Source Code-Based Recommendation Systems

Chapter · January 2014

DOI: 10.1007/978-3-642-45135-5_5

Kim Mens
Université Catholique de Louvain
226 PUBLICATIONS 1,840 CITATIONS
SEE PROFILE

Angela Marie Lozano
Augusta University
23 PUBLICATIONS 313 CITATIONS
SEE PROFILE

Chapter ~~1~~
~~Building~~ Source Code-Based Recommendation Systems

Kim Mens, Angela Lozano

Early draft of a chapter which appeared as Chapter 5:
“Source Code-Based Recommendation Systems”
of the book
“Recommendation Systems in Software Engineering”
published by Springer and edited by
Martin P. Robillard, Walid Maalej, Robert J. Walker &
Thomas Zimmermann

Abstract Although today’s software systems are composed of a diversity of software artifacts, source code arguably remains the most up-to-date software artifact, and therefore the most reliable data source. It provides a rich and structured source of information, upon which recommendation systems can rely to provide useful recommendations to software developers. Source code based recommendation systems can provide support for tasks such as how to use a given API or framework, provide hints on things missing from the code, suggest how to reuse or correct existing code, or help novices learn a new project, programming paradigm, language or style. This chapter highlights relevant decisions involved in developing source code based recommendation systems. An in-depth presentation of a particular system we developed serves as a concrete illustration of some of the issues that can be encountered, and of the development choices that need to be made, when building such a system.

1.1 Introduction

In general, **recommendation systems** aid people to find relevant information and to make the right decisions when performing particular tasks. **Recommendation Systems for Software Engineering (RSSEs)** [29] in particular are software tools that can assist developers in a wide range of activities, from reusing code to writing effective bug reports. This chapter’s focus is on **Source Code Based Recommendation Systems (SCoReS)**, that is, recommendation systems that produce their recommendations essentially by analyzing the source code of a software system. Since programming lies at the heart of software development, it is no surprise that

Kim Mens (e-mail: kim.mens@uclouvain.be)
Angela Lozano (e-mail: angela.lozano@uclouvain.be)
ICTEAM, Université catholique de Louvain,
Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium

recommendation systems based on source code analysis form an important class of RSSEs.

Many current SCoReS are aimed at supporting the correct *usage of APIs* (e.g., [8, 22, 34]), *libraries or frameworks* (e.g., [3]). They typically rely upon a corpus of examples of known API usage, obtained by analyzing several applications that use a particular API. The recommendation tools that use this corpus often consist of a front-end embedded in the **IDE**, and a back-end that stores the **corpus**. The front-end is in charge of displaying results and getting the right information from the IDE or development context in order to formulate appropriate queries. The back-end is in charge of producing results that match these queries and of updating the corpus. When a developer edits a program, the recommendation system can suggest how the API should be used, based on similar, previous uses of the API. Such recommendation systems can be considered as ‘smart’ code completion tools. The match-making process typically uses information similar to that used by code completion tools, such as the type of the target object (the expected return type) and the type of the object from which the developer expects to access the required functionality.

In addition to this class of SCoReS, many other kinds of SCoReS can be distinguished, not only depending on the kind of recommendations they extract from source code or on what technique they use to do so, but also on the kind of design choices that went into building them. Although a wide variety of source code analysis techniques exists, and even though the information needs of developers when evolving their code have been identified [30], it is not always obvious how to leverage a particular technique into a successful recommendation tool. Even when the most appropriate technique is chosen that best suits the recommendation needs for a particular development task, this is no guarantee that it will lead to a successful recommendation tool. We do not claim to have discovered the philosopher’s stone that will lead to successful recommendation tools that are adopted by a broad user base. We do argue, that when building a recommendation system, one needs to be aware of, and think carefully about, all relevant development choices and criteria, in order to be able to make educated decisions that may lead to better systems. All too often, development choices that may seem unimportant at first, when not thought about carefully, in the end may hinder the potential impact of a SCoReS.

The goal of this chapter is to present novices to the domain of source code based recommendation systems a quick overview of some existing approaches, to give them a feeling of what the issues and difficulties are in building such systems, and to suggest a more systematic process to develop SCoReS. This process was distilled from on a non-exhaustive comparison and classification of currently existing approaches, as well as from our own experience with building tools to support a variety of software development and maintenance tasks. [1]

To achieve this goal, the chapter is structured as follows. After presenting a few concrete examples of SCoReS in Section 1.2, Section 1.3 lists the main decisions

[1] These tools range from source code querying tools like SOUL [14], over source code based validation tools like IntensiVE [25] and eContracts [19], to source code mining and recommendation tools like Heal [4], Mendel [18] and Mentor [20].

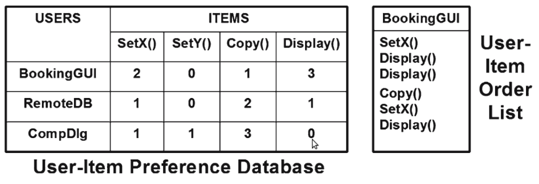
that need to be taken when developing such systems, suggests an order in which these development choices could be made, and illustrates this with the different choices that have actually been made (whether explicitly or implicitly) by currently existing SCoReS, and those introduced in Section 1.2 in particular. Being aware of the possible choices to make, as well as of the potential impact of those choices, is of key importance to novices in the area, so that they can make the right choices when building their own SCoReS. To illustrate this, Section 1.4 zooms in on a set of actual recommendation systems we built, highlighting the different design choices chosen or rejected throughout their development and evolution. We conclude the chapter and summarize its highlights in Section 1.5.

1.2 A selection of source code based recommendation systems

Before proposing our design process for SCoReS in the next section, we briefly present a selection of five recommendation systems that will later help us illustrate the importance of certain design decisions. The selected cases are *Rascal*, *FrUiT*, *Strathcona*, *Hipikat* and *CodeBroker*. We selected these systems because they are complementary in terms of the covered design approaches.

1.2.1 Rascal

Rascal [23] is a recommendation system that aims at predicting the next method that a developer could use, by analyzing classes similar to the one currently being developed. Rascal relies on the traditional recommendation technique of **collaborative filtering**. Collaborative filtering is based on the assumption that users can be clustered in groups of users that prefer the same items. Rascal's 'users' are classes and the items to be recommended are methods to be called. The similarity between the current class and other classes is essentially based on the methods they call.



USERS	ITEMS			
	SetX()	SetY()	Copy()	Display()
BookingGUI	2	0	1	3
RemoteDB	1	0	2	1
CompDlg	1	1	3	0

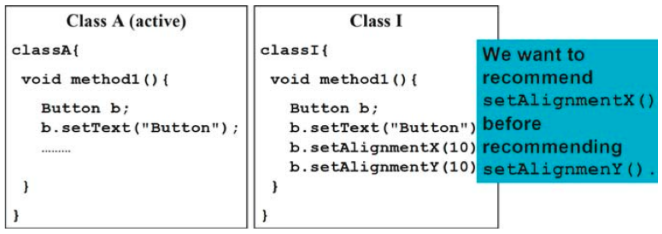
User-Item Preference Database

BookingGUI
SetX()
Display()
Display()
Copy()
SetX()
Display()

User-Item Order List

Fig. 1.1: RASCAL database – reproduced with permission from [23].

Rascal is divided in four parts, in charge of different stages of the recommendation process. First, the *active user*, which identifies the class that is currently being developed. Second, the *usage history collector*, which mines automatically a class–method usage matrix and a class–method order list (Fig. 1.1) for all classes in a set of APIs. Each cell in the matrix represent the number of times a particular method is called by a class. Third, the *code repository*, which stores the data mined by the history collector. And fourth, the *recommender agent*, which recommends the next method for the user to call in the implementation of the method and class where the cursor is currently located. The recommender agent starts by locating classes similar to the one that is currently selected. Similarity between two classes is calculated by comparing the methods they call. The frequency in which a method is used is taken into account to identify significant similarities. Commonly used methods like `toString()` get lower weight than other methods when comparing two classes. To order the recommended methods, RASCAL selects among the most similar classes those that contain the last method call used in the current class, and then recommends the method calls following that one by priority in why they appear after that method call (Fig. 1.2).



```
classA{
 void method1(){
   Button b;
   b.setText("Button");
   .........
 }
}
```

```
classI{
 void method1(){
   Button b;
   b.setText("Button")
   b.setAlignmentX(10)
   b.setAlignmentY(10)
 }
}
```

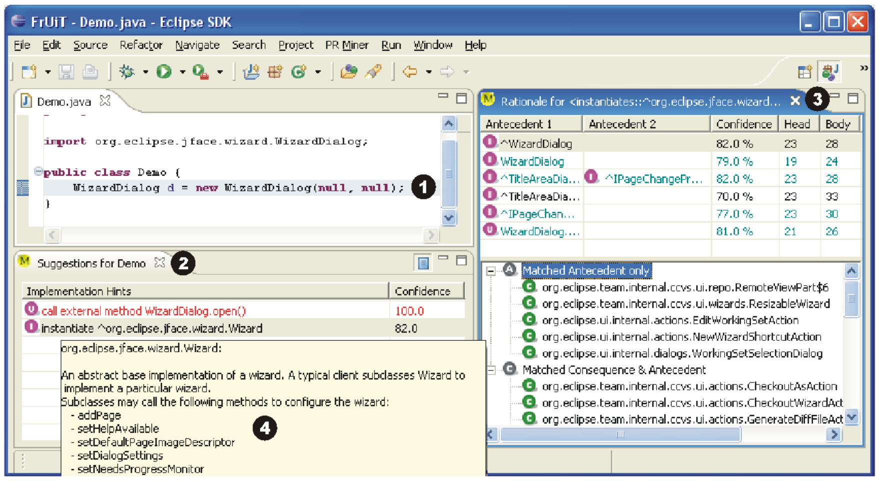
We want to recommend `setAlignmentX()` before recommending `setAlignmenY()`.

Fig. 1.2: RASCAL recommendation ordering – reproduced with permission from [23].

1.2.2 FrUiT

FrUiT [3] is an Eclipse plugin to support the usage of frameworks. FrUiT shows source code relations of the framework that tend to occur in contexts similar to the file that a developer is currently editing. Figure 1.3 shows a screenshot of FrUiT. The source code editor (1) determines the developer's context, in this case the file `Demo.java`. FrUiT's implementation hints for this file are shown on a peripheral view in the IDE (2), and the rationale for each implementation hint (e.g., 'instantiate Wizard') are also shown (3). For each implementation hint the developer can also read the Javadoc of the suggested class or method (4).

