

*The Road to ASCRARAD: The Development of
Agent Support for a Case-based Reuse
Application for Rapid Application Development*

Stein Inge Morisbak

June 22, 2000

Contents

1	Introduction	1
2	Theory and Method	3
3	The Environment and Agents	15
4	Agent Organization	33
5	Agent Interaction	39
6	Comparison and Test Results	47
7	Conclusions and Further Development	53
A	Usage / user manual	61
B	Installing and running the application	73
	References	77

Chapter 1

Introduction

To support programming in Java, a tool, Case-based support for Rapid Application Development [65], has been developed which makes it possible to search through potentially reusable Java classes. The tool is based on case-based methods to support retrieval and reuse.

The purpose of *this* project, ASCRARAD, is to add agent support to the retrieval mechanism of this tool. This should optimize retrieval efficiency and free the user's hands and mind making it possible for the user to concentrate fully on her task.

Rapid Application Development is a technique used to obtain insight into system requirements by experimenting with partial implementations. To be effective, it must be possible to create a prototype with a minimal investment of time and resources. The purpose of this approach is to develop a prototype of portions of the proposed system to determine the necessity, desirability, or feasibility of certain requirements. The pur-

pose of such a prototype is to help the end-user and developer to come to a mutual understanding of the system requirements and decide on a final design.

One mechanism for decreasing 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. 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.

The ASCRARAD tool helps the developer to locate code for potential reuse in an automated way and in the background. The tool aids in program understanding and adaptation. It allows an exploratory approach to program development and optimizes reuse efficiency.

I will in this thesis present "*The Road to ASCRARAD*". In other words, *the theory behind* and *the process of the development* of the agent support. I will start with a theory chapter describing the concepts of *AI Method*, *Agent Orientation*, "*Intelligence*", *Case-based Reasoning* and the *Java programming language* (Chapter 2). I will continue with a brief presentation of the tool developed by Bjørnar Tessem et al before I go on to a more detailed description of the Agent support and architecture. I will try to describe my chosen implementation strategy and to justify this choice (Chapter 3 and 4). Chapter 5 is about agent language and interaction. Some example runs will be presented in Chapter 6. I will conclude with a summary and some thoughts about further development and possibilities in Chapter 7.

Chapter 2

Theory and Method

The main purpose of this project is to implement an agent architecture suitable for retrieval of cases in a Case-based Reasoning tool for Rapid Application Development. I have decided to create an architecture where each case in the repository is an active case (or agent) and where a hierarchical structure of agents provide an organization analogy useful to implement the retrieval mechanisms and rules. The big question was; "can *agent support* prove to be useful for retrieval of cases in a Rapid Application Development environment based on CBR-techniques?"

2.1 AI Method

Scientific method in artificial intelligence (AI) is often characterized by an experimental or explorative angle of incidence. I have chosen to follow an approach based on

this method.

AI is the study of the mechanisms underlying intelligent behavior through the construction and evaluation of artifacts that enact those mechanisms. [36, p 755]

This definition of artificial intelligence (AI) is a description of a method for experimenting rather than a theory about those mechanisms underlying intelligence. It tells us that the mechanisms are best studied through design, execution and evaluation of experiments with improvement of models and further experimentation as a goal. As an empirical science, AI takes a constructive approach; we attempt to understand intelligence by building a working model of it. Hence, AI is also engineering.

AI uses much of the theory we find in cognitive psychology [43, 54, 4, 16], evolutionary psychology [5, 11, 12] and organizational and social psychology [40, 21, 35]. We wish to let computers perform tasks which, if humans carried them out, would be viewed upon as something demanding intelligence. The task for AI research is to find tasks demanding intelligence and to see what processes are sufficient to accomplish these. Through experimentation with alternative models and implementations you can find out much about the principals for intelligence, both individual and corporate. Principles that are best understood through repetitively experimentation, trying and failing.

The moment of truth is a running program [59, p 96]

Herbert A. Simon [59] states that the main method within AI consists of building and studying systems that exhibit intelligence. Three points sum up the method:

1. Choose a task incorporating a feature of intelligence that is of substantial practical importance or that exhibits features and complexities that have not yet been simulated by AI systems.

2. Build a system exhibiting this feature of intelligence.
3. Examine the behavior of the system in different task environments and with different initial conditions.

I decided to build an agent-architecture performing "intelligent" actions in a complex environment where experimentation with different initial conditions and different task environments is possible.

2.2 Agent Orientation vs. Object Orientation

Objects and agents share many things in common, but in my 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" [8]. In other words, when and how an agent acts, is determined by the agent.

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" [45].

It is not entirely correct that OO doesn't support many of the properties ascribed to agents. OO technology can be extended in various ways to support many of the prop-

erties ascribed to agents. (E. g. `process` and `thread` can be considered active objects). The point here is that the agent-based approach is an extension to how we think in an OO world. Agents, then, are an evolution rather than a revolution. It is another way of thinking about systems and their implementations, i. e. an extension to the way of thinking in the OO approach.

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. As an example, the ASCRARAD solution uses an organization analogy. The system is organized in departments where each department consists of a group of workers with special features that deviates them from the rest of the organization's workforce. A coordinator leads each department and the Manager leads the whole organization. Each member has the potentiality of surviving and succeeding but also the possibility of replacement and death. They communicate by sending messages to each other and the command line is top-down (Chapter 4).

There are many other areas than autonomy and interaction that can differentiate the agent-based approach from traditional OO [45], but I think the most important distinction is the computational model used to solve the problem. In most cases the problem can be solved both ways. The resulting program may look just the same externally, but agent-based programming should be able to deal with complexity and adaptability better. Agents can be viewed as an extension of objects. An object is something that encapsulates its identity (who), its state (what), and its behavior (how). An active object encapsulates its own thread of control (when). An agent may have all of these, but it must have something else. A common approach is that an agent encapsulates *why* it does something. Objects have a specific contract that defines their behavior,

while agents, like humans, have only a generic interface to interact with others and the environment. Contracts between agents are negotiated at a meta-level. They are not implicit in the interface. Agents (like humans) can only attempt to get other agents to believe or do something.

2.3 "Intelligence"

After decades, the term *intelligent* has still not been 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 tables or objects, thought of as somewhat "smarter". Or, it could just be marketing hype. However, 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.

If the science of artificial intelligence has made any contribution to human knowledge, it is in confirming that intelligence is not some mystic vapor permeating men and angels, but rather the effect of a set of general principles that can be understood and applied to the design of intelligent machines.[36, p 75]

Whatever the term *intelligent agent* means, such agents will require a set of attributes and facilities. E. g. an intelligent agents state must be formalized by knowledge (i. e. *beliefs, goals, desires, intentions, plans, assumptions*) and be able to act on this knowledge. It should be able to examine its beliefs and desires, form its intentions, plan what actions it will perform based on certain assumptions, and eventually act on

its plans [14]. Furthermore, intelligent agents must be able to interact with other agents using symbolic language (Chapter 5).

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.

ASCRARAD uses a metaphor from the organizational structure of an enterprise (see chapter 4). The intelligence in the solution arise from the interactions of all the simple, individual, embodied agents.

All this sounds like models of rational human thinking, and that is exactly what it is. Our understanding of how humans think and how they act together in societies is used as a model for designing agents and agent systems, or as James Odell says it, we should use *life as a metaphor*[44] to build these systems.

Since the term intelligent is somewhat controversial in computer science and perhaps not applicable to the type of agents I have developed, I will use *agent*, not *intelligent agent*, as the term for the type of agents in this system. Instead I 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. It's probably far to the former side of the spectrum.

2.4 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 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 and Plazas ideas as they are expressed in the article *Case-based Reasoning: Foundational issues, methodological variations, and system approaches* [1].

2.5 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 [26]. 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 [26]. 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 class¹.

2.5.1 Extending Java's Reuseability?

The software developer is left without 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 [46]. 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

¹Java 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.[30]

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.

It is wrong to say that it *extends* Java's reuseability since the case descriptions are constructed using Javas own artifacts. One could say though, that it *enhances* reuseability in that it automates, ensures the quality of -, and aids in understanding of - the process.

