Agents for Case-based Software Reuse

Stein Inge Morisbak & Bjørnar Tessem,

Department of Information Science,

University of Bergen,

5020 Bergen, Norway

email: {stein,bjornar}@ifi.uib.no

August 23, 2000

Abstract

The effective reuse of previously engineered components has become a core activity in any object-oriented software development project. The task is however often problematic when it comes to actual retrieval and understanding of the class library in e. g. Java. Especially newcomers in a software company or novices in Java programming will need time to obtain a good overview of available components. This article explores the use of intelligent agents in a case-based tool for software reuse. By introducing agent support to the retrieval mechanism of the tool we show how retrieval efficiency may be improved. The cooperating agents assist the user in retrieval of code for potential reuse in an automated way and in the background. This makes it possible for the developer to concentrate fully on her task. The tool aids in program understanding and adaptation. Thus it allows an exploratory approach to program development and increases reuse efficiency.

1 Introduction

The need for higher productivity in software development has led to a lot of research on how to reuse already made and tested software components (Krueger 1992). The introduction of object-oriented programming languages is perhaps the most useful step in the pursuit of more effective reuse in software development, but there are problems when it comes to retrieving and understanding object-oriented classes. In a object-oriented programming language like Java one of the main problems for a software developer is to get an overview of the class library. Especially newcomers in a Java software company or novices in Java programming will need time to obtain a good overview of available components.

To support programming in Java, Tessem, Whitehurst & Powell (1999) have developed a tool, which makes it possible to search through potentially reusable Java classes. The tool is based on case-based methods to support retrieval and reuse. In this paper, we describe an approach to add agent support to the retrieval mechanism of this tool. This goal is to increase retrieval efficiency and free the user's hands and mind making it possible for the user to concentrate fully on her task.

The tool helps the developer to locate code for potential reuse in an automated way and in the background. The tool further aids in program understanding and adaptation. Thus it allows an exploratory approach to program development and increases reuse efficiency.

A module in the editor of the tool extract data fields, constructor and method signatures from the code under development. An agent system running in the background is then able to use these signatures to identify potential candidates for reuse. Several variations are possible when it comes to organization of the multi-agent system running in the background. The approaches we discuss here are systems where each class is represented as a case in a case-based reasoning system (Aamodt & Plaza 1994), and further that each case in the system is represented by an agent.

In section two we will present some background material for this research, presenting the earlier version of the tool, case-based reasoning and softare agents. We continue with a description of the agent system in section three, emphasizing on the architecture of the multi-agent system and communication among agents. We then present some experiments in section four, and conclude with references to related work and ideas for further development in section five.

2 Background

The main purpose of this project is to implement an agent architecture suitable for retrieval of cases in a case-based reasoning tool for software reuse support. The work is an extension of the tool described by Tessem et al. (1999).

2.1 The original tool

The environment consists of an integrated editor, and a reuse tool. The reuse tool can be used on partial class specifications to identify components that may be candidates for reuse. In addition it supports the reuse process of the chosen components. The case-based reuse tool supports retrieval and reuse of classes based on their signatures (methods return types and arguments etc.), which in this case is viewed upon as cases. From these signatures one may also extract some knowledge about what kind of component this is. For instance, whether it is a collection of some kind. The reuse component may suggest mappings between signatures of a retrieved case and the target, and the user may accept or discard the suggestions. In addition the reuse tool suggests *how* to reuse a class, either by extension, or by lexical reuse of source code (if it is available). Java's reflective capabilities are used to extract case descriptions from compiled Java classes, and case-based reasoning is applied to support retrieval and adaptation of reusable components. The purpose of the tool is to localize potentially reusable code and to support the programmer in her program understanding and adaptation of the code.

Research on the original tool has shown that the set of features that can be automatically extracted utilizing the Java reflective capabilities (e. g., method signatures, field types, inheritance information, etc.) can be effectively used to retrieve components for subsequent reuse. The fact that case descriptions are extracted automatically increases the chances for acceptance of the approach in real programming environments and circumvents the high start-up costs traditionally associated with establishing repositories of reusable components.

Figure 1 is a conceptual diagram of the tool as presented by Tessem et al. (1999).

The system is also meant to include a Java single line interpreter. As development progresses, portions of the software could be sent to the interpreter for immediate feedback. This is shown in the figure as a grey box since it is for the time being not a part of the environment.

2.2 Case-based Reasoning

Case-based reasoning (CBR) is a general method for reasoning on the background of experience. CBR is a retainment model for representation, indexing and organization of previous cases, and a process model for retrieval and modification of old cases and assimilation of new ones. A case usually describes a problem situation.

A previously experienced situation, which has been captured and learned in a way

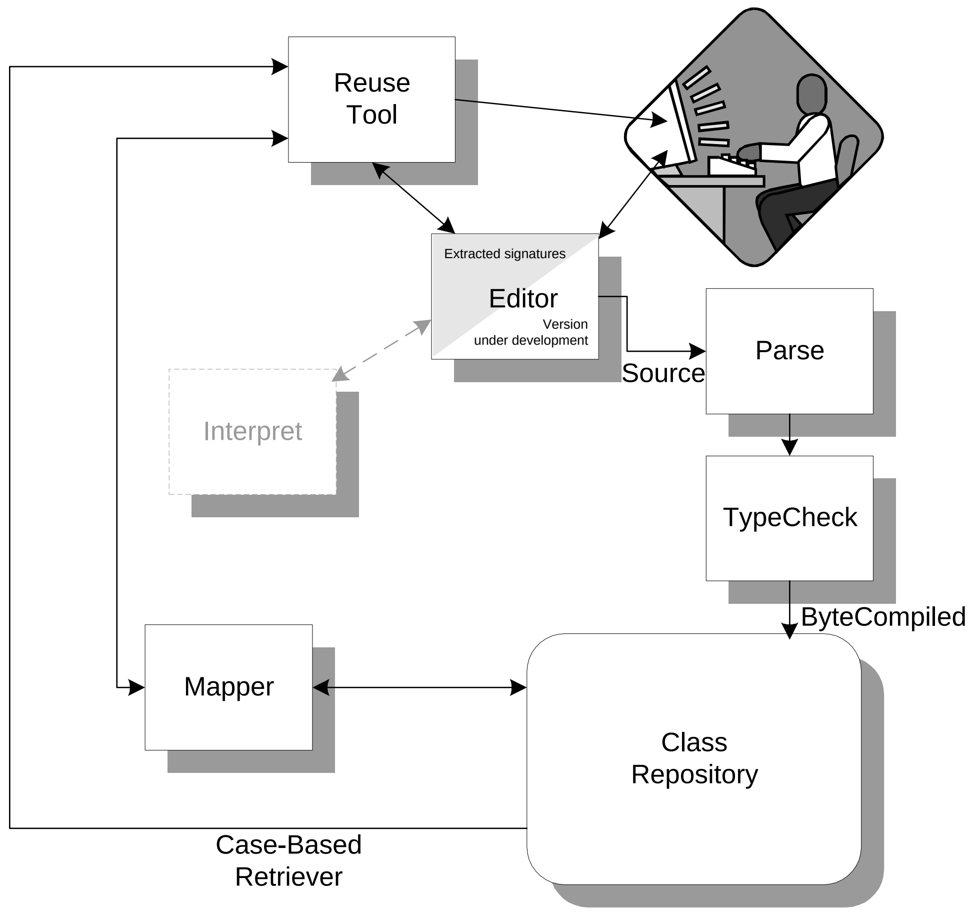


Figure 1: Conceptual Diagram of Rapid Development Environment

that it can be reused in the solving of future problems, is referred to as a base case, past case, previous case, stored case, or retained case. A new case (target case) is the description of a new problem to be solved. The technique utilizes similarities in problem descriptions to select and adapt previous solutions to new situations. Case-based reasoning is therefore a cyclic and integrated process that consists of solving a problem, learning from this experience, solving a new problem, etc.

Generally a CBR cycle may be described by the following four processes:

1. RETRIEVAL of the most similar case (base case).

2. REUSE of the information and knowledge in the base case. Adaptation and mapping of the base case to the present problem (target case).

3. REVISE the proposed solution (evaluate and correct).

4. RETAINMENT of the parts of this experience that may be useful for future problem solving. The new case (problem description and adapted solution) is stored in the case base for future reuses (the learning process).

Instead of general knowledge about a domain the system stores case descriptions and their associated solutions that are retrieved, adapted and utilized in the solution of new problems. There are many advantages with this approach. Some of them are:

- *Simple knowledge acquisition*: We record a human expert's solutions or experiences from real life to a number of problems and let a case-based reasoner select and reason from the appropriate case.

- *Generalization using cases*: This saves us the trouble of building general rules from the examples. Instead the reasoner would generalize the rules automatically through the process of applying them to new situations.

- *Learning*: After reaching a search based solution to a problem, a system can save that solution, so that next time a similar situation occurs, search would not be necessary. It can also be important to retain in the case base information about the success or failure of previous solution attempts; thus, CBR offers a powerful model of learning.

This description of CBR is based on Aamodt & Plaza (1994).

2.3 Why Java?

Java is a class-based object-oriented programming language supporting encapsulation, inheritance and polymorphism. These features guide system organization and are intended to promote the reuseability of the resulting software components.

Java is intended to allow application developers to write a program once and then be able to run it anywhere (Gosling, Joy & Steele 1996). This, and that the Java naming scheme facilitates the sharing of object classes among distributed developers without having to duplicate software modules, contributes to reuseability. Software development with Java can therefore be viewed as a constructive activity where system functionality is composed from a set of existing components.

Java Reflection enables Java code to discover information about the fields, methods, and constructors of loaded classes, and to use reflected fields, methods and constructors to operate on their underlying counterparts at runtime (Gosling et al. 1996). This capability allows us to extract feature descriptions from compiled classes without having access to the source code.

Java has a set of powerful mechanisms that directly support software reuse. However, the developer must have a sufficient knowledge of the language environment to be able to construct a mental mapping from existing object classes to the class that he wishes to construct. Java supports this to some extent in that it is possible to inherit the structure and functionality of an existing class and only specify new behavioral features in the new object class[1].

2.4 Agent technology

2.4.1 Agent Orientation vs. Object Orientation

Objects and agents share many things in common, but in our view differences exist which makes a useful and important perspective for system development. In traditional object orientation, objects are considered passive because their methods are invoked only when some external entity sends them a message. Software agents have their own thread of control, localizing not only code and state but their invocation as well. Such agents can also have individual rules and goals, making them appear like ”active objects with initiative” or as Jeffrey Bradshaw puts it ”objects with an attitude” (Bradshaw 1997). In other words, when and how an agent acts, is determined by the agent.

