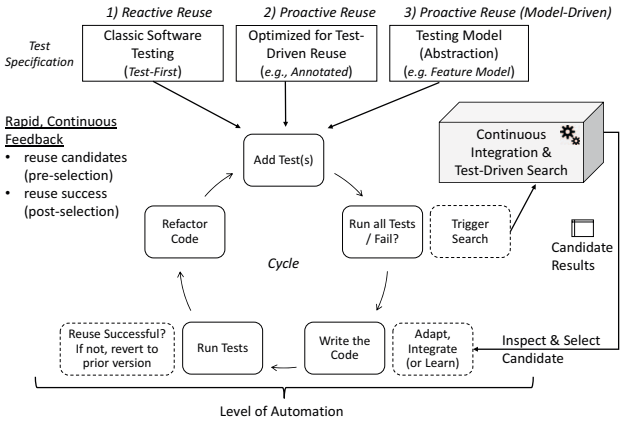


**Figure 1: Integrating Rapid, Continuous Code Reuse into Test-First Development**

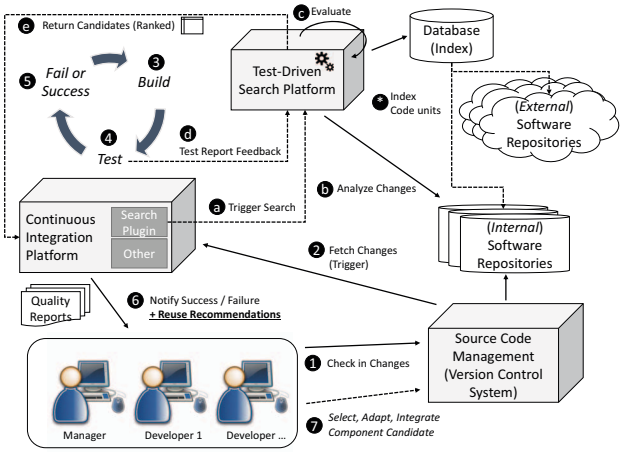


**Figure 2: Test-Driven Search and Continuous Integration**

Figure 1 illustrates the typical development cycle in test-first development. Moreover, it exemplifies the integration of rapid, continuous reuse of software components as an *alternative* to "from-scratch" development (i.e to complement the development process). The first step in the classic testing approach is to define one or more tests of desired functional behavior and then to execute them to observe their failure. When conducting "from-scratch" development, the developer continues to add only as much application code as needed to make the tests pass. This is where our reuse technology offers an additional option to developers to improve the development process. If one or more tests fail, a test-driven component search can be triggered to retrieve a set of component candidates that pass the tests. The developer is then presented with a list of high quality, recommended components which are guaranteed to provide the functionality characterized by the tests since they have passed them. Without any additional effort or divergence from his normal development activities, therefore, the developer is presented with the option of reusing one of the recommended components or of continuing to build the component from scratch. The search engine will often also offer a specialization (adaptation) of the chosen candidate's reusable code to support its integration into the developer's code base. The cycle then proceeds with the re-execution of the tests and, if necessary, the recommendation of refactoring step to further improve the quality of the code. Since the cycle is repeated many times, the success of the reuse choices can be monitored using the reporting mechanisms offered by the continuous integration system. This also provides a way of judging the impact of the reused component on the overall quality of the system (e.g., whether there is a negative impact on run time properties revealed by running the tests).

The aforementioned scenario can vary with respect to the level of automation employed and the way tests are specified by the developer. We envision three basic levels of automation. The most unobtrusive level of automation at which to integrate reuse into the development process is to let developers know about reuse opportunities without disturbing their normal development activities in any way. Reuse opportunities are thus identified automatically, and continuously, in the background as soon as the developer has checked in his code changes. We refer to this as the (fully)"reactive" approach, therefore. Assuming that new and/or modified tests appear in the shared code base, new test-driven component searches are triggered that extract the component under test and retrieve existing component candidates that pass the tests. Like other reporting mechanisms such as quality reports which are pushed to developers, developers may be notified about potential reuse opportunities in an asynchronous manner (e.g., by e-mail, once the system has been built and a search has generated a list of reuse recommendations).

