



UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
<http://www.dit.unitn.it>

A MULTI-AGENT SYSTEM FOR CHOOSING
SOFTWARE PATTERNS.

Aliaksandr Birukou, Enrico Blanzieri, Paolo Giorgini, and Michael Weiss

October 2006

Technical Report # DIT-06-065

A Multi-Agent System for Choosing Software Patterns*

Aliaksandr Birukou, Enrico Blanzieri,
Paolo Giorgini
Department of Information
and Communication Technology
University of Trento - Italy

{birukou, blanzier, pgiorgio}@dit.unitn.it

Michael Weiss
School of Computer Science
Carleton University, Ottawa - Canada
weiss@scs.carleton.ca

ABSTRACT

Software patterns enable an efficient transfer of design experience by documenting common solutions to recurring design problems. They contain valuable knowledge that can be reused by others, in particular, by less experienced developers. Patterns have been published for system architecture and detailed design, as well as for specific application domains (e.g. agents and security). However, given the steadily growing number of patterns in the literature and online repositories, it can be hard for non-experts to select patterns appropriate to their needs, or even to be aware of the existing patterns. In this paper, we present a multi-agent system that supports developers in choosing patterns that are suitable for a given design problem. The system implements an implicit culture approach for recommending patterns to developers based on the history of decisions made by other developers regarding which patterns to use in related design problems. The recommendations are complemented with the documents from a pattern repository that can be accessed by the agents. The paper includes a set of experimental results obtained using a repository of security patterns. The results prove the viability of the proposed approach.

Categories and Subject Descriptors

H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—*Information filtering, Relevance feedback, Search process*; I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence—*Intelligent agents, Multiagent systems*; K.6.3 [Management of Computing and Information Systems]: Software Management—*Software development*

Keywords

Multi-agent system, implicit culture, patterns, Lucene

*The primary author of the paper is a PhD student

1. INTRODUCTION

A little more than ten years ago the authors of the book *Design Patterns* [9], the first major publication on software patterns, stated the problem of selecting patterns: “With more than 20 design patterns in the catalog to choose from, it might be hard to find the one that addresses a particular design problem, especially if the catalog is new and unfamiliar to you”. As time has passed, patterns have become a staple of current software development approaches. However, the problem of selecting patterns still exists. Moreover, it has become much more critical as the number of documented patterns is continuously increasing: for instance, the *Pattern Almanac* [15] lists more than 1200 patterns. And in the past six years since its publication, many new patterns and books on patterns have been published. The problem of choosing the appropriate pattern is particularly hard to solve for inexperienced programmers [17], and tools assisting in this process become of utmost importance.

In this paper, we address the problem of selecting patterns from a social point of view. To help a developer make a decision about which patterns to use, getting suggestions from her group of peers is important. We present a multi-agent system aimed at helping developers choose the patterns suitable for a given design problem. The system is based on the implicit culture framework [5]. This framework has been implemented within the *IC-Service* [4], which provides recommendations on patterns. These recommendations are created using a history of previous user interactions with the system, namely with their personal agents. The task of personal agents in the system is to distribute the knowledge about the use of patterns within a community of developers without their direct involvement. Namely, agents provide their users with suggestions on which patterns are suitable for a specified problem. The suggestions are complemented with a description of patterns from the pattern repository accessible by the agents. Currently, we use a Lucene-based implementation of the repository that contains a set of security patterns published on patterns.org [10], one of several popular online repositories for patterns.

A preliminary description of our approach has been presented in [14]. The main contributions of this paper are a description of an actual prototype of a system for selecting patterns, and an experimental evaluation of the approach.