[1]Java does not provide multiple inheritance. The `interface` construct allows classes and their descendants to define and `implement` several interfaces as a set of methods. An interface can then be used by other classes as a form of contract. So, while Java only allows single inheritance for classes, a single class can implement multiple interfaces, and an interface can be defined as an extension of multiple other interfaces.(Jacobsen, Griss & Jonsson 1997)

Agents are regarded as *autonomous* entities because they can watch out for their own set of internal responsibilities. Furthermore, agents are *interactive* entities that are capable of using rich forms of messages. These messages can support method invocation as well as informing the agents of particular events, asking something of the agent, or receiving a response to an earlier query. Lastly, because agents are autonomous they can initiate interaction and respond to a message in anyway they choose. In other words, agents can be thought of as objects that can say ”No” as well as ”Go” (Odell 1999).

The autonomous and interactive character of agents more closely resembles natural systems than do objects. Since nature has long been very successful, identifying analogous situations to be used in agent-based systems is sensible. We can e. g. think of agents as being a part of an environment where they have the potentiality of surviving and succeeding, the ability to command resources and cooperate with others, and the possibility of failure, replacement and even death.

After decades, the term *intelligent* has still not been fully defined (or understood) for artificial systems and applying it to agents may not be appropriate. Most tend to regard the term *agent* and *intelligent agent* as equivalent. Perhaps this is just an attempt to communicate that agents have more power than conventional approaches. Agents could be, e. g. in comparison to relational data base tables or objects, thought of as somewhat ”smarter”. Anyhow, it would be fair to say that the notion of intelligence for agents could very well be different than for humans. We are not creating agents to replace humans; instead, we are creating them to assist or supplement humans. A different kind of intelligence, then, would be entirely appropriate.

An alternative view is that the interaction of many individual agents can give rise to secondary “intelligent” effects where groups of agents behave as a single entity. An agent organization consists of individual agents acting according to their own rules and individual goals, but the achievements as a whole can be viewed as greater than the sum of its individual contributors.

Emergent intelligence is viewed as a phenomenon resident in and emerging from a society and not just a property of an individual. Intelligence is reflected by the collective behaviors of large numbers of simple interacting agent. So whether we take these agents to be neural cells, individual members of species, or single persons in a society, their interactions produce intelligence.

Since the term intelligent is somewhat controversial in computer science and perhaps not applicable to the type of agents we have developed, we will use *agent*, not *intelligent agent*, as the term for the type of agents in this system. Instead we will assume a level of agent competency sufficient to allow them to communicate and work together to perform useful tasks, to accept or infer instructions or requests regarding its activities, and to use these to shape its autonomous activity decisions. Such agents are the building blocks of the agent organization. The agent system falls somewhere between a simple event-triggered program and one with human collaborative abilities.

2.4.2 Java Agents

Java provides all of the functionality required to design and implement software agents. Besides *Mobility*[2], a feature not used here, it provides support for *Autonomy* and *Artificial Intelligence*.

Autonomy - For a software program to be autonomous, it has to be a separate process or thread. Java applications are separate processes and may therefore last long and be autonomous. An agent can be a single thread. Java supports multithreaded applications and hence autonomy using both techniques.

Pattie Maes, head of MIT's Media Labs agents group defines an agent to be

> a process that lives in the world of computers and computer networks and can operate autonomously to fulfill one or more tasks (Maes 1994).

Agents are autonomous programs or processes. These processes are always ready to respond to a user's action or a change in the environment. The agents are informed about changes in the environment by messages being sent to them.

Intelligence - Two main aspects of Artificial Intelligence (AI) are knowledge representation and algorithms manipulating these. Knowledge representation is often based on the use of slots or attributes that store information about some entity, and chains or

[2]The Java Virtual Machine offers a homogeneous interface for Java processes, something that lets Java agents move between heterogeneous hardware systems.

references to other entities. Java objects may be used to code this data and behavior as well as relations between objects. Standard AI knowledge representation such as frames, semantic networks, and if-then rules may easily and naturally be implemented in Java.

3 Agents: Organisation and Inner Workings

As pointed out, a Java software developer is left with little support for acquiring an intimate knowledge of the contents of all the class libraries that are available for reuse. This may explain why object-oriented programming has such a steep learning curve (O'Shea 1986). Much of the effort of becoming an effective programmer in an object-oriented environment is expended in becoming familiar with the existing class libraries. This is made even worse if the class repository is dynamic (i. e., constantly being extended by multiple programmers).

The purpose of a case-based retrieval and reuse tool is to help the developer to locate reusable code and to aid in program understanding and adaptation. The tool matches Java classes from the class repository (base cases) to the target case (the class under construction) and then suggests similarities between them.

3.1 Agent Supported Case-based Retrieval

A user of the tool presented by Tessem et al. (1999) must repetitively and actively ask for reuse support. An alternative to this is to add to the system a set of agents observing the implementation the user builds, and when the similarity to a class is good enough notifies the programmer.

Figure 2 is a conceptual diagram of the Java programming environment with agents.

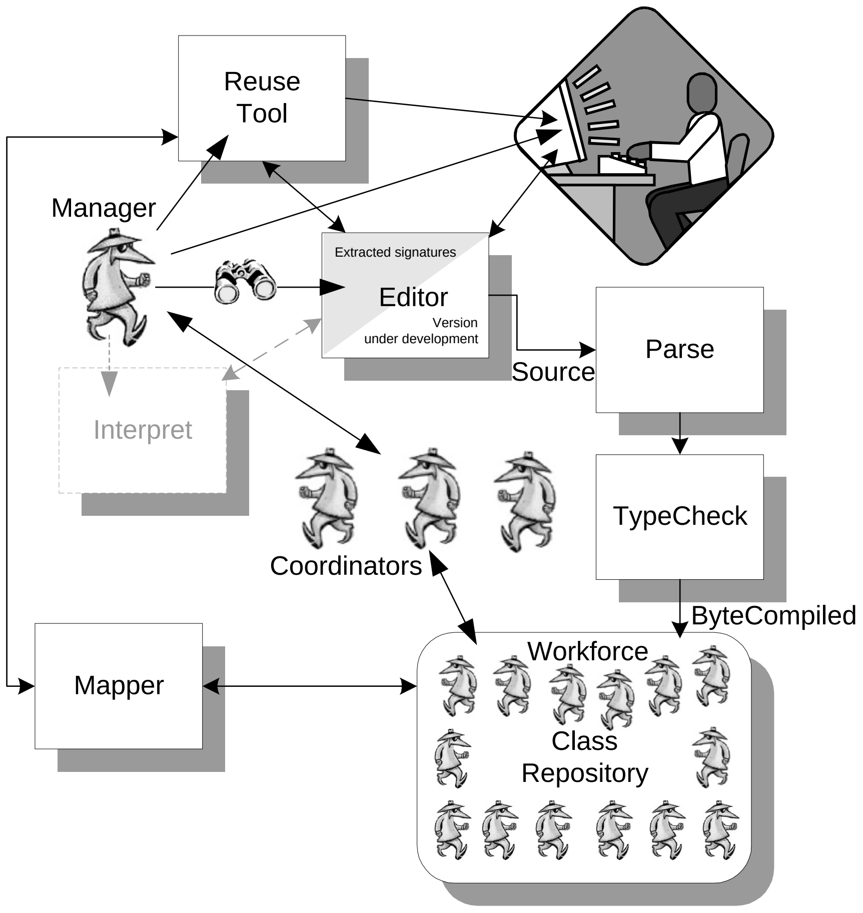


Figure 2: Conceptual diagram of a Java programming environment with agents

The Manager agent monitors the programmer's implementation via the extracted signatures of the partial class specification and extracts the target case when needed. The Manager agent then passes the task on to the appropriate Coordinator agents. Each Coordinator have more specific knowledge about a certain group of cases. If a Coordinator thinks that his group of cases may be relevant for the user, it sends a message to the workforce agents in this group. Such a group is in reality the class members of a Java package, and a single workforce agent is in reality a single case description of a Java class.

3.2 The Manager Agent

On the top level we find ”the Manager agent”. The Manager handles the interaction with the user and controls the retrieval process. When reusable cases are found the user is notified by a ”Case Found!” message in a small window at the bottom of the source editor. A menu pops up on the menu line containing to choices. The user may choose to ignore the agents message by pressing ”Ignore”, or she may start the reuse tool.

The Manager also monitors the coding. The user specifies how often (a time interval) or under what condition (number of code lines in the editor) the Manager should interpret the code and make a target case for the retrieval process. The Manager interprets the code following the user specification and passes a message to the next layer of agents, namely the Coordinators.

3.3 The Coordinator Agents

Coordinators are ”package cases”. The logical grouping of cases and the concept of Coordinators are based on the Java concept ”Packages” (Gosling et al. 1996). Java programs are organized as sets of packages. Each set has its own set of names for types, which help to prevent name conflicts. The naming structure for packages is hierarchical which is convenient for organizing related packages in a conventional manner. A Coordinator agent represents a single package in the repository as a ”package case”. A package case consists of all the types (method return types, fields and argument lists) referenced to in all the classes in a Java package. Each value has a significance attribution. The significance of a type in a certain package is a calculation of its number of occurrences in a case in relation to occurrences in each other package and in the whole repository. The significance of a type is hence a value used in the matching with the target case's types. If a type in the target case matches a highly significant type in a package it increases the Coordinators ”willingness” to fire an event to its workforce. If it finds that classes in the package contain highly significant types for this particular target case it will pass the target case to the workforce for further matching. Coordinators that have good match, but not the best, will keep on living in memory, but they won't fire events (See *Estimating The Relevance of a Target Type*). There's reason to believe that these Coordinators may be relevant when the user continues with extending her class. Therefore we keep them in memory for the next round of matching. If the Coordinators workforce is found to not contain many significant types for the target case, it dies.

3.3.1 Estimating the Significance of a Type

The estimation of a type's significance in a package is calculated in the following steps. The package `java.applet`, the class `java.applet.Applet` and the type `URL` is used as an example in a JDK1.1.8 repository containing 78 packages. `java.applet` contains four classes:

1. Count the number of occurrences $c(\kappa, \tau)$ of each particular type τ in all classes κ. For example, the number of references to `URL` in `java.applet.Applet` is

$$c(\texttt{java.applet.Applet}, \texttt{URL}) = 8$$

2. Divide the number of occurrences of a particular type in a single class by the total number of references to types in the class to get a normalized count $\bar{c}(\kappa, \tau)$ for the types. The formula below ensures that these counts add up to 1.