Figure 2 provides a schematic illustration of the continuous integration cycle where the continuous integration platform is extended by test-driven search technology to enable the reuse of software components. To start with, on a frequent basis, developers check in the changes made to the shared code base to a source code management system (e.g., GIT, subversion etc.). The changes may include code modifications, additions and removals to the underlying system code and/or tests. Depending on the level of automation in the continuous integration platform, a new software build may be triggered either directly by new code pushes to the software repository or in a periodic manner (i.e using cron jobs, every 30 minutes for instance). In parallel to the start of a new build, new test-driven searches may be triggered based on the analysis and extraction of new test cases from the code changes. In order to enable more flexible search strategies, test-driven searches can also include further information from reports generated for the current build (e.g., test reports). Since the test-driven search platform mines software components from the internal software repository (private, shared code base) as well as from external software repositories (e.g., Open source software repositories like Github), software components can be found from a variety of sources. In case a test-driven search returns a list of matching components, a reuse report is then pushed through the continuous integration platform to the developer(s) and or manager, hence making reuse a reactive activity.

In addition to this fully reactive reuse scenario, we also envision two further levels of automation which require the developer to place some minor additional effort into supporting the component search effort. We therefore refer to these as "proactive" approaches. First, developers can explicitly mark (i.e annotate) tests to indicate that they define the functionality for components which they wish

to search for and ultimately reuse. Second, a more radical approach to proactive reuse is to further abstract from concrete, unit level test cases (e.g. test case procedures) and use more abstract representations like feature models in SPLE [2] from which executable specifications can be generated. One of the main advantages of these abstractions is that they can be used to trace features in code.

## 4 CHALLENGES

In this section we discuss some of the factors that have an important bearing on the successful introduction of the rapid, continuous reuse of software components as explained above. First, and most importantly, reuse must be a viable alternative to "from-scratch' development in continuous engineering.

*Reuse.* The reuse activity will only enhance continuous engineering if it can be applied successfully. The approach, therefore, has to support a variety of reuse scenarios. This not only includes the reuse of software components "as-is", but also includes the reuse of software components that need to be (invasively) modified and specialized for the new application in hand. Since developers usually need to satisfy both functional and non-functional requirements, they need to be able to carefully and efficiently judge a reuse candidate's quality and functional suitability [12]. We believe that the sorting (i.e ranking [15]) of components plays an important role in making the reuse-decision process more reliable and efficient.

*Principles and Policies.* Agile teams typically agree on a set of principles and responsibilities. Since in our scenario we encourage the team to check in test(s) before little if any application code has been written, the approach may violate enforced policies. For instance, even though developers are supposed to check in their changes frequently into the repository, they are usually not allowed to check in broken or untested code. One way to tackle this non-compliance is to rely on code markings in terms of code annotations or test profiles in conjunction with code skeletons (cf. proactive reuse scenario). Test profiles are already used today to instruct the system to ignore certain tests in various stages of a build. Alternatively, code annotations can explicitly mark tests for reuse scenarios or to trigger the use of code generators to generate stubs (like mocks in unit testing) for missing system code in order to avoid broken code.

*Strategies.* The next challenge is the development of scalable and efficient strategies for initiating test-driven searches. Frequent triggers play an important role in today's continuous integration systems, but high-frequency triggering comes at the cost of higher computing resources. Efficient search strategies are therefore required that only initiate test-driven searches once new test cases have been identified. Since developers typically have different practices regarding the frequency of code changes, the development of good strategies is challenging. Similarly, the analysis and extraction of software components and new tests from code changes is non-trivial as well. Even though our own test-driven search platform MEROBASE employs a set of heuristics based on today's best test case specification practices to determine the component under test, we need more robust and efficient strategies that can be applied in continuous integration environments to keep the additional test-driven search "testing" load to a minimum. More specifically, we need efficient ways to judge which classes and methods are actually tested. Varying strategies in the design of test cases make it even harder to derive information about what is actually tested by each test case. Recent advances in mining software repositories, for instance, may solve the problem through the mining of historical (i.e. evolutionary) information from version control systems [17].

