Chapter 14
Reuse-Oriented Code Recommendation Systems

Werner Janjic, Oliver Hummel, and Colin Atkinson

Abstract Effective software reuse has long been regarded as an important foundation for a more engineering-like approach to software development. Proactive recommendation systems that have the ability to unobtrusively suggest immediately applicable reuse opportunities can become a crucial step toward realizing this goal and making reuse more practical. This chapter focuses on tools that support reuse through the recommendation of source code—*reuse-oriented code recommendation systems* (ROCR). These support a large variety of common code reuse approaches from the copy-and-paste metaphor to other techniques such as automatically generating code using the knowledge gained by mining source code repositories. In this chapter, we discuss the foundations of software search and reuse, provide an overview of the main characteristics of ROCR systems, and describe how they can be built.

14.1 Introduction

Although the idea of software reuse is not new, it has yet to take off in practice. The basic problem is that the perceived benefits of systematic software reuse still do not clearly outweigh the effort, risks, and uncertainties involved. Developers are faced with the dilemma of whether to first create a detailed system design and then try to find matching coarse-grained components relatively late in the development process or to invest a great deal of effort discovering what components already

W. Janjic (✉) • C. Atkinson
Software-Engineering Group, University of Mannheim, Mannheim, Germany
e-mail: werner.janjic@informatik.uni-mannheim.de; atkinson@informatik.uni-mannheim.de

O. Hummel
Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology, Karlsruhe, Germany
e-mail: hummel@kit.edu

M.P. Robillard et al. (eds.), *Recommendation Systems in Software Engineering*,
DOI 10.1007/978-3-642-45135-5_14,

exist and then try to tailor and combine them to meet the requirements. In either case, it is not always certain that all system requirements can be fulfilled and that something reusable can actually be found. It is therefore no surprise that the reuse approaches that have recently gained the most attention are *pragmatic reuse* approaches [10] that focus on the non-preplanned reuse of source code assets mainly during implementation.

Regardless of the exact motivation for reuse, researchers and practitioners have traditionally faced three main obstacles to implementing an effective reuse program for mainstream software development:

- The *repository problem* [8, 29], that is, where to find a sufficient amount of reusable material
- The *representation problem* [9], that is, how to optimally store and represent the reusable material
- The *retrieval problem* [25], that is, how to formulate and execute queries for a repository in a simple and precise manner

A great deal of progress has been made in all these areas recently (partly due to the open source "revolution" that made literally millions of potentially reusable files freely available), and new solutions to these problems laid the foundation for a new generation of *internet-scale software search engines*.

Although software search engines are an essential prerequisite for reuse recommendation tools, in their simple (mostly web-based) form they cannot be regarded as recommendation engines since they will only retrieve exactly what they are asked to retrieve. Thus, they are a necessary but not sufficient part of the whole solution; additional features would be needed to move into the realm of practical, large-scale software reuse. An ideal reuse recommendation engine would automate the whole process of searching, adapting, and evaluating reuse candidates as well as validating that they seamlessly integrate into the application under development. Obviously, in general this process becomes more challenging the larger and more complex the reuse candidates.

The main obstacle to software reuse is no longer the lack of components to reuse or the ability to retrieve them efficiently. Many projects have shown that this is feasible with modern technology [3, 11, 14, 27]. The main obstacle is rather the balance between the effort required to evaluate and incorporate components into new applications and the likely benefit (including the risk that a reuse candidate will turn out to be unsuitable). This is where code recommendation tools can come in handy. Their role is to nonintrusively and reliably find and recommend high quality code artifacts leveraging software reuse and to help developers integrate them into their systems with minimal effort.

There are different forms of recommendation system supporting different services and use cases involved in software reuse. Nevertheless, based on the generic definition of a recommendation system from Robillard et al. [28], we can define a

reuse-oriented code recommendation system (ROCR[1]) as *any tool that recommends code artifacts of any kind and size for the sake of supporting reuse tasks*. In general, ROCRs are assistant tools for developers, which are seamlessly integrated into the developers' software development process and environment. Based on observations of the pros and cons of example ROCRs, we can identify a minimum set of requirements that have to be met by modern code recommendation tools to make code reuse more convenient. These "best practices" should be standard features of ROCR systems as they contribute to higher acceptance of such systems among users.

Proactive recommendation systems that unobtrusively suggest code with a high likelihood of being beneficial in a given situation provide a promising way of supporting reuse in the implementation phase of software development projects. The artifacts recommended by such systems need not just be functional production code but can include all different kinds of executable software used in the lifecycle of a project such as tests, prototypes, frameworks, libraries, or small code snippets that can be retrieved, recommended, and reused. Furthermore, a code artifact can be reused in different ways ranging from direct inclusion in a new software product to using it as an oracle to drive the software testing process [2, 17].

The remainder of this chapter discusses opportunities, challenges, and techniques associated with the creation of ROCR systems and describes the basic technologies needed to build such a system. Many examples of ROCRs have been produced, including Code Finder [8], CodeBroker [32], Strathcona [11], Prospector [22], Code Genie [21], Code Conjurer [15], and Code Recommenders [5]. Surveying all such tools is beyond the scope of this chapter; two archetypal examples (Strathcona and Code Conjurer) are outlined in Sect. 14.2. Section 14.3 describes the process of software reuse as well as the basic characteristics and range of different search services that state-of-the-art code search engines can provide. Section 14.4 takes a closer look at different forms of code-related reuse that provides the motivation for variants of code recommendation systems with different foci. It then introduces the important characteristics of these variants along with implications for their usage. Building on the provided foundations, Sect. 14.5 focuses on the implementation of a recommendation system using the open source tool Code Conjurer [14] to provide concrete examples for the discussed aspects. Finally, a discussion and some thoughts on the future of reuse recommendation technology—emphasizing open issues and current developments—are presented in the last two sections.

14.2 Introductory Examples

To provide an intuitive introduction to the subject of this chapter, we present two ROCR systems by briefly describing their background and characterizing their features. More detailed information on each of them can be found in the provided

[1]The acronym ROCR is meant to be pronounced "rocker."

```
public class MyClass {
  public CompilationUnit createASTFromSource(String source) {
    ASTParser.setSource(source.toCharArray());
  }
}
```

Listing 14.1 Example skeleton used to query Strathcona

literature sources. The first one, Strathcona, is a recommendation system that suggests examples of actual usage scenarios based on information extracted from existing software components, while the second, Code Conjurer, is a "classic" reuse tool that recommends reusable code in a copy-and-paste manner, leveraging a particular code search engine (Merobase).

14.2.1 Code Recommendation for API Usage with Strathcona

Strathcona is an *example recommendation tool* [11]. Instead of following the established source code reuse approaches that target component reuse, the Strathcona recommendation system focuses on the lack of documentation accompanying the wide variety of frameworks and software libraries that are used in modern software systems. The example recommender assists users by recommending usage and invocation examples relevant to the developer's context without imposing new hurdles for users such as learning a new query language. It achieves this by extracting all necessary search parameters directly from the developer's code. An illustrating example that is familiar to most developers who work within the Eclipse IDE is the question of how to create an abstract syntax tree (AST) from a piece of source code. A first quick look at the documentation for the application programming interface (API) provided by Eclipse suggests that the `setSource(...)` method of the `ASTParser` class would be helpful in achieving this goal, resulting in the developer trying to write an implementation like that in Listing 14.1.