FrUiT's recommendations are calculated in three phases. First, FrUiT extracts structural relations from a set of applications using a given framework or API. These structural relations include *extends* (class A extends class B), *implements* (class A implements an interface B), overrides (class A overrides an inherited method m),


FrUiT - Demo.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project PR Miner Run Window Help
Demo.java
1
Suggestions for Demo
2
Rationale for <instantiates::^org.eclipse.jface.wizard...
3
4

```java
import org.eclipse.jface.wizard.WizardDialog;

public class Demo {
    WizardDialog d = new WizardDialog(null, null);
}
```

Implementation Hints	Confidence
call external method WizardDialog.open()	100.0
instantiate ^org.eclipse.jface.wizard.Wizard	82.0

org.eclipse.jface.wizard.Wizard:

An abstract base implementation of a wizard. A typical client subclasses Wizard to implement a particular wizard.
Subclasses may call the following methods to configure the wizard:
- addPage
- setHelpAvailable
- setDefaultPageImageDescriptor
- setDialogSettings
- setNeedsProgressMonitor

Antecedent 1	Antecedent 2	Confidence	Head	Body
^WizardDialog		82.0 %	23	28
WizardDialog		79.0 %	19	24
^TitleAreaDia...	^IPageChangePr...	82.0 %	23	28
^TitleAreaDia...		70.0 %	23	33
^IPageChan...		77.0 %	23	30
WizardDialog....		81.0 %	21	26

- Matched Antecedent only
  - org.eclipse.team.internal.ccvs.ui.repo.RemoteViewPart$6
  - org.eclipse.team.internal.ccvs.ui.wizards.ResizableWizard
  - org.eclipse.ui.internal.actions.EditWorkingSetAction
  - org.eclipse.ui.internal.actions.NewWizardShortcutAction
  - org.eclipse.ui.internal.dialogs.WorkingSetSelectionDialog
- Matched Consequence & Antecedent
  - org.eclipse.team.internal.ccvs.ui.actions.CheckoutAsAction
  - org.eclipse.team.internal.ccvs.ui.actions.CheckoutWizardAct
  - org.eclipse.team.internal.ccvs.ui.actions.GenerateDiffFileAct

Fig. 1.3: A screenshot of FrUiT at work – reproduced with permission from [3].

calls (any of the methods of class A call a method m), and *instantiates* (a constructor of class B is invoked from the implementation of class A).

Second, FrUiT uses **association rule mining** to identify structural relations that are common whenever the framework or API is used. **Association rules** are of the form ***antecedent*** $\xrightarrow{confidence}$ ***consequent***, which can be interpreted as *if-then* rules. For instance, the rule $call : m \xrightarrow{80\%} instantiate : B$ would mean that whenever an application using the framework calls the method m, it also tends to (in 80% of the cases) call a constructor of class B. Notice that FrUiT also shows each of the cases in which the association rule holds or not (see bottom-right side of the IDE in Fig. 1.3). Given that association rule mining tends to produce a combinatorial explosion of results, it is necessary to filter the results to ensure that the tool produces only relevant information. FrUiT's filters for association rules include:

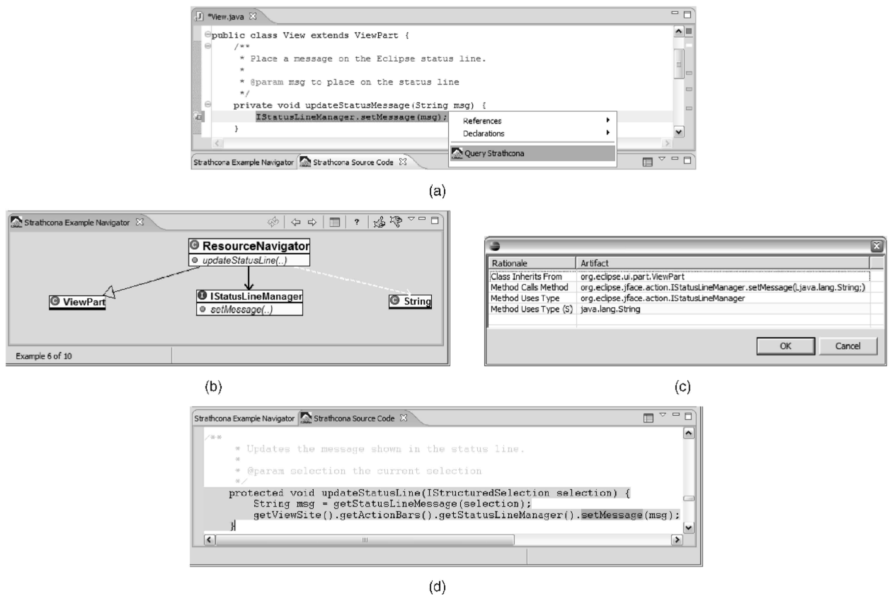
- Minimum **support**: there should be at least s cases for which the antecedent of the rule is true.
- Minimum **confidence**: the cases for which both antecedent *and* consequent of the rule are true over the cases for which the antecedent is true should be at least $c\%$.
- Misleading rules: whenever there are rules with the same consequent that have overlapping antecedents (e.g., $y \xrightarrow{c1\%} z$ and $y \wedge x \xrightarrow{c2\%} z$), the rule with the least prerequisites in the antecedent is preferred (i.e., $y \xrightarrow{c1\%} z$) because the additional prerequisite for the antecedent (i.e., x) decreases the likelihood of z to hold.
- **Overfitting** rules: even if in the case of the previous filter $c2$ would be marginally greater than $c1$, then the rule with the more detailed antecedent would still be rejected, to keep the rule simpler and because the increase in confidence is not significantly higher.

- Specific rules: imagine two rules with the same consequent and same confidence, whose antecedents are related by a third rule with 100% confidence. For instance, $x \xrightarrow{c\%} z$, $y \xrightarrow{c\%} z$, and $y \xrightarrow{100\%} x$). Then the second rule can be discarded ($y \xrightarrow{c\%} z$) because it is already covered by the first one ($x \xrightarrow{c\%} z$).

In the third and final phase, for the file currently in focus of the Eclipse editor, FrUiT recommends all rules that mention in their antecedent or consequent any of the source code entities of the framework mentioned in that file.

1.2.3 Strathcona

Strathcona [12] is also an Eclipse plugin that recommends examples (cf. Fig. 1.4 (d)) on how to complete a method that uses a third-party library. The user is supposed to highlight a source code fragment that uses the third party library to request Strathcona for examples that use the same functionality of the library (cf. Fig. 1.4 (a)). The examples are taken from other applications using the same third party library.



```
public class View extends ViewPart {
    /**
     * Place a message on the Eclipse status line.
     *
     * @param msg to place on the status line
     */
    private void updateStatusMessage(String msg) {
        IStatusLineManager.setMessage(msg);
    }
```

Rationale	Artifact
Class Inherits From	org.eclipse.ui.part.ViewPart
Method Calls Method	org.eclipse.jface.action.IStatusLineManager.setMessage(Ljava.lang.String;)
Method Uses Type	org.eclipse.jface.action.IStatusLineManager
Method Uses Type (S)	java.lang.String

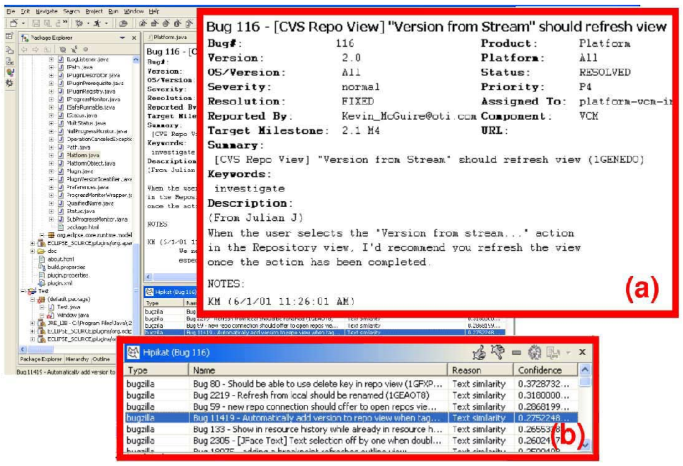
```
/**
     * Updates the message shown in the status line.
     *
     * @param selection the current selection
     */
    protected void updateStatusLine(IStructuredSelection selection) {
        String msg = getStatusLineMessage(selection);
        getViewSite().getActionBars().getStatusLineManager().setMessage(msg);
    }
```

Fig. 1.4: Strathcona screenshots – reproduced with permission from [12]: (a) Querying Strathcona; Similarity between the example suggested and the queried code (b) as UML graph, and (c) as list of structural facts; (d) Example suggested by Strathcona.

Strathcona works by extracting the *structural context* of source code entities. This structural context includes the method's signature, the declared type and parent type, the methods called, the names of fields accessed, and the types referred to by each method. The extracted structural context can be used in two ways. First, as an automatic query that describes the source code fragment for which the developer requests support. Second, to build a database containing the structural contexts of classes using the libraries that the tool supports. Relevant examples are located by identifying entities in Strathcona's database with a structural context similar to the one of the fragment being analyzed (cf. Fig. 1.4 (b) and (c)). Similarity is based on heuristics that take into account entities extending the same types, calling the same methods, using the same types, and using the same fields. Entities in the result set that match more heuristics are ranked higher than those that match less.

1.2.4 Hipikat

Hipikat [6] helps newcomers to a software project to find relevant information for a maintenance task at hand. The idea behind *Hipikat* is to collect and retrieve relevant parts of the project's history based on its source code, email discussions, bug reports, change history and documentation. Fig. 1.5 shows a screenshot of Hipikat. The main



```
Bug 116 - [CVS Repo View] "Version from Stream" should refresh view
Bug#:               116                       Product:      Platform
Version:            2.0                       Platform:     All
OS/Version:         All                       Status:       RESOLVED
Severity:           normal                    Priority:     P4
Resolution:         FIXED                     Assigned To:  platform-vcm-i
Reported By:        Kevin_McGuire@oti.com     Component:    VCM
Target Milestone:   2.1 M4                    URL:
Summary:
 [CVS Repo View] "Version from Stream" should refresh view (1GENEDO)
Keywords:
 investigate
Description:
(From Julian J)
When the user selects the "Version from stream..." action
in the Repository view, I'd recommend you refresh the view
once the action has been completed.

NOTES:
KM (6/1/01 11:26:01 AM)
```

Type	Name	Reason	Confidence
bugzilla	Bug 80 - Should be able to use delete key in repo view (1GFXP...	Text similarity	0.3728732...
bugzilla	Bug 2219 - Refresh from local should be renamed (1GEAOT8)	Text similarity	0.3180000...
bugzilla	Bug 59 - new repo connection should offer to open repos vie...	Text similarity	0.2868199...
bugzilla	Bug 11419 - Automatically add version to repo view when tag...	Text similarity	0.2752248...
bugzilla	Bug 133 - Show in resource history while already in resource h...	Text similarity	0.26553[illegible]
bugzilla	Bug 2305 - [JFace Text] Text selection off by one when doubl...	Text similarity	0.26024[illegible]

Fig. 1.5: A screenshot of Hipikat - reproduced with permission from [6].

window in the IDE shows a bug report that the developer wants to fix (cf. Fig. 1.5 (a)). Once the developer chooses 'Query Hipikat' from the context menu, the tool will return a list of related artifacts. The artifacts related to the bug are shown in a separate view (Fig. 1.5 (b)). For each artifact found, Hipikat shows its name, type (web page, news article, CVS revision, or Bugzilla item), the rationale for recommending it, and an estimate of its relevance to the queried artifact. Any of the artifacts found can be opened in the main window of the IDE to continue querying Hipikat, so the user can keep looking for solutions for problems similar to the one described in the bug report.