Since the tool uses Java reflection and reuseability it doesn't require additional work on the part of the software developers. Furthermore it ensures that the automated retrieval and adaption strategies will be immediately useful and work with existing software repositories. However, it should be noted that the same approach to feature extraction could be implemented in any other programming environment in which access to the source code for the components is available.

2.5.2 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.

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

Pattie Maes [37], 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 [38].

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 (see Chapter 5) 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 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.

2.6 Bringing it all together

Following AI method I have identified a main task for the agents and system as a whole. This task is to retrieve software components for reuse. The task can be viewed as demanding "intelligence" since the location of such code requires search, recognition and comparison of code features and which, if it should have been carried out by a human, would demand knowledge, experience and skill to accomplish.

Recalling Pattie Maes' definition from Section 2.5.2, an agent is

a process that lives in the world of computers and computer networks and can operate autonomously to fulfill one or more tasks [38].

This definition describes the features an agent has to possess, but in relation to ASCRARAD it also expresses why agents are particularly suitable in this type of environment. The concept of agents being autonomous makes it possible for the user to concentrate fully on her task. Agent-orientation is a great computational model for creating autonomous systems, and I believe that agent support could become excellent help for the user.

Case-based Reasoning(CBR) is a problem solving paradigm that uses knowledge of relevant past cases to interpret or to solve a new problem case. The technique constitutes a natural approach to represent the experience acquired in the design of past software components. CBR is especially suitable when dealing with object-oriented design where software development usually relies on previously developed object classes.

In the Java programming language the base case descriptions can be constructed from the software artifacts themselves using Java reflection. Java is also suitable for implementing agent societies since it supports multithreaded applications.

These choices of method, techniques and models have been used as the basis for the development of ASCRARAD.

Chapter 3

The Environment and Agents

3.1 Case-based Support for RAD

Rapid Application Development(RAD) is a technique for achieving insight into system requirements by experimenting with partial implementations. Reuse of previously developed components assists this process. Tessem et al [65] have developed a tool where the reuse method is case-based and integrated in an environment specialized for rapid prototyping.

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 done by Tessem et al 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 3.1 is a conceptual diagram of the Rapid Application Development Environment from the article *Case-based Support for Rapid Application Development* [65].

An earlier version of the system included a Java interpreter. This is shown in the figure as a grey box since it is no longer part of the environment. As development progresses, portions of the software could be sent to the interpreter for immediate feedback. This is a great feature in RAD tools and should be introduced again (See section 7.2.2 *Editor Improvements*).

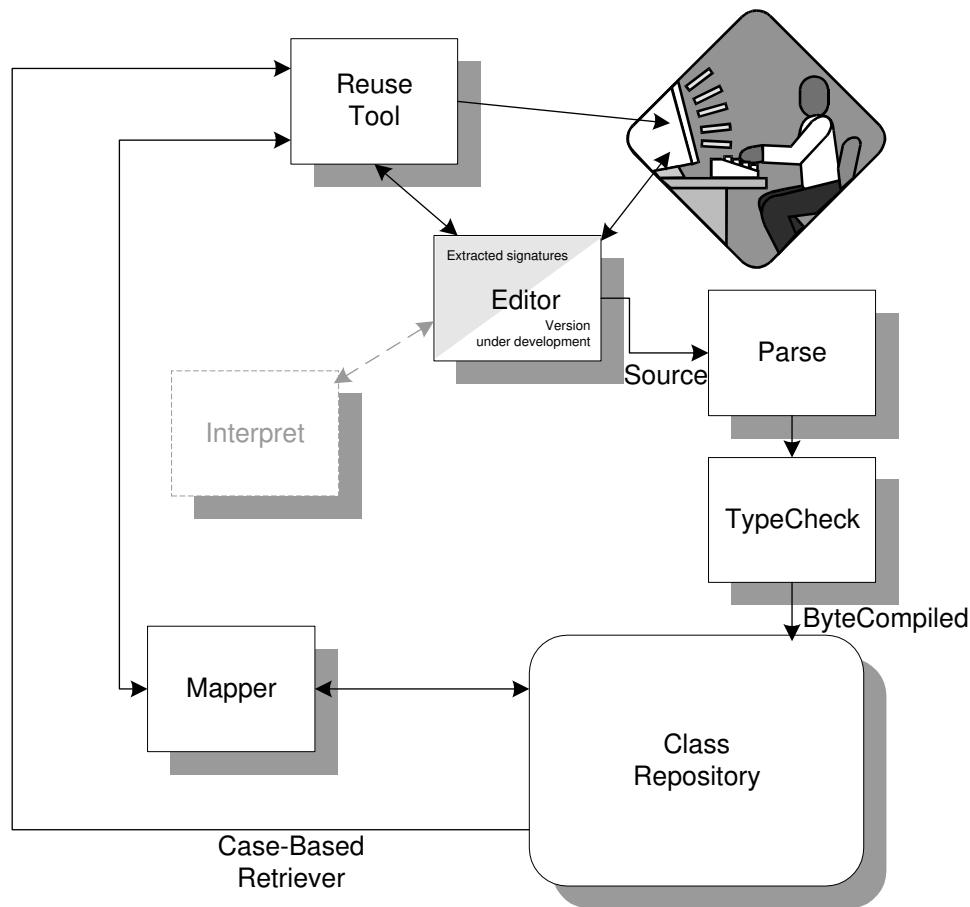


Figure 3.1: Conceptual Diagram of Rapid Development Environment

3.2 Agent Supported Case-based Retrieval

A user of such a tool must repetitively and actively ask for reuse support. An alternative to this is one or more intelligent agents observing the implementation the user builds, and when the similarity to a class is good enough notifies the programmer. The Agent Supported Case-based Reasoning Application for Rapid Application Development(ASCRARAD) tool is an attempt to support these features. It is based on the work done by Tessem et al and extends their work with the implementation of agent support.

Figure 3.2 is a conceptual diagram of the Rapid Application Development Environment with Agents.

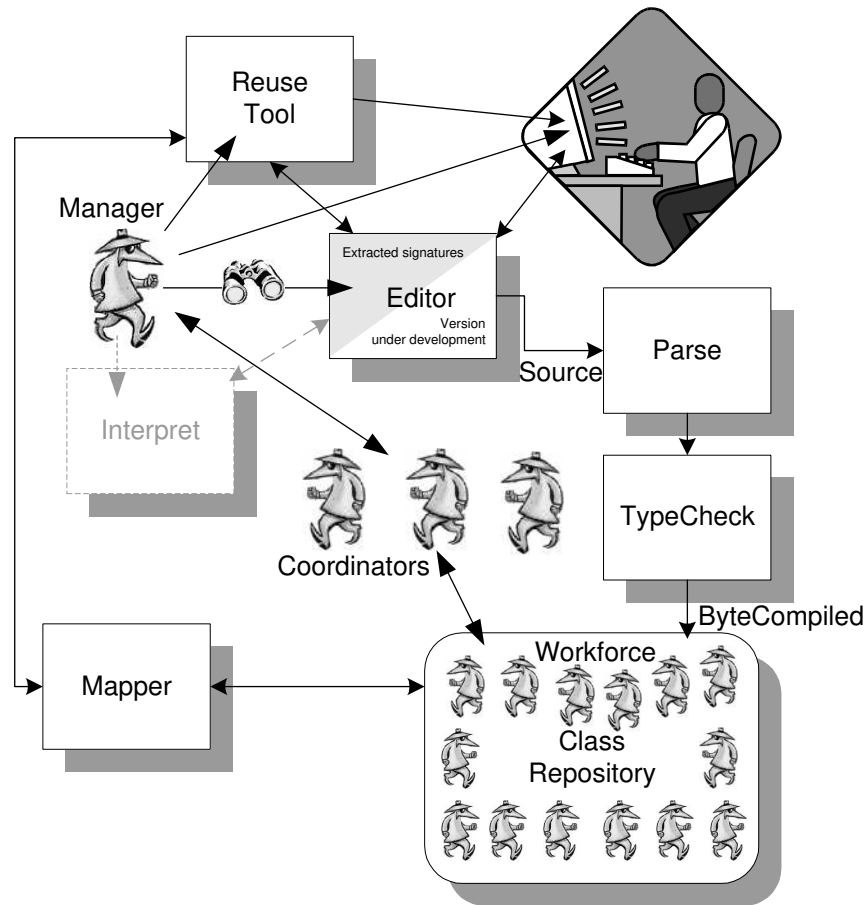


Figure 3.2: Conceptual Diagram of Rapid Development Environment with Agents

The Manager agent monitors the programmer's implementation via the extracted signatures of the partial class specification and compiles the code when needed. A re-introduction of the interpreter would allow the Manager to test-run uncompiled Java statements; thus check the syntax of the expressions. This is however not implemented, but may be a great idea for further development (See section 7.2.2). 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

[63], and a single workforce agent is in reality a single case description of a Java class.