However, the documentation does not describe the three steps necessary to complete the task, namely: (1) the parser needs to be created by using a factory method, (2) the parser needs to be made aware of the source code, and (3) the AST has to be created. Strathcona's client (provided as a plugin for Eclipse) will extract the structure of the developer context to identify the class, its parents, method calls, and possibly existing field declarations to form a query for its backend, where different matching heuristics can be applied. The server looks up possible example recommendations and returns the top ten examples to the recommender client; Listing 14.2 gives an example (shown in the source view, one of several presentations provided by Strathcona). The examples serve both to solve the developer's immediate problem, and to provide context about additional issues about which they may be unaware (like the possibility of setting preferences on generating bindings or on fault tolerance, as in this example).

```
public class ASTResolving {
  public static CompilationUnit createQuickFixAST(
      ICompilationUnit compilationUnit, IProgressMonitor monitor)
       {
    ASTParser astParser = ASTParser.newParser(ASTProvider.
        SHARED_AST_LEVEL);
    astParser.setSource(compilationUnit);
    astParser.setResolveBindings(true);
    astParser.setStatementsRecovery(ASTProvider.
        SHARED_AST_STATEMENT_RECOVERY);
    astParser.setBindingsRecovery(ASTProvider.
        SHARED_BINDING_RECOVERY);
    return (CompilationUnit) astParser.createAST(monitor);
  }
}
```

Listing 14.2 Example result (source view without highlighting) delivered by Strathcona

14.2.2 Code Reuse with Code Conjurer

Many developers experience the feeling when implementing a piece of code that "this must have been already implemented by someone else." It is certainly possible to find existing implementations of frequently used components by using a web-based search engine, but this is typically a haphazard process that disturbs the natural workflow of developers and requires the explicit cognitive decision to search for reusable artifacts. In most cases, attempts to use "raw" code search engines either lead to frustration because nothing reusable can be found or to a decrease in productivity since searches take too much time. Moreover, developers often miss possible reuse opportunities because they did not expect reusable artifacts to be available. To support this "classic" code reuse scenario, the Code Conjurer recommendation system nonintrusively suggests reusable artifacts by examining developers' code and autonomously querying the Merobase code search engine for results [14]. As an example, imagine a developer writing a simple text editor that requires the contents of a file to be loaded into a string object and the changes to be written back to the file. Code Conjurer can help to find a routine that does all of this based on the method declarations in the source code. For example, solely relying on the information in Listing 14.3, Code Conjurer will autonomously query Merobase for reusable artifacts without the user noticing.

Upon receiving the search results, the Code Conjurer client will then present the developer with results of the kind shown in Listing 14.4.

This code recommendation may be directly reused in the developer's project via a simple drag-and-drop action, thus imposing no additional effort on the developer related to searching and reusing code. In other words, Code Conjurer seamlessly integrates code reuse into the "natural" workflow of software developers. In addition to that, Code Conjurer integrates reuse with test-driven development: if developers

```
public class TextDocument {
  public String loadFile(String filename) {
  }

  public void saveFile(String filename) {
  }
}
```

Listing 14.3 Example class stub used by Code Conjurer for code recommendation

```
private String loadFile(String fName) throws Exception {
   FileReader fr = new FileReader(fName);
   BufferedReader br = new BufferedReader(fr);
   StringBuffer sb = new StringBuffer();
   String line;

   while ((line = br.readLine()) != null) {
      sb.ppend(line);
   }
   br.close();
   fr.close();

   return sb.toString();
}
```

Listing 14.4 Example result delivered by Code Conjurer

write JUnit tests before they write the actual code, Code Conjurer is able to find reusable assets that fulfill the requirements manifested in the test cases and therewith to recommend semantically matching components.

14.3 Foundations

Before a ROCR can be beneficial and reuse can actually be carried out, it is necessary to have a critical mass of potentially reusable artifacts. These artifacts need to be mined for interesting information and an effective search-engine that can efficiently support searches needs to be built. But why do developers actually want to find something reusable? The essential motivation for *software search* and *reuse* is clearly captured by a frequently-cited quotation from Krueger [19] who was strongly opposed to the continuous "reinvention of the wheel" in software development:

> Software reuse is the process of creating software systems from existing software rather than building software systems from scratch. [...] Simply stated, software reuse is using existing software artifacts during the construction of a new software system.

This simple vision was built upon the suggestions made even earlier by McIlroy [23], which are often regarded as the starting point for research in the area of software reuse. Since software reuse depends on the ability to discover reusable artifacts, it is necessary to take a closer look at software search engines that form the "backend" for most ROCR systems. In the past, the development of tools designed to support reuse has usually been preceded or accompanied by the creation of a search engine focused on the particular kind of reuse to be supported. This separation of concerns helped their developers to ensure best quality in both fields: search engines focusing on optimizing the processing of search queries and client systems providing a convenient way for users to benefit from this functionality within their development environments.

Before taking a closer look at the whys and hows of ROCRs, the following section covers the basic foundations on code reuse itself as well as some basic knowledge on software search engines, as a prerequisite for the creation of ROCR systems. In this context, the term *software search engine* is used in a broader sense than for just plain source code search since there are different categories/variants of ROCR systems and not all of them focus purely on code; some also provide automatically generated code recommendations based on the use of sophisticated data mining techniques to harvest the knowledge embedded in existing code.

14.3.1 Software Reuse Process

In the literature, there are numerous publications dealing with software reuse, its foundations, and possible improvements. For example, de Almeida et al. [7] define a comprehensive framework that cleanly describes the key ingredients for software reuse in general. Besides the need for a repository and search infrastructure, they describe a generic software reuse process and various best practices for effective software reuse. As with classical software development, for effective software reuse it is necessary to have a specification of *what* should be built or reused as it forms the foundation for a query to the search backend that looks for reusable candidates. The simplest way to do this in a search engine is the "Google approach" of looking for keywords like "getDistance int" to find a distance calculator, for example. Nevertheless, such a simple hand-crafted query does not convey much information beyond the meaning of names and will most probably lead to poor (i.e., rather imprecise) results [13].

Thus, it is necessary to improve and enrich pure name-matching with additional information from the context of the environment in which the reused asset should be integrated. An overview of the improvements in precision that can be achieved with enhanced query formulation is depicted in Table 14.1, which compares four textual software search techniques. The table illustrates that all techniques except the interface-based search deliver a large number of false-positives. Therefore, they make it hard for developers to identify concretely reusable candidates without the additional effort of examining many useless ones.

Table 14.1 Precision of code retrieval techniques [13]

	signature matching	keyword-based	name-based	interface-based
average precision	0.9%	16.3%	17.2%	53.7%
standard deviation	1.8%	21.9%	19.3%	22.4%

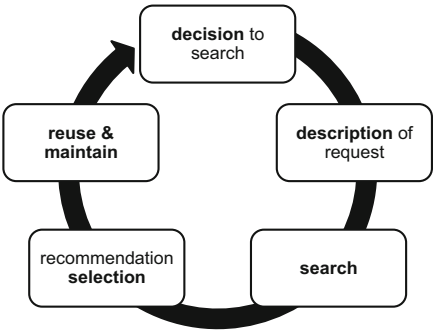

Fig. 14.1 Overview of the microprocess of software reuse. Ideally a software reuse action is followed by a new one

When the search returns a set of candidate results, these are usually not directly *fit for purpose* for various reasons such as missing dependencies or API mismatches. Thus, the process of software reuse involves their examination as well as possibly their reengineering and adaptation to support seamless integration into the developers' software projects. By reusing a previously created piece of code, the lifecycle of the reused asset is tied to that of the whole project. In other words, the incorporated code is subject to modifications or refactoring within its new environment and tests may reveal issue. All these aspects need to be reflected within a ROCR system that ideally supports the full automation of this process as well as the responses to developers' inputs.