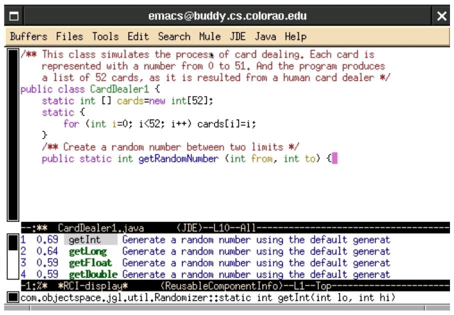
Hipikat is composed of two parts. The first part is an Eclipse plugin that sends the query (by artifact of interest or a set of keywords) and presents the results (related artifacts). The second part is the back-end which builds a relationship graph of diverse software artifacts and calculates the artifacts relevant to the query. In contrast to the previously described SCoReS, *Hipikat*'s recommendations are based on the links established between different artifacts and their similarity. The links are inferred by custom-made heuristics. For instance, bug report IDs are matched to change logs by using regular expressions on their associated message, and change time-stamps on source code files are matched to the closing time of bug reports.

1.2.5 CodeBroker

CodeBroker [33] recommends methods to call based on the comments and signature of the method where the developer's cursor is currently located. The goal of the system is to support the development of a new method by finding methods that already (partially) implement the functionality that the developer aims to implement.

CodeBroker is also divided in a front-end and a back-end. The front-end monitors the cursor, queries the back-end with relevant information from the current context, and shows the results. The back-end is divided in two parts. First, a *discourse model* that stores and updates the comments and signatures of methods from a set of libraries, API's and frameworks to reuse. Second, a *user model* to remove methods, classes or packages from the recommendation list, while remembering if the removal should be done for a session (irrelevant for the task) or for all sessions (irrelevant for the user, i.e., the developer is aware of the method and does not need to be reminded of it). For instance, Fig. 1.6 (c) illustrates how it suffices to right-click on a recommendation to be able to eliminate it from the recommendation list.

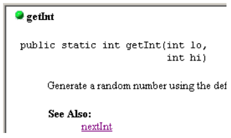
CodeBroker is a proactive recommendation tool integrated with the source code editor (emacs). That is, the tool proactively identifies the need for recommendations by monitoring method declarations and their description as soon as the developer starts writing them. This means that every time the cursor changes its position it automatically queries the back-end repository. The query describes the method currently being developed in terms of the words used in its comments, of the words in its signature, and of the types used in its signature. This query aims at finding methods with similar comments and signatures. The technique used is the same one used by basic text search engines (namely LSA, or Latent Semantic Analysis) which represents each method as a boolean vector that indicates which words are mentioned by the method's signature or comments, and a vector of weights which indicate which words are significant, to decide the similarity between two methods



```
emacs@buddy.cs.colorao.edu
Buffers Files Tools Edit Search Mule JDE Java Help
/** This class simulates the process of card dealing. Each card is
    represented with a number from 0 to 51. And the program produces
    a list of 52 cards, as it is resulted from a human card dealer */
public class CardDealer1 {
    static int [] cards=new int[52];
    static {
        for (int i=0; i<52; i++) cards[i]=i;
    }
    /** Create a random number between two limits */
    public static int getRandomNumber (int from, int to) {

--:**  CardDealer1.java      (JDE)--L10--All--------------------------------
1  0.69  getInt    Generate a random number using the default generat
2  0.64  getLong   Generate a random number using the default generat
3  0.59  getFloat  Generate a random number using the default generat
4  0.59  getDouble Generate a random number using the default generat
-1:%*  *RCI-display*      (ReusableComponentInfo)--L1--Top------------------
com.objectspace.jgl.util.Randomizer::static int getInt(int lo, int hi)
```

(a) CodeBroker's recommended methods (first recommended method is selected).



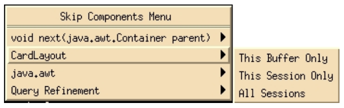
getInt

```
public static int getInt(int lo,
                         int hi)
```

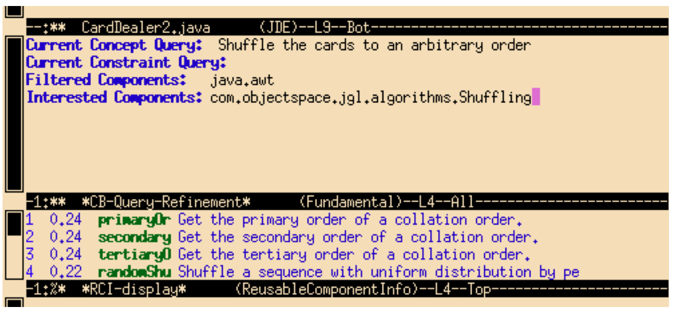
Generate a random number using the def

See Also:
nextInt

(b) Left-clicking on the selected recommendation shows the Javadoc of the method.



(c) Right-clicking on the selected recommendation allows to remove it from the recommendation list or to refine the query.



```
--:**  CardDealer2.java      (JDE)--L9--Bot-------------------------------------
Current Concept Query:  Shuffle the cards to an arbitrary order
Current Constraint Query:
Filtered Components:   java.awt
Interested Components: com.objectspace.jgl.algorithms.Shuffling

-1:**  *CB-Query-Refinement*      (Fundamental)--L4--All----------------------
1  0.24  primaryOr Get the primary order of a collation order.
2  0.24  secondary Get the secondary order of a collation order.
3  0.24  tertiaryO Get the tertiary order of a collation order.
4  0.22  randomShu Shuffle a sequence with uniform distribution by pe
-1:%*  *RCI-display*      (ReusableComponentInfo)--L4--Top----------------------
```

(d) Query refinement.

Fig. 1.6: Screenshots of CodeBroker - adapted with permission from [33].

of an application. The text search results are then filtered depending on the similarity between the types used in the method in focus and those that matched. As soon as there is a response for the query, matching methods are presented, ordered by relevance, in a peripheral view of the source code editor (cf. Fig. 1.6 (a)). Finally, Fig. 1.6 (d) shows how CodeBroker allows the developer to *guide* the back-end in finding relevant recommendations by adding the names of methods, classes or packages to the Filtered Components so that they get excluded from the results, or to the Interested Components so that the search is limited to these entities.

1.3 Development decisions when building a SCoReS

Throughout the development of a source code based recommendation system, many important decisions need to be made. These decisions can be classified along two main axes. A first axis is the phase of the development cycle when the decision needs to be made: is it a decision that relates to the system's *requirements*, to its *design*, to its actual *implementation*, or rather to its eventual *validation*? [2] The other axis is whether these decisions apply to the recommendation *approach* or rather are about how the system *interacts* with the *user*. [3] Table 1.1 summarizes these axes and the main classes of decisions that can be found along these axes.

Table 1.1: Kinds of development decisions to be taken when building a SCoReS

	Requirements	**Design**	**Implementation**	**Validation**
Approach	1. Intent	3. Corpus	5. Method	7. Support
User interaction	2. HCI	4. General I/O	6. Detailed I/O	8. Interaction

1.3.1 Process

Assuming that we follow a traditional development cycle starting from the requirements down, the numbers in Table 1.1 suggest a possible *process* of how and when to make these decisions :

1. Start by thinking about the system's **Intent** (§1.3.2), which are all decisions related to the *purpose* of the recommendation approach.
2. Then decide upon the **Human Computer Interaction** (§1.3.3), i.e. how the end user (typically, a developer) is expected to *interact* with the SCoReS.
3. Once the functional and user interaction requirements have been decided upon, choose what **Corpus** (§1.3.4) of data sources the system will use to provide its recommendations.
4. **General Input-Output** (§1.3.5) decisions then refine the Human Computer Interaction decisions taken previously.
5. Now decide upon the details of the recommendation **Method** (§1.3.6), i.e. all details related to how the recommendation process is implemented.
6. Since the decisions of the previous step may shed more light on the **Detailed Input-Output** (§1.3.7), next we can decide what precise information the approach requires as input and what and how it produces its output.

[2] More details about validation decisions can be found in Chapters **??**,**??**, **??** and **??**.

[3] See Chapter **??** for more details about user interaction.

7. After having implemented the system, we need to choose how to validate it. On the approach axis, the main question is how well the system provides **Support** (§1.3.8) for the tasks it aims to provide recommendations for.
8. On the user interaction axis, regarding the validation some decisions need to be made on how to assess the **Interaction** (§1.3.9) of different users with the SCoReS.

In the next subsections, we zoom in on each of these classes of decisions, one by one, providing examples taken from a selection of SCoReS approaches which we analyzed [4], and the selected approaches presented in Section 1.2 in particular.

1.3.2 Intent

The *intent* decisions define the purpose of the system: who is the intended **user**, what tasks need to be supported by the system, what kind of cognitive support can it provide, what kind of information does it produce to support that task?

Intended user Regarding the intended audience, first a decision should be made on the *role* of the expected users of the system. Although most SCoReS, such as those shown in Section 1.2, are targeted at **developers**, some others (whose supported task is not directly linked to modification requests) also consider other roles [1, 16]. German et al. [10] distinguish roles such as maintainers, reverse engineers, reengineers, managers, testers, documenters or researchers. A second decision is related to the level of *experience* of the intended users. Whereas some recommendation systems make no assumptions about their users' level of experience, others are targeted specifically at novices or experts in either the source code or the programming language being studied. For instance, Hipikat helps newcomers to a project,whereas CodeBroker can be **personalized** depending on the experience level of its users.

Supported task This decision is about what task the system aims to support. As already mentioned in Section 1.1, many SCoReS aim at supporting the correct *usage of APIs, libraries or frameworks*. Others support *mappings* for API evolution [7] or for language migration [35]. Yet others recommend *code corrections* (source code entities that are associated to bugs) [4], *code to be reused* (methods to be reused that contain a functionality that needs to be implemented) like CodeBroker, *code changes* (source code entities impacted by the change of another source code entity) [36] or *code suggestions* (suggested changes to code entities to complete or standardize their implementation) [18, 20, 21]. Some SCoReS, like Hipikat, can support novices in *learning a new project*. Others provide relevant information that

[4] Many approaches exist that extract relevant information for developers from source code. For the purpose of this chapter we analyzed a non-exhaustive selection of approaches that call themselves or that have been cited by others as ‘recommendation systems’. We filtered this selection further to those systems that rely explicitly on source code to produce their recommendations, and privileged approaches published in well-known software engineering venues.

can help developers ***understand*** how certain concerns are implemented [11, 28] or how certain bugs have been solved [1].