(1) $$\bar{c}(\kappa, \tau) = \frac{c(\kappa, \tau)}{\sum_{i=1}^{N} c(\kappa, \tau_i)}$$

`java.applet.Applet` has 55 different types and a total of 488 type references, so

$$\bar{c}(\texttt{java.applet.Applet}, \texttt{URL}) = \frac{8}{\sum_{i=1}^{55} c(\kappa, \tau_i)} = \frac{8}{488} = 0.016$$

3. Sum up all the results from step 2 for all classes in a package and receive a sum of all normalized counts $\bar{c}(\kappa, \tau_i)$ to get a *package count* for a type in a package.

   (2) $$pc(\mathcal{P}, \tau_i) = \sum_{j=1}^{M} \bar{c}(\kappa_j, \tau_i)$$

   where M is the number of classes in package $\mathcal{P}$. `java.applet` contains 4 classes and we get:

   $$pc(\texttt{java.applet}, \texttt{URL}) = \sum_{j=1}^{4} \bar{c}(\kappa_j, \texttt{URL}) = 0.524$$

4. Normalize the package count by dividing each $pc(\mathcal{P}, \tau_i)$ by the number of classes in the package

   (3) $$\bar{pc}(\mathcal{P}, \tau_i) = \frac{pc(\mathcal{P}, \tau_i)}{M}$$

   where M is the number of classes in $\mathcal{P}$. In our example we get

   $$\bar{pc}(\texttt{java.applet}, \texttt{URL}) = \frac{0.524}{4} = 0.131$$

5. Get an average $\bar{pc}$ for each type in the whole repository by dividing on the number Q of packages in the repository

   (4) $$\bar{\bar{pc}}(\tau) = \frac{\sum_{i=1}^{Q} \bar{pc}(\mathcal{P}_i, \tau)}{Q}$$

   JDK1.1.8 contains 78 packages and the total sum of all $\bar{pc}(\mathcal{P}_i, \texttt{URL})$ equals 1.790

so we get

$$\bar{\bar{pc}}(\texttt{URL}) = \frac{1.790}{78} = 0.023$$

6. We are now ready to calculate the significance of a particluar type for a package using the following formula:

$$\text{sign}(\mathcal{P}, \tau) = \frac{\bar{pc}(\mathcal{P}, \tau) - \bar{\bar{pc}}(\tau)}{\bar{\bar{pc}}(\tau)} \tag{5}$$

   The significance of `URL` in `java.applet` is then

$$\text{sign}(\texttt{java.applet}, \texttt{URL}) = \frac{0.131 - 0.023}{0.023} = 4.696$$

The significance value sign is used in a Coordinators matching of a *target case* with a *package case*. In other words, the Coordinator estimates a relevance value for the target case's types based on the significance of types in the classes contained in a package. This process assists in finding out if it is reasonable to believe that packages may contain case candidates for reuse.

3.3.2 Estimating the Relevance of a Target Type

The estimation of a package's relevance to a target case is calculated in the following steps:

1. Count the number $c(\varphi, \tau)$ of occurrences of the types in the target case φ

2. Calculate the relevance of each package $\mathcal{P}$ to the target case using the following formula:

   $$\text{rel}(\mathcal{P}, \varphi) = \sum_{i=1}^{N} c(\varphi, \tau_i) \cdot \text{sign}(\mathcal{P}, \tau_i) \qquad (6)$$

   where N is the number of classes referred to in φ.

3. If the total relevance rel$(\mathcal{P}, \varphi)$ is a positive number (larger than 0), it is a match. If the total similarity is larger than -0.5, the coordinator agent continues to live in memory. If else, the agent dies.

Unfortunately these heuristics do not give the wanted results alone. Some packages that contain relevant classes do not always get selected. We have therefore provided the Coordinator agents with some more knowledge.

Packages where the contained classes don't "belong together" firmly based on their relatedness, and where few special types exist (e. g. `java.lang` and `java.util`), will hardly ever be selected. We have therefore ensured that these very general packages should always be selected. The `java.lang` package contains the classes that are most central to the Java language, and Java depends directly on several of the classes (mostly data types) in the `java.util` package (Flanagan 1997). We believe this is a reasonable solution since these packages contain core classes used in almost any Java program.

We have also added two checks for class name similarity. The similarities are calculated based on a string alignment algorithm where, if a good alignment is found, similarity is set to the length of the longest identical substring relative to the length of the whole alignment. If the name of the target class is similar to the name of a package (e. g. `MyApplet`'s similarity to `java.applet` is 0.43), the package gets selected. If the name of the target class is similar to any name of a class contained in a package (e. g. `MyApplet`'s similarity to `JApplet` from `javax.swing` is 0.75), the package also gets selected. The target name similarity has to exceed 0.40 for package name similarity or 0.50 for class name similarity to select the package. According to *the Java Language Specification* (Gosling et al. 1996, p 106) it is recommended that naming conventions should be used in all Java programs, so we also believe this is a reasonable solution.

If a Coordinator agent, based on these calculations, thinks that his Workforce may contain potentially reusable cases, it passes a message to all the Workforce agents in the package.

3.4 The Workforce Agents

The individual case, or Workforce agent, possesses its own case description. The descriptions are created using Java's reflective facilities. Java allows any class to be asked for its methods, fields, constructors, inheritance information, and other information at run time. Java's syntactic reuse construct is the `import` statement. Java uses an environmental variable called `CLASSPATH` to establish where to search for classes that are mentioned as `import` statements. The agent supported case-based retriever traverses the directories on the `CLASSPATH` environmental variable, extracts all the feature information for each class in a pre-processing step and stores that information in a `.case` file associated with the class for later use. Each `.case` file is associated with a Workforce agent. When a base case is matched with a target case it obtains a value (see *Estimating the Similarity* below). This value (between 0-1) determines if the case is a candidate for reuse. If the match is good (greater than a predefined threshold) the Workforce agent offers itself as a potential case for retrieval. The user specifies the threshold the case has to match to be considered as a potential case for reuse. If the match evaluates to half of the threshold, the Workforce agent continues to live in memory but does not send an event. If not it ”kills” itself. There’s also here, as for the Coordinators, reason to believe that these Workforce agents may be relevant when the user continues with extending her class.

3.4.1 Estimating the Similarity

The estimation of the similarity between the target and the base is developed by Tessem et al. (1999).

The Workforce agents (base cases) estimate a similarity to the target class using similarities between pairs of methods, constructors, and data fields. To establish a similarity for a base case it does the following steps:

1. For each method, constructor, and data field in the base class use its signature to compute a similarity to each of the method signatures of the target.
2. For each method, constructor, and data field in the target select the most similar entry in the base class description and match it to this entry. As the entries in the base class are selected, mark them not-selectable.
3. The total similarity is the sum of the similarities of the selected matches in the target case. All similarities are represented as a number in the interval $[0, 1]$ where 1 indicates maximal similarity.

Similarity between each pair of entries (methods, constructors, fields) are computed using string identity for types, and string similarity for names. For methods the similarity is computed from three parts:

1. Similarity in return type. Identity gives similarity 1, otherwise 0.
2. Similarity in method name. Identity gives similarity 1, substring containment gives a similarity which is the size of the contained string relative to the containing string. It also runs a string alignment algorithm on the names, and if a good alignment is found, similarity is set to the length of the longest identical substring relative to the length of the whole alignment.
3. Similarity in arguments. If we let the collections of arguments for the two methods be represented as two bags A and B (bags are like sets, but may have multiple occurrences of same element), then the argument similarity can be given by

the formula:

$$\text{sim}(A,B) = 1 - \frac{|A-B| + |B-A|}{|A| + |B|} \tag{7}$$

The three similarities are weighted and added into a total similarity for the methods.

Similar approaches are used for constructors and data fields. For constructors only argument similarity counts, whereas for data fields type and name similarity counts.

3.5 Back to the Manager

The Manager collects all retrieved cases from the different packages. The best cases are sorted by how well they match the target case and are presented to the user. The leftover cases (or workforce agents) are kept alive, as they may become potential cases for reuse in the further development of the target case. In the next round of matching these leftover cases will be re-matched without having to re-read their features and invoke them again.

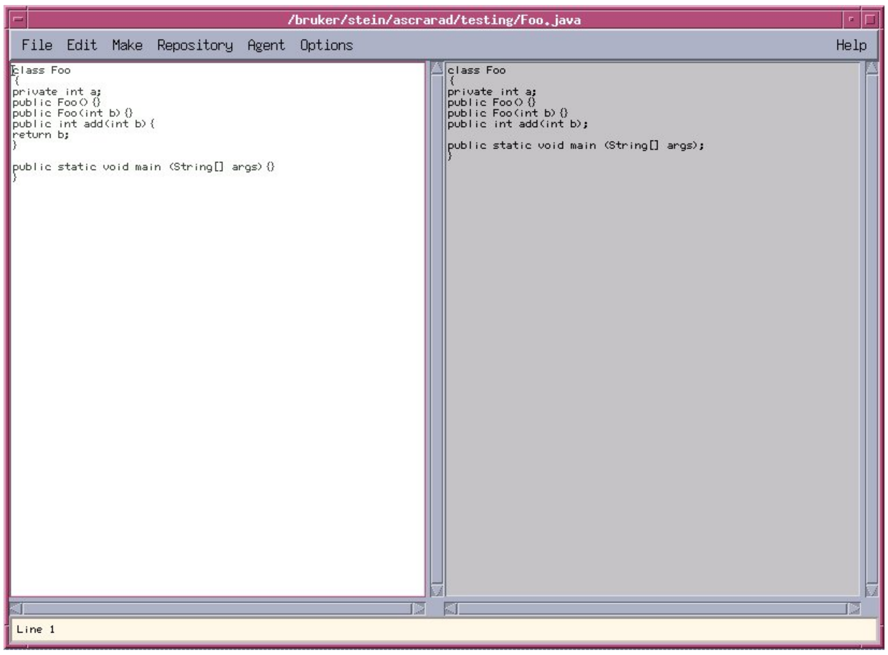
The manager agent's responsibility in this environment is to independently execute the case-based matching cycle at the right time and with satisfying feedback.

The agent is able to monitor the users' code, and when the similarity to a case class is satisfying enough, give feedback. For this to work smoothly the agent has to know when to pass the problem to the right coordinator(s), when to disturb the programmers concentration, and when to start retrieving a case proposed by the coordinators. The standards for this should be based on the nature of the problem and the programmers skill level and experience. It is possible for the users to adjust the system to their individual needs.

After retrieving a candidate case the agent provides the user with feedback about what it has carried out. The agent gives information about which alternatives the programmer has. The alternatives consist in either proposing adaptation of the retrieved case(s) with help from the reuse assistant, to continue the adaptation independently from the assistant, or to continue the programming with new searches for others and maybe more appropriate cases for potential reuse. It is of course also possible to stop the agent, put it to sleep for a while or to change other settings. The programmer is completely free to follow the agents' advice or to ignore it.