## 5 CONCLUSION

In this paper we have motivated and introduced the idea of rapid, continuous reuse of software components – an approach that aims to exploit the tests constructed in continuous software engineering activities to unobtrusively retrieve and recommend software components for reuse. In particular we have argued that test-driven search engines have now reached a level of maturity that makes rapid, continuous reuse a viable technology. When fully integrated into the agile life-cycle, as a continual and frequently performed activity, test-driven search can offer various attractive alternatives to developing components from scratch. As well as describing some of these reuse scenarios, we have described how test-driven search platforms can be integrated into existing continuous integration platforms. As part of our ongoing work we plan to integrate our test-driven search engine into a leading open source continuous development platform and explore scenarios by which software reuse, especially the reuse of unanticipated software components mined from software repositories, can become a viable alternative to "from-scratch" development of software components.

## REFERENCES


[1] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing* (1 ed.). Cambridge University Press.

[2] Sven Apel, Don Batory, Christian Kaestner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer.

[3] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. *Encyclopedia of Software Engineering.* Wiley, Chapter The Goal Question Metric Approach.

[4] Veronika Bauer and Antonio Vetro'. 2016. Comparing reuse practices in two large software-producing companies. *Journal of Systems and Software* (2016).

[5] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional.

[6] Kent Beck and Cynthia Andres. 2004. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional.

[7] Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, and Massimo Tivoli. 2006. Towards an Engineering Approach to Component Adaptation. Springer, 193–215.

[8] M. Beller, G. Gousios, and A. Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *IEEE/ACM MSR'17.* 447–450.

[9] T. Biggerstaff and C. Richter. 1987. Reusability Framework, Assessment, and Directions. *IEEE Software* 4, 2 (Mar 1987), 41–49. Copyright - Copyright IEEE Computer Society Mar/Apr 1987; Last updated - 2014-05-17; CODEN - IESOEG.

[10] Martin Fowler. 2006. Continuous Integration. (2006). https://martinfowler.com/articles/continuousIntegration.html (accessed 2018-01-20).

[11] W. B. Frakes and Kyo Kang. 2005. Software reuse research: status and future. *IEEE Transactions on Software Engineering* 31, 7 (July 2005), 529–536.

[12] Reid Holmes and Robert J. Walker. 2013. Systematizing Pragmatic Software Reuse. *ACM Trans. Softw. Eng. Methodol.* 21, 4, Article 20 (Feb. 2013), 44 pages.

[13] Oliver Hummel and Werner Janjic. 2013. *Test-Driven Reuse: Key to Improving Precision of Search Engines for Software Reuse.* Springer New York, 227–250.

[14] O. Hummel, W. Janjic, and C. Atkinson. 2008. Code Conjurer: Pulling Reusable Software out of Thin Air. *IEEE Software* 25, 5 (Sept 2008), 45–52.

[15] Marcus Kessel and Colin Atkinson. 2016. Ranking software components for reuse based on non-functional properties. *Information Systems Frontiers* 18, 5 (01 Oct 2016), 825–853.

[16] Charles W. Krueger. 1992. Software Reuse. *ACM Comput. Surv.* 24, 2 (June 1992), 131–183.

[17] Y. Li, C. Zhu, J. Rubin, and M. Chechik. 2018. Semantic Slicing of Software Version Histories. *IEEE Transactions on Software Engineering* 44, 2 (Feb 2018), 182–201.

[18] Steven P. Reiss. 2009. Semantics-based Code Search. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09).* IEEE, 243–253.

[19] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. 2014. *Recommendation Systems in Software Engineering.* Springer.

[20] Amy Moormann Zaremski and Jeannette M. Wing. 1995. Signature Matching: A Tool for Using Software Libraries. *ACM Trans. Softw. Eng. Methodol.* 4, 2 (1995).