Cognitive support According to German et al. [10], another important decision is how recommendation systems support human cognition to achieve certain tasks. In other words, which questions about the task can the system help answering? Five categories of questions can be distinguished: *who*, *why*, *when*, *what* and *how*.

Many systems aim at finding artifacts or source code entities relevant to developers to perform their task (i.e., they provide an answer to the question of *what* information is needed to complete the task) [6, 23].

The amount of information a system provides to **explain** its recommendations tends to depend on the context in which it is applied. Systems that offer API support and that act as a kind of code completion tool integrated with the IDE, usually do not (need to) provide detailed information that could help a user evaluate the appropriateness of their recommendation. API support tools that are not embedded as code completion tools, on the other hand, do tend to provide diverse views that explain the rationale for each recommendation they propose. Nevertheless, the level of detail offered does not depend *only* on how the system is integrated with the IDE. While some SCoReS, like Hipikat and CodeBroker, provide information to explain *why* an entity is part of their recommendation set, others require the developer to figure out the rationale of the recommendation [11, 17, 28].

Another popular cognitive support provided by SCoReS, as exemplified by FrUiT and Strathcona, is information for developers on *how* to complete their task. The level of detail offered depends on the kind of support provided by the system. Systems with specific goals like how to replace deprecated methods [7] or how to obtain an instance of a type from another instance of another type [8, 22] need to provide less background information on their recommendations than systems concerned with broader goals (like how to complete a method [18, 20]), systems that require user input (how to map statements in a method [5]), or how to use an API method [3, 12].

Only few systems provide support for answering questions about the reason behind particular implementation choices (why something is implemented in a certain way); the closest ones are those that look for related information to an artifact/entity [1, 11, 28] but the relation among those entities must still be discovered by the developer.

Similarly, few SCoReS seem to answer temporal questions (*when*), which is probably due to the fact that they often do not rely on change information. Such temporal information could however be useful for answering rationale questions, since it can uncover the trace behind a change.

Finally, SCoReS rarely permit to answer authorship (*who*) questions (a notable exception is DebugAdvisor [1]), due to the fact that source code alone often does not have information on its author, unless complemented by other sources that do have this.

Proposed information This decision pertains to what kind of information a recommendation system proposes. The information proposed affects how easy it is for a user to use a recommendation.

Some recommendation systems propose concrete *actions* [2, 5, 8, 22, 34], that is, they do not only advise *what* to do, but also *how* to do it. For most recommenders proposing actions, the only thing that remains to be done by users (typically developers), is to select the change that they judge adequate for the current context or task, and the system will perform it automatically (e.g., inserting a code snippet after the last statement added to the source code entity being implemented). Most systems providing API support or code completion propose actions as output.

Another common type of output, like in FrUiT, Rascal and CodeBroker, is *source code entities* that is, *what* to change or use but not *how* to do it. Depending on the task at hand and the type of recommendation system, a further interpretation of the proposed results may then be required. For example, the recommendation system may propose to modify a particular source code entity in a very specific way, but the system's integration with the IDE may not be sophisticated enough to automate this action, thus requiring the user to perform this action manually.

Examples can help a user understand how a similar problem was solved in similar situations [12]. It is the user's job of deciding whether and which part of the example is relevant for the task at hand, and how to adapt the example to the current situation.

Finally, SCoReS can suggest related *software artifacts* (i.e., any by-product of the software development process that is not a source code entity) to help a developer in better understanding the implications of the task at hand [1, 6]. The usefulness and efficacy of these recommendations depend on the relevance of the information and links that the developer can abstract from the information the system proposes.

1.3.3 Human Computer Interaction

After having focused on decisions related to the system's intent, we now turn our attention to the **human computer interaction**. These decisions permit us to establish the expected interaction between a user and a SCoReS. They comprise the type of system, the type of recommendations given by the system, and the expected input from the user.

Type of system A recommendation system can be an *IDE plugin*, a *stand alone application* or some other type such as a *command line* tool or an internal or external *DSL* to be used by the user of the system. Given that most of the SCoReS we analyzed are targeted at developers, they tend to be integrated with the development environment, as is the case with FrUiT, Hipikat, Strathcona and CodeBroker. This choice allows for an easier detection of the user's context or activity, and sometimes easier access to the syntactic entities and relations being used in that context. Another alternative that is sometimes chosen for recommendation systems is to make them stand alone applications, like Rascal. Both standalone SCoReS and those integrated with the IDE tend to have a graphical interface. Programmatic SCoReS are seldom an implementation choice probably to reduce their intrusiveness.

Type of recommender This decision characterizes what *type* of recommender a SCRS is: a 'finder', an 'advisor' or a 'validator'.

Some recommendation systems, like Hipikat and Strathcona, are dedicated search engines that *find* relevant information for a particular task at hand. For example, Strathcona finds examples on how to use a given API, while Hipikat finds information related to a source code artifact, for instance a change request. Other systems, like FrUiT, Rascal and CodeBroker, focus on *advising* certain courses of action to the user. Rascal, for example, recommends what method calls may need to be added [23]. The advice produced may be sufficiently clear to end users only if it is part of their workflow, as is the case with code completion tools providing code suggestion to developers, or if the user knows exactly how to interpret the results produced by the recommendation system.

Yet other systems focus more on verifying or *validating* certain properties of the source code, such as adherence to certain conventions or design regularities [19, 25].

An interesting observation is that it sometimes happens that developers misuse a SCoReS. For instance, if the methods recommended by some SCoReS often have words similar to those in the identifier or comments of the method currently being implemented, developers can be tempted into abusing the system as a kind of search tool, by changing the signature or comments of their methods to resemble the methods they anticipate to be within the APIs used. Such a case was reported on in [33]. This change in the user's expected behavior (using the system as a finder rather than as a validator) could probably be explained by the fact that the developer found it frustrating to use the SCoReS due to repeated unsuccessful use because many of the anticipated methods did not exist in the APIs used. When building a SCoReS, being aware of the possibility of such abuses is important.

User involvement Another important choice that needs to be made is what kind of *user involvement* the SCoReS requires. For example, does it require manual input or does it require user involvement for filtering the output?

Usually, the input required for a SCoReS can be extracted automatically from the programming context. For instance, the method being implemented, the methods it has called so far, the types it has used so far, the order of the methods called, the last statement or identifier created, etc. Given that the majority of SCoReS only rely on the analysis of source code entities related to the one being edited or selected, there is no need of manual input. However, if a SCoReS being developed requires data that cannot be extracted from the programming context, it is necessary to decide in which way the user will provide the information required. For instance, using a query made by the user to select a set of seed entities [11] or by asking the user explicitly to provide the source and target for a mapping [5].

Another important choice to make is whether or not the user can or should filter out recommendations. This filtering can be done iteratively, like in FrUiT or CodeBroker, and/or from the group of final results, like in Strathcona, Rascal and

CodeBroker. Interestingly, filtering of results can also be used to exclude the user's explicit input. For instance, CodeBroker eliminates results that the user knows by looking at their local history [33]. Another interesting alternative to consider is allowing result filtering per user session or per module [33].

1.3.4 Corpus

Deciding on the *corpus* of a SCoReS amounts to deciding what data sources the system needs to provide its recommendations. The main source of information used by SCoReS is *program code*, yet is sometimes complemented with additional sources of information such as change management or defect tracking repositories, informal communications, local history, etc. When combining several sources of data, it is also important to explicitly decide how these sources are *correlated*.

Program code A first important decision to make regarding the corpus is whether or not the system will calculate its recommendations based on the application on which a developer is currently *working*, as in Hipikat, or on an application that they are currently *using*, as in the other SCoReS we illustrated. The former set of systems build their corpus from the code of a *single application* (i.e., the one that is being built) while the latter build their corpus from the code of *multiple applications* (i.e., those that also use the application that is being used by the application being built).

This choice has repercussions on the techniques and methods that can be used to calculate recommendations, on storage requirements, and on their usefulness. SCoReS with a corpus based on *multiple applications* can use generic data analysis techniques, but require storage to hold the data collected previously and their recommendations are often restricted to the analysis of specific library or framework. Therefore their usefulness depends on the match between the APIs used by the user of the SCoReS and those available on the server. In contrast, SCoReS that analyze a *single application* do not need separate storage nor pre-fetched data, nevertheless they usually require specialized or heuristic-based data analysis techniques. These SCoReS are also well suited for closed source applications.

Complementary information In addition to source code or a source code repository, a recommendation system may also distill useful information from ***program execution traces*** [1], from ***software configuration management systems*** [1, 6, 7, 36], from ***issue management systems*** [1, 6], from *informal communication* [6] transcripts between developers (taken from e-mail discussions, mailing lists, instant messages, discussion fora, etc.), or from the ***user's history*** by tracking and analyzing the user's navigation and edition behavior. Mylar [15] provides an example of the latter by providing program navigation support that gives artifacts of more recent interest a higher relevance.

Correlated information All SCoReS that use data not originated from the source code correlate it to source code. Indeed, there is little added value of having several data sources if the system does not attempt to combine the sources into something that is more than the sum of its independent facts. A relevant decision to take is thus *how* to mix the information coming from *program code* with the information extracted from *complementary* sources. For example, author information and timestamps of files in a CVS repository can be correlated to the source code contained in those files to infer what source code entities may have changed together [36], or the identifier of a bug report can be linked to the messages in CVS commits to locate the source code entities that were modified when fixing a bug [6].

1.3.5 General Input/Output

The general input/output decisions listed below define the interaction between a user and the recommendation system, and refine the Human Computer Interaction decisions discussed in Subsection 1.3.3.

Input mechanism A first decision regards whether or not the SCoReS infers its input from the currently active source code entity or artifact whenever the recommendation system is triggered. This decision depends on whether or not the information available from the source code development context is needed to calculate recommendations for the task to support. While most SCoReS infer their input *implicitly* from the current context or activity of the user, like in FrUiT, Strathcona and Rascal, others rely either partially or completely on the *user's input* to identify for which task or entity the recommendation is requested, like CodeBroker and Hipikat.

Nature of input The nature of the input directly affects the user's perception of how easy it is to use the system. The majority of SCoReS require as input a (fragment of) *source code* to provide concrete recommendations regarding that source code, as in FrUiT or Rascal. Other systems implicitly or explicitly take the users' *context* and *activity* into account in order to support them in that activity, like Strathcona or CodeBroker. Extracting the programming context is usually limited however to identifying the source code entity in focus or currently being used: only a few of the recommendation systems go beyond that. API-Explorer, for instance, finds source and target types in a code completion operation [8]. Finally, some systems take input under the form of a *natural language queries* [11], or other *non code artifacts*, like Hipikat.