3.2.1 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 (See Section 3.3 *The 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.

3.2.2 The Coordinator Agents

Coordinators are "package cases". The logical grouping of cases and the concept of Coordinators are based on the Java concept "Packages" [26]. 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) of 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.

Estimating the Significance of a Type

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

1. Count the number of occurrences of a particular type in a single class:

<p>FORMULA 3.1</p> <p><i>class</i> κ</p> <p><i>count</i> c</p> <p><i>type</i> τ</p> $c(\kappa, \tau)$ <p style="text-align: right;">△</p>
--

EXAMPLE 3.2

```
class java.applet.Applet(Applet for short)
```

```
type URL
```

$$c(\text{Applet}, \text{URL}) = 8$$



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 (`java.applet.Applet` has 55 different types):

FORMULA 3.3

normalized count \bar{c}

$N =$ number of references to types in κ

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



EXAMPLE 3.4

$N = 55$

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



3. Sum up all the results from 2. for all classes in a package and receive a sum of all normalized count $\bar{c}(\kappa, \tau_i)$'s (`java.applet` contains 4 classes):

FORMULA 3.5

package \mathcal{P} *package count* pc $M =$ number of classes in \mathcal{P}

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



EXAMPLE 3.6

package java.applet $M = 4$

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



4. Divide the sum of all normalized \bar{c} 's ($pc(\mathcal{P}, \tau_i)$) by the sum of the results from 3. for all packages in the repository:

FORMULA 3.7

normalized packagecount \bar{pc}

$$\bar{pc}(\mathcal{P}, \tau_i) = \frac{pc(\mathcal{P}, \tau_i)}{\sum_{k=1}^N pc(\mathcal{P}, \tau_k)}$$



EXAMPLE 3.8

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



5. Divide the sum of all normalized $\bar{p}c$'s ($\bar{p}c(\mathcal{P}, \tau_i)$) by the total number of packages in the repository (JDK1.1.8 contains 78 packages)::

FORMULA 3.9

normalized $\bar{p}c$ $\bar{p}c$

$Q =$ number of packages(\mathcal{P}) in the repository

$$\bar{p}c(\tau_i) = \frac{\sum_{i=1}^Q \bar{p}c(\mathcal{P}, \tau_i)}{Q}$$

△

EXAMPLE 3.10

Sum of all $\bar{p}c$ for the type URL = 1.790

$Q = 78$

$$\bar{p}c(\text{URL}) = \frac{1.790}{78} = 0.023$$

♣

6. Calculate the significance of a type in a package by the following formula:

FORMULA 3.11

significance $sign$

$$sign(\mathcal{P}, \tau) = \frac{\bar{p}c(\mathcal{P}, \tau) - \bar{p}c(\tau)}{\bar{p}c(\tau)}$$

△

EXAMPLE 3.12

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

♣

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.

Estimating the Relevance of a Target Type

The estimation of a target case types relevance in a package is calculated in the following steps:

1. Count the number of occurrences of a particular type in the target case:

<p>FORMULA 3.13</p> <p><i>target case</i> φ</p> $c(\varphi, \tau)$ <p style="text-align: right;">△</p>

2. Calculate the relevance of the type by the following formula:

<p>FORMULA 3.14</p> <p><i>relevance</i> rel</p> $rel(\mathcal{P}, \varphi) = \sum_{i=1}^N c(\varphi, \tau_i) * sign(\mathcal{P}, \tau_i)$ <p style="text-align: right;">△</p>
--

3. The total similarity is the sum of the selected relevances:

FORMULA 3.15

similarity sim

$$sim(\mathcal{P}, \varphi_i) = \sum_{i=1}^N rel(\mathcal{P}, \varphi_i)$$

△

4. If the total similarity ($sim(\mathcal{P}, \varphi_i)$) is a positive number (larger than 0), it's a match. If the total similarity is larger than -0.5, the agent continues to live in memory. If else, the agent dies.

FORMULA 3.16

If ($sim(\mathcal{P}, \varphi_i) > 0$)

then it's a match!

else if ($sim(\mathcal{P}, \varphi_i) > -0.5$)

then it's a partial match

else

die!

△

Unfortunately these heuristics does not give the wanted results alone. Some packages that contain relevant classes doesn't always get selected. I 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. I have therefore hard coded 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 [20]. I believe this is a reasonable solution since these packages contain core classes used in almost any Java program.

I 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* [26, p 106] it is recommended that naming conventions should be used in all Java programs, so I also believe this is a reasonable solution. In Katalagarianos and Vassiliou's work on software reuse [31] semantic similarity is the only feature taken into account in the matching.

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.

3.2.3 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 [63]. 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 it self 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.

Estimating the Similarity

The estimation of the similarity between the target and the base is developed by Tessem et al and is described in the article *Case-based Support for Rapid Application Development* [65].

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:

FORMULA 3.17

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

△

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.

3.2.4 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 (see Section 3.3).

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.

3.3 The Editor

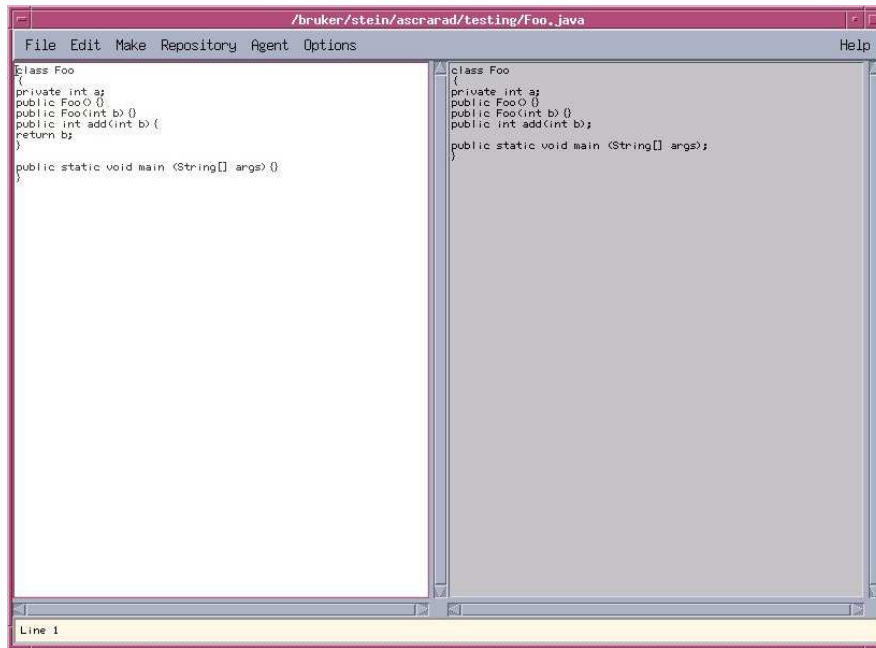


Figure 3.3: Agent supported Rapid Prototyping Environment

The editor's graphical display is separated into two views (see Figure 3.3). 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 agent re-reads the extracted signatures for each round of matching.

The Agent Settings Dialog is shown in Figure 3.4. 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" (See Section 3.2.2) 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.