A simplified representation of the microprocess of software reuse is depicted in Fig. 14.1. The process itself is generic and applies to manual as well as tool-supported software search and reuse. The particular elements of this process are as follows:

Decision. During a software project, developers decide to actively search for a reusable asset. Therefore, they need to decide what kind of asset they want to reuse. The different kinds of search scenarios/assets that users search for will be described in the subsequent section.

Description. Once a developer has decided to look for reusable assets, a clear description of *what* should be reused needs to be created. This specification ideally should comprise all required information that is necessary to find useful reusable assets.

Search. The description serves as the query to a search engine. Sophisticated algorithms should be able to automatically refine and adapt queries in order to filter out all useless artifacts and ensure that no useful ones are missed. This is almost impossible without tool support, as it would consume a lot of time to

create a query, inspect the results, refine and re-issue the query, etc. This cycle may have to be repeated several times and is obviously not very efficient when done manually.

Selection. From the "raw" set of search results, the developer needs to choose whether any of the results are useful and if there are different candidates that fulfill the given criteria. In that case, the developer has to select the best match from the list, which can be a very tedious task since it may involve trial uses of a large number of possible candidates. If this is carried out manually, it involves to copy of the code from the search engine, look for necessary dependencies, eventually adapt the provided interface of a reused class and finally try it out. This must be performed for every candidate in order to find the best matching one.

Reuse and Maintain. Once a candidate has been selected for reuse and integrated into the developer's system, the microprocess of code reuse is completed. Nevertheless, the reused candidates are now part of the developer's project development lifecycle and should be subject to all the same actions and processes as the other parts of the system like testing and maintenance.

Although the microprocess of reuse is complete, Fig. 14.1 reflects that reuse should not be a one-off event but should rather be continuously applied throughout project development [e.g., 19].

14.3.2 Software Search

A recommendation system's ability to provide reusable code assets to the developer is mainly based on a repository of previously written code, which has been indexed and made efficiently searchable. In the past, there have been many commercial and scientific attempts to provide web-based search engines for code. Examples include Google Code Search, Koders, Krugle, Sourcerer, and Merobase. However, none of them ever reported significant numbers of users comparable to mainstream search engines. In fact, Google even shut down their code search engine in 2012, clearly illustrating that developers need some other form of support for code reuse. This is where ROCR systems become an interesting alternative to web-based search engines as they offer a large range of potential usage scenarios that are very similar to the archetypal usage scenarios of software search described by Janjic et al. [17].

Detailed understandings of the different use cases for software search have only emerged recently through studies and online surveys such as those described by Umarji et al. [30]. A prominent example from this survey is the use of search engines to provide guidance in the use of libraries—a topic that led to the creation of a couple of recommendation systems that have received significant research attention (like, e.g., Strathcona [11]) and their approaches inspired the official Eclipse Code Recommenders project.

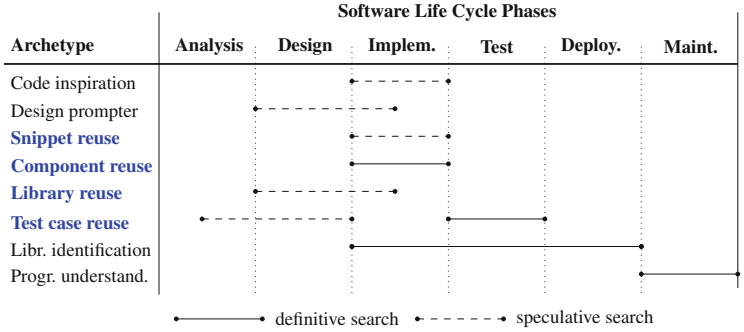

Fig. 14.2 Search scenarios in software engineering [adapted from 17]

Searches motivated by the goal of *reusing code without modification* are subject to the following four categories:

- Code snippets, wrappers or parsers
- Reusable data structures, algorithms and graphical user interface (GUI) widgets to be incorporated into an implementation
- Reusable libraries to be incorporated into an implementation
- A reusable system to be used as a starting point for an implementation

Searches motivated by the goal of *finding reference examples* are categorized by the following four categories:

- A block of code to be used as an example
- Examples for how to implement a data structure, an algorithm or a GUI widget
- Examples for how to use a library
- Looking at similar systems for ideas and inspiration

Figure 14.2 visualizes the eight archetypal search scenarios assigned to the traditional software development life cycle. Searches are grouped into speculative or definitive searches, represented by dashed or solid lines respectively. While the former are likely to occur early during the software development process, giving users an idea about how to solve particular tasks, the latter are more likely to occur late in the design and implementation phases when a concrete specification of a required component is typically available. Since our focus is source code recommendation, the tools presented in this chapter focus on recommending artifacts originating from the following four archetypal usage-scenarios of software search:

1. Snippet reuse
2. Component reuse
3. Library reuse
4. Test case reuse

Having clarified the motivation for software searches and having described concrete use cases in which they are typically applied, we deal with the characteristics of ROCRs in general in the next section.

14.4 Common Characteristics of ROCRs

In this section, we discuss common characteristics of ROCR systems. These criteria were largely distilled from previous research work capturing the "best practices" that should be considered for newly built ROCR systems. It is important to emphasize that the characteristics presented in this section are valid for all the different categories of ROCR systems that we introduce hereafter.

14.4.1 Use Case Characteristics

The term software reuse is usually associated with the integration of existing software (i.e., code) into a project under development utilizing a copy-and-paste approach to reuse [20]. This is also known as *code scavenging* when contiguous blocks of source code are copied to the new system [19]. The underlying goal of these techniques, which are known by different names and are subsumed under the term *pragmatic reuse* [10], is to copy as much code as possible from already existing projects. However, this is not the only kind of reuse that is possible. There are many other forms of software reuse like *design scavenging*, where large blocks of code are reused and subject to major internal changes. This diversity in motivation for reuse leads to different varieties of ROCR systems. ROCR systems were designed to support other forms of reuse than just to copy pre-existing code. For example, some systems recommend automatically created code fragments by leveraging knowledge from pre-existing source code or other software artifacts.

Component Reuse

The most obvious use case for a ROCR system is to present previously written code assets to developers. These artifacts may have different levels of granularity ranging from code snippets, methods, and classes up to whole subsystems and systems. A well known member of this family is Code Conjurer, which offers developers the possibility to find reusable code artifacts from the Merobase component finder [18]. When using this Eclipse plugin in its proactive mode, developers are offered suggestions for reusable methods and classes that fit into their programming context; they can simply drag-and-drop the best match into their project. By offering the possibility of automatic dependency resolution, where classes are accompanied by those classes that they make use of (e.g., by instantiation or method invocation), Code Conjurer even offers the automated reuse of (smaller) systems, which we call components in the sense of component-based software development [1].