Response triggers When building a SCoReS one also needs to decide when the system should be *triggered* to calculate its recommendations. Recommendations can be calculated either upon *explicit request* by the user (known as **reactive recommendation**), as in FrUiT, Hipikat, Strathcona and Rascal, or ***proactively*** when a certain *contextual* situation occurs, such as saving a file. But even for those SCoReS that discover their input implicitly from the development environment, the request for calculating a recommendation typically remains explicit, except for a few systems like CodeBroker which *continuously* update their results. This design decision is related to whether the system's responses are considered as *intrusive* or not, in other words whether they interfere directly with how developers normally perform their tasks. Most SCoReS try to be not too intrusive by triggering responses only upon explicit request by the developer and by presenting the recommendations and their rationale in peripheral views and not in the main programming view.

Nature of output The nature of the output, too, may affect significantly the usability perception of a SCoReS. For example, for a source code fragment given as input, Strathcona [12] provides similar source code examples to a developer. Each such example consists of three parts: a graphical UML-like overview illustrating the structural similarity of the example to the queried code fragment, a textual description of why the example was selected, and source code with structural details highlighted that are similar to the queried fragment.

There is no typical type of output for SCoReS. While some of them are restricted to deliver the information requested in some primitive *textual* description [3, 12, 22], others aim at integrating the recommendation into the IDE so it can be used as seamlessly as possible. This seamless information includes navigation and browsing aids like *text links*[5] [18, 20], *clickable links* [6] as in FrUiT, Hipikat, Strathcona and CodeBroker, *annotations or highlights* of relevant parts of the source code as in Strathcona and CodeBroker [12, 33], or *graphical* notations as in Strathcona [12], for the results.

There are some missing opportunities at graphical outputs given that existing SCoReS tend to be limited to UML graphs, but the intermediate data representations used by the SCoReS (e.g., graphs, trees, vectors) so far have been neglected as a way to deliver the recommendations. We believe that, in some cases, presenting the recommendations in such a format, as opposed to presenting a mere list of recommendations, may provide a more intuitive way for the end-user to understand the results.

Type of output The usefulness of SCoReS also dependq on the appropriateness of the output produced to the task to support. Examples of these outputs include existing *software artifacts* [1, 6] or *source code entities* relevant to the user's current task, *missing source code* entities or fragments [3, 18, 20], or suggestions for *mapping* entities (like statements [5] or methods [7]). Existing relevant entities usually indicate other entities that were typically used or changed whenever the source code entity(ies) given as input are used. For instance, methods to reuse [33], methods to change [36], the next method to call [2, 23], or a set of methods to call (as a list [11, 17, 28], as a sequence [32], using an example [12] or as a code snippet [8, 22]). Mappings indicate how to use or replace the components of one entity with the components of the other entity. And implementation suggestions indicate

[5] Textual references such as corresponding files and line numbers.

[6] Links in the IDE allowing to jump directly to the corresponding code.

syntactic or structural relations that may have been forgotten in an entity's implementation (such as implementing an interface, calling a method, using a type, etc.).

1.3.6 Method

We now discuss all design choices specific to the software recommendation process.

Data selection Once the data to be used as input by a SCoReS is chosen (e.g., all applications that use a certain API), the developer of the SCoReS must decide on the level of detail in which the data will be collected. This decision affects which analyses are possible and which information will be available for the user of the SCoReS. As such, the data selected as relevant for the recommendations can provide a differentiating factor among similar SCoReS. It is therefore important to describe up front the rationale for the data to be collected, and in particular how it can be used to provide a useful recommendation, so that the appropriate level of granularity can be chosen. While a too coarse granularity may negatively affect the support the SCoReS aims to provide, there is no use in choosing a too fine granularity either.

Not surprisingly, when analyzing object-oriented source code, the preferred level of detail of most SCoReS seems to be at the level of *methods* [3, 12, 23, 33] and *relations between methods* (like call statements [3, 12, 23], overriding information [2], extends [2, 8, 12, 18, 20, 22], and implements [2, 22]). Nevertheless, *type information* like the return type of a method [8, 12, 22] or the type of its parameters [8, 12, 22] can be very useful to connect different source code entities as well. Tokens within the names of source code entities are another significant piece of information to consider [5, 11, 18, 20, 32–35] as a way to analyze the issues of diverse vocabulary to describe the same concept or feature, and to match the concrete concepts that developers use when dealing with modification requests versus the high level concepts used when implementing reusable source code entities. Other levels of source code information used include comments [33], files [1, 6, 36], types [8, 17, 22], fields [12, 17, 28, 35], literals [34], signatures [33], contains / instantiates statements [2, 3, 5, 8, 12, 17, 18, 20, 22], access information [12, 17, 22, 28, 34], order of statements [23, 32], and changes to source code [1, 6, 7, 36].

Type of analysis The simplest kind of analyses are *textual approaches* that process raw source code directly. *Lexical approaches* or token-based techniques transform code into a sequence of lexical tokens using compiler-style lexical analysis, as in Hipikat and CodeBroker. *Syntactic approaches* convert source code into parse trees or abstract syntax trees and perform their subsequent analyses on those trees, as in FrUiT, Strathcona, Rascal, and CodeBroker too. *Semantics-aware approaches* perform their analysis at a more semantic level by relying for example on dynamic program analysis techniques [1].

Data requirements This decision states any particular constraints on the data for the system to be able to work properly, such as the *programming paradigm* targeted by the system (e.g., object-oriented or procedural) or the *particular programming language* (e.g., Java, Smalltalk, C) supported by the system. This decision is usually taken for pragmatic reasons like being able to compare the results of a SCoReS with another one, which requires being able to analyze the same source code, or being able to use a certain auxiliary tool that can handle the data extraction or data processing.

The most common paradigm analyzed has been the object oriented [3, 12, 23], probably because it is more prevalent in current industrial practice, although a few are limited to the procedural paradigm [17]. However, many of the assumptions made for object-oriented code can be translated to procedural and vice versa. Java seems to be one of the languages most commonly supported by SCoReS [3, 6, 12, 23, 33]. Other languages tackled include C [17] and SmallTalk [18, 20, 21]. Although the choice of the actual implementation language may seem insignificant, dynamically typed languages like Smalltalk may make it easier to quickly prototype (recommendation or other) tools, but on the other hand dynamically typed languages cannot ensure all assumptions made for statically typed languages like Java. For instance, that a method signature can be uniquely linked to a type, that a method returns variables of a particular type, or that its parameters are of a particular type. Similarly, while the name of a method can indicate similar concerns in a language like Java, for languages like Smalltalk (where there is no difference between the name and the signature of a method) such indications are more difficult to validate.

In addition to the language, there may be *other* particular requirements on the data needed by the system, such as that there should be sufficient example code available that uses a particular API, as is the case for Strathcona, for example.

Intermediate data representation This decision concerns the actual acquisition of the raw data into the most appropriate format for facilitating further processing. Regarding this intermediate format, a SCoReS' designer can choose among *graph-based approaches* [1, 8, 11, 17, 22, 28] by using for example program-dependence or call graphs, *tree-based approaches* [5] which reason about parse trees or other tree structures, approaches like Rascal or CodeBroker that use *vectors or matrices*as intermediate representation, or *fact-based approaches* like FrUiT, Hipikat or Strathcona, which organize their data as sets of logic or database facts. Yet other approaches may reason about ***metrics*** or simply about some *textual data*. *Hybrid* approaches combine several of these internal representations.

Graphs and fact bases seem to be among the most popular representations, probably because they provide flexible and well-known mechanisms to find interesting relations, and because these relations can be used to explain the rationale behind a recommendation. Vectors or matrix representations provide high performance operations while keeping an exact account of the relevant facts. Textual and numeric representations seem less appropriate for SCoReS probably due to the lack of traceability they suffer from. Search-based techniques, for example, usually operate atop matrix or vector representations instead of text representations.

Analysis technique The next key decision is the choice of the actual algorithm or *technique* that produces relevant recommendations from the data extracted.

Traditional recommendation techniques are typically based on two sets: the *items* to recommend and the *users* of those items. Recommendation systems aim at matching users to items, by identifying the user's needs or preferences, and then locating the best items that fulfill those needs. To identify the user's need, recommendation techniques can take into account previous ratings of the current user (**social tagging**), demographic information, features of the items, or the fitness of the items to the current user. Once the matching is done, items can be prioritized by ratings of their users. The combination of these strategies results leads to different kinds of recommendation techniques. Rascal [23] is a typical example of a SCoReS that relies on a traditional recommendation technique. In Chapter **??**, **(author?)** [9] provide a more elaborate discussion on traditional recommendation techniques.

In practice, however, unless a SCoReS recommends API calls (*items*) on a large and diverse set of example applications that use the API (*users*), it is often difficult for a SCoReS to rely on traditional recommendation techniques, because the likelihood of having enough users for a given source code entity is small. The most popular analysis techniques used by SCoReS therefore seem to be those that calculate similarities between the input and the rest of the entities analyzed, by using traditional *classification techniques* like **cluster analysis** [2, 17, 32], association rule mining [3] and identification of **frequent itemsets** [2, 36]. Yet other approaches essentially rely on basic [22] or advanced [23, 33] *search techniques*, take inspiration from **machine learning** techniques or *logic reasoning* [4, 14, 25], or, like Hipikat and Strathcona, rely on ***heuristics***. Heuristics are a popular choice because they can describe the expected similarity among entities that are not related by frequent relations. Among classification techniques, those that are resilient to minor exceptions (like frequent item-set analysis, or association rules), are privileged over those that take into account all information (like formal concept analysis) so that idiosyncratic characteristics that distinguish different source code entities do not disrupt the result. In Chapter **??**, **(author?)** [27] provide a more elaborate discussion on data mining techniques often used in software engineering and recommendation systems. In Chapter **??**, **(author?)** [13] provide a more elaborate discussion on recommendation systems that rely on heursitics.

Filtering Filtering (a.k.a. **data cleaning**) aims at avoiding the analysis of code entities that are likely to produce **noise** in the results (**false positives**, irrelevant or uninteresting recommendations). Many SCoReS use a two-phase filtering approach. The first phase finds entities or attributes related to the user context. Discarding information in this first phase is also known as *pre-filtering*. Examples of pre-filtering include: using a blacklist of stop words, not including imported libraries, excluding system classes or generated code, etc. The second phase selects the most appropriate results from the first phase. Discarding information in this second phase is called *post-filtering*. Post-filtering aims at eliminating trivial, irrelevant or wrong recommendations, ordering them by relevance, and presenting only the most appropriate ones.

Filtering is usually done by comparing source code entities or other artifacts to the current context of the developer. This comparison can be done *by similarity* based on some measure of similarity [3, 6, 12, 23, 33], *by frequency* based on the number of occurrences as in FrUiT or Strathcona, or *by rating* based on some scoring mechanism as in CodeBroker or Strathcona. Both pre-filtering and post-filtering can be done by using similarity, frequency or rating information. Frequency is usually a proxy for rating [7] but frequencies tend to be preferred over ratings so that it is possible to provide recommendation even at early stages of the recommendation system (when no ratings have been given yet). Nevertheless, in order to use frequencies it is necessary to have several examples, and that might be one of the reasons for having API support as the preferred task supported by recommendation systems, at the expense of programming tasks and developer's questions [30] that might have more impact on their productivity. Another reason that might explain the few approaches using ratings is that early SCoReS reported them as being intrusive and a source of introducing noise [12]. Nevertheless, ratings can be a good way of collecting data on the usefulness of the approach and to move further towards using traditional recommendation techniques as opposed to mere mining techniques.