3.6 The Editor

The editor's graphical display is separated into two views (see Figure 3). The edit window, shown on the left, allows the developer to build and edit a class. The developer may either open a previously saved Java file or use the default class. The class window, shown on the right, reflects the class being developed in the edit window. The class window contains extracted signatures from the class being developed. This view is used to compile and/or interpret the current implementation. When the developer invokes the agents the extracted signatures of the software is used as the basis for the target class. As the developer extends, changes or starts on a new class the Manager



```
class Foo
{
private int a;
public Foo() {}
public Foo(int b) {}
public int add(int b) {
return b;
}

public static void main (String[] args) {}
}
```

```
class Foo
{
private int a;
public Foo() {}
public Foo(int b) {}
public int add(int b);

public static void main (String[] args);
}
```

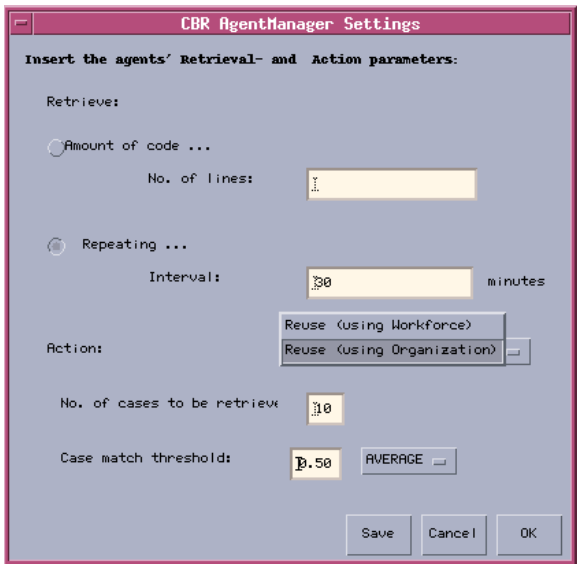
```
Line 1
```

Figure 3: Agent supported Rapid Prototyping Environment

agent re-reads the extracted signatures for each round of matching.

The Agent Settings Dialog is shown in Figure 4. In this dialog you specify how often the Manager Agent should invoke the reuse cycle in minutes. Alternatively you can specify that the agent should start when a certain number of code lines has been exceeded. When you've reached the limit of e. g. 30 code lines, the Manager will start the retrieval process. Next time it will start after reaching 30 more (60 code lines).

You can also specify which type of retrieval the Agent should use. You can choose between "Reuse (using Workforce)" which means that you try to match the target with all cases in the repository, or "Reuse (using Organization)" which means that you use "package matching" to decide which cases the target should be matched against. Furthermore you may specify number of cases to be retrieved. Lastly you specify the threshold. This value tells the agents how well the cases has to match the target to be considered for retrieval and reuse.


CBR AgentManager Settings

Insert the agents' Retrieval- and Action parameters:

Retrieve:

Amount of code ...

No. of lines:

Repeating ...

Interval: 30 minutes

Action: Reuse (using Workforce) / Reuse (using Organization)

No. of cases to be retrieve 10

Case match threshold: 0.50 AVERAGE

Figure 4: Agent Settings Dialog

3.7 Agent Organization

In order to get a large number of software agents to work together to perform complex activities effectively you have to choose some form of agent organization or architecture.

Competent agents that share common communication skills, viewpoints, and purposes form an agent society. An agent society is a very robust and flexible entity. Agents within a society can interact with one another (due to common language (see Chapter 3.8), identify the abilities and needs of each other (due to common viewpoints), and request or perform activities on behalf of others (due to common goals). Societal agents can determine that activities needs to be performed and enlist other agents to help out. The flexibility of an agent society comes at a cost, however: a substantial inefficiency in handling ongoing or repetitive activities. This cost begins with the problem of finding appropriate agents for an activity and continues through planning and executing the activity. The flexibility of an agent society starting at square one incurs the cost of always starting at square one.

Agent societies provide a basis for developing more structured agent organizations. What is an agent organization? For Galbraith (1977), an agent organization is;

1. composed of competent agents,
2. working together to achieve a shared purpose
3. through a division of labor,
4. integrated by decision processes
5. continuously through time.

An organization consists of patterns of behavior and interaction that are relatively stable and change slowly over time.

Sufficiently competent agents (humans) have the ability to organize themselves naturally. This is a direct result of the need to coordinate the work divided among the agents. In other words, organizing is a basic activity of competent agents. If agent organizations emerge naturally, why should we bother to design them? The standard answer for human organizations is that *properly designed organizations perform better than those that emerge naturally* (Corkill & Lander 1998). It is reasonable to expect this answer to hold for software agent organizations as well.

We will in the following try to explain the features of our choice of organization and why we have chosen it. Given an organization design, there must be a mechanism to implement it, both within each agent and in the agent-organization architecture. Three candidates, and explanations to why two of them where rejected, will be presented. I did however implement two of them (A comparison and test results may be found in Chapter 4).

3.7.1 Single Agent Architecture

The first model (Figure 5) proposed is a single-agent architecture and can hardly be called an organizational model. It consists of a single agent (A1) controlling the interaction with the user, the interpretation of the user's actions and work (the construction of a software component), and the retrieval of cases. The agent is dependent of a global retrieval mechanism similar to the one proposed in the original tool (Tessem et al. 1999). The advantage with this architecture is that the Agent is capable of controlling the retrieval process in the background without user intervention due to its autonomous capabilities.

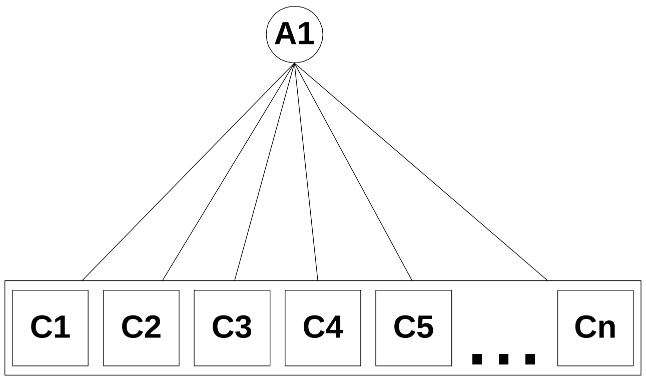


Figure 5: Single Agent Architecture.

3.7.2 Manager/Workforce Architecture (implemented)

A single agent may be pretty smart, but the value gained from agents coordinating their actions by working in cooperation is greater than that gained from any individual agent. This is where agents really come into their element.

The second approach (Figure 6) is a multi-agent system involving relatively uniform agents (Workforce agents (Aw1 to Awn)) with a Manager agent (Am) as facilitator, which handles the interaction with the user and controls the retrieval process. The Workforce agents have interaction capabilities for communicating with other agents, local decision-making capabilities for controlling the activities of the agent and the communications with the Manager agent, and task-level capabilities for performing the work of the agent. The difference between the Workforce agents in this architectural structure is only their content. The content of each Workforce agent is an actual case description.

Traditionally cases are viewed upon as passive objects waiting to be retrieved for fur-

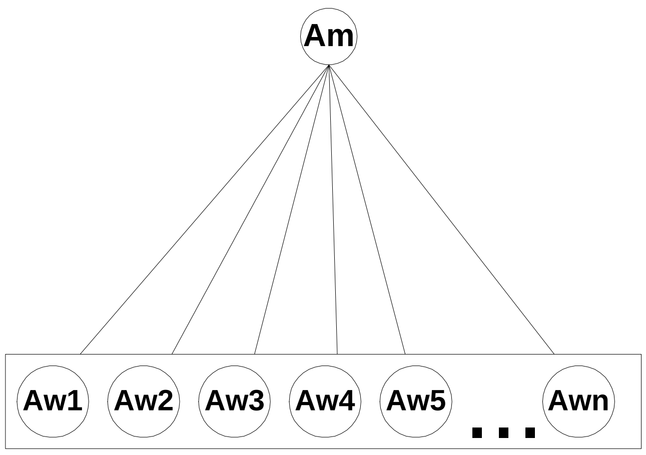


Figure 6: Manager/Workforce Architecture (Active Cases).

ther decision taking or problem solving. The decision whether cases should be retrieved is taken by a global mechanism. Ye Huang proposes in; *An Evolutionary Agent Model of Case-Based Classification* (Huang 1996), to let the stored cases play a more active role. The idea is to increase the flexibility of the retrieval process, and to let each case reflect over its own context and to let cases work together to obtain global goals. The active cases (or Workforce agents) are capable of matching themselves with a target case description received from the Manager Agent. Based on the matching results they may decide to offer themselves as potential cases for reuse, decide to wait for a further development of the target case that may match better its description, or simply die. If an agent decides to offer its case description as a potential candidate for reuse it has to compete with other agents. The Manager agent controls this ”war” and decides who will be offered to the user, who will get a second chance in the next round of retrieval and who will die. Instead of treating the cases in the case base as a passive mass of previous experience, this model expands cases to become active agents with knowledge and intentions.

A uniform-agent architecture is appealing. A single computational framework is developed for one agent and then replicated as needed. Each agent possesses the same social abilities as every other agent.

3.7.3 Manager/Coordinators/Workforce Architecture (implemented)

The last model is the organisation shortly described early in this section and consists of three agent layers where the agents are organized hierarchically using an enterprise analogy (Figure 7). In the ”enterprise” model the Workforce is split into ”departments” (p1 to pn) led by Coordinator agents (Ac1 to Acn). On top of the organization is the Manager agent (Am).