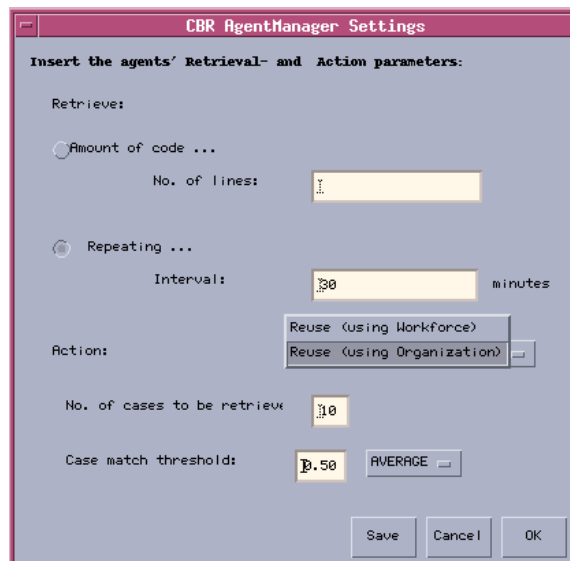


Figure 3.4: Agent Settings Dialog

A complete description of the menus of the graphical interface can be found in Appendix A.2.

Chapter 4

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 5)), 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 [23], 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* [10]. It is reasonable to expect this answer to hold for software agent organizations as well.

4.1 Choice of organization

I will in the following try to explain the features of my choice of organization and why I 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 were rejected, will be presented. I did however implement two of them (A comparison and test results may be found in Chapter 6).

4.1.1 Single Agent Architecture

The first model (Figure 4.1) 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 by Tessem et al [65]. 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.

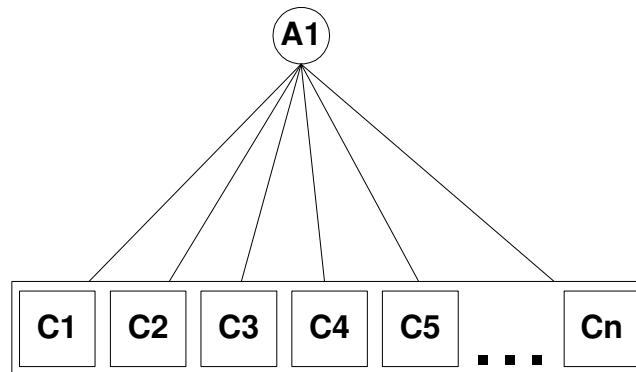


Figure 4.1: Single Agent Architecture.

4.1.2 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 4.2) is a multi-agent system involving relatively uniform agents (Workforce agents (Aw_1 to Aw_n)) 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.

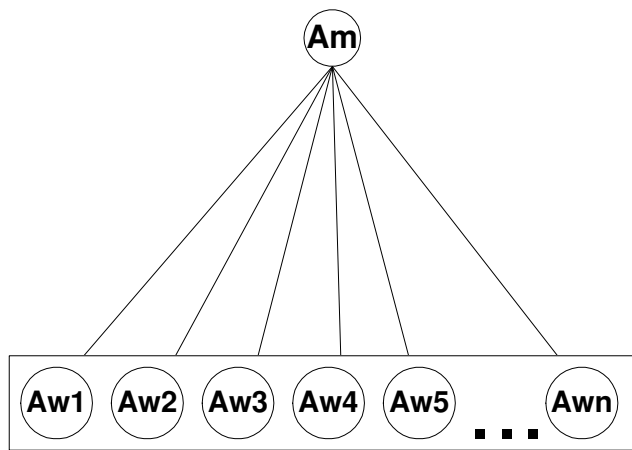


Figure 4.2: Manager/Workforce Architecture (Active Cases).

Traditionally cases are viewed upon as passive objects waiting to be retrieved for further 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* [28], 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. In stead 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.

4.1.3 Manager/Coordinators/Workforce Architecture (implemented)

The last model consists of three agent layers where the agents are organized hierarchically using an enterprise analogy (Figure 4.3). 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 Manger 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 (see Section 3.2.2). 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

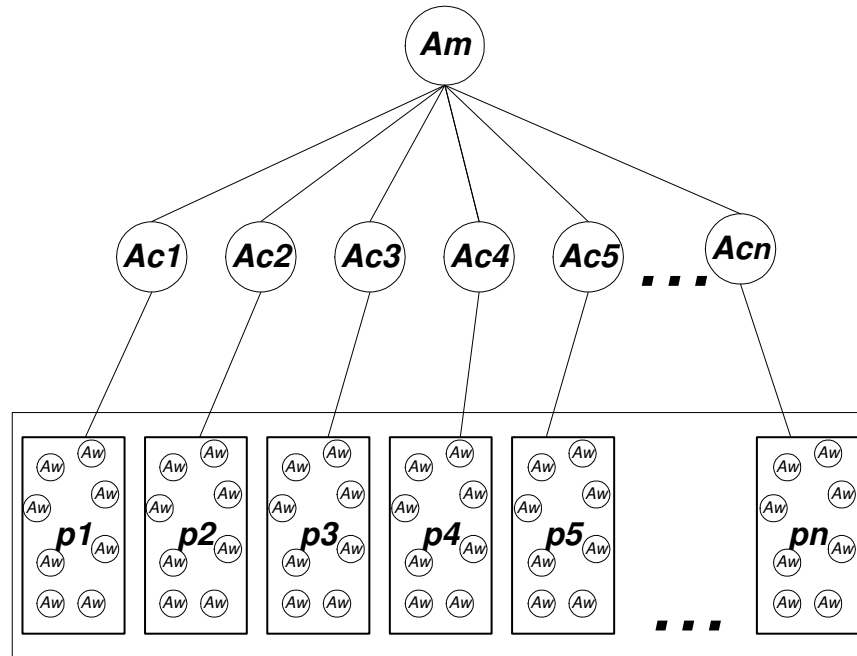


Figure 4.3: Manager/Coordinators/Workforce Architecture.

types for the type of cases in the 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 (see 3.2.2 and [26]).

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.

Chapter 5

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.

5.0.4 KQML

KQML, which is an acronym for Knowledge Query and Manipulation Language [33], was conceived both as a message format and a message handling protocol to support run-time knowledge sharing among agents [19]. 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 [33]:

- 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 in ASCRARAD.)
- to: final recipient; used when a message is sent using intermediary agents. (Not used in ASCRARAD.)
- in-reply-to: identifier of the message that triggered this message submission. (Not used in ASCRARAD.)
- reply-with: identifier to be used by a message replying to this message. (Not used in ASCRARAD.)
- language: language for interpreting the information in the content field of this message. (Not used in ASCRARAD.)
- ontology: identifies the ontology to interpret the information in the content field of this message. (Not used in ASCRARAD.)
- content: context-specific information describing the specifics of this message.

5.0.5 ASCRARAD Agent Messages

I have added a couple of extra fields for the ASCRARAD environment agent messages.