1.3.7 Detailed Input/Output

Decisions regarding the detailed input/output concern the detailed information that is required by the SCoReS (usually extracted from the developer's context or explicitly requested) as well as the quantity of results provided by the SCoReS.

Type of input In addition to the corpus (Subsection 1.3.4), the builder of a SCoReS needs to decide what additional information the system needs for its analysis. This decision is a refinement of the *input mechanism* and the *nature of the input* decisions described in Subsection 1.3.5.

Whereas some recommendation systems like Hipikat may start from a *search query* given by the user, others may start from a particular *source code entity* (partial implementations, as Strathcona or Rascal [12, 23] , or empty implementations, as CodeBroker [33]), *pairs of entities or artifacts* [5, 7, 8, 22] that need to mapped in some way, or *sets of entities* [11, 28] that may together represent some higher-level concept.

In case the user does not have a clear starting point, some systems allow finding it using any information that can be reached from the information provided, for instance, any identifier or literal in the file currently being edited.

[7] If something is used by a lot of people, it may be a sign that people like it, and thus that it deserves a good rating.

Multiplicity of output The quantity of recommendations to be produced by a SCoReS should be chosen with care. Providing concise and accurate results is of utmost importance in the design of recommendation systems. Limiting the results to a reasonable set allows developers to evaluate their appropriateness and value and to choose or to discard them accordingly. If a SCoReS produces a *single result* [5] this avoids the burden for the user to select the most relevant result, potentially at the risk of missing other relevant results. When *multiple results* [3, 6, 12, 23, 33] are returned, the burden of selecting the most appropriate ones can be reduced by *prioritizing* them rather than just showing an *unordered* list of results, so that the amount of time spent on evaluating recommendations to find a suitable one is minimized.

1.3.8 Support

This set of decisions focuses on aspects related to the suitability of the recommendations given by a SCoReS, with respect to the task it supports. It helps to foresee ways to *validate* how *useful* or *correct* the system is.

Empirical validation In case the builders of a SCoReS want to evaluate their system beyond example-based argumentation, it is important to decide what kind of empirical validations will be conducted to evaluate their system. Typical ways of validating a SCoReS are ***case studies*** (or in-depth analysis of particular applications) [11, 20, 28], ***benchmarks*** and/or *comparisons* with other systems [2, 7, 11, 12, 17], automated ***simulation of usage*** of the system [12], comparing the system's results against some kind of *oracle or gold standard* [7, 18, 22], ***controlled experiments*** with dependent and independent variables [22, 33], or perhaps even a *validation with practitioners* [1, 2, 5, 6, 12, 22, 28, 33] or a **field study**. Chapters **??**, **??**, **??** and **??** provide more details on such validation issues.

The most common types of validation include analyzing how developers use the system, or comparison with similar systems. The majority of approaches choose open source systems for their validation. In such cases it is essential to mention precisely the versions used, to allow for easy replicability of the results of the validation. Also consider whether or not to provide the corpus collected and the results obtained. Giving others access to this information allows them to validate and replicate each argumentation step, and facilitates the construction of benchmarks and simulations. It also offers the possibility to validate if the corpus was correctly built, and to identify differences between recommendations given by different SCoReS using the same corpus.

Usefulness Validating how relevant a SCoReS is for the task it supports [3, 20, 33] can be done along three axes: assessing whether the SCoReS *enables users to complete a given task* satisfactorily [6, 12, 22], *faster* [22], or with *less intellectual effort*.

Most often, to validate their system, SCoReS builders only conduct a case study to argue how useful the recommendations provided by their system are, but refrain from performing a more *quantitative validation* to provide more empirical support for their claims. Nevertheless, regardless of the type of validation chosen, be aware of the requirements that entail a particular validation choice. For instance, being able to claim satisfactory task achievement requires an unambiguous specification of the task and when it is considered completed, as well as a unique, correct response. Another example is the need for a valid proxy to assess claims (like intellectual effort). Finally, it is important to be aware that recommendations neglected by a user do not necessarily imply incorrect or invalid recommendations. From our analysis of existing SCoReS and their validation, we believe that there still remain a lot of opportunities for better assessment of the usefulness of recommendation systems.

Correctness Regarding the quality of the results produced by a SCoReS, typical correctness measurements include ***precision*** (percentage of correct recommendations) [6, 23, 33] and ***recall*** (to what extent the recommendations found cover the recommendations needed) [6, 23, 33]. *Relevance* [6, 12, 33] could also be measured but is usually a subjective view of the user, and therefore difficult to gather automatically.

Notice that in order to measure precision or recall it is necessary to establish *the* correct recommendations to compare against, for each user context. In practice, these ideal correct answers might be unattainable depending on the nature of the recommendations. Moreover, when conducting this type of validation, it is important to be able to argue that the user contexts analyzed are significant examples of typical user contexts.

1.3.9 Interaction

A final set of decisions focuses on how different types of users can interact with the SCoReS. The first type of users are the *developers* that will use the SCoReS as support for their work. Regarding this type of users it is important to assess how easy it is for them to interact with the SCoReS (***usability***) and to what extent they can easily get hold of the SCoReS (*availability*). A second type of users are *researchers* that may want to compare their own approach with the SCoReS. These users are also concerned by the *system's availability* but more importantly by the *data availability* that would allow them to reproduce and compare results.

Usability Regarding the usability of a SCoReS for its intended end-users (typically, developers), several criteria should be assessed carefully such as the system's **response time** (is it sufficiently fast for practical usage?) [1, 5, 17], *conciseness* of the results (are the system's recommendations sufficiently succinct and relevant for the end-user to analyze and use?) [12, 22, 28], *ease of use* (is the system sufficiently easy to use by its intended audience?) and ***scalability*** (is the system capable of handling larger software systems?).

Measuring the usability of a SCoReS is not straightforward, however. The easier aspects of usability that could be assessed by measuring are its conciseness or its response time. However, even then, unless the SCoReS's measurements are compared against those of similar systems (which might not be possible), by themselves these measurements may not provide a significant argument to use the SCoReS.

System Availability This decision describes under what form the system will be made available: as *source code* like FrUiT, as *binaries* only [7, 8, 11, 28], by providing only a *description* of the system as for Hipikat and Strathcona, or keeping it *unavailable* like Rascal and CodeBroker. This decision may vary significantly depending on what the intended goal of the system builders is, e.g., whether the SCoReS is a research prototype or a commercial system.

Availability of Recommendation Data The last decision to consider is whether or not the empirical results of the validation of the SCoReS will be made available. Leaving all data produced public allows other researchers to reproduce the validation, and to compare results between systems. There are several levels of data availability to contemplate. The *corpus* [18] gathered or used by the SCoReS, or the *results* [18] produced by the SCoReS on different subject systems could be published with the system or distributed upon request. Finally, the versions of the *subject systems* [3, 6, 12] analyzed could also be stored so that they could easily be used to create benchmarks or to compare the results already gathered for a given SCoReS with latter SCoReS (or future versions of the same SCoReS).

1.4 Building a code-based recommendation system

Having elaborated in Section 1.3 on the decision process and design decisions involved in building SCoReS, in this section we walk through a set of SCoReS that we have built, and discuss some of the design choices taken throughout their development and evolution. Figure 1.7 summarizes the history of these systems, starting from MEnToR [20] and Clairvoyant, both implemented in the Smalltalk language, via a first prototype of the Mendel system in Smalltalk [18], to its most recent reincarnations for the Java language (cf. decision `D5c` in Table 1.2). The case study serves a double purpose: it aims to illustrate the impact that certain design decisions can have, but also serves as an illustration of an actual process of building SCoReS. Tables 1.2 and 1.3 summarize the key decision choices for three of the systems in our case study (i.e., MEnToR and both the Smalltalk and Java implementation of Mendel). As we walk through the case study we refer to these tables to highlight the key design choices made. For instance, decision `D1a` refers to the *Intended user* decision in the *Intent* category of Table 1.2 and we can observe that all systems we developed intend to provide support for software *developers or maintainers*.

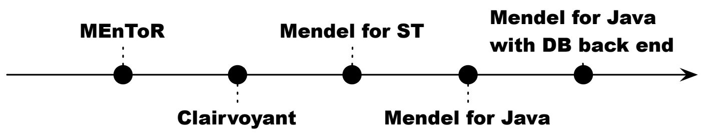


Fig. 1.7: From MEnToR to Mendel.

The recommendation systems which we built are based on four initial assumptions. First, we assume that source code is the most reliable and most up to date artifact in the software development process. This first assumption motivated us to consider only program code as input (`D4b`) and no complementary information (`D3b`). Furthermore, our SCoReS are particular in the sense that they don't use a corpus consisting of multiple applications but focus on the application under analysis alone to provide their recommendations (`D3a`). Second, we assume that a lot of implicit and undocumented design knowledge somehow gets codified in the code. More specifically, our third assumption is that they get encoded as coding idioms, naming conventions and design patterns that tend to be adhered to more or less uniformly across the application code. We refer to such codified design knowledge in the code as 'implementation regularities' (`D1d`). Our fourth assumption is then that understanding and maintaining those regularities consistent across the application (`D1b`) is beneficial to a program's comprehensibility and its evolution.

1.4.1 MEnToR

The first SCoReS we built upon these assumptions was called MEnToR (which stands for Mining Entities To Rules) [20, 21].