The paper has the following structure. Section 2 provides a brief background on patterns and the ideas underlying them, while Section 3 illustrates the implicit culture framework. In Section 4 we describe the general architecture of our system, document a scenario of using the system, and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

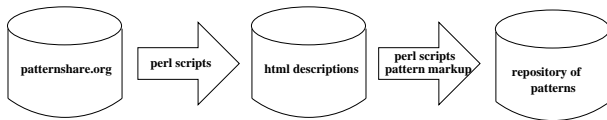


Figure 1: The pattern extraction process. (1) Information about the security patterns is extracted from the patternshare.org repository using Perl scripts; (2) the pattern descriptions are then converted to an XML format using Perl scripts and a pattern markup language.

provide details on the implementation. Experimental results are presented in Section 5, and related work is discussed in Section 6. Finally, Section 7 concludes the paper.

2. BACKGROUND

Patterns enable an efficient transfer of design experience by documenting common solutions to recurring design problems in a specific context [2]. Each pattern describes the situation when the pattern can be applied in its context. The context can be thought of as a precondition for the pattern. This precondition is further refined in the problem description with its elaboration of the forces that push and pull the system to which the pattern is applied in different directions. Here, the problem is a precise statement of the design issue to be solved. Forces are design trade-offs affected by the pattern. One of the most significant contributions of patterns is that they intend to make the trade-offs between the forces involved explicit. The trade-offs can be documented in various forms. One popular approach is to document them as sentences like “on one hand ..., but on the other ...”. The solution describes a way of resolving the forces. Some forces may not be resolved by a single pattern. In this case, a pattern often includes references to other patterns, which help resolve forces that were unresolved by the current pattern. Together, patterns connected in this way are often referred to as a pattern language. Links between patterns can be of different types, including uses, refines, and conflicts [12, 16]. Patterns that need another pattern link to that pattern with uses. Patterns specializing the context or problem of another pattern refine it. Patterns that offer alternative solutions conflict, and should not be used together.

Recently, there have been several efforts in making patterns available in online pattern repositories, where they can be browsed and searched by various criteria. An early example was the Pattern Almanac [15], which is available in electronic form (www.smallmemory.com/almanac). A more recent example is the patternshare.org site hosted by Microsoft (patternshare.org). In order to store patterns in a repository, a structured pattern representation must be adopted. There have been several proposals, most notably the Pattern Language Markup Language (PLML) [7].

In this work we have adopted a format that is specific to a set of security patterns published on patternshare.org [10]. We have defined an XML representation for these patterns and extracted the content of the subset of this repository from the website. The extraction process is illustrated in Figure 1. Our current representation contains the following elements: `Pattern.Context`, `Pattern.Problem`, `Pattern.Solution`, `Pattern.KnownUses`, and `Pattern.RelatedPatterns`, as well as elements specific to the patternshare.org site, but not required for our purposes (see Figure 2 for an example of the

```
<Pattern id="SingleAccessPoint">
  <Pattern.Name>Single Access Point</Pattern.Name>
  <Pattern.View>Application Architecture</Pattern.View>
  <Pattern.Role>Architecture</Pattern.Role>
  <Pattern.Aspect>Function</Pattern.Aspect>
  <Pattern.Summary>Single entry point for each process.
  </Pattern.Summary>
  <Pattern.Context>You are planning to secure a system from
  outside intrusion. The system provides a bunch of services
  but you want to secure the system as a whole.
  </Pattern.Context>
  <Pattern.Problem>A security model is difficult to validate
  when there are multiple ways for entering the application.
  How can we secure a system from outside intrusion?
  </Pattern.Problem>
  <Pattern.Solution>Set up only one way to get into the system
  and if necessary, create a mechanism to decide which sub-
  application to launch. Typically most applications use a log
  in screen to accomplish the single access point.
  </Pattern.Solution>
  <Pattern.RelatedPatterns>Single Access Point validates the
  user's login information through a <Pattern idref=
  "PolicyEnforcementPoint"/> and uses that information to
  initialize the user's Roles and Session. A Singleton can be
  used to implement a Single Access Point.
  </Pattern.RelatedPatterns>
  <Pattern.Publication>This pattern appeared in the paper titled
  "Architectural Patterns for Enabling Application Security" by
  Joseph Yoder and Jeffrey Barcalow in Pattern Languages of
  Programmers conference in 1997. Peter Sommerlad integrated
  the material in the Security Pattern book titled "Security
  Patterns: Integrating Security and Systems Engineering".
  </Pattern.Publication>
</Pattern>
```

Figure 2: An example of the XML representation of a pattern (our pattern markup language).

pattern representation). However, our approach does not depend on a specific pattern representation.