Library Reuse

Especially within object-oriented development projects, developers constantly utilize prefabricated building blocks provided in the form of libraries by invoking some of their functionality. This is very convenient at first sight, since libraries form a cohesive piece of software that usually incorporates a lot of reusable objects with their dependencies. Although they can make the development of new software much easier, there are, however, numerous obstacles to their usage that every developer experiences on a regular basis. Questions like "how is this library used," "which objects do I need," "how are they created," and "what sequence of calls do I have to make" arise almost every time a new framework, API, or library is used. Tools like Strathcona [11] or Prospector [22] explicitly address this problem by recommending code snippets that show examples of how libraries can be used or which call sequence is necessary to transform an object from one into another type (e.g., a `File` into an `AbstractSyntaxTree`).

Test Case Reuse

Modern code search engines index vast quantities and varieties of reusable code artifacts. This also includes a large number of test cases along with production code. For instance, JUnit tests are written as plain Java code and can even be built into and shipped with a component. This opens up another form of code recommendation—the recommendation of test code for a newly created system. Appropriate recommendation techniques for JUnit test code were introduced by Janjic and Atkinson [16]. They focus on predicting the best possible "next test" based on a repository of previously written test cases and the knowledge extracted from them. Such systems do not recommend reusable code per se, but generate reusable test code by assembling the previously analyzed knowledge bound up in existing tests and their accompanied production code.

14.4.2 Design Characteristics

Building ROCR systems is a challenging task. Users are sensitive to the usability of such systems and the quality of the recommendations they provide. If ROCR systems do not work or behave the way that users expect them to, if they start to annoy developers with too many suggestions—especially if these are useless or incorrect—they can quickly get deactivated or uninstalled. To name an example, a "Clippy-style" intrusive user-interface will most likely cause users to dislike even the best system (see Murphy-Hill and Murphy [26] in Chap. 9), since it disturbs them in their primary tasks and forces them to additional cognitive decisions combined with additional effort (even if this only means to move/click the mouse to hide an unsuitable recommendation). Therefore, the first important characteristic of a ROCR system is how these systems should be integrated into users' development environments.

Integration and Usability

An environment for code reuse (sometimes also called *software reuse environment*) [7] should ensure full integration of the reuse process into developers' personal development processes and IDEs. To be successful, the microprocess of code reuse, which comprises similar tasks to classic software development, has to be non-intrusively adopted and integrated into the development process of the users' software projects. For a ROCR system this means that it should be unnoticeable to the developers unless it has something useful to recommend. And even in the case that the system can be helpful it must make its recommendation as clearly, concisely, and unobtrusively as possible. ROCR systems must also make it easy for developers to reject the recommendation and continue their work with no additional effort should they decide that the suggested recommendations are not of interest.

In Chap. 9, Murphy-Hill and Murphy [26] present the general characteristics of recommendation systems' user interfaces (UIs) in more detail. The characteristics presented there almost fully apply to ROCR systems as well, so we do not repeat them here.

Autonomous Background Agent

One of the key problems of web-based code search engines is that the developers usually have to leave their current working environment (i.e., the active code editor and project), which obviously interrupts their workflow. Moreover, because queries have to be defined in a completely different environment (the web-browser) without access to the immediate context of the user's work, there is very little space to formulate queries that fully match the developer's goal. In addition, developers have to understand how a search engine works to be able to formulate adequate queries that deliver precise results. And, last but not least, developers have to invest a significant amount of effort to manually evaluate and integrate reusable assets into their new applications. In particular, to try out any of the recommendations, users have to switch between (at least) two windows, and may even lose track in the process.

Reuse-oriented code recommendation systems should therefore operate in a completely automatic manner in the background, constantly monitoring the developers' actions. More specifically, an autonomous background agent process is required to observe all changes made to the system under development and to proactively decide when to trigger a search for recommendable artifacts. This should happen without any user involvement. A ROCR system may in fact be much better at timing a search than the user would be, as it can take into account different factors like network and system load, the time necessary for the creation of the recommendations, etc. If a recommendation system needs some time to examine recommendations from a list of search results for fitness for purpose or has to create the recommendations on the fly by extracting information from the search, it makes even more sense that it initiates the recommendation process at the earliest possible moment so that the recommendations are ready should the user request them.

The timing and smartness of the background agent in issuing searches and providing valuable information to the search engine is a key feature of the recommendation system. This proactive behavior therefore needs to be well designed since it plays a major role in determining how a ROCR system is perceived by its users.

Context Awareness

In order to be able to efficiently find potential recommendations, a ROCR system needs to access as much information about the development context as possible. Depending on the kind of recommendation system, context awareness may range from the immediate environment of the cursor to the source code of the whole project. As the ability to analyze context data is an important driver to the proactive behavior of a recommendation system, it should be directly embedded into a developer's working environment with full project access. This enables the aforementioned background agent to autonomously decide, when to issue searches and to deliver recommendations to the system's user.

Traditional code search engines usually offer users a small text field where they can write a short query describing the desired reusable assets. This query can either be in the form of a sequence of keywords or a sophisticated query language to provide a full description of the interface the requested assets should provide. While the former case is quite easy to use, the latter involves additional effort in the formulation of the query. As the evaluations in Table 14.1 show, keyword-based searches tend to be rather imprecise, while the interface-based ones seem to provide more precise search results. Further manual refinements of a query may make the whole searching process more time-consuming and inefficient. This can frustrate the user who, dissatisfied by this experience, might be tempted to revert to "reinventing the wheel" again.

To address this problem, context aware ROCR systems should remove the responsibility for query formulation from developers, instead performing such tasks on their behalf. In conjunction with a background agent, context awareness allows the system to perform search tasks hidden from the users' view. Depending on the kind of code recommendation system, context awareness may have different foci. One application is when a recommendation system aims at simplifying API usage of a framework or library. In this case the recommendation system is usually more interested on the immediate context of the cursor than on other classes in the developers' project. More specifically, it uses the last few lines created to issue a search based on such information as the type of a newly instantiated object (i.e., source type), method invocations associated with that object and the allocation of a method's return value to another new object (target type). Another example of the usage of context awareness can be found in Sect. 14.5 where Code Conjurer [14] is used to illustrate how code recommendation systems can be implemented.

Since the possibilities for investigating the context of the code under development are uncountable and mining it for relevant information during query formulation is a time consuming task, this process has to be carried out automatically if it is to be efficient. In Chap. 3, Menzies [24] provides further insights into the topic of data-mining.

Evaluation and Ranking

To be useful for developers, recommendation systems in general and ROCR systems in particular should not present the raw data acquired from the search backend, but should provide a meaningful ranked overview of the recommendations. The consequences of providing the user with incorrect or unordered recommendations are similar to those mentioned in the section on user-interface characteristics. Furthermore, following the ideas of Brun et al. [6], the users of a recommendation system should not be required to inspect large numbers of options before they (hopefully) find a useful asset. Therefore the IDE should autonomously perform an evaluation of the consequences of the application of the assets within the developers' context. This approach is called *speculative analysis* and it enables ROCR systems to investigate and predict the consequences of the inclusion of any of the suggested options in the developer's project.