In this architecture the basic approach involves developing specialized agents whose organizational role is to handle the problem of finding the ”best men for the job”. This role has been realized in multiple facilitators called Coordinators. The Coordinators represent a layer of diversity in the organization, providing a connectivity and routing layer, in addition to the task level represented by the Workforce agents. Coordinators extract information about some basic features of the target case and decide whether the ”staff” in her department has the qualifications for solving the problem. The basic features extracted and reasoned about are data types used in the target case code. Furthermore, the Coordinators also check for string similarity in package- and class names. If a target case doesn’t contain common data types for the type of cases in the

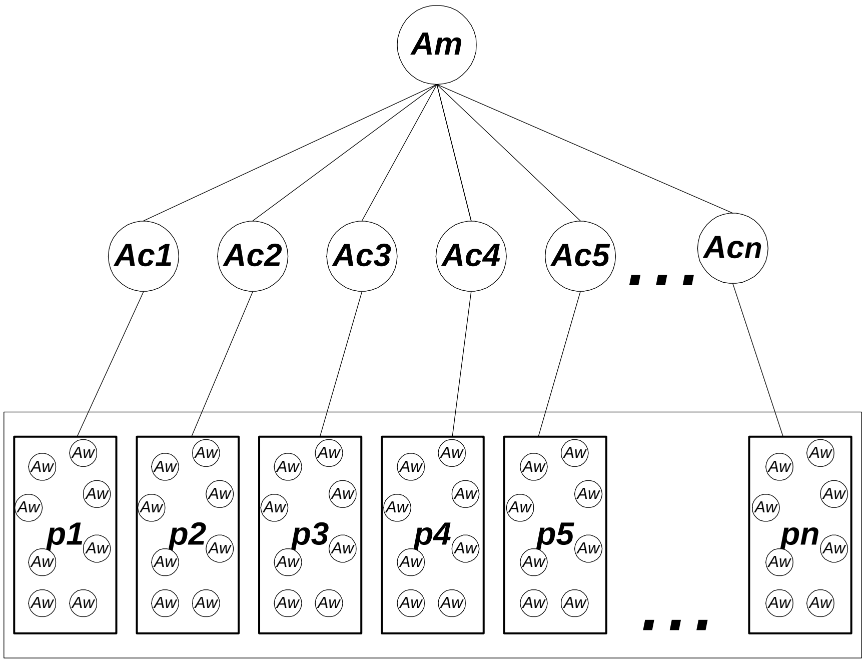


Figure 7: Manager/Coordinators/Workforce Architecture.

department, or the package doesn't contain classes with similar names to the target, the Coordinator will avoid sending messages to its workforce. Hence the Coordinator layer's task consists of forwarding messages to appropriate agents. Each department is constructed based on the package concept of Java.

The ”enterprise” model provides greater efficiency than a completely uniform multi-agent system, since the invocation of all the task-level agents is avoided. However, it recognizes only a single type of organizational diversity: that of the coordinators. In other words, only the coordinators may decide if a problem is ”their department”. The task-level agents have nothing to contribute with unless their coordinator sends them a message.

3.8 Agent Interaction

The agents communicate by sending messages to each other. All of the communications between the Coordinators, Workforce, and the Manager is done by sending *Agent Messages*. These messages are basically a collection of data that corresponds to some of the major slots of a KQML message.

KQML, which is an acronym for Knowledge Query and Manipulation Language (Labrou & Finin 1997), was conceived both as a message format and a message handling protocol to support run-time knowledge sharing among agents (Finin, Labrou & Mayfield 1997). This language can be thought of as consisting of three layers: a communication layer (which describes low level communication parameters, such as sender and recipient), a message layer (which contains a performative and indicates the protocol of interpretation); and a content layer (which contains information pertaining to the performative submitted).

Keywords used in KQML messages are defined as follows (Labrou & Finin 1997):

- performative: action, such as requesting or commanding.
- sender: agent sending the message.
- receiver: agent receiving the message.

- from: original sender; used when a message is sent using intermediary agents. (Not used here.)

- to: final recipient; used when a message is sent using intermediary agents. (Not used here.)

- in-reply-to: identifier of the message that triggered this message submission. (Not used here.)

- reply-with: identifier to be used by a message replying to this message. (Not used here.)

- language: language for interpreting the information in the content field of this message. (Not used here.)

- ontology: identifies the ontology to interpret the information in the content field of this message. (Not used here.)

- content: context-specific information describing the specifics of this message.

3.9 Some Extra Agent Messages

We have added a couple of extra fields for agent messages in this environment. These are:

- noCases: number of cases to be retrieved; the user may specify how many cases she wants to retrieve.

- targetCase: the target case

- agentCase: the base case

- threshold: the user specifies the threshold the case has to match to be considered a potential case for reuse.

EXAMPLE 3.1

```
(find-case
  :noCases 8
  :content null
  :receiver coAgent[i].getName()
  :sender this.getName()
  :targetCase MyString
  :agentCase null
  :threshold 0.75
)
```

♣

An example agent message is shown above (Example 3.1). The message starts with "find-case" which is the *action* (performative) intended for the message. There is no need for a *protocol of interpretation* (language and ontology) since our tool is a homogeneous environment with a single well defined language for interpreting the information in the content fields (i. e. the agents know how to handle the content of the message based on the performative, and the only language used in the environment is Java and text). The remainder of the message contains keywords needed for the content and communication layers.

The `:content` of a message is a Workforce agent which in this case is `null`, since the message is a query from the Manager agent and has not yet reached a Workforce agent. The message does not contain a `:agentCase` for the same reason. A `:agentCase` is the description of a retrieved *base case* from the repository and is added by the appropriate Workforce agent. The last field belonging to the content layer is the `:targetCase` field which is the case extracted from the class under development. The target case is in fact the content of the Manager agent; a single *base case* is a Workforce Agent while the *target case* is a Manager.

The `:sender` and `:receiver` parameters specify information at the communication level; `:sender` is the name of the agent sending the message, while `:receiver` is the name of the agent the message is aimed at. A Workforce agents `:receiver` parameter is always “the Manager”.

The following is an example of a message from a Workforce agent.

EXAMPLE 3.2

```
(case-matched
  :noCases 8
  :content this
  :receiver Manager
  :sender this.getName()
  :targetCase MyString
  :agentCase java.lang.String
  :threshold 0.75
)
```

♣

Example 3.3 shows a message from a Coordinator to a Workforce agent:

EXAMPLE 3.3

```
(match
  :noCases 8
  :content null
  :receiver workAgent[i].getName()
  :sender this.getName()
  :targetCase MyString
  :agentCase null
  :threshold 0.75
)
```

♣

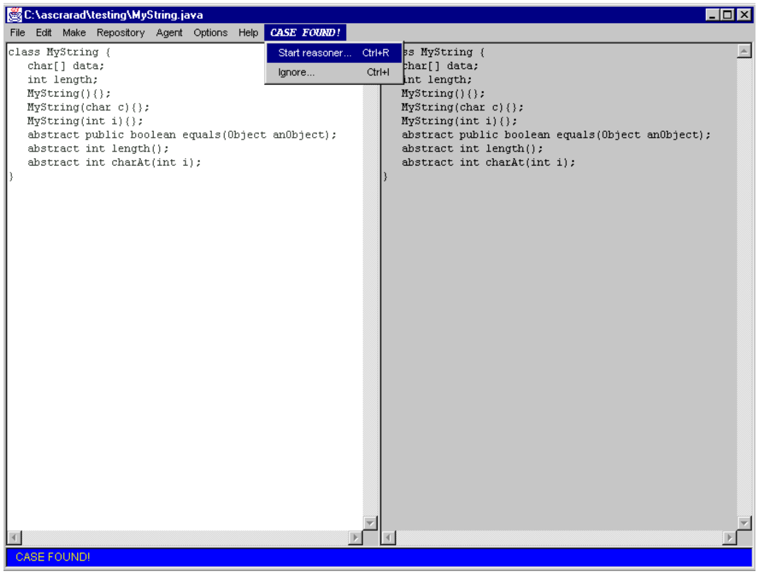
We have decided to maintain only one format for our agent messages. This results in an information overflow between the agents and also null values. Workforce agents for instance don't need information about how many cases should be retrieved(`:noCases`). The `:agentCase` and `:content` slots will have null values until it reaches a Workforce agent. The choice is made for simplicity, flexibility and extendibility reasons. A uniform message format is easier to maintain and extend. Furthermore the meaning of the slots is easier to understand when they are common to all messages in the system. The optional fields, such as `:agentCase` and `:content`, can be viewed as containers requesting to be filled according to the performative submitted. I believe that our choice also makes it easier to extend the system to become networked (see Section 5.2 *Further Development*). A common format facilitating high–level communication is important and essential for distributed agents on the internet.

4 Example Run and Efficiency Comparisons

In the following we will present an example run illustrating how a user might experience working with the tool. The example run in section 4.1 also gives more insight into the use of the reuse tool/assistant. We will also present results indicating the efficiency for both of the implemented architectures (See section 3.7). The tests are run under the WinNT operating system.

4.1 Example Run (users experience)

Suppose the user is working on the `MyString` class. After adjusting the settings in the Agent Settings Dialog (see Figure 4) and starting the agent, the Manager agent autonomously, creates a target case description from the partial class specification and starts a retrieval cycle. The user continues her work while the agents run continuously without intervention. After a while, the agents report that cases are found.

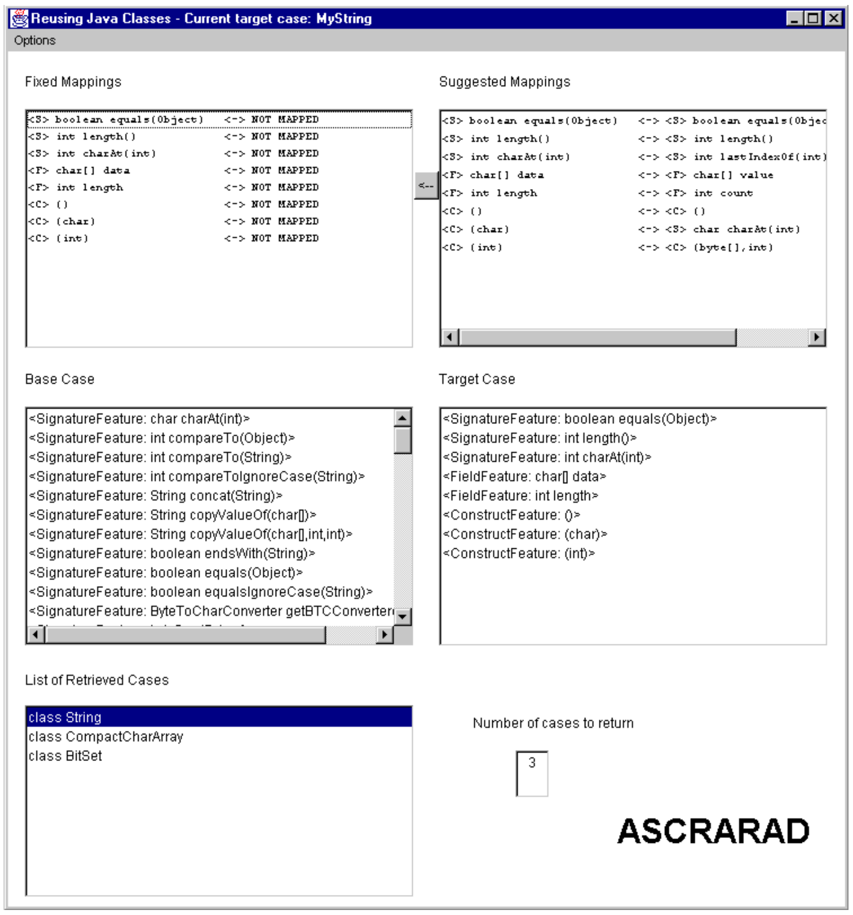


```
class MyString {
   char[] data;
   int length;
   MyString(){};
   MyString(char c){};
   MyString(int i){};
   abstract public boolean equals(Object anObject);
   abstract int length();
   abstract int charAt(int i);
}
```

```
ss MyString {
char[] data;
int length;
   MyString(){};
   MyString(char c){};
   MyString(int i){};
   abstract public boolean equals(Object anObject);
   abstract int length();
   abstract int charAt(int i);
}
```

Figure 8: Cases are found

The user starts the Reuse Tool by choosing the *Start reasoner...* option from the newly arrived *Case Found* menu on the menu bar as shown in Figure 8. The user may also choose to ignore the *Case Found* message if she wants to continue programming without intervention.