We are also not concerned, at this stage of development, with how easy it is to deploy our approach; however, in the future; we plan to converge towards a standard like PLML.

3. IMPLICIT CULTURE

This section presents an overview of the general idea of the implicit culture framework and Systems for Implicit Culture Support (SICS) that provide the basis for the *IC-Service* we have used to provide recommendation facilities within our system. A more thorough description of the *IC-Service* can be found in the work of Birukou et al. [4]. The paper by Blanzieri et al. [5] contains a detailed description of implicit culture theory and the SICS architecture.

Our motivation for adopting an implicit culture approach stems from the difficulty less experienced developers face in using patterns. Developers who wish to apply patterns from a domain that is not their main area of expertise encounter similar difficulties. A good example is the security domain. For any but trivial applications, security is a key concern, however, making the application secure is not the main concern of the application developer. Security patterns [16] can help developers with this task: they provide guidance to non-experts in security for designing secure application. However, a significant challenge remains: how do developers decide which patterns they should use? The following quote from [17] is indicative of the difficulty inherent in using patterns:

Only experienced software engineers who have a deep knowledge of patterns can use them effec-

tively. These developers can recognize generic situations where a pattern can be applied. Inexperienced programmers, even if they have read the pattern books, will always find it hard to decide whether they can reuse a pattern or need to develop a special-purpose solution.

The difference between these two types of developers is that an experienced developer uses implicit knowledge (in particular, her own experience) about the problem (see [6] for a more general discussion on this point). When we look at the community of a developer's peers, knowledge is called *implicit* when it is embodied in the capabilities and abilities of the community members (developers). It is *explicit* when it is possible to describe and share it through documents or knowledge bases. To select appropriate patterns inexperienced developers should acquire the implicit knowledge that more experienced developers have.

We argue that it is possible to shift the pattern selection behavior exhibited by inexperienced developers towards the behavior of more experienced developers by suggesting patterns suitable for their current design task. To determine which patterns are suitable we use the history of previous interactions with the system, i.e. which patterns other developers have chosen in a similar situation. We refer to the pattern selection behavior of experienced developers as the *culture* of that developer community. When inexperienced developers start behaving in agreement with the community culture, a knowledge transfer from experienced to inexperienced developers occurs. The relation characterized by this knowledge transfer is called "*implicit culture*".

For example, let us consider a programmer that needs to improve access control in a system that offers multiple services. Let us suppose that for an experienced developer knowledgeable in security it is apparent to use the Single Access Point pattern. If the system is able to use previous history to suggest that the novice uses the Single Access Point pattern and she actually uses it, then we say that she behaves in accordance with community culture and the implicit culture relation is established. We will use this example as a running example throughout the paper.

In our system, personal agents use the *IC-Service* to recommend patterns. The *IC-Service* provides an interface for accessing a SICS that is dealing with observations coming from the system and produces recommendations. The general architecture of a SICS consists of the following three components:

- an *observer*, which stores information about actions performed by the user in a database of observations;
- an *inductive module*, which analyzes the stored observations and applies learning techniques (namely, data mining or machine learning) to develop a theory about actions performed in different situations;
- a *composer*, which exploits the information collected by the observer and derived by the inductive module to suggest actions in a given situation.

In terms of our problem domain, the observer saves information about the problem (a textual description plus some characteristics of the project, in whose context the problem occurred), which patterns have been proposed as a solution,

and which pattern has been chosen in return. The inductive module discovers problem-solution pairs by analyzing the history of the interaction of users with the system. A set of problem-pattern pairs (which patterns are selected for what problems) form a *theory*. The goal of the composer is twofold. Firstly, it compares the description of a problem faced by a developer with the problem part of the theory mined by the inductive module in order to suggest the corresponding pattern. Secondly, the composer tries to match the problem with the pattern by analyzing the history of observations and calculating the similarity between the problem description given by the user and the problem descriptions which users provided for patterns previously selected for similar problems.