Requirements MEnToR aims to provide support to make *developers or maintainers* (`D1a`) discover and *understand* relevant *implementation regularities* (`D1b`, `D1d`). One problem is that, given that implementation regularities are usually implicit, often they are not perfectly adhered to. This observation lead us to establish some initial requirements for MEnToR. First of all, the underlying mining technique chosen had to be robust towards slight deviations. A technique tolerant to irregularities but still capable of *finding* (`D2b`) relevant implementation regularities was necessary to reduce false positives and false negatives. Secondly, in order to be useful for developers, we wanted MEnToR's results to be concise (`D8a`). Indeed, previous experiences with using techniques like formal concept analysis to mine source code for regularities [24, 26, 31] taught us that, due to redundancy and noise, the amount of results produced by such techniques was often too prohibitive to be usable in practice. Having few and unique results facilitate the adoption of the SCoReS by developers because it does not require much extra effort for analyzing the final results (`D2c`). Third, the result should indicate the intention or design decision behind

Table 1.2: Development decisions for three selected SCoReS (part 1)

Decisions	**MEnToR**	**Mendel for ST**	**Mendel for Java**
1. Intent			
a. Intended user	Developers or maintainers		
b. Supported task	Understand implementation regularities	Provide code suggestions about implementation regularities	
c. Cognitive support	What are the design decisions hidden in the implementation?	How to implement / improve a source code entity?	
d. Proposed information	Implementation regularities		
2. Human Computer Interaction			
a. Type of system	IDE plugin		
b. Type of recommender	Finder	Advisor	
c. User involvement	Evaluation of final results	Selection of final results	
3. Corpus			
a. Program code	Single application		
b. Complementary information	None		
c. Correlated information	*(not applicable)*		
4. General Input/Output			
a. Input mechanism	User chooses the application to analyze	Implicit: source code entities opened in the IDE's editor	Implicit: source code entities selected in IDE's editor
b. Nature of input	Source code		
c. Response triggers	Reactive	Proactive	Reactive
d. Nature of output	Textual description + UML-based visualization	Text links	
e. Type of output	Code regularities and entities that match them (or not)	Source code entities to add or modify	
5. Method			
a. Data selection	Identifiers, Extends, Implements	Identifiers, Extends, Implements, Calls, Types, Signatures, Protocol*	Identifiers, Fields declared, Methods implemented, Interfaces implemented, Classes extended, Types used, Methods called, Super calls, Exceptions thrown, Modifiers
b. Type of analysis	Syntactic and lexical		
c. Data requirements	Object Oriented / Smalltalk		OO / Java
d. Intermediate representation	Fact base		
e. Analysis technique	Association rule mining	Heuristics	
f. Filtering	By confidence, by support, using structural and heuristic filters	By similarity (family), by frequency (dominant/recessive recommendations), by rating.	By similarity (family) and by frequency (dominant/recessive recommendations)

* Protocols are tags used in Smalltalk to indicate the role of a method, and can also be regarded as indicators of the interface that a method implements.

Table 1.3: Development decisions for three selected SCoReS (part 2)

Decisions	**MEnToR**	**Mendel for ST**	**Mendel for Java**
6. Detailed Input/Output			
a. Type of input	Application to analyze	Source code entities in focus	
b. Multiplicity of output	Multiple results prioritized		
7. Support			
a. Empirical validation	Two case studies	Five comparisons against an oracle (simulation of SCoReS usage)	One comparison against an oracle (recommendations implemented during evolution of the subject system)
b. Usefulness	Less intellectual effort	Quantitative validations: complete code	
c. Correctness	Relevance	Precision and recall	
8. Interaction			
a. Usability	Concise results	Concise results, ease of use, scalability	
b. System Availability	Unavailable	Source code[*]	Unavailable
c. Data Availability	None	Corpus, results, subject systems	Subject system

[*] `http://ss3.gemstone.com/ss/Mendel.html`

the discovered regularities. The system needs to provide the developers with clues to let them understand the rationale of *why* certain source code entities are involved in some regularity (`D1c`).

Approach To fulfill these requirements MEnToR *mines association rules* (`D5e`) from diverse implementation characteristics (`D5a`) of the *application analyzed*, which is given as input by the user (`D4a`, `D6a`). MEnToR's approach bears a lot of resemblance with FrUIT (cf. Section 1.2.2), which mines association rules that represent structural relations in the source code of applications that use a given framework.

A concrete example of a rule found found by MEnToR is the rule:

$$\texttt{Id:`Collection'} \xrightarrow{75\%} \texttt{H:SequenceableCollection}$$

which indicates that 75% of the classes that have the keyword `'Collection'` in their name also belong to the hierarchy of class `SequenceableCollection`. Other characteristics taken into account (`D5a`) are the methods implemented. Note that, given that the set of classes that belong to the hierarchy of `Sequenceable-Collection` and the set of classes that have the word `'Collection'` in their name overlap, the following association rule could be mined as well:

$$\texttt{H:SequenceableCollection} \xrightarrow{60\%} \texttt{Id:`Collection'}$$

However, to reduce redundancy in the results, after postfiltering (`D5f`) MEnToR would report only the first rule, because it has a higher confidence. Furthermore, in order to discover more high-level design decisions that are more concise (D8a) and require less effort to understand (D7b), MEnToR also merges different association rules that affect an overlapping set of source code entities, into single, higher-level, regularities. But before merging association rules into regularities, they are first filtered (D5f) to eliminate rules that are too trivial or that have too low confidence. An example of a trivial rule is that all classes in the hierarchy of a subclass are also in the hierarchy of its super-class, while a rule with low confidence represents either very few entities or has a minimal overlap between characteristics. MEnToR also filters other information that can produce noise before and after calculating the regularities. Before calculating association rules (pre-filtering), MEnToR eliminates from the analysis source code entities that are likely to be too generic (like the class Object) as well as implementation characteristics that are very common (like the keyword 'get' which appears in many method names). After calculating the regularities, MEnToR then prioritizes them (D6b) by amount of entities they cover, by amount of implementation characteristics shared by those entities, and by taking into account the rating given previously to the regularity by other developers.

MEnToR is implemented as an IDE plugin (D2a) that performs its analysis upon explicit request by the user (D4c). Every time a reported association rule or regularity is selected, MEnToR updates a view showing the source code entities that respect the rule, as well as some of those that don't respect it but should, together with the characteristics that those entities do and don't share (D4e). This view helps developers in better understanding the regularities and how well they are adhered to in the code. Although the discovered rules are essentially shown in textual format, they can also be visualized as a UML diagram that marks with different colors the entities that respect the rule and those that do not, thus giving a visual clue of the extent of the rule and its deviations (D4d).

Limitations As useful as MEnToR could seem, after an initial validation on two case studies (D7a) we realized two main flaws of MEnToR. A first one was that it still required too much user involvement to evaluate the relevance of the discovered rules and regularities (D2c, D7c). The system was showing relevant regularities to increase the user's awareness of hidden design decisions in the code, even if the user was already aware of those regularities and even if they were perfectly respected by the code (in practice, the user was often more interested in those entities that were breaking the rules). As such, the system was not giving actionable information to the user. And for those cases where a regularity was not perfectly respected, even though the system highlighted those implementation characteristic that a source code entity was missing according to that regularity, the association rules often were often too verbose so that it was difficult for the user to assess how to use that information to improve code quality.

A second issue was that although MEnToR could show a developer whether or not a source code entity that was currently selected in the IDE lacked certain implementation characteristics, MEnToR did not cope well with the evolution of the application. Each time even a small change was made to the application, the fact base (D5d) of implementation characteristics could change and the association rule mining algorithm (as well as all subsequent processing) needed to be retriggered for the entire application, possibly leading to slightly different rules and regularities.

To solve the first issue we developed a new front-end for MEnToR called Clairvoyant. To solve the second issue we developed an entirely new SCoReS using a different technique inspired by a closer analysis of MEnToR's results.

1.4.2 From MEnToR to Clairvoyant

Clairvoyant is a new front-end for MEnToR that, depending on the adherence of the source code entity in focus in the IDE to a rule, provides more actionable information to the developer. Rather than seeing association rules as implications, we can also regard them as overlapping sets of entities that satisfy different characteristics. For example, the association rule `Id:'Collection'` $\xrightarrow{75\%}$ `H:SequenceableCollection` can be seen as two overlapping sets: the set of all classes that have the keyword '`Collection`' in their name and the set of all classes in the hierarchy of `SequenceableCollection`, where 75% of the entities in the first set also belong to the second set. We call the first set the *antecedent* of the rule, the second set its *consequent*, and the intersection of both sets the *matches*. Clairvoyant will flag an implementation characteristics as a likely *error* if the source code entity in focus is part of the antecedent but not part of the matches, as *satisfied* if the entity is part of the matching set, and as an implementation *suggestion* if the entity is part of the consequent but not part of the matches. Moreover, every time we click on a recommendation, Clairvoyant updates a view showing the matching source code entities as well as those that are in the antecedent but not in the consequent. Although Clairvoyant improved the appreciation of the system by developers, the fact that their changes were not reflected in the output of the system as soon as they made them, but still required the recalculation of all recommendations, made it an unrealistic system to support software developers and maintainers.

1.4.3 From Clairvoyant to Mendel

Mendel was developed to provide code suggestions about implementation regularities (`D1b`) to developers or maintainers (`D1a`) continuously (`D4c`). That is, as soon as some code entity is changed, Mendel would update its recommendations. However, Mendel also aimed at tackling three shortcomings in the results proposed by Clairvoyant. A first shortcoming is the amount of noise produced. Although developers showed an interest in the suggestions and errors recommended by Clairvoyant, some rules are redundant. Often, Clairvoyant finds regularities that are very similar to, or even subsets of, other regularities. In such cases, it is difficult to choose automatically which of those regularities make more sense to recommend or which are most informative while minimizing noisy information. Noisy information could be caused, amongst others, by implementation characteristics that are part of the regularity by accident, for example, because all entities sharing an important characteristic also happen to share a less relevant characteristic. What recommendation is perceived as 'best' may vary from one developer to another. Second, some regularities are accidental (e.g., 'accidental polymorphism' when a bunch of methods have the same name even though they don't implement a similar behaviour) whereas others are essential and capture entities implementing a similar concept, naming and coding conventions, or important protocols and usage-patterns among entities. Third, the relevance of what implementation characteristics to consider may depend on the type of source code entity analyzed (e.g., reasoning about message sends may be more relevant when analyzing methods than when analyzing classes).

To overcome some of these problems, we decided to build a new SCoReS called Mendel. Mendel is based on the concept of 'family' of a source code entity. The concept of family aims at eliminating regularities found by chance. Entities belonging to a same family are more likely to share implementation characteristics that indicate essential design decisions. Given that many of the relevant regularities recommended by MEnToR/Clairvoyant described entities belonging to a same class hierarchy, we compute the family of class by taking the direct superclass of that selected class and returning all of this superclass' direct subclasses, as well as the subclasses of these direct subclasses, except for the class analyzed (the class analyzed is excluded from the family). In other words, the family of a class are its siblings and nieces. The family of a method is defined as all methods with the same name that are in the family of its class.

The characteristics analyzed for a source code entity depend on its type. For methods, Mendel takes into account methods called, the method protocol[8], supercalls, referred types, and the AST structure. For classes, Mendel takes into account the keywords appearing in their name, implemented methods, and types used. These characteristics were chosen because they provide useful information to developers or maintainers to improve their code `(D1c)`. Based on Mendel's metaphor of a family's genetic heritage, the characteristics of a source code entity are called *traits*. The traits chosen were not exhaustive and we aimed to explore different characteristics depending on the obtained results. Finally, the frequency of occurrence of an implementation characteristic in the family indicates the likelihood of that characteristic being relevant. Therefore, we call implementation characteristics that are shared by the majority of the family the *dominant* traits of the family, while those that are shared by at least half of the members of the family (and that are not dominant) are called the *recessive* traits of the family. Dominant traits are shown as 'must' recommendations while recessive traits are shown as 'may' recommendations.