Reusing Java Classes - Current target case: MyString

Options

Fixed Mappings

```
<S> boolean equals(Object)    <-> NOT MAPPED
<S> int length()              <-> NOT MAPPED
<S> int charAt(int)           <-> NOT MAPPED
<F> char[] data               <-> NOT MAPPED
<F> int length                <-> NOT MAPPED
<C> ()                        <-> NOT MAPPED
<C> (char)                    <-> NOT MAPPED
<C> (int)                     <-> NOT MAPPED
```

Suggested Mappings

```
<S> boolean equals(Object)    <-> <S> boolean equals(Objec
<S> int length()              <-> <S> int length()
<S> int charAt(int)           <-> <S> int lastIndexOf(int)
<F> char[] data               <-> <F> char[] value
<F> int length                <-> <F> int count
<C> ()                        <-> <C> ()
<C> (char)                    <-> <S> char charAt(int)
<C> (int)                     <-> <C> (byte[],int)
```

Base Case

```
<SignatureFeature: char charAt(int)>
<SignatureFeature: int compareTo(Object)>
<SignatureFeature: int compareTo(String)>
<SignatureFeature: int compareToIgnoreCase(String)>
<SignatureFeature: String concat(String)>
<SignatureFeature: String copyValueOf(char[])>
<SignatureFeature: String copyValueOf(char[],int,int)>
<SignatureFeature: boolean endsWith(String)>
<SignatureFeature: boolean equals(Object)>
<SignatureFeature: boolean equalsIgnoreCase(String)>
<SignatureFeature: ByteToCharConverter getBTCConverter
```

Target Case

```
<SignatureFeature: boolean equals(Object)>
<SignatureFeature: int length()>
<SignatureFeature: int charAt(int)>
<FieldFeature: char[] data>
<FieldFeature: int length>
<ConstructFeature: ()>
<ConstructFeature: (char)>
<ConstructFeature: (int)>
```

List of Retrieved Cases

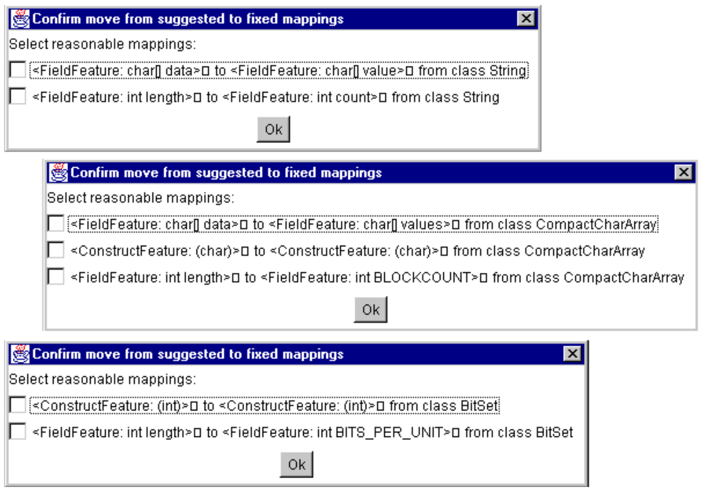
class String
class CompactCharArray
class BitSet

Number of cases to return

3

Figure 9: The Reuse Tool

The Reuse Tool shown in Figure 9 suggests mappings between signatures of retrieved cases and the target. The user may accept the suggestion by choosing it and pressing the button marked with an arrow. Notice the suggested mapping on the third line in the *Suggested Mappings* list. The return type of `int charAt(int)` does not match to `java.lang.String`'s `char charAt(int)` and hence the match is not found. However, the developer should detect her mistake at this point and correct her specification before the agents start a new retrieval cycle or just continue the reuse process with the retrieved classes.


Confirm move from suggested to fixed mappings

Select reasonable mappings:

- [ ] <FieldFeature: char[] data>□ to <FieldFeature: char[] value>□ from class String
- [ ] <FieldFeature: int length>□ to <FieldFeature: int count>□ from class String

Ok

Confirm move from suggested to fixed mappings

Select reasonable mappings:

- [ ] <FieldFeature: char[] data>□ to <FieldFeature: char[] values>□ from class CompactCharArray
- [ ] <ConstructFeature: (char)>□ to <ConstructFeature: (char)>□ from class CompactCharArray
- [ ] <FieldFeature: int length>□ to <FieldFeature: int BLOCKCOUNT>□ from class CompactCharArray

Ok

Confirm move from suggested to fixed mappings

Select reasonable mappings:

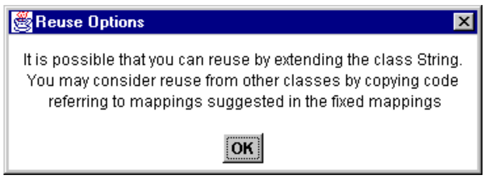
- [ ] <ConstructFeature: (int)>□ to <ConstructFeature: (int)>□ from class BitSet
- [ ] <FieldFeature: int length>□ to <FieldFeature: int BITS_PER_UNIT>□ from class BitSet

Ok

Figure 10: Suggestions

The user may also ask for the system to run in a more interactive manner where the tool asks the user whether she would like to keep the fixed mappings and continues by presenting mapping suggestions for the user to accept or discard. In this mode it still suggests mappings between less similar pairs, and asks for verification (Figure 10). The user is of course allowed to assert and retract mappings throughout the reuse process. In addition the reuse tool suggests how to reuse a class. Depending on the total similarity between the cases the system suggests direct reuse, subclassing (extension), or lexical reuse of source code (copy-and-paste).

For the signatures that were not mapped from the `String` class, the reuse tool searches in the rest of the retrieved classes, and tries to find appropriate mappings. After all cases have been tried, the reuser suggests how to reuse by printing a statement like in Figure 11


Reuse Options

It is possible that you can reuse by extending the class String.
You may consider reuse from other classes by copying code
referring to mappings suggested in the fixed mappings

Figure 11: Reuse Options

When the user is done she may continue extending her class. Hopefuly she's come a little further with help from the agents and the reuse tool. If she's set the agent settings to restart after a certain interval or number of code lines the agents will start their retrieval cycle again and hopefully come up with new and improved reuse suggestions.

4.2 Efficiency comparisons

We created two repositories for comparing the approaches with regard to time used to retrieve a set of cases:

- Version 1.1.8 of the Java^{TM} Development Kit with 1581 cases and 78 packages.

- Version 1.3 of the Java^{TM} Standard Development Kit with originally 7174 cases and 241 packages. We have created a smaller repository by removing all sun.* packages, all com.* packages and the netscape.javascript package. Only packages that are part of the Java^{TM} 2 Platform, Standard Edition, v 1.3 API Specification are included in the repository (Other packages may be a part of Sun's Java 2 SDK and Java 2 Runtime Environment distributions). The repository used in the tests consists of 3256 cases and 79 packages.

The example target class we used with the repository built from version 1.1.8 of the JDK consisting of method prototypes is shown below.This is the same case as used by Tessem et al. (1999).

```
class MyString {
   char[] data;
   int length;
   MyString(){}
   MyString(char c){}
   MyString(int i){}
   abstract boolean equals(Object anObject);
   abstract int length();
   abstract int charAt(int i);
 }
```

The example target class we tried with the repository built from version 1.3 of the JSDK is:

```
class MyApplet
{
   String urlName;
   URL myUrl;
   int width;
   int heigth;
   MyApplet(){};
   abstract URL getCodeBase();
   abstract void play(URL url);
   abstract void resize(int width, int heigth);
 }
```

The agents were asked to retrieve 8 cases with a case match threshold of 0.5:

4.2.1 Results Manager/Workforce

The eight classes retrieved for the `MyString` class from version 1.1.8 of the Java^{TM} Development Kit were:

1. java.lang.String
2. java.text.CompactCharArray
3. java.util.BitSet
4. java.lang.StringBuffer
5. java.util.Vector
6. java.util.GregorianCalendar
7. java.util.Hashtable
8. java.text.CompactIntArray

It took 1 min. and 4 sec. to run on find these cases.

The eight classes retrieved for the `MyApplet` class from version 1.3 of the Java^{TM} Standard Development Kit were:

1. javax.swing.JApplet
2. java.applet.Applet
3. javax.swing.JEditorPane
4. javax.swing.JTextPane
5. javax.swing.text.html.HTMLDocument
6. javax.swing.text.html.StyleSheet
7. javax.swing.JDialog
8. javax.swing.JTextComponent

It took 3 min. and 11 sec. to find these classes

4.3 Results Manager/Coordinators/Workforce

The eight classes retrieved for the `MyString` version 1.1.8 of the Java^{TM} Development Kit were in this case exactly the same as under the Manager/Workforce Architecture (4.2.1) It took 49 sec. to run the retrieval.

The eight classes retrieved for the `MyApplet` class from version 1.3 of the Java^{TM} Standard Development Kit differed slightly and were:

1. java.applet.Applet
2. javax.swing.JApplet
3. javax.swing.JEditorPane
4. javax.swing.JTextPane
5. javax.swing.text.html.HTMLDocument
6. javax.swing.text.html.StyleSheet
7. javax.swing.JDialog
8. javax.swing.JToggleButton

In this case it took 2 min. and 20 sec. to retrieve the cases.

4.4 Test Results Summary

The efficiency comparison runs and the example run described above show that the agents are capable of retrieving appropriate components for subsequent reuse autonomously and in the background. The system as a whole identifies the components which are candidates for reuse, and also provides support in the process of adapting them to the target class. The unobtrusive interface allows the user to ignore or pursue the agents suggestions, and the reuse tool guides the adaptation and reuse process by identifying good candidates if the user wants it.

The timing results presented in Tessem et al.'s earlier version were that it took about 2 minutes on a Sun SPARC-10 on a repository consisting of over 1700 cases. The results are hardly comparable for the Java 2 tests since the size of the repositories differ and since the versions of the JDK's are not the same (Tessem et al. used version 1.1.4 of the JDK). And of course, the tests were run on different computer platforms.