Above, we described the basic idea behind using an implicit culture approach for selecting an individual pattern. However, it is clear that this is only part of a larger process, where a developer selects patterns in an iterative manner. Patterns are never used in isolation, rather patterns are selected in the context of the other patterns that have already been applied (see e.g. [2] for an overview of this general process). This means that we can also make use of links between patterns (the information in the `Pattern.RelatedPatterns` section of our pattern representation) to recommend patterns. In this paper, however, we focus on the former, the selection of individual patterns, in the understanding that we will be able to use this as a step of a larger, iterative process.

4. THE ARCHITECTURE OF THE SYSTEM

This section gives a description of the system and of the search process. The system is intended for the use within an IT-company, or just within a project group, and it should adapt the suggestions to the specificity of the software development process adopted within the company or project group, converging to the "community culture".

The architecture of the system is given in Figure 3. The system consists of a web-based user interface at the client side and a multi-agent platform at the server side. A user accesses the system by submitting a description of the problem via the web-based interface in her browser. Apart from the problem description, a description of the project, in which the problem is encountered, is submitted. In the remainder of the paper, we will refer to the problem description together with the project description as a user *query*. The problem is described by a set of keywords, optionally restricted to specific elements of the pattern markup language. The project description can be represented as a set of properties (e.g. project size, required level of data protection, etc.). In our running example, the user could submit a query with the following problem description: "access control in a system that offers multiple services" related to the project that has the following set of properties: {Name: OnlineBanking, SecurityLevel: High, ProjectSize: Medium}.

Each user is assisted by a personal agent. The goal of a personal agent is to help the user choose a pattern suitable for the submitted query. In order to fulfill this goal, the agent is capable of accessing one or more information sources. It can also obtain recommendations from the *IC-Service*. In the current implementation there is only one information source: a Lucene-based repository of patterns. The personal agents in the system are software agents running on the multi-agent platform at the server side.

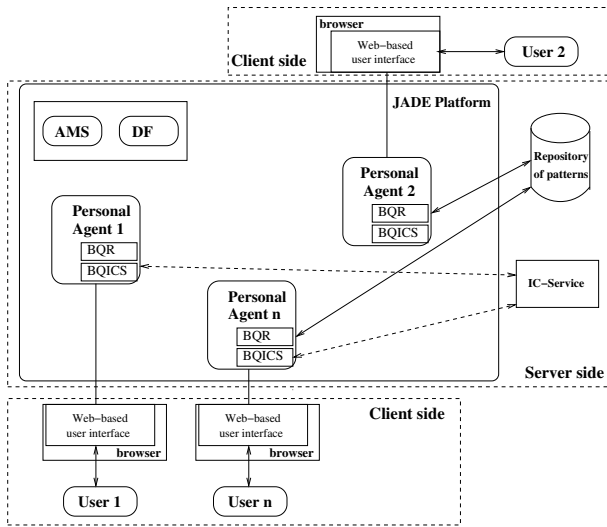


Figure 3: The architecture of the system. *Personal agents* process queries from *users* and access the repository of patterns to retrieve potentially relevant patterns; the *IC-Service* is exploited by the agents in order to create recommendations from the history of past interactions; the *Agent Management System (AMS)* exerts supervisory control over the platform: it provides agent registration, search, etc.; the *Directory Facilitator (DF)* provides agents with other personal agents’ IDs; *BQR* stands for BehaviourQueryRepository used to access the repository, and *BQICS* stands for BehaviourQueryICService respectively.

Table 1: The actions observed by the system.

| action | objects | attributes |
|---------|------------------------------|--------------------|
| request | problem_description | project_properties |
| apply | pattern, problem_description | project_properties |
| reject | pattern, problem_description | project_properties |

The use of agents and of the implicit culture ideas allows for the distribution of the knowledge without the direct involvement of the users. In our example, the user’s personal agent should suggest using the Single Access Point pattern. If the agent does so because someone else has already used this pattern for similar problems, it distributes the knowledge about the use of patterns within the community.