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 5.1

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



A ASCRARAD Agent Message is shown above (Example 5.1). 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 ASCRARAD 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 5.2


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



Example 5.3 shows a message from a Coordinator to a Workforce agent:

```
EXAMPLE 5.3

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



I have decided to maintain only one format for ASCRARAD agent messages. This results in an information overflow between the agents and 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 my choice also makes it easier to extend the system to become networked (see Section 7.2 *Further Development*). A common format facilitating high-level communication is important and essential for distributed agents on the internet.

5.0.6 Interacting Agents

Interaction is one of the most important features of the agents. Researchers investigating agent communication languages mention three key elements to achieve multi-agent

interaction [19, 29]:

- A common agent communication language and protocol
- A common format for the content of communication
- A shared ontology

The *common agent communication language and protocol* in ASCRARAD is “hard coded” using the Java programming language. There is no need for a flexible solution to this problem since ASCRARAD is a stand-alone application with a well defined task specification. Java is however more portable than other languages, can move between heterogeneous hardware systems, supports encapsulation, inheritance and polymorphism, and has high reuseability (see Section 2.5). Extending and/or adapting the environment may therefore not be very difficult tasks.

A *common format for the content of communication* is provided by the “quasi” KQML modeling of the ASCRARAD Agent Messages described in Section 5.0.5.

Ontologies

Ontologies are defined as specification schemes for describing concepts and their relationships in a domain of discourse [19]. It is important that agents not only have ontologies to conceptualize a domain, but also that they have ontologies with similar constructions. Such ontologies, when they exist, are called common ontologies. Once interacting agents have committed to a common ontology, it is expected that they will use this ontology to interpret communication interactions, thereby leading to mutual understanding and (ultimately) to predictable behaviors.

The *shared ontology* in ASCRARAD is easily understood by the agents since we know that all agent interactions are concerned with retrieving base cases (Java classes) that

match a target case, where a target case is a Java class developed by the user of the tool. The vocabulary used in this process is small and the meaning of the “words” is easily understood.

An example of a situation where agents might misunderstand each other because of two slightly different ontologies is two English-speaking agents getting confused if one talks about the boot and bonnet, and the other about the hood and trunk of an automobile. They have to have a shared vocabulary of words and their meaning. The agents does not share the same ontology.

Chapter 6

Comparison and Test Results

In the following I will present test results for both of the implemented architectures (See Chapter 4). Only the test results under the WinNT environment will be presented (see Appendix A.1 for specifications).

I have created two repositories for testing the tool:

- The first is created from version 1.1.8 of the JavaTM Development Kit[62]. The repository consists of 1581 cases and 78 packages.
- The second is created from version 1.3 of the JavaTM Standard Development Kit. The repository consists originally of 7174 cases and 241 packages. I 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 JavaTM 2 Platform, Standard Edition, v 1.3 API Specification [62] 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 I will use with the repository built from version 1.1.8 of the JDK consisting of method prototypes is illustrated below. This is the same test case used in *Case-Based Support for Rapid Application Development* [65] by Tessem et al.

```
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 I will use with the repository built from version 1.3 of the JSDK consisting of method prototypes is:

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

The Agent settings are:

- Number of Cases to be retrieved: 8.
- Case match threshold: 0.50 (Average).

6.1 Results Manager/Workforce

6.1.1 JDK1.1.8

The eight classes retrieved for the `MyString` class from a repository built from version 1.1.8 of the JavaTM 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 the repository built from version 1.1.8 of the JavaTM Development Kit. The repository consists of 1581 cases and 78 packages.

6.1.2 JSDK1.3

The eight classes retrieved for the `MyApplet` class from a repository built from version 1.3 of the JavaTM 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 run on the repository built from version 1.3 of the JavaTM Standard Development Kit. The repository consists of 3256 cases and 79 packages.

6.2 Results Manager/Coordinators/Workforce

6.2.1 JDK1.1.8

The eight classes retrieved for the `MyString` class from a repository built from version 1.1.8 of the JavaTM Development Kit were exactly the same as under the Manager/Workforce Architecture (6.1)

It took 49 sec. to run on the repository built from version 1.1.8 of the JavaTM Development Kit. The repository consists of 1581 cases and 78 packages.

6.2.2 JSDK1.3

The eight classes retrieved for the `MyApplet` class from a repository built from version 1.3 of the JavaTM Standard Development Kit 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

It took 2 min. and 20 sec. to run on the repository built from version 1.3 of the JavaTM Standard Development Kit. The repository consists of 3256 cases and 79 packages.

6.3 Test Results Summary

The test done by Tessem et al. gave the following results: 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).

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 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 (see Section 7.2.3 *CBR Improvements*).

To create the complete class repository takes several hours for Java 2 libraries, but

since this is done only when the class library is updated I don't consider the efficiency of this to be of particular significance.

Chapter 7

Conclusions and Further Development

7.1 Conclusions

Rapid Application Development is a technique used to obtain insight into system requirements by experimenting with partial implementations. To be effective, it must be possible to create a prototype with a minimal investment of time and resources. The purpose of this approach is to develop a prototype of portions of the proposed system to determine the necessity, desirability, or feasibility of certain requirements. The purpose of such a prototype is to help the end-user and developer to come to a mutual understanding of the system requirements and decide on a final design.

One mechanism for decreasing 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 [65] 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.

I believe I have reached my goals in developing a tool that fulfills the requirements I have 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.

"Can *agent support* prove to be useful for retrieval of cases in a Rapid Application Development environment based on CBR-techniques?". I would say yes.

7.2 Further Development

I 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, a bit far-fetched and requiring future research, and some that could have been developed right away.

7.2.1 Agent Improvements

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 [51, 53, 32, 48, 49, 50, 52].

7.2.2 Editor Improvements

I would like to improve the editor with more features. On the Make menu I would like to add a "Run..." method for running the application under development, a "View Applet..." for viewing applets, and also a class browser. Furthermore I would like to re-introduce the Java interpreter from an earlier version of the Case-based RAD tool developed by Tessem et al. mentioned in the *Case-Based Support for Rapid Application Development*[65] article. The Java interpreter allows the user to test-run uncompiled Java statements; thus the user can import Java classes to the environment, and test their functionality in the interpreter without the need for compiling the application. This feature could also be helpful for the Manager agent of the ASCRARAD. The Manager could test-run uncompiled Java statements and thereby check the syntax of the expressions in an automated way.

The goal is to create a full featured Rapid Application Development Environment.

7.2.3 CBR Improvements

Possible CBR improvements are:

- 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 features from design patterns [24].
- 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 coordinators and workforce) would then fire more often than cases we don’t have very good reuse experiences with. Sánchez-Marré et al [57] 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.

7.2.4 Evaluation

An evaluation of the tool in a realistic environment is important 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.

7.3 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.

7.3.1 CBR and Software Engineering

Some approaches to support software reuse using CBR focus on the earlier phases of a software development process [41, 39, 60, 64], others discuss the possibility of reuse of code by identifying similarity in designs or specifications, assuming that this similarity leads to similarity in code [13, 66, 3], and a couple of authors focus on

case-based reasoning as a tool to enhance organizational learning in software factories [27, 2].

Some case-based tools to support the coding phase are Bhansali and Harandi's work on synthesis of UNIX scripts [7], Fouqué and Matwin's work on the case-based reuse of C code blocks [22], Gomes and Bento's work on automatic conversion of procedural VHDL programs into cases [25] using extracted functional and behavioral knowledge from basic language constructs, Bergmann and Eisenecker's work on reuse of object-oriented software in Smalltalk-80 and C++ [55] 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 [31], 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 [18, 17]; integrating code, natural language documentation and expert domain knowledge under the same component representation scheme, and of course the approach taken by Tessem et al [65], extracting features directly from the class definitions without the need for neither high-level code or documentation.

7.3.2 Agents and Software Reuse

A multi-agent system for searching and retrieving reusable software components has been developed by Erdur et al [15]. 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 [15].

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) [9] 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 et al [58] 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 my 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 ASCRARAD, 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 ASCRARAD doesn’t facilitate distributed

repositories and network agents is of course a major difference.

7.3.3 CBR Agents

Most general CBR agents research focus on communication and cooperation in distributed case bases [51, 49, 53, 48, 50, 47, 56, 6, 52, 34, 32, 42]. Although ASCRARAD 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 [51, 53] uses facilitator agents to gather a group of competent agents based on the specification of the problem. The solution differs from ASCRARAD in that the facilitator doesn't know the capabilities of the individual agents prior to the gathering. In ASCRARAD the task is to find which coordinator has the workforce group suitable for solving the problem.

Huang [28] 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. ASCRARAD's cases are all agents capable of autonomous activity and communication.

Sánchez-Marré et al [57] describes a similar approach to the ASCRARAD hierarchical structure of the case repository. During retrieval meta-cases containing discriminating attributes are queried to select which groups of cases should be candidates for reuse.

Appendix A

Usage / user manual

A.1 Specifications

- The tool is build using Microsoft Visual J++ 6.0 and Emacs 20.6.1 with Java Developer Environment (JDE 2.5.1). I started developing the product with MS Visual J++. It has great debugging facilities, but doesn't support Java 2. I have therefore ported the solution to Emacs. See Appendix B for installation specifications.
- Can be run under Suns JavaTM Development Kit v1.1.8 or higher. It is however recommended to use the latest JDK. If you want to create repositories from Development Kits higher than v1.1.8 you must include the paths to these in your CLASSPATH variable.
 - Has been tested on WinNT 4.0 with service pack 6.