Implementing this approach, however, only does half the work since the information obtained through speculative analysis is just a basis for ranking the recommendations. The detailed ranking criteria strongly depend on the focus of the ROCR system and need to be optimized on a domain-by-domain basis. If the reuse system is used in conjunction with a private reuse repository, it is also possible to filter out the classes and types that are in the so-called *reuse-by-memory* space of a developer and thus need not be recommended by the system [31]. This is done by CodeBroker, for instance, where the system removes recommendations well-known to the developer to save time.

Ready on Demand

The introduction to this chapter already mentioned the dilemma of *make-or-reuse* from which code search and reuse considerably suffered in the past. Yet, it is still a challenge to convince developers that this approach can make them more efficient during system development since issuing a query to a search engine typically comes at the price of interrupting their cognitive work on a program. In contrast, ROCR systems do not create this problem since they are integrated into the developers' IDE and have to be ready on demand in order to be successful. Imagine a reuse-oriented recommendation system that is tightly integrated into the code editor and which

forces the developer to stop typing while it is performing a time consuming task. This would result in immediate deactivation of the system and adoption of the *make* option described above.

To avoid such situations, ROCR systems have to be ready on demand and must not cause any delays on the developer's work. If they cannot deliver any appropriate recommendations in a particular situation, they need to stay silent and invisible.

Traceability

Recommendation systems in general and code recommendation systems in particular need to be easy to understand and self-explanatory when they are used. The aforementioned requirement of integrating the system into the developer's IDE is a key consequence of this. Thereby a tight integration means as well, that the usage, the "look-and-feel" and the behavior of the recommendation system has to be similar to what developers are used to from their IDE. No new design or usage metaphors should be imposed, as they may impose an extra hurdle to the usage of the system.

Beyond plain UI design criteria, however, it is also important that the recommendations themselves are understandable and reasonable from the developers' point of view. Recommendation systems should present their information in a clean and transparent manner so the users can clearly comprehend their value. This also involves the aforementioned ranking, where users should easily understand why an option outranks others and in the ideal case should also be able to adapt the ranking criteria to their needs. When a recommendation is finally integrated into the system under development, the system should highlight that fact and should not perform any action that cannot easily be undone and observed by users in case that the reused component needs to be removed from the system at a future point in time for unforeseen reasons.

Summary

To sum up the characteristics discussed in this section, imagine a ROCR system as an adviser that provides a developer with an easy to use interface for the most sophisticated search engines and mining tools available. It must silently monitor the developers' activities and present recommendations only when there is a high likelihood that they will actually be useful and fit into the current system in an effective way. "Less is more" is probably also an important motto for a ROCR system, since too much visible activity can easily annoy users and cause them to switch off all or part of the functionality. The subsequent section discusses how to tackle this none trivial challenge from an implementation point of view.

14.5 Implementing ROCRs

After having laid out the basic foundations and having discussed the generic characteristics of ROCR systems, this section focuses on how such systems can be implemented for, and incorporated, into modern IDEs. More specifically, it provides an overview of the architectural organization of a recommender-enhanced IDE and briefly touches on the question of which technologies can be helpful to build ROCR systems.

14.5.1 Architecture of ROCR Systems

A ROCR system is supposed to automate the micro-process of code reuse. Therefore, it is not composed of a single building block, but synthesizes various modules that need to fit together in order to create a ROCR system as illustrated in Fig. 14.3. Let us take a closer look at the individual parts of such an architecture and describe them in more detail by reviewing the process steps outlined previously in Fig. 14.1:

decision → description → search → selection → reuse and maintain

While the introduction outlined this process in general terms, the following reflects it in the specific context of ROCR systems.

Decision. A ROCR system is integrated into the IDE of the developer and includes an autonomous background agent. The system constantly monitors developer actions within the IDE and autonomously decides when it should trigger the process of searching for a recommendation.

Description. Considering the full context of the developers' projects, the ROCR system collects all relevant information that is necessary to create a description of the current task for which it aims to create a reuse recommendation. The result is the formulation of a query that can be sent to the underlying search engine.

Search. Utilizing the information gathered from the project, the search infrastructure performs a search for reusable assets. ROCR systems take into account that the results at that stage can only be regarded as raw material (i.e., candidates) for further examination in the subsequent selection process and are by no means ready to use.

Selection. The selection part of the considered microprocess can also involve a *conversion* step in the context of ROCR systems. Since these are, as discussed before, not solely focused on copy-and-paste reuse of code, in this step they may also generate code recommendations from information contained in the search results. At this stage, the ROCR system automatically evaluates the set of candidates to provide a ranked selection of recommendations. Based on the information gained from this, the system should rank the recommendations and reject those that are presumably not useful to the developer. This ensures that the

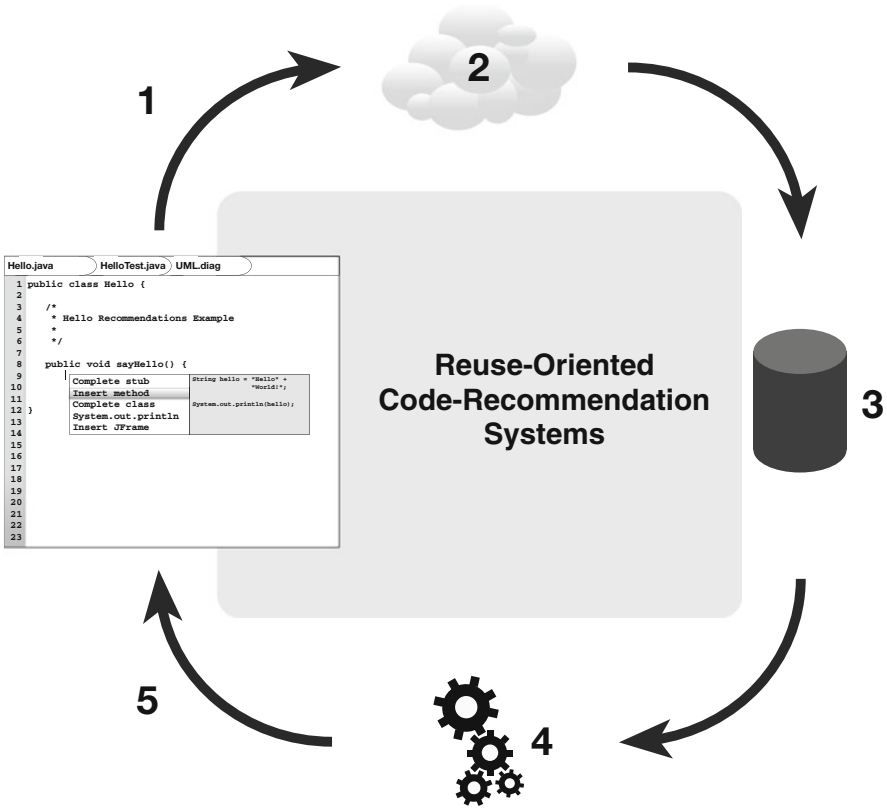

Fig. 14.3 Architectural overview on a ROCR system

users of the system get a precise list of recommendations from which they can choose the most beneficial for their purposes.

Reuse. When a recommendation is selected it becomes part of the developer's system. The recommended asset is integrated into the development lifecycle of the project and becomes subject to the same quality assurance criteria and maintenance tasks as the rest of the system under development.