4.1 The Configuration of the IC-Service

In terms of the implicit culture framework user queries are objects in *situations*. The goal of the SICS within the *IC-Service* is to find the most similar situations. Besides abstractions of situations we need to introduce several terms. We treat developers as *agents* who perform *actions* on *objects*. We further suppose that actions have *attributes*, which are features helpful for the analysis of those actions. In the implicit culture framework, actions are assumed to be performed in situations, therefore we can speak of *situated actions* [18]. In our application, the SICS analyzes the actions presented in Table 1. Since all the actions are performed by developers, we omit agents from the table.

We explain the information contained in the table in detail. A developer *requests* the system to find patterns that are suitable for her task. The query contains a description

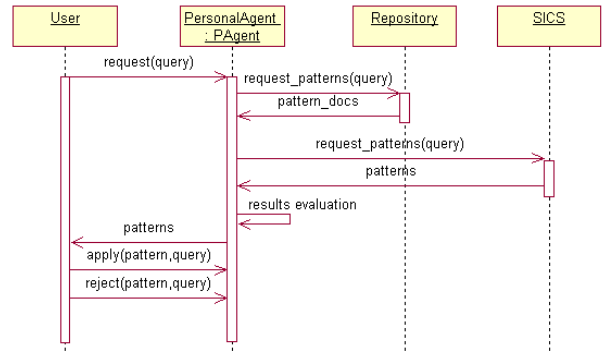


Figure 4: Sequence diagram of the search process.

of the problem and the properties of the project where the problem has been encountered.

The developer *applies* the pattern when she implements it in code. To observe this action, for now, we request explicit feedback from the developer that a pattern has been used, but in principle, a case tool which support the subsequent (semi-)automatic implementation of the patterns (e.g., [13]) could provide this kind of feedback indirectly. Alternatively, techniques for automatic detection of the patterns in a piece of software can be used, for instance, the PTIDEJ prototype tool [1] provides such a functionality.

It is possible for the developer to specify the inapplicability of a pattern to the task explicitly. Such the indication is recorded as a *reject* action.

In the running example, examples of actions are:

```
request(query)
apply(SingleAccessPoint, query)
reject(Authenticator, query)
```

where *query* has the problem description “access control in a system that offers multiple services” and project properties {Name: OnlineBanking, SecurityLevel: High, ProjectSize: Medium}.

Obviously, the main problem lies in the “observability” of the users’ actions. The most problematic action to observe is the action of using a pattern for a problem. In the current implementation we assume the user explicitly indicates this action in the system, specifying that she selected the pattern X for the problem A, where the problem corresponds to a search in the history of searches. This is a reasonable assumption, since the amount of the input required from the user is very low. However, implicit sources of feedback such as clicking on a pattern description, time spent reading the description, etc. can be used together with or instead of the explicit feedback. Fox et al. have shown [8] that a combination of multiple sources of implicit feedback can produce results comparable to those based on explicit feedback.

4.2 Search in the System

The search scenario is given in Figure 4. A user submits a query via the user interface, from where the query is forwarded to the user’s personal agent. In the first step of the search process, the personal agent accesses the pattern repository and retrieves a set of patterns relevant to the query. In the second step, the personal agent submits a query to the SICS and receives a list of recommended pat-

terns. Thus, the result consists of patterns retrieved from the repository and patterns recommended by the SICS. The feedback from the user is collected via the *apply* and *reject* actions, which mark a pattern as suitable or unsuitable for the problem, respectively.

The SICS inside the *IC-Service* processes the query within two steps. In the first step, the SICS matches the action contained in the query, i.e. the *request* action, with the theory and determines the action that must follow, i.e. the *apply* action. In the second step, the SICS finds situations where the *apply* action has been previously performed, thus determining the patterns used for similar problems in the past. In this step, the similarity between the current query and the previously submitted queries is calculated. As a result, the SICS returns a set of patterns that have been used for similar problems in the past.¹ A pattern is recorded as “applied” or “rejected” if a user indicates so explicitly.