Even though the retrieval seems to give the wanted results, it takes some time using the Java 2 based repository. The time issue is however no longer as critical when using agents. The agents do their work in the background without interfering with the users work.

The results show that the Manager/Coordinators/Workforce architecture is faster than the Manager/Workforce architecture. Most of the time they also retrieve the same cases, but it would be reasonable to conclude that the Manager/Workforce Architecture is more secure (no cases are left un-matched). Some adjustments to the package matching heuristics should be considered.

To create the complete class repository takes hours for Java 2 libraries, but since this is done only when the class library is updated we don't consider the efficiency of this to be of particular significance.

5 Conclusions and Further Development

5.1 Conclusions

One mechanism for decreasing software development time is the effective reuse of previously engineered components. A software company may have a large repository of previously developed components, and to make the reuse of these more effective, will contribute a great deal to the development process. Not only will it be more effective, it will also contribute to a development where a certain ”company style” will be followed. An example of this is ”look and feel”. Software coming from the same firm should have the same ”look and feel” and similar basic features. These mechanisms are especially helpful for newly employed and novices since they don’t have the experience working with and the overview of the firm’s repository.

This study, and that of Tessem et al. (1999) has shown that the set of features that can be automatically extracted utilizing the Java reflective capabilities (e. g. method signatures, field types, inheritance information, etc.) can be effectively used to retrieve appropriate components for subsequent reuse.

The introduction of agents in the environment has been successful. They have proven to provide their retrieval tasks in time and more efficiently than if the user were controlling the process. The autonomous agents free the users’ hands and mind and makes it possible for the user to concentrate fully on her task. The user may continue working while the agents provide helpful assistance in the background.

Thus, we have reached the goals of developing a tool that fulfills the requirements we set forth. The tool helps the developer to locate code for potential reuse in an automated way, and the tool aids in program understanding and adaptation. It allows an exploratory approach to program development and optimizes reuse efficiency.

5.2 Further Development

We will in the following include some suggestions and thoughts concerning further development of this tool. There are many possibilities for extending and enhancing the solution. Some of them would require future research, whereas others could have been developed right away, for instance the completion of the environment to a full featured tool for editing, reusing, testing, and running Java applications.

Distributed case libraries on the Internet is a challenging possibility for further development of the agent support. This can be made possible utilizing Javas capabilities of letting agents move between heterogeneous hardware systems, KQML messages and Java reflection. A lot of research is going on in the field of corporate memories and distributed case libraries (Prasad, Lesser & Lander 1995, Prasad & Plaza 1995, Knoblock, Arens & Hsu: 1994, Plaza, Arcos & Martín 1997, Plaza, Arcos & Martín 1999, Prasad 1998, Prasad, Lesser & Lander Japan).

Within CBR possible improvements could be:

- Improve the package matching. It should be possible to improve the package matching so that fewer packages are selected during retrieval. An example of how this could be done is to somehow find out what *kind* of class is being developed (Is it e. g. a GUI class, a collection of some sort or an IO class). By extracting knowledge to recognize different types of classes either automatically using some heuristics, or by letting the user specify it herself, we should be able to select candidate packages better and to leave non-candidates unselected.

- Inferring features of a class to other kinds of features. It should e. g. be possible to extract knowledge about design patterns (Gamma, Helm, Johnson & Vlissides 1995) in which a class participate.

- Should also be possible to extend the case-based approach to include cases containing reuse experiences and not only classes. This could be done by indexing a case with a specification and perhaps a class reused and *how* it was reused. Similarity can then be estimated based on similarity between the specifications in two cases.

- To avoid the process of querying all coordinator agents each time a new retrieval cycle starts and all workforce agents for selected packages it could be possible to rank the cases based on how often they are selected and how well experiences with the reuse of them are. “Good” cases (both co-ordinators and workforce) would then fire more often than cases we don’t have very good reuse experiences with. Sànchez-Marrè, Béjar & Cortés (1997) proposes an evolutionary solution where agents “forget” seldom used cases and prefer often used cases.

- Convert the repository from files to some sort of database for efficiency.

- Lastly, and perhaps a bit far-fetched at the moment, it could be possible to extract case features from the comments in a class. This would require advanced language interpretation.

5.2.1 Evaluation

An evaluation of the tool in a realistic environment is important in order to see the effects case-based reuse support may have on target users. These experiments should be executed with “real users” in a “real environment” where their tasks are similar to what they do every day. The perfect user would be a newly employed software programmer working in a software company where rapid prototyping is an important task. The company should have a repository of previously developed software components and a wish to improve reuse and efficiency in the software development process.

5.3 Related Research

There have been several attempts to use case-based reasoning (CBR) to support software engineering, but few combine the technique with agent support.

5.3.1 CBR and Software Engineering

Some approaches to support software reuse using CBR focus on the earlier phases of a software development process (Miriyala & Harandi 1991, Maiden & Sutcliffe 1992, Spanoudakis & Constantopoulos 1996, Tessem & Bjørnestad 1997), others discuss the possibility of reuse of code by identifying similarity in designs or specifications, assuming that this similarity leads to similarity in code (Dershowitz 1986, Whitehurst 1995, Althoff, Nick & Tautz 1998), and a couple of authors focus on case-based reasoning as a tool to enhance organizational learning in software factories (Henninger 1997, Althoff, Birk, Gresse von Wangenheim & Tautz 1998).

Some case-based tools to support the coding phase are Bhansali and Harandi's work on synthesis of UNIX scripts (Bhansali & Harandi 1993), Fouqué and Matwin's work on the case-based reuse of C code blocks (Fouqué & Matwin 1993), Gomes and Bento's work on automatic conversion of procedural VHDL programs into cases (Gomes & Bento 1999) using extracted functional and behavioral knowledge from basic language constructs, Bergmann & Eisenecker's (1995) work on reuse of object-oriented software in Smalltalk-80 and C++ with weight on the definition and structuring of relevant program features used for indexing, Katalagarianos and Vassiliou's work on reuse in an object-oriented repository (Katalagarianos & Vassiliou 1995), taking into account only semantic similarity to the name of the component, and Fernández-Chamizo et al's work on retrieval and adaptation of object-oriented classes (Fernández-Chamizo, González-Calero, Hernández-Yánez & Urech-Baqué 1995, Fernández-Chamizo, Gonzáles-Calero, Gómez-Albarrán & Hernández-Yánez 1996); integrating code, natural language documentation and expert domain knowledge under the same component representation scheme, and of course the approach taken Tessem et al. (1999), extracting features directly from the class definitions without the need for neither high-level code or documentation.

5.3.2 Agents and Software Reuse

A multi-agent system for searching and retrieving reusable software components has been developed by Erdur, Dikenelli & Şengonca (1999). Their work aims at designing and implementing a component retrieval module of a component based reuse environment as a multi-agent system consisting of different types of software agents. The author truthfully claims that;

> It can be expected that in a very near future thousands of reusable component servers will be marketing their components on Internet and Internet will become the software repository of any organization (Erdur et al. 1999).

The agents in this system are capable of retrieving components based on different ontologies. Although the component ontology is based on the Basic Interoperability Model (BIDM) (Browne & Moore 1997) there is no support for automated component creation or feature extraction from the software components themselves. Hence, the descriptions will have to be created manually. Other ontologies used by the agents are domain ontology (e. g. Type of component: abstraction, operations, data types, programming language) and system ontology (e. g. Server: ip-address, domain, type of components stored, location).

The approach does not use Case-based Reasoning for retrieval of components. The searching is based on a "traditional" browse and surf mode facilitating agent capabilities for search, collaboration (using KQML) and handling of the dynamically changing nature of the Internet.

Silverman, Bedewi & Morales (1995) explores the role for intelligent agents to support reuse in distributed software repositories. The author has directed four separate repository design and evaluation projects where the architectures contain a variety of intelligent agents. The most related types are the CBR ones. These agents are not described in great detail, but the general idea is similar to our approach. The agents examine the user's description (however incomplete) of the target problem, compares it with a repository of previous cases, and retrieves the most similar case(s). Furthermore, the author suggests an adaptation facility. Other learning agents such as design rationale capture, learning by interviewing and analytical learning after tracking and studying outcomes are present in the system.

The author does not mention anything about how the reusable software artifact repositories are created.

The last approach seems to be the most similar to ours, but since the specific details on the CBR agents and the repository creation is not explained in great detail it is hard to compare the two systems. That our tool doesn't facilitate distributed repositories and network agents is of course a major difference.

5.3.3 CBR Agents

Most general CBR agents research focus on communication and cooperation in distributed case bases (Prasad et al. 1995, Plaza et al. 1999, Prasad & Plaza 1995, Plaza et al. 1997, Prasad 1998, Plaza, Arcos & Martín 1996, Redmond 1990, Barletta & Mark 1998, Prasad et al. Japan, Lander 1994, Knoblock et al. 1994, Morisbak 1999). Although our tool is not a networked system facilitating distributed case bases the architectural solution, communication and collaboration techniques could be extended to meet requirements for distributed CBR. Prasad et al (Prasad et al. 1995, Prasad & Plaza 1995) uses facilitator agents to gather a group of competent agents based on the specification of the problem. The solution differs from ours in that the facilitator doesn't know the capabilities of the individual agents prior to the gathering. In our tool the task is to find which coordinator has the workforce group suitable for solving the problem.

Huang (1996) proposes that the stored cases could play a more active role in the retrieval process by letting each case be represented as an active agent. Our cases are all agents capable of autonomous activity and communication.

Sànchez-Marrè et al. (1997) describes a similar approach to the hierarchical structure of the case repository found here. During retrieval meta-cases containing discriminating attributes are queried to select which groups of cases should be candidates for reuse.

5.3.4 Expert Assistants

Our system is also somewhat related to approaches in the field of “Expert Assistants” in that we’re creating an agent system providing active assistance to a user performing computer-based tasks. Maes (1997) describes *personal assistants* who is *collaborating with the user* in the same work environment. Our system is however limited when it comes to agents learning skills. Nor have we emphasized HCI concerns beyond that of background autonomy and avoidance from disturbing the users concentration. Pattie Maes is head of the MIT Media Lab’s software agents group which works on a variety of projects concerning software agents.

Rhodes & Starner (1996) describes *The Remembrance Agent*, one of the projects being developed by the MIT Media Lab’s software agents group. Given a collection of the user’s accumulated email, usenet news articles, papers, saved HTML files and other text notes, an agent attempts to find those documents which are most relevant to the user’s current context. That is, it searches this collection of text for the documents which bear the highest word-for-word similarity to the text the user is currently editing, in the hope that they will also bear high conceptual similarity and thus be useful to the user’s current work. These suggestions are continuously displayed in a small buffer at the bottom of the user’s emacs buffer. If a suggestion looks useful, the full text can be retrieved with a single command.