This generic architecture is naturally only an outline of the specific architecture for any single ROCR system implementation tailored to a specific usage scenario. Possible refinement options may address the inclusion of user feedback on recommendations, the association of the recommendations with artifacts from the development context (e.g., for caching purposes), social aspects like reporting of users' recommendations of reused assets to other users [e.g., 4] and reporting changes to reused assets back to the original source, etc. Some of these aspects are addressed in the discussion in Sect. 14.6.

14.5.2 Implementation Outline

As with any software system, the implementation of a ROCR system must be driven by the specification of functionality the system should deliver. In our case, this is the recommendation of any form of source code that is automatically derived from previous implementations. The Eclipse plug-in *Code Conjurer* [14] therefore serves as a reference system, which supports the reuse of Java code based on interface-based or test-driven searches. The system utilizes the Merobase code search engine[2] and recommends reusable code assets that fit into the context of the class under development. Developers may choose to integrate only portions of the recommendations into their project (by dragging a method into a class) or to reuse a class with all its dependencies and, if necessary, appropriate adapters.

Background Agent for Search Initiation

The decision when to search for recommendable assets is the responsibility of the autonomous background agent that "intelligently" triggers the recommendation process. In this context, "intelligent" means that it should perform its task in a smart way in that sense that not every event, such as a keystroke, should initiate a search and drain the resources of the development system or the network and search infrastructure.

As Code Conjurer is a Java-based, code-centric tool, its background agent is merely activated when a developer is working in a Java Editor and remains inactive otherwise. If a developer changes any structural property of the class under development, such as any of its interface defining parts, the system switches to a state where it waits for further user action within a given timeframe. If no further user interaction is observed, the system initiates a search in due consideration of the development context. Therefore, Code Conjurer examines the development context looking for tests that accompany the class under development and if it finds any, it accordingly creates a query and issues a search to the search infrastructure.

Thus, the "smartness" of a background agent for ROCR systems has several facets. It must be able to construct a query that goes beyond the information a user would provide to a code search engine, issues queries autonomously (which helps to prevent delays induced by the search infrastructure), and it considers the limitation of resources by incorporating a grace timer for user interaction. This is introduced to prevent the system from initiating too many search requests one after another that would either be carried out in parallel with minimal differences in content or would have to be frequently canceled, both resulting in a waste of resources.

[2]The Merobase repository of reusable assets contains approximately 2.5 million Java source files with around 22 million methods [18].

Search Infrastructure

There is an obvious synergy between search engines and recommendation systems. As previously mentioned, on the one hand practically all ROCR systems have user interfaces, which are integrated into an IDE, but need to rely on some kind of search engine (or database) as their source of information. Code search engines on the other hand are good in delivering a large amount of information, but often require relatively complex queries that need to be manually prepared by users. Hence, simply by providing a semi-automatic and context aware way of invoking search engines, recommendation systems already improve the reuse process.

To a certain extent, the creation of search engines nowadays is a straightforward task and there are many tutorials available that describe this process in detail. Besides the use of relational databases (which, e.g., also underpinned the Sourcerer [21] code search engine), document-driven full-text databases such as Lucene or MongoDB have recently gained a lot of popularity in projects like Code Recommenders [5], Merobase [18], and Sentre [16]. Since it is very important for a recommendation system to be responsive and provide recommendations to users on an ad hoc on-demand basis, it is important that the underlying search infrastructure supports this goal.

The search infrastructure can be distributed in many different ways. If the recommendations are created from a small and static pool of data, it is possible to ship the search infrastructure with the recommendation system itself. This is, however, the least common case, since the code base from which the search indices are created usually changes rapidly (after all, it primarily consists of source code) and search infrastructure should aim to incorporate short update cycles. Therefore, it is helpful to separate the search infrastructure from the ROCR system and to locate it on a centrally maintained and operated server that provides enough resources to store the data and execute the updates as well as the searches. Additionally, another positive effect of this separation is the ability to store user feedback from the reuse process to improve the user experience of all clients.

In our exemplary implementation, Code Conjurer queries the Merobase search engine via a web service in order to receive potentially reusable code assets. Merobase itself is a web application implemented using J2EE utilizing Lucene and runs on a JBoss application server. The queries arriving at the server are translated (parsed) into the Lucene query language in order to use Lucene to drive the code search process [12]. The results of a search (which usually takes less than a second) are immediately returned to Code Conjurer for further analysis.

Selection and Ranking of Recommendations

When receiving the search results, a recommendation system should evaluate and process them before they are presented to the user. This helps to elevate the users' perception of the ROCR system and makes its application more effective and efficient. The context awareness of ROCR systems is one of the key features that

make them superior to traditional search engines by enabling them, for instance, to autonomously issue queries or to evaluate the effects of accepting a recommendation before it is actually selected. In the literature, the latter is referred to as *speculative analysis* [6].

Code Conjurer supports speculative analysis, when it comes to the recommendation of JUnit test cases, that is, test code [16]. In general, there are two main ways in which this feature can be implemented for classic code recommendation:

Distance Measure. A naive measure that can be used to rank the results is a distance metric between an issued query and the results delivered by the search backend. It subsumes a comparison of the interface-description provided by the class under development with the interface-description of the elements in the result list. The smaller the deviation of a reuse candidate's interface from the developer's class' interface, the higher the ranking of the particular candidate in the list of recommendations. As an example, consider a generic class that comprises a set of methods with input parameters. If the interface-description of this class perfectly matches the interface-description of a candidate the distance between them is zero and thus the candidate will be ranked highly. However, if only the class names of the query and the candidate match, it is assigned a low ranking.

Test-Driven Reuse. The main difference between code and other (textual) documents is that code is executable. As described in the characterization of the background agent Code Conjurer therefore uses its context awareness and examines the project's workspace in order to look for test cases that have been written for the class under development. If the system is able to identify accompanying tests, they are executed against the reuse candidates and used as a means of evaluating the candidate's fitness for purpose. Thereby a set of running candidates is obtained and the system can distinguish between those that provide an interface that matches the one defined in the test and those that need an adapter for their interface in order to execute tests. In both approaches, additional metrics like LOC, cyclomatic complexity and execution times can also be used to influence the final ranking.

Convenient Integration of Recommendations

In Chap. 9, Murphy-Hill and Murphy [26] discuss the importance of an effective user-interface design and of making recommendation systems as easy to use and access as possible. This applies in particular to ROCR systems, since developers can quickly get frustrated by popup-windows or other UI effects that disturb their creative work and distract them from their main task—the creation of software. Thus, the recommendations should seamlessly integrate into the IDE and be intuitive to use as well as to not use. As mentioned before, an example of how this can be achieved is shown in Fig. 14.4, where the recommendation system is integrated into the auto-completion feature of the IDE.