Let us illustrate how the search process takes place in our example. The user submits the *request* action with the following query: {ProblemDescription: “access control in a system that offers multiple services”; Project: {Name: OnlineBanking, SecurityLevel: High, ProjectSize: Medium}}. In the first step the agent retrieves patterns from the repository: SingleAccessPoint, PolicyEnforcementPoint, and RoleBasedAccessControl. In the second step, the agent queries the *IC-Service*. The SICS matches the *request* action with the theory. The theory contains rules of essentially the following form:

if *request*(query) then *apply*(pattern-X,query)

This means that the *apply* (and not, e.g. a *reject*) action must follow the *request* action. So, the SICS matches the *request* action with that part of the theory that represents a problem, and searches for situations where the *apply* action has been performed. It finds the following situations (situation_id, the action, problem description, project, pattern):

| | | | | |
|---|-------|-----|----|------------------------|
| 1 | apply | pd1 | pp | SingleAccessPoint |
| 2 | apply | pd2 | pp | PolicyEnforcementPoint |

where pd1=“access control in a system that offers multiple services”, pd2=“only authorized clients should access the system”, pp={Name: e-BookShop, SecurityLevel: Medium, ProjectSize: Medium}. As a result, the SICS returns the SingleAccessPoint pattern, chosen in the most similar situation. After the evaluation of the results, the following list of patterns is displayed in the user interface: {SingleAccessPoint, PolicyEnforcementPoint, RoleBasedAccessControl }. Having analyzed the proposed patterns, the user *applies* the SingleAccessPoint pattern and indicates this in the user interface. She also marks the RoleBasedAccessControl pattern as unsuitable, thus performing the *reject* action.

4.3 Implementation Details

The system is implemented using JADE 3.4 (Java Agent DEvelopment framework). For creating recommendations the *IC-Service* is used. Although called a “service”, the *IC-Service* can be used in a number of ways, in particular as a Java library (the way we use it in our system).

¹If the database of observations is large, then patterns *generally used* for such problems must be returned. This information is in the theory developed by the inductive module.

To build the repository of patterns we took the following steps: (1) the descriptions of security patterns are extracted from patternshare.org using scripts; (2) the pattern descriptions are converted to the XML format using scripts; (3) the XML documents representing patterns are indexed with Apache Lucene 2.0. Apache Lucene is a fully-featured text search engine library available as an open source Java project(<http://lucene.apache.org/>). The Lucene library is also used to access the repository of patterns from the personal agents. However, our approach does not depend on a particular repository or a tool for accessing the repository. Moreover, the repository can be further extended with adding other patterns.

5. EVALUATION

The goal of the experiment is to compare the performance of the system with and without the SICS.

In the experiment we implemented in each agent a class that simulates the querying behavior of the real user. The main functions of this class are: (1) provide pseudo-user input in order to enable the personal agent’s recommendations, and (2) generate pseudo-user response to the recommendations. The input is provided and the responses are generated according to a user profile. The user profile contains a sequence of sets of keywords (queries) and a set of patterns. The intuition behind the user profile is as follows: the user has a single problem to solve using patterns, the problem can be described in a number of ways (each set of keywords in the sequence describes the problem), and the problem can be solved with the use of one of the patterns contained in the user profile. So, in the experiments, a query contains only a problem description in free-text form and does not contain information about the project.

During the simulation the multi-agent platform contains a special agent, the simulation manager, which is responsible for the simulation. The task of the simulation manager is to create a number of personal agents and to collect the information regarding searches. The number of agents is specified in the simulation scenario. Personal agents send the information about the simulated searches to the simulation manager using the FIPA-Request protocol.