- The WinNT box I've used is a Pentium II, 450MHz with 256MB RAM.
- Two repositories are built and tested on WinNT. They have the following features:
 - The first is created from Suns JavaTM DK 1.1.8 class library and consists of 78 package cases and 1581 class cases. The size of the repository is 3.84 MB.
 - The second is created from Suns JavaTM 2 SDK 1.3 class library and consists of 241 package cases and 7174 class cases. The size of the repository is 18.5 MB. I have also created a smaller 1.3 repository for testing purposes (see Chapter 6 for details).

- Has also been tested on Solaris7.
 - The Solaris box is a Sun Enterprise 450, 2x UltraSPARC-II 400MHz with 1024MB RAM.
 - The repository on Solaris is created from Sun's JavaTM 2 SDK 1.2.2 class library and consists of 154 package cases and 5370 class cases. The size of the repository is 18 MB. I have created a smaller 1.2.2 repository by removing all sun.* packages and all com.* packages. Only packages that are part of the JavaTM 2 Platform, Standard Edition, v 1.2.2 API Specification [61] are included in the repository (Other packages may be a part of Sun's Java 2 SDK and Java 2 Runtime Environment distributions). The final repository consists of 2827 cases and 63 packages. The size of the repository is 8 MB.
- IBM's Jikes Compiler v. 1.10 is used for compiling target case classes.
<http://ibm.com/developerworks/opensource>
- This document is created using L^AT_EX.

A.2 Menus

A.2.1 The File Menu:

Open File

General usage: Open File brings a open file dialog box onto the screen, in the dialog box you can traverse through the file system and pick the file to be opened into the editor. If there is already a file in the editor, the file will be replaced. If the existing file in the editor has not been saved, you will be reminded to do so.

Known bugs: When the file is very big, the text editor will crash. Don't try to load files more than 1,000 lines! If you accidentally load a big file, it sometimes gets truncated. In this case, don't save the file. Quit immediately.

Insert File

General usage: Insert File brings a file dialog box onto the screen, in the dialog box you can traverse through the file system and pick the file to be inserted into the editor. The contents of the file are inserted at the current cursor position.

Known bugs: When the file is very big, the text editor will crash. Don't try to insert files more than 1,000 lines! If you accidentally insert a big file, it sometimes gets truncated. In this case, don't save the file. Quit immediately.

New

General usage: New starts a blank text file. The current file in the editor will be replaced. If the existing file in the editor has not been saved, you will be reminded to do so.

Known bugs: None

Save

General usage: Save saves the file in the editor. If the file is new and does not have a name yet, Save works the same as Save As.

Known bugs: None

Save as

General usage: Save As brings a Save As file dialog box onto the screen. You can choose a directory from the directory list in the dialog box and give the file a name. The file in the editor will be saved in the directory and the name you just specified. If the file name you specified is already being used, you will be asked whether you want to overwrite the file or not.

Known bugs: None

Quit

General usage: Quit terminates the program. If there is a unsaved file in the editor, you will be reminded to save.

Known bugs: None

A.2.2 The Edit menu:

Search

General usage: Search brings a Search dialog box on the screen, you can fill either the "Find" field or the "Replace with" field or both. "Find Next" button highlights the next matched string from the current cursor position. It will automatically pass the end

of the file and go to the beginning of the file. If there is no matched string, a warning message will appear. After a matched string has been found, the "Replace button" is enabled. "Replace button" replaces the matched string with the replacing string. "Replace all" replaces all the occurrences of the searching string with the replacing string. If there is no matched string, a warning message will appear. The radio buttons "Up" and "Down" specify the search direction. If "Match whole word only" is checked, the words instead of character streams are compared with the search String. If "Match case" is checked the search is case sensitive.

Known bugs: 1. If you click "Find Next" continuously, some of the clicks will be ignored. 2. In Search Dialog box, when you do a search and find the string, the replace button is enabled. Before you replace the string, if you either delete or insert some characters, then replace the string, the result are not what you expected. So don't do anything else between find and replace.

Delete

General usage: If there is selected text (highlighted) in the text editor, it is deleted. Otherwise nothing happens.

Known bugs: None

Cut

General usage: If there is selected text (highlighted) in the text editor, the selected text is deleted and stored in the buffer (later it can be retrieved using paste). Otherwise nothing happens.

Known bugs: None

Copy

General usage: If there is text selected(highlighted) in the text editor, the selected text is copied into in the buffer (later can be retrieved using paste). Otherwise nothing happens.

Known bugs: None

Paste

General usage: If there is text in the buffer(a cut operation or a copy operation has been done before the paste operation) the text in the buffer is inserted into the text editor at the cursor position. Otherwise nothing happens.

Known bugs: None

Undo

General usage: undo the operation just performed. For example, if a paste operation is just performed, after the undo, the pasted text disappears as if the paste operation never happened. There is only one level of undo. The operations that can be undone include: insert, delete, cut, paste, replace and replace all. For insert operation, undo deletes the word you just typed. A word is determined by two ways: (1) undo deletes all the characters you typed until a space or a tab or a new line. (2) after the cursor is moved(by mouse or the arrow keys) and some non-space characters are typed, undo deletes these characters. For delete operation, if you use backspace or delete key to delete a character, undo reinserts the character. If a chunk of text is deleted (highlight and delete), then the chunk of text reappears. Replace and replace all operations happen in the search operation. After the text is replaced, the undo operation is able to undo the replacement.

Known bugs: None

Redo

General usage: Redo reverses the undo operation.

Known bugs: None

A.2.3 The Make menu:

Compile

General usage: Compile brings a dialog box in which the users can enter options and arguments for the compiler. For a description of these see the menus help file.

Known bugs: When there are spaces in the file path directories it will try to convert them to a dos path. For example, d:/java source code/test.java will be converted to d:/javaso 1/test.java since javac does not parse space in the file path. However, the conversion is not always correct, and you have to correct it manually. You can't have space in the source file name, since javac won't recognize it.

A.2.4 The Repository menu:

Create Repository

General usage: Opens the Repository Maker Dialog. In this dialog you can create a new repository or add cases to an existing. The current CLASSPATH environment variable will by default appear in the CLASSPATH text field. The "Repository Maker" uses this path to search for Java classes. You may change this value by entering your own path to Java classes. However it is not recommended to put "untrusted" classes into the repository. This does NOT change the systems CLASSPATH variable. *Warning: It takes long time to create a Repository of e.g. Suns JDK1.X*

Known bugs: None

A.2.5 The Agent menu:

Start

General usage: Starts the Manager Agent, which in turn starts other agents. There are three kinds of agents: Manager Agent, Coordinator Agents and Workforce Agents.

Known bugs: The Manager Agent is unable to compile classes contained in packages. You can avoid this problem in some occasions by commenting out the package statement.

Stop

General usage: Stops the Manager Agent. The Manager Agent itself controls all other starting and stopping of agents.

Known bugs: If you stop the agent while it's working some agents may survive the "killing of all" invoked by the Manager.

Settings

General usage: Opens the Agent Settings Dialog. You can 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 reached. 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. You can 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.

Known bugs: None

A.2.6 The Options menu:

Font

General usage: Font brings a dialog box on the screen where you can change the font of the text Editor. Default button restores the options defined by the author.

Known bugs: If the option file can't be read or written, then the changes don't show. In this case, the default options will be used.

Foreground

General usage: Foreground brings a dialog box on the screen where you can change the foreground color of the text Editor. Default button restores the options defined by the program.

Known bugs: If the option file can't be read or written, then the changes won't show. In this case, the default options will be used.

Background

General usage: Background brings a dialog box on the screen where you can change the background color of the text Editor. Default button restores the options defined by the program.

Known bugs: If the option file can't be read or written, then the changes won't show. In this case, the default options will be used.