Both Mendel and Clairvoyant prioritize suggestions `(D6b)`. The key suggestions in Mendel are the dominant traits, which correspond to the likely errors in Clairvoyant, while the recommendations with lower confidence are the recessive traits in Mendel, which correspond to the suggestions in Clairvoyant.

[8] Smalltalk allows methods to be annotated with the category to which they belong, called a protocol.

Mendel's initial validation consisted of simulating Mendel's usage with a partial implementation. For each source code entity (i.e., class or method), the simulator temporarily removed its implementation (leaving only an empty declaration), asked Mendel for recommendations, and compared Mendel's recommendation with the entity that was removed. We ran such a simulation over five Open Source Systems of different domains `(D7a)`, and showed that Mendel proposed a limited amount of recommendations that were calculated quickly in real time `(D8a)`. We also concluded that at least half of the proposed recommendations were correct, and that Mendel discovered between 50 and 75% of missing traits `(7c)`.

The concept of family had two purposes: reducing the set of entities analyzed to give a recommendation so that the system could be responsive to source code changes 'on the fly', and reducing noise in the recommendations. As expected, Mendel's results are sensitive to the size of the family, which in turn depends on the depth and width of hierarchies in the analyzed software system. Moreover, manual inspection of some of the proposed recommendations indicated that the definition of family might not be the most appropriate one for all types of implementation characteristics.

We also did not manage to conduct a study with real developers, mainly because of the chosen programming language `(D5c)`. Although in the research labs where Mendel was conceived a few researchers were Smalltalk programmers and could thus have been invited as participants in a user study, it turned out that most of them had recently switched to another Smalltalk IDE (namely Pharo), making the first version of Mendel irrelevant (it was implemented in and integrated with VisualWorks Smalltalk). Therefore, Mendel was ported to Pharo. However, it then turned out that those developers who were willing to be part of our user study, were working on very small Smalltalk applications only, which would have resulted in too small families without dominant traits. For these reasons we decided to port Mendel to the Java programming language.

1.4.4 Porting Mendel from Smalltalk to Java

In addition to expanding the potential user base of our system (a.o., for validation purposes), our port of Mendel to Java had another goal. While porting it, we decided to extend it to be able to experiment with different alternative definitions of family, for recommending different implementation characteristics. For example, for a family of classes it could be more interesting to look at class-related characteristics such as inheritance, whereas for a family of methods it could be more interesting too look at method-related characteristics such as message sends. One of our master students thus implemented a prototype of an Eclipse plugin which supported several definitions of family, and storing the user rating of recommendations so that recommendations could be prioritized depending on how useful they had been for other developers.

As was already the case for the Smalltalk version, Mendel was designed to reduce the amount of computation needed for proposing recommendations and being capable of updating the recommendations ‘on the fly’ as soon as any change took place in the source code. An important performance problem was encountered to conduct automated validations, however. In order to validate and compare the recommendations given by different family definitions, it would be necessary to calculate the implementation characteristics for each family member, for each family definition, for each source code entity and this on each application analyzed. This approach proved to be very inefficient because it would recalculate the implementation characteristics several times for each source code entity. A more efficient approach therefore would be to calculate all implementation characteristics of each source code entity only once beforehand, and only then study the effect of choosing a different family definition. That was the motivation for implementing the latest version of Mendel: Mendel with a database storage back end.

1.4.5 Adding a database storage back end to Mendel

Several existing Java code repositories were considered to study the effect of Mendel’s family definitions on the quality of its recommendations. None of them, however, offered the level of detail Mendel required, while at the same time containing several versions of the code. The reason why we wanted to have several versions of the same systems being analyzed was because of the particular set-up of the validation we had in mind. Inspired by the kind of automated validation we conducted for the Smalltalk version of Mendel, we now wanted to validate whether a recommendation for a given source code entity in some version of a software system was relevant, not by first removing it and then checking it against itself, but rather by checking if it would actually be implemented in a later version of that system (and how many versions it took before that recommendation actually got implemented). To have the necessary information to conduct that experiment, we implemented another Eclipse plugin to gather all implementation characteristics of all source code entities for all versions of all systems to be analyzed, and stored these as structural facts in a database. We then experimented with and compared different alternative definitions of families to assess if they gave rise to significant differences in precision and recall per type of implementation characteristic on which the family definition was based. Although more experimentation is still needed, partial results of this analysis indicate that the choice of family definition indeed affects the correctness of the results (depending on the type of recommended characteristic) but also that families probably should not be described by a single implementation characteristic, but rather by a combination of different characteristics (as MEnToR’s regularities did).

1.5 Conclusion

This chapter provided a brief overview of existing source code based recommendation systems, including some we built. This overview illustrated the large variety of decision points and alternatives, when building a source code based recommendation system. We used this overview to highlight some of the key design decisions involved in building such systems. We organized these decisions along eight main categories, which were divided along two orthogonal dimensions. One dimension corresponds more or less to the development life cycle of the system, ranging from the elaboration of its requirements, through its design and implementation, to its validation. The other dimension focused either on the underlying approach, or on how the user interacts with the system. Regarding the approach, the main categories of design decisions to address are related to the intent of the system, the corpus of data it uses to provide its recommendations, the underlying recommendation method it uses, and how to validate how well the system supports the task it aims to provide recommendations for. Regarding the user interaction, the design decisions that need to be taken involve how the end user is expected to interact with the system, at different levels of detail. We also suggested a waterfall-like process in which to address all these decisions, but this process should not be regarded as restrictive. The design decisions could be visited in any other order that best fits the needs of the system builder. Our main message, however, is that it is important to address all these design decisions carefully and up front because, as we had to learn, making the wrong decision can have a significant impact on the quality and perceived or actual usefulness of the developed system. This set of key design decisions can also offer a useful frame of reference against which to compare different systems, to understand why one system is better, worse, or simply different from another one, or to steer the development of one's own system to better suit certain needs. In any case, we hope that our set of design decisions and proposed process can be of use to guide unexperienced builders of source code based recommendation systems in making them ask the right questions at the right time.

References

1. B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. Debugadvisor: a recommender system for debugging. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 373–382, New York, NY, USA, 2009. ACM.
2. Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.
3. Marcel Bruch, Thorsten Schäfer, and Mira Mezini. Fruit: Ide support for framework understanding. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, eclipse '06, pages 55–59, New York, NY, USA, 2006. ACM.

4. Sergio Castro, Coen De Roover, Andy Kellens, Angela Lozano, Kim Mens, and Theo D'Hondt. Diagnosing and correcting design inconsistencies in source code with logical abduction. *Science of Computer Programming*, 76(12):1113–1129, 2011. Special Issue on Software Evolution, Adaptability and Variability.
5. Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 214–225, New York, NY, USA, 2008. ACM.
6. Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, 31(6):446–465, June 2005.
7. Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 481–490, New York, NY, USA, 2008. ACM.
8. Ekwa Duala-Ekoko and Martin P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 79–104, Berlin, Heidelberg, 2011. Springer-Verlag.
9. Alexander Felfernig, Michael Jeran, Gerald Ninaus, Florian Reinfrank, Stefan Reitererand, and Martin Stettinger. Basic approaches in recommendation systems. In Martin Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann, editors, *Recommendation Systems in Software Engineering*, chapter ? Springer, 2014.
10. Daniel M. German, Davor Cubranić, and Margaret-Anne D. Storey. A framework for describing and understanding mining tools in software development. In *Proceedings of the 2005 international workshop on Mining software repositories (MSR'05)*, SIGSOFT Softw. Eng. Notes, 30(4), pages 1–5, New York, NY, USA, 2005. ACM.
11. Emily Hill, Lori Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 14–23, New York, NY, USA, 2007. ACM.
12. Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32(12):952–970, December 2006.
13. Laura Inozemtseva, Reid Holmes, and Robert J. Walker. Recommendation-in-the-small. In Martin Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann, editors, *Recommendation Systems in Software Engineering*, chapter ? Springer, 2014.
14. R. Wuyts K. Mens, I. Michiels. Supporting software development through declaratively codified programming patterns. *Journal on Expert Systems with Applications*, 23(4):405–431, November 2002.
15. Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 159–168, New York, NY, USA, 2005. ACM.
16. Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 306–315, New York, NY, USA, 2005. ACM.
17. Fan Long, Xi Wang, and Yang Cai. API hyperlinking via structural overlap. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 203–212, New York, NY, USA, 2009. ACM.
18. Angela Lozano, Andy Kellens, and Kim Mens. Mendel: Source code recommendation based on a genetic metaphor. In *Proceedings of the 26th IEEE/ACM international conference on Automated Software Engineering (ASE 2011)*, pages 384–387, Lawrence, Kansas, USA, 2011.

19. Angela Lozano, Andy Kellens, and Kim Mens. Usage contracts: Offering immediate feedback on violations of structural source-code regularities. *Science of Computer Programming*, 2013. Under review.
20. Angela Lozano, Andy Kellens, Kim Mens, and Gabriela Arévalo. Mentor: Mining entities to rules. In *9th BElgian-Netherlands EVOLution Workshop (BENEVOL), Lille*, 2010. http://hdl.handle.net/2078.1/91168.
21. Angela Lozano, Andy Kellens, Kim Mens, and Gabriela Arévalo. Mining source code for structural regularities. In *WCRE'10: Proceedings of the 2010 17th Working Conference on Reverse Engineering*, pages 22–31, Washington, DC, USA, 2010. IEEE Computer Society.
22. David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM.
23. Frank Mccarey, Mel Ó. Cinnéide, and Nicholas Kushmerick. Rascal: A recommender agent for agile reuse. *Artif. Intell. Rev.*, 24(3-4):253–276, November 2005.
24. Kim Mens, Andy Kellens, and Jens Krinke. Pitfalls in aspect mining. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE)*, pages 113–122. IEEE Computer Society, 2008.
25. Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views a case study. *Journal on Computer Languages, Systems & Structures*, 32(2–3):140–156, 2006. Special Issue: Smalltalk.
26. Kim Mens and Tom Tourwé. Delving source code with formal concept analysis. *Elsevier Journal on Computer Languages, Systems & Structures*, 31(3-4):183–197, 2005.
27. Tim Menzies. Data mining techniques in software engineering. In Martin Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann, editors, *Recommendation Systems in Software Engineering*, chapter ? Springer, 2014.
28. Martin P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 11–20, New York, NY, USA, 2005. ACM.
29. Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, July/August 2010.
30. Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.
31. Tom Tourwé and Kim Mens. Mining aspectual views using formal concept analysis. In *Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 97–106. IEEE Computer Society, 2004.
32. Tao Xie and Jian Pei. MAPO: mining API usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 54–57, New York, NY, USA, 2006. ACM.
33. Yunwen Ye and Gerhard Fischer. Reuse-conducive development environments. *Automated Software Engg.*, 12(2):199–235, April 2005.
34. Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical API usage. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 826–836, Piscataway, NJ, USA, 2012. IEEE Press.
35. Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 195–204, New York, NY, USA, 2010. ACM.
36. Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on*

Software Engineering, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.