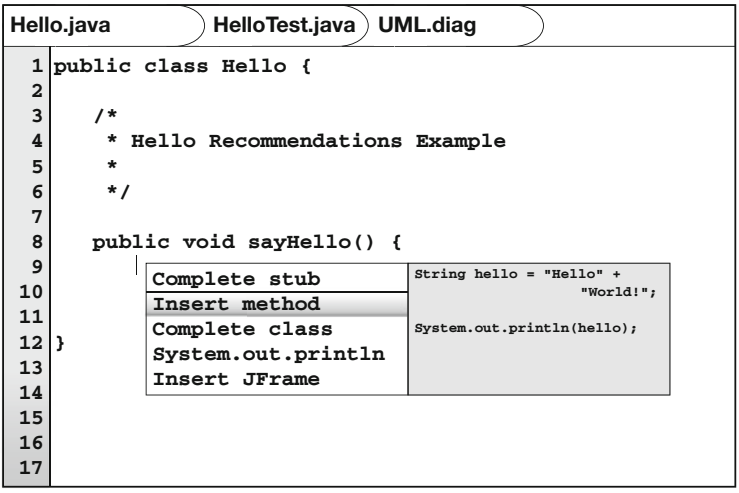

Fig. 14.4 Recommendations integrated in the IDE's auto-completion

In this case, the system does not distract the developer with unwanted popup windows and there is no need to activate any special views. Moreover the recommendations themselves can be examined using the arrow keys, and if users want to use the recommendation they can simply integrate it by pressing the enter key. Similarly, discarding the recommendations only involves the pressing of the escape key. This is a very convenient way for developers to interact with the system that takes into account the fact that during the creation of code users usually have their fingers on the keyboard and are not in contact with other input devices.

Code Conjurer also provides a convenient way of integrating reuse candidates in the system under development. If a reusable asset calls some functionality of another class or object, the system tries to automatically resolve this dependency and offers the developer the option of integrating that artifact into the system as well. In addition to that, it automatically adapts search results to the developer's context: if a reuse candidate in test-driven search provides a different interface to the one required by the test, an adapter generator tries to produce the necessary glue code to allow the candidate to be invoked. If this is successful, Code Conjurer supports the automatic integration of the component and the adapter into the developer's Eclipse workspace.

14.6 Discussion

The principles, practices, and examples presented in this chapter provide a basic overview of ROCR systems and their creation. Many of the characteristics described are more or less "best practices" distilled through a constant process of improvement

and learning about how these systems can be improved and enriched. This short section is intended to take a look at the implications arising from the usage of ROCR systems and point to possible improvements in future systems.

14.6.1 Responsibility

Software development is a labor-intensive task, involving creativity and endurance. It is thus unfortunate that developers continue to invest a huge amount of effort in re-creating similar code over and over again. Software reuse, however, promises to ease this burden on developers and to provide more room for the creation of truly new components. This is, nevertheless, only one side of the coin, since the reuse of code imposes a great deal of responsibility on developers.

Although modern systems are able to perform initial checks on the reused code like filtering malicious code (as it is performed, for instance, with the server-side execution in Code Conjurer) or evaluating the system's state by applying speculative analysis, developers have to ensure that the recommended and reused code does not introduce any (possibly malicious) unwanted behavior into the system under development. Furthermore, they need to inspect the code for any potentially harmful modules and ensure that the quality of the code at least matches that of a "self-made" system.

As explained before, when integrated into the system, the recommended code assets have to adhere to the same process and quality standards as the code written by the project developers themselves. The developers must understand what the reused code does, which beside code-inspection involves reading additional comments and documentation about the component at hand and, as a side effect, identifying possibly superfluous statements, that is, dead code.

An example of how this can be implemented is provided by Code Conjurer, which supports the identification of superfluous parts of code in reuse candidates. During a test-driven search the system inspects the reuse candidates and examines whether dead-code can be removed before compilation. If the reused component executes in the context of the developer's test case, the system allows only the necessary parts of the reuse candidate to be integrated. In addition, it provides dependency resolution when necessary and cleans up imports and declarations after the code has been inserted into the new system.

Although parts of the code inspection and cleanup actions have been considered and partially implemented in contemporary tools, in general the question of quality assurance in ROCR systems has been somewhat neglected in the past.

14.6.2 Feedback

In the same way that context awareness is critical to the processes of query formulation and result evaluation, the collection of user created feedback is critical to the quality and improvement of future recommendations provided by ROCR

systems. This not only involves processing intentional feedback from users like the manual rating of a recommended asset. It also means collecting indirect feedback derived from the users' behavior and interaction with the system under development.

As an example, one aspect of (intentional) user feedback includes ideas from social media and networks. Users may want to tell other developers within their project that a particular piece of code they reused does a great job and that they recommend its reuse. Additionally, this may help to motivate other users in their decision to exploit reuse in their projects. Developers of ROCR systems are therefore encouraged to enable users to share their experience with the system and the code assets it recommends.

Information acquired from automatically collected feedback can help to adjust the process of evaluating and ranking the retrieved results of the search, which usually relies on algorithms that grade the results with the help of a set of weighted criteria. Ideally, when the system offers a list of ranked recommendations to its users, the first item on the list should be the most suitable. It may, however, happen that users pick some other candidate from the list in accordance with their own evaluation criteria. Although this is not likely to be an issue for a small list of recommendations, the users' confidence in the recommendation system would be higher if it learned from their decisions and improved its recommendations accordingly. To achieve this, the system may for instance analyze how the different internal ranking criteria can be re-adjusted to put a user's choice first in the list and store this information in a data model for learning algorithms.

In addition, it is important to keep in mind that the process of code reuse also involves the maintenance of the integrated assets. This means, that users will invariably create new versions of the code recommended by the ROCR system by fixing bugs, improving efficiency, ... These changes can be monitored and processed in order to archive the new version of the code and provide it to other users who reused an older version. In this way the overall quality of the code in software projects applying reuse should rise, since the more often a piece of code is reused the more it will be refined and cleaned of bugs.

14.6.3 Privacy

The overall application of ROCR systems, as well as the specific issue of user feedback, cannot be considered without a look at privacy issues. The following list of issues provides an impression of some problems that may arise.

Query Formulation. Whenever a ROCR system relies on a server-side search infrastructure, the queries extracted from the developer's code are sent to the network and thus potentially exposed to others. Users need to be aware of this and ROCR systems should incorporate mechanisms to establish a trust relationship with developers. The wide range of possibilities includes the anonymization of user-related information (i.e., removal of user-id, client IP, etc.) in the server logs,

as well as the usage of secure connections. In addition, the query may contain sensitive data (like the inner design structure of a developer's system) that should not be stored or exposed to others.

Test-Driven Reuse. With the availability of test-driven reuse, a balance has to be found between the aforementioned privacy rights of the users of a system and the protection against attacks. Since test-driven reuse involves the execution of the user's code (on the server infrastructure in the case of Code Conjurer), it must be possible to track the sources of possibly malicious code and intentional attacks.

Versioning. Automating the tracking of versions may not be as easy as it seems at first sight. Reused artifacts may become deeply integrated into a developer's system and thus be tightly connected to the intellectual property of the developer and/or owning company. Since not all open-source licenses have a strong copyleft, users might not want to share their valuable code and thus it may be hard if not impossible to track changes just to the reused code without revealing more code from the project.

14.7 Conclusion

Software engineering has benefited greatly from the open source movement, especially the nascent genre of ROCR systems. Without open source it would have been much harder if not impossible to build the ROCR systems described in this chapter since a key prerequisite for them is a large set of source code that can be used as a basis for recommendations. As discussed, the ROCR systems that have been built to date index all kinds of software artifacts ranging from small code snippets, API usage, coarse-grained components and even test cases.

All tools presented in this chapter generally need to support the simplified software reuse process presented in Sect. 14.3.1 that requires developers to carry out five steps: First they need to consciously decide to reuse an artifact. Once this decision has been made they need to describe what they are looking for so that the search tool is able to find candidates for reuse. In most cases, a search will deliver a number of candidate results so that the next step is to select and tailor the most useful artifact. Once it has been integrated into the project, it needs to be maintained and updated like all other artifacts in the developer's code base.