The approach taken by Rhodes & Starner (1996) is similar to ours in that the agent system augments human memory and automatically and in the background helps the user to locate material for potential (re-)use. However, our approach uses more advanced heuristics and knowledge for retrieval of cases than semantic matching. A major difference is that we are not dealing with the complexity of natural language.

References

Aamodt, A. & Plaza, E. (1994), ‘Case-based reasoning: Foundational issues, methodological variations, and system approaches’, *AI Communications* **7**(1), 39–59.

Althoff, K.-D., Birk, A., Gresse von Wangenheim & Tautz, C. (1998), Case-based reasoning for experimental software engineering, *in* M. Lenz, B. Bartch-Spörl, H.-D. Burkhard & S. Wess, eds, ‘Case-Based Reasoning Technology - From Foundations to Applications’, Springer-Verlag, pp. 235–254.

Althoff, K.-D., Nick, M. & Tautz, C. (1998), Concepts for reuse in the experience factory and their implementation for cbr system development, *in* ‘Proceedings of the 11th German Workshop on Machine Learning’.

Barletta, R. & Mark, W. (1998), Breaking cases into pieces, *in* ‘Proceedings of Case-Based Reasoning Workshop’, St. Paul, MN.

Bergmann, R. & Eisenecker, U. (1995), Fallbasiertes schlieen zur untersttzung der wiederverwendung objektorientierter software: Eine fallstudie, *in* ‘Proceedings der 3. Deutschen Expertensystemtagung XPS-95’.

Bhansali, S. & Harandi, M. T. (1993), ‘Synthesis of UNIX programs using derivational analogy’, *Machine Learning* **10**, 7–55.

Bradshaw, J. (1997), An introduction to software agents, *in* J. Bradshaw, ed., ‘Software Agents’, The AAAI Press/The MIT Press.

Browne, S. & Moore, J. W. (1997), Reuse library interoperability and the world wide web, *in* ‘Proceedings of the 1997 international conference on Software engineering’, Boston, United States, pp. 684–691.

Corkill, D. D. & Lander, S. E. (1998), ‘Organising software agents: The importance of design to effective system performance’, *Object Magazine* **8**(2), 41–47.

Dershowitz, N. (1986), Programming by analogy, *in* R. S. Michalski, J. G. Carbonell & T. M. Mitchell, eds, ‘Machine Learning: An artificial intelligence approach’, Vol. II, Morgan Kaufmann Inc., Los Altos, California, chapter 15, pp. 393–421.

Erdur, R. C., Dikenelli, O. & Şengonca, H. (1999), A multiagent system for searching and retrieving reusable software components, *in* ‘14th International Symposium on Computer and Information Sciences’, Kusadasi, Izmir, Turkey.

Fernández-Chamizo, C., Gonzáles-Calero, P. A., Gómez-Albarrán, M. & Hernández-Yánez, L. (1996), Supporting object reuse through case-based reasoning, *in* I. Smith & B. Faltings, eds, ‘Advances in case-based reasoning : third European workshop, EWCBR-96’, Lecture Notes in Artificial Intelligence, Springer-Verlag.

Fernández-Chamizo, C., González-Calero, P. A., Hernández-Yánez, L. & Urech-Baqué, A. (1995), ‘Case-based retrieval of software components’, *Expert Systems with Applications* **9**(3), 397–405.

Finin, T., Labrou, Y. & Mayfield, J. (1997), Kqml as an agent communication language, *in* J. Bradshaw, ed., 'Software Agents', The AAAI Press/The MIT Press, pp. 291–316.

Flanagan, D. (1997), *Java in a Nutshell*, 2 edn, O'Reilly, Sebastopol, CA. ISBN 1-56592-262-X.

Fouqué, G. & Matwin, S. (1993), 'A case-based approach to software reuse', *Journal of Intelligent Information Systems* **1**, 165–197.

Galbraith, J. R. (1977), *Organization Design*, Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley professional computing series, Addison-Wesley Publishing Company, Reading, MA.

Gomes, P. & Bento, C. (1999), Automatic conversion of vhdl programs into cases, *in* 'Challenges for Case-based Reasoning - Proceedings of the ICCBR'99 Workshops', University of Kaiserslautern, Centre for Learning Systems and Applications.

Gosling, J., Joy, B. & Steele, G. (1996), *The Java Language Specification, The Java Series*, The Java Series, 1 edn, Addison-Wesley, Reading MA. ISBN 0-201-63451-1.

Henninger, S. (1997), Tools supporting the creation and evolution of software development knowledge, *in* ‘Proceedings of the 12th Annual Conference on Automated Software Engineering’, IEEE Computer Society Press, pp. 46–53.

Huang, Y. (1996), An evolutionary agent model of case-based classification, *in* I. Smith & B. Faltings, eds, ‘Advances in case-based reasoning : third European workshop, EWCBR-96’, Lecture Notes in Artificial Intelligence, Springer-Verlag.

Jacobsen, I., Griss, M. & Jonsson, P. (1997), *Software Reuse: Architecture Process and Organization for Business Success*, ACM Press, New York.

Katalagarianos, P. & Vassiliou, Y. (1995), On the reuse of software: A case-based approach employing a repository, *in* ‘Automated Software Engineering’, Vol. 2, Kluwer Academic Publisher, Dordrecht, Netherlands, pp. 55–86.

Knoblock, C. A., Arens, Y. & Hsu:, C.-N. (1994), Cooperating agents for information retrieval, *in* ‘Proceedings of the Second International Conference on Cooperative Information Systems’, UoToronto Press, Toronto.

Krueger, C. W. (1992), ‘Software reuse’, *ACM Computing Surveys* **24**(2), 131–181.

Labrou, Y. & Finin, T. A. (1997), A proposal for a new kqml specification. tr cs-97-03, Technical report, Computer Science and Electrical Engineering Department, University of Maryland, Baltimore County, Baltimore.

Lander, S. E. (1994), Distributed Search and Conflict Management Among Reusable Heterogeneous Agents, PhD thesis, University of Massachusetts.

Maes, P. (1994), ‘Interacting with virtual pets, digital alter-egos and other software agents’, Doors of Perception II Conference, Amsterdam. http://www.doorsofperception.com/doors/doors2/transcripts/maes.html.

Maes, P. (1997), Agents that reduce work and information overload, *in* J. Bradshaw, ed., ‘Software Agents’, The AAAI Press/The MIT Press.

Maiden, N. & Sutcliffe, A. (1992), ‘Exploiting reusable specifications through analogy’, *Communications of the ACM* **35**(4), 55–64.

Miriyala, K. & Harandi, M. T. (1991), ‘Automatic derivation of formal software specifications from informal descriptions’, *IEEE Trans. Software Engineering* **17**(10), 1126–1142.

Morisbak, S. I. (1999), Samarbeidende cbr-agenter (in norwegian). Assignment written in the course Artificial Intelligence(iv281) at the Department Of Information Science, University of Bergen. Available at http://www.ifi.uib.no/staff/stein/CBR_agent.htm).

Odell, J. (1999), Objects and agents: How do they differ? Working paper v2.2, URL: http://jamesodell.com/publications.html.

O’Shea, T. (1986), The learnability of object-oriented programming systems, *in* ‘OOPSLA ’86 Proceedings’, ACM, ACM Press, New York, NY, pp. 502–504.

Plaza, E., Arcos, J. L. & Martín, F. (1996), 'Inference and reflection in the object-centered representation language noos', *Journal of Future Generation Computer Systems* pp. 73–188.

Plaza, E., Arcos, J. L. & Martín, F. (1997), Cooperative case-based reasoning, *in* G. Weiss, ed., 'Distributed Artificial Intelligence meets Machine Learning, Lecture Notes in Artificial Intelligence', Springer-Verlag, pp. 180–201.

Plaza, E., Arcos, J. L. & Martín, F. (1999), 'Knowledge and experience reuse through communication among competent (peer) agents', *International Journal of Software Engineering and Knowledge Engineering* .

Prasad, M. N. (1998), 'Distributed case-based learning', Article written for Andersen Consulting, Center for Strategic Technology Research, Thought Leadership. http://www.ac.com/services/cstar/cstrdcb13.html.

Prasad, M. N., Lesser, V. R. & Lander, S. E. (1995), 'Retrieval and reasoning in distributed case bases', *Journal of Visual Communication and Image Representation, Special Issue on Digital Libraries* **7**(1), 74–87.

Prasad, M. N., Lesser, V. R. & Lander, S. E. (Japan), Learning organizational roles in a heterogeneous multi-agent system, *in* 'Proceedings of the International Conference on MultiAgent Systems', 1996.

Prasad, M. N. & Plaza, E. (1995), Corporate memories as distributed case libraries, *in* ‘Procceding of the 10th Banff knowledge acquisition for knowledge-based system workshop’, number 40 *in* ‘2’, Banff, Canada, pp. 1–19.

Redmond, M. (1990), Distributed cases for case-based reasoning: facilitating use of multiple cases, *in* ‘Proceedings AAAI-90’.

Rhodes, B. & Starner, T. (1996), The remembrance agent: A continuously running automated information retrieval system, *in* ‘Proceedings of The First International Conference on The Practical Application Of Intelligent Agents and Multi Agent Technology (PAAM ’96)’, London, UK, pp. 487–495.

Sànchez-Marrè, M., Béjar, J. & Cortés, U. (1997), Reflective reasoning in a cbr agent, *in* ‘Procc. of the VIM Project Spring Workshop on Collaboration Between Human and Artificial Societies’, Spain.

Silverman, B. G., Bedewi, N. & Morales, A. (1995), Intelligent agents in software reuse repositories, *in* ‘Proc. of ACM 4th InternationalConference on Information and Knowledge Management (CIKM’95)’, Baltimore, Maryland, USA. Workshop on Intelligent Information Agents.

Spanoudakis, G. & Constantopoulos, P. (1996), ‘Analogical reuse of requirements specifications: A computational models’, *Applied Artificial Intelligence* **10**(4), 281–306.

Tessem, B. & Bjørnestad, S. (1997), ‘Analogy and complex software modeling’, *Computers in Human Behavior* **13**(4), 465–486.

Tessem, B., Whitehurst, R. A. & Powell, C. L. (1999), Case-based support for rapid application development, *in* ‘SEKE’99 Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering’, Knowledge Systems Institute, Skokie, Illinois, pp. 192–196.

Whitehurst, R. A. (1995), Systematic Software Reuse Through Analogical Reasoning, PhD thesis, University of Illinois, Urbana-Champaign, IL.