A.2.7 The Help menu:

Help describes the function of every Menu item.

A.2.8 The CASE FOUND! menu (Only appears if a reusable case is found):

Start reasoner

General usage: Starts the reuse tool. The reuse tool assists in matching signature features from the retrieved base cases to the class under development (target case). 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). In this part of the tool the similarities between signatures are used in the mapping process. When the tool is asked to establish mappings it only establishes those with a similarity higher than some threshold (0.75), asks for verification of those not so similar (0.5 - 0.75), and discards the rest. Depending on the total similarity between the cases the system suggests direct reuse, subclassing (extension), or literal reuse (copy-and-paste).

Known bugs: None

Ignore

General usage: Ignores the "CASE FOUND!" message from the Manager Agent and removes the "CASE FOUND!" menu.

Know bugs: None

A.3 Scenario

- Open the `MyApplet.java` file in the `C:/ascrarad/testing` directory (`$HOME/ascrarad/testing` on UNIX). (File|Open...)
- Go to the Agent menu and start the Agent Settings dialog. (Agent|Agent Settings...)
- Type in:
 - The number of cases you want to retrieve (e. g. 5).
 - The time interval for the Manager Agent, i. e. how often it should start the retrieval processes (e.g. 5 minutes).
 - Set the retrieval threshold (e.g. 0.50 "AVERAGE").
 - Choose "Using Organization" from the Action menu.
 - Click "OK"
- Go to the Agent menu and start the Agent. (Agent|Start...)
- While the agent is working you may extend the class, change it or open a new file.
- When and if the "CASE FOUND" message appears start the adaptation tool. (CASE FOUND|Start Reasoner...)
- Check out the adaptation features of the adaptation tool.

Appendix B

Installing and running the application

All files needed for the installation can be downloaded from:

<http://www.ifi.uib.no/staff/stein/ascrarad/>

B.1 WinNT/2000

1. Create the directory `C:/ascrarad`
2. Copy `ASCRARAD.zip` to that directory.
3. Unzip the `ASCRARAD.zip` file inside the installation directory using WinZip (can be obtained from www.winzip.com).
4. Install Suns SDK 2 v 1.2.2 if you haven't already installed it (can be obtained from <http://java.sun.com/products/jdk/1.2/>).

5. Include

```
c:/jdk1.2.2/lib/dt.jar;c:/jdk1.2.2/lib/tools.jar;  
c:/jdk1.2.2/jre/lib/i18n.jar;c:/jdk1.2.2/  
jre/lib/jaws.jar;c:/jdk1.2.2/jre/lib/rt.jar;  
c:/jdk1.2.2/jre/lib/pluginprov.jar
```

in the systems CLASSPATH variable. It is important that you include the complete paths to the jar files to make sure the jikes compiler works.

6. Include C:/jdk1.2.2/bin;C:/ascrarad/jikes in the systems PATH variable.

7. Double-click the run_ascrarad.bat file.

B.2 UNIX (Solaris)

1. Create the directory ascrarad/ in your home directory.

2. Copy ASCRARAD.tar.gz to that directory.

3. Unzip the ASCRARAD.tar.gz file inside the installation directory (*gzip -d ASCRARAD.tar.gz*).

4. Untar the ASCRARAD.tar file inside the installation directory (*tar xf ASCRARAD.tar*).

5. Install Suns SDK 2 v 1.2.2 if you haven't already installed it (can be obtained from <http://java.sun.com/products/jdk/1.2/>).

6. Include

```
/usr/java/lib/dt.jar:/usr/java/lib/tools.jar:  
/usr/java/jre/lib/exactvm.jar:  
/usr/java/jre/lib/i18n.jar:
```

`/usr/java/jre/lib/rt.jar`

in your `CLASSPATH` variable. It is important that you include the complete paths to the jar files to make sure the jikes compiler works.

7. Include `/usr/jdk1.2.2/bin:$HOME/ascrarad/jikes` in your `PATH` variable.
8. Execute the `run_ascrarad.sh` file.

References

- [1] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.
- [2] K.-D. Althoff, A. Birk, Gresse von Wangenheim, and C. Tautz. Case-based reasoning for experimental software engineering. In M. Lenz, B. Bartch-Spörl, H.-D. Burkhard, and S. Wess, editors, *Case-Based Reasoning Technology - From Foundations to Applications*, pages 235–254. Springer-Verlag, 1998.
- [3] Klaus-Dieter Althoff, Markus Nick, and Carsten Tautz. Concepts for reuse in the experience factory and their implementation for cbr system development. In *Proceedings of the 11th German Workshop on Machine Learning*, August 1998.
- [4] J. R. Anderson. Arguments concerning representations for mental imagery. *Psychological Review*, pages 249–277, 1978.
- [5] J. H. Barkow, L. Cosmides, and J. Tooby. *The Adapted Mind*. Oxford Univ. Press, New York, 1992.
- [6] R. Barletta and W. Mark. Breaking cases into pieces. In *Proceedings of Case-Based Reasoning Workshop*, St. Paul, MN., 1998.

- [7] Sanjay Bhansali and Mehdi T. Harandi. Synthesis of UNIX programs using derivational analogy. *Machine Learning*, 10:7–55, 1993.
- [8] Jeffrey Bradshaw. An introduction to software agents. In Jeffrey Bradshaw, editor, *Software Agents*. The AAAI Press/The MIT Press, 1997.
- [9] Shirley Browne and James W. Moore. Reuse library interoperability and the world wide web. In *Proceedings of the 1997 international conference on Software engineering*, pages 684–691, Boston, United States, May 1997.
- [10] Daniel D. Corkill and Susan E. Lander. Organising software agents: The importance of design to effective system performance. *Object Magazine*, 8(2):41–47, April 1998.
- [11] Leda Cosmides and John Tooby. *The Adapted Mind: Evolutionary Psychology and the Generation of Culture*, chapter Cognitive adaptations for social exchange. Oxford University Press, New York, 1992. Jerome H. Barkow, Leda Cosmides and John Tooby Eds.
- [12] Leda Cosmides and John Tooby. *Mapping the mind: Domain specificity in cognition and culture*, chapter Origins of domain specificity: the evolution of functional organization, pages 85–116. Cambridge University Press, Cambridge, 1994. Lawrence A. Hirschfeld and Susan A. Gelman Eds.
- [13] Nachum Dershowitz. Programming by analogy. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An artificial intelligence approach*, volume II, chapter 15, pages 393–421. Morgan Kaufmann Inc., Los Altos, California, 1986.
- [14] Agent Working Group (ed. James Odell). Agent technology green paper. Working paper v0.91, <http://jamesodell.com/publications.html>, 2000.
- [15] Riza Cenk Erdur, Oğuz Dikenelli, and Halil Şengonca. A multiagent system for searching and retrieving reusable software components. In *14th International*

- Symposium on Computer and Information Sciences*, Kusadasi, Izmir, Turkey, 1999.
- [16] Luger G. F. *Cognitive Science: The Science of Intelligent Systems*. Academic Press, San Diego and New York, 1994.
- [17] Carmen Fernández-Chamizo, Pedro Antonio González-Calero, Mercedes Gómez-Albarrán, and Luis Hernández-Yáñez. Supporting object reuse through case-based reasoning. In Ian Smith and Boi Faltings, editors, *Advances in case-based reasoning : third European workshop, EWCBR-96*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.
- [18] Carmen Fernández-Chamizo, Pedro A. González-Calero, Luis Hernández-Yáñez, and Alvaro Urech-Baqué. Case-based retrieval of software components. *Expert Systems with Applications*, 9(3):397–405, 1995.
- [19] T. Finin, Y. Labrou, and J Mayfield. Kqml as an agent communication language. In Jeffrey Bradshaw, editor, *Software Agents*, pages 291–316. The AAAI Press/The MIT Press, 1997.
- [20] David Flanagan. *Java in a Nutshell*. O’Reilly, Sebastopol, CA, 2 edition, May 1997. ISBN 1-56592-262-X.
- [21] Ken Ford, Clark Glymour, and Patrick J. Hayes. *Android Epistemology*. MIT Press, Cambridge, 1995. ISBN 0-262-06184-8.
- [22] Gilles Fouqué and Stan Matwin. A case-based approach to software reuse. *Journal of Intelligent Information Systems*, 1:165–197, 1993.
- [23] Jay R. Galbraith. *Organization Design*. Addison-Wesley, 1977.
- [24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison-Wesley Publishing Company, Reading, MA, 1995.