In principle, a full-fledged ROCR system should be able to support all these steps automatically so that the developer is not burdened with them. This means that the system needs to monitor the developer's activities and must be able to independently decide when it is worthwhile to execute a search query. Obviously, it needs to be able to generate an appropriative query for the underlying search engine and rank the results according to their usefulness for a given context. Ideally, the developer then simply needs to choose the most appropriate result and the corresponding artifact is then integrated into the project automatically by the ROCR system. In a perfect world, the ROCR system would keep track of changes to the reused artifact from then on and would at least notify the developer when they occur.

A ROCR system basically consists of two main parts, namely a search engine or repository hosting the code base used to search for recommendations (the back-end if you will) and the actual recommendation engine (the front-end) that is responsible for the user and IDE interaction. If stable, generic, and mature software search engines were available it would be possible to drive several recommender systems from one search engine. However, all ROCR systems created in the last 15 years have been built in the context of academic theses and incorporated their own specialized search engine. As a result, many of them are no longer operational due to the rapid change in the landscape of code search and reuse technologies. Hence, building novel and more sustainable ROCR systems that finally accomplish the transition to a production-ready tool is still a challenge for an upcoming generation of students or industrial developers. We hope this chapter will provide them with the historical context and background knowledge needed to create them.

References

1. Atkinson, C., Bostan, P., Brenner, D., Falcone, G., Gutheil, M., Hummel, O., Juhasz, M., Stoll, D.: Modeling components and component-based systems in KobrA. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. Lecture Notes in Computer Science, vol. 5153, pp. 54–84. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85289-6_4
2. Atkinson, C., Hummel, O., Janjic, W.: Search-enhanced testing. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 880–883 (2011). doi:10.1145/1985793.1985932
3. Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P., Lopes, C.: Sourcerer: a search engine for open source code supporting structure-based search. In: Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 681–682 (2006). doi:10.1145/1176617.1176671
4. Begel, A., Phang, K.Y., Zimmermann, T.: Codebook: discovering and exploiting relationships in software repositories. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 125–134 (2010). doi:10.1145/1806799.1806821
5. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 213–222 (2009). doi:10.1145/1595696.1595728
6. Brun, Y., Holmes, R., Ernst, M.D., Notkin, D.: Speculative analysis: exploring future development states of software. In: Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research, pp. 59–64 (2010). doi:10.1145/1882362.1882375
7. de Almeida, E.S., Alvaro, A., Lucrédio, D., Garcia, V., de Lemos Meira, S.R.: RiSE project: towards a robust framework for software reuse. In: Proceedings of the IEEE International Conference on Information Reuse and Integration, pp. 48–53 (2004). doi:10.1109/IRI.2004.1431435
8. Fischer, G., Henninger, S., Redmiles, D.: Cognitive tools for locating and comprehending software objects for reuse. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 318–328 (1991). doi:10.1109/ICSE.1991.130658
9. Frakes, W.B., Pole, T.: An empirical study of representation methods for reusable software components. IEEE Trans. Software Eng. **20**(8), 617–630 (1994). doi:10.1109/32.310671

10. Holmes, R., Walker, R.J.: Systematizing pragmatic software reuse. ACM Trans. Software Eng. Meth. **21**(4), 20:1–20:44 (2012). doi:10.1145/2377656.2377657
11. Holmes, R., Walker, R.J., Murphy, G.C.: Approximate structural context matching: an approach to recommend relevant examples. IEEE Trans. Software Eng. **32**(12), 952–970 (2006). doi:10.1109/TSE.2006.117
12. Hummel, O., Atkinson, C., Schumacher, M.: Artifact representation techniques for large-scale software search engines. In: Sim, S.E., Gallardo-Valencia, R.E. (eds.) Finding Source Code on the Web for Remix and Reuse. Springer, Heidelberg (2013). doi:10.1007/978-1-4614-6596-6_5
13. Hummel, O., Janjic, W., Atkinson, C.: Evaluating the efficiency of retrieval methods for component repositories. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering, pp. 404–409 (2007)
14. Hummel, O., Janjic, W., Atkinson, C.: Code Conjurer: pulling reusable software out of thin air. IEEE Software **25**(5), 45–52 (2008). doi:10.1109/MS.2008.110
15. Janjic, W.: Realising high-precision component recommendations for software-development environments. Diploma thesis, University of Mannheim (2007)
16. Janjic, W., Atkinson, C.: Utilizing software reuse experience for automated test recommendation. In: Proceedings of the International Workshop on Automation of Software Test (2013)
17. Janjic, W., Hummel, O., Atkinson, C.: More archetypal usage scenarios for software search engines. In: Proceedings of the Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation, pp. 21–24 (2010). doi:10.1145/1809175.1809181
18. Janjic, W., Hummel, O., Schumacher, M., Atkinson, C.: An unabridged source code dataset for research in software reuse. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 339–342 (2013)
19. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2), 131–183 (1992). doi:10.1145/130844.130856
20. Lange, B.M., Moher, T.G.: Some strategies of reuse in an object-oriented programming environment. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 69–73 (1989). doi:10.1145/67449.67465
21. Lazzarini Lemos, O.A., Bajracharya, S., Ossher, J., Masiero, P.C., Lopes, C.: A test-driven approach to code search and its application to the reuse of auxiliary functionality. Inform. Software Tech. **53**(4), 294–306 (2011). doi:10.1016/j.infsof.2010.11.009
22. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 48–61 (2005). doi:10.1145/1065010.1065018
23. McIlroy, M.D.: Mass-produced software components. In: Software Engineering: Report on a Conference by the NATO Science Committee, pp. 138–155 (1968)
24. Menzies, T.: Data mining: a tutorial. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Springer, Heidelberg, Chap. 3 (2014)
25. Mili, A., Mili, R., Mittermeir, R.T.: A survey of software reuse libraries. Ann. Software Eng. **5**, 349–414 (1998). doi:10.1023/A:1018964121953
26. Murphy-Hill, E., Murphy, G.C.: Recommendation delivery: getting the user interface just right. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Springer, Heidelberg, Chap. 9 (2014)
27. Reiss, S.P.: Semantics-based code search. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 243–253 (2009). doi:10.1109/ICSE.2009.5070525
28. Robillard, M.P., Walker, R.J., Zimmermann, T.: Recommendation systems for software engineering. IEEE Software **27**(4), 80–86 (2010). doi:10.1109/MS.2009.161
29. Seacord, R.: Software engineering component repositories. In: Proceedings of the International Workshop on Component-Based Software Engineering (1999)

30. Umarji, M., Sim, S.E., Lopes, C.V.: Archetypal internet-scale source code searching. In: Proceedings of the IFIP World Computer Conference, *IFIP—The International Federation for Information Processing*, vol. 275, pp. 257–263. Springer, Heidelberg (2008). doi:10.1007/978-0-387-09684-1_21
31. Ye, Y.: Supporting component-based software development with active component repository systems. Ph.D. thesis, Department of Computer Science, University of Colorado, Boulder (2001)
32. Ye, Y., Fischer, G.: Supporting reuse by delivering task-relevant and personalized information. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 513–523 (2002). doi:10.1145/581339.581402