- [25] Paulo Gomes and Carlos Bento. 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, 1999.
- [26] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification, The Java Series*. The Java Series. Addison-Wesley, Reading MA, 1 edition, 1996. ISBN 0-201-63451-1.
- [27] Scott Henninger. Tools supporting the creation and evolution of software development knowledge. In *Proceedings of the 12th Annual Conference on Automated Software Engineering*, pages 46–53. IEEE Computer Society Press, 1997.
- [28] Ye Huang. An evolutionary agent model of case-based classification. In Ian Smith and Boi Faltings, editors, *Advances in case-based reasoning : third European workshop, EWCBR-96*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.
- [29] Michael N. Huhns and Munindar P. Singh. Agents and multi-agent systems: Themes, approaches and challenges. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, pages 1–23. Morgan Kaufmann, 1998.
- [30] Ivar Jacobsen, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture Process and Organization for Business Success*. ACM Press, New York, 1997.
- [31] P. Katalagarianos and Y. Vassiliou. On the reuse of software: A case-based approach employing a repository. In *Automated Software Engineering*, volume 2, pages 55–86. Kluwer Academic Publisher, Dordrecht, Netherlands, 1995.
- [32] Craig A. Knoblock, Yigal Arens, and Chun-Nan Hsu:. Cooperating agents for information retrieval. In *Proceedings of the Second International Conference on Cooperative Information Systems*, UoToronto Press, Toronto, 1994.

- [33] Y. Labrou and T. A. Finin. 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, February 1997.
- [34] Susan E. Lander. *Distributed Search and Conflict Management Among Reusable Heterogeneous Agents*. PhD thesis, University of Massachusetts, 1994.
- [35] C. G. Langton. *Artificial Life: An Overview*. MIT Press, Cambridge, MA, 1995.
- [36] George F. Luger and William A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison-Wesley, 1997.
- [37] Pattie Maes. <http://pattie.www.media.mit.edu/people/pattie/homepage.html>.
- [38] Pattie Maes. Interacting with virtual pets, digital alter-egos and other software agents. Doors of Perception II Conference, Amsterdam, November 1994. <http://www.doorsofperception.com/doors/doors2/transcripts/maes.html>.
- [39] N.A. Maiden and A.G. Sutcliffe. Exploiting reusable specifications through analogy. *Communications of the ACM*, 35(4):55–64, 1992.
- [40] M. Minsky. *Society of Mind*. Simon and Schuster, New York, 1985.
- [41] Kanth Miriyala and Mehdi T. Harandi. Automatic derivation of formal software specifications from informal descriptions. *IEEE Trans. Software Engineering*, 17(10):1126–1142, 1991.
- [42] Stein Inge Morisbak. Samarbeidende cbr-agenter. Assignment written in the course Artificial Intelligence(iv281) at the Department Of Information Science, University of Bergen, May 1999.
- [43] A. Newell and H. Simon. *Human Problem Solving*. Prentice Hall, Engelwood Cliffs, N. J., 1972.

- [44] James Odell. Designing agents: Using life as a metaphor. *Distributed Computing*, pages 51–56, July 1998.
- [45] James Odell. Objects and agents: How do they differ? Working paper v2.2, URL: <http://jamesodell.com/publications.html>, 1999.
- [46] Tim O’Shea. The learnability of object-oriented programming systems. In *OOP-SLA ’86 Proceedings*, pages 502–504, New York, NY, September 1986. ACM, ACM Press.
- [47] Enric Plaza, Josep Lluís Arcos, and Francisco Martín. Inference and reflection in the object-centered representation language noos. *Journal of Future Generation Computer Systems*, pages 73–188, 1996.
- [48] Enric Plaza, Josep Lluís Arcos, and Francisco Martín. Cooperative case-based reasoning. In G Weiss, editor, *Distributed Artificial Intelligence meets Machine Learning, Lecture Notes in Artificial Intelligence*, pages 180–201. Springer-Verlag, 1997.
- [49] Enric Plaza, Josep Lluís Arcos, and Francisco Martín. Knowledge and experience reuse through communication among competent (peer) agents. *International Journal of Software Engineering and Knowledge Engineering*, 1999.
- [50] M.V. Nagendra Prasad. Distributed case-based learning. Article written for Andersen Consulting, Center for Strategic Technology Research, Thought Leadership, 1998. <http://www.ac.com/services/cstar/cstrdcb13.html>.
- [51] M.V. Nagendra Prasad, Victor R. Lesser, and Susan E. Lander. Retrieval and reasoning in distributed case bases. *Journal of Visual Communication and Image Representation, Special Issue on Digital Libraries*, 7(1):74–87, 1995.
- [52] M.V. Nagendra Prasad, Victor R. Lesser, and Susan E. Lander. Learning organizational roles in a heterogeneous multi-agent system. In *Proceedings of the International Conference on MultiAgent Systems*, 1996, Japan.

- [53] M.V. Nagendra Prasad and Enric Plaza. Corporate memories as distributed case libraries. In *Proceeding of the 10th Banff knowledge acquisition for knowledge-based system workshop*, number 40 in 2, pages 1–19, Banff, Canada, 1995.
- [54] Z. W. Pylyshyn. *Computation and Cognition: Toward a Foundation for Cognitive Science*. MIT Press, Cambridge, 1984.
- [55] Bergmann R. and Eisenecker U. Fallbasiertes schlieen zur untersttzung der wiederverwendung objektorientierter software: Eine fallstudie. In *Proceedings der 3. Deutschen Expertensystemtagung XPS-95*, 1995.
- [56] M. Redmond. Distributed cases for case-based reasoning: facilitating use of multiple cases. In *Proceedings AAAI-90*, 1990.
- [57] M. Sànchez-Marrè, J. Béjar, and U. Cortés. Reflective reasoning in a cbr agent. In *Procc. of the VIM Project Spring Workshop on Collaboration Between Human and Artificial Societies*, Spain, 1997.
- [58] Barry G. Silverman, Nabil Bedewi, and Alfredo Morales. Intelligent agents in software reuse repositories. In *Proc. of ACM 4th International Conference on Information and Knowledge Management (CIKM'95)*, Baltimore, Maryland, USA, December 1995. Workshop on Intelligent Information Agents.
- [59] Herbert A. Simon. Artificial intelligence: an empirical science. *Artificial Intelligence*, 77(1):95–177, 1995.
- [60] George Spanoudakis and Panos Constantopoulos. Analogical reuse of requirements specifications: A computational models. *Applied Artificial Intelligence*, 10(4):281–306, 1996.
- [61] Sun Microsystems, Mountain View, CA. *Java™ 2 Platform, Standard Edition, v 1.2.2 API Specification*. <http://java.sun.com/j2se/1.2/docs/api/>.

- [62] Sun Microsystems, Mountain View, CA. *JavaTM Platform 1.1 API Specification*. <http://java.sun.com/products/jdk/1.1/docs/api/packages.html>.
- [63] Sun Microsystems, Mountain View, CA. *Java 2 SDK, Standard Edition Documentation*, 1999. <http://java.sun.com/products/jdk/1.2/docs>.
- [64] Bjørnar Tessem and Solveig Bjørnstad. Analogy and complex software modeling. *Computers in Human Behavior*, 13(4):465–486, 1997.
- [65] Bjørnar Tessem, R. Alan Whitehurst, and Christopher L. Powell. Case-based support for rapid application development. In *SEKE'99 Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering*, pages 192–196, Skokie, Illinois, 1999. Knowledge Systems Institute.
- [66] R. Alan Whitehurst. *Systematic Software Reuse Through Analogical Reasoning*. PhD thesis, University of Illinois, Urbana-Champaign, IL, 1995.