We use the following measures [3] in order to evaluate the quality of suggestions:

- We call a pattern **relevant** to a problem if it can be applied for solving the problem. More specifically, pattern(s) contained in a user profile are relevant for the problem described with the keywords from this profile.
- **Precision** is the ratio of the number of suggested relevant patterns to the total number of suggested patterns, relevant and irrelevant.
- **Recall** is the ratio of the number of proposed relevant patterns to the total number of relevant patterns.
- **F-measure** is a trade-off between precision and recall. It is calculated as follows:

$$\text{F-measure} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

In our experiment, we have not used the inductive module of the SICS to update the theory and the recommendations

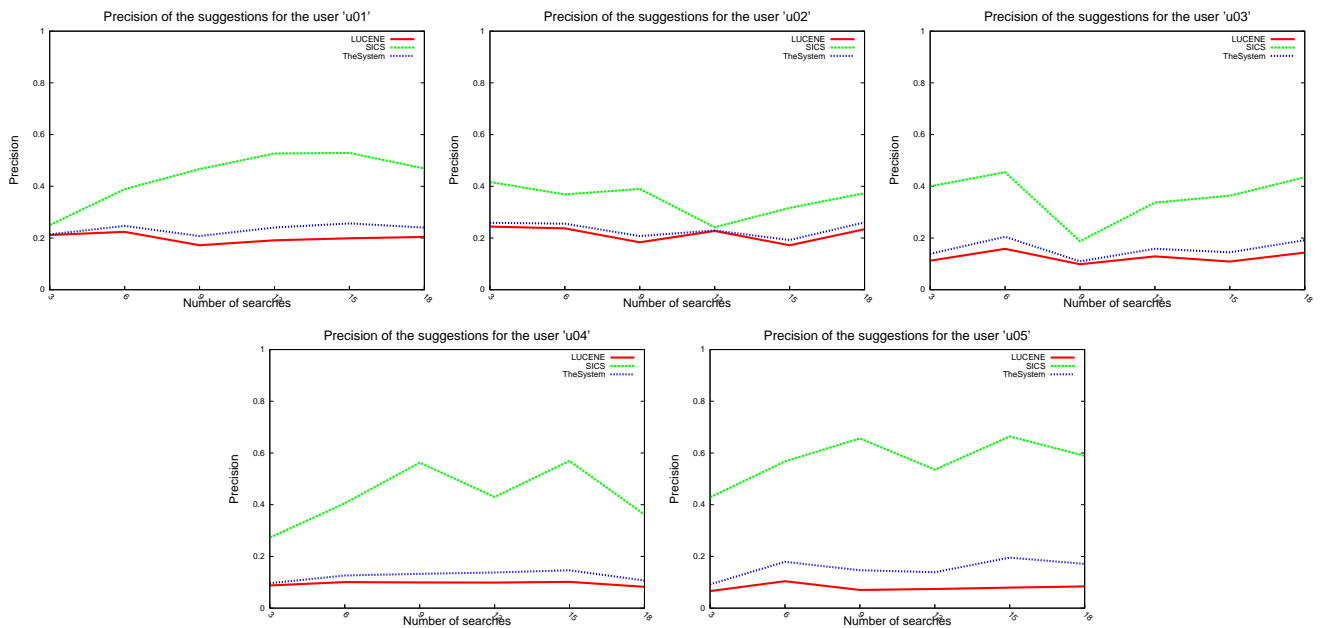


Figure 5: The precision of suggestions in the experiment

Table 2: The user profiles

| ID | profile pattern | relevant pattern |
|-----|----------------------------------|---|
| u01 | ControlledProcessCreator | ControlledObjectCreator Execution Domain |
| u02 | StatefulFirewall | PacketFilterFirewall ProxyBasedFirewall |
| u03 | VirtualAddressSpaceAccessControl | ControlledProcessCreator ExecutionDomain |
| u04 | Authorization | ReferenceMonitor RoleBasedAccessControl |
| u05 | MultilevelSecurity | ReferenceMonitor RoleBasedAccessControl |

are generated entirely by the composer module. Also, models of the users have not produced *reject* actions, just *request* and *apply*.

To build a small community of five developers, we considered five patterns from the repository of Security Patterns² and assigned them to each of the developers as shown in Table 2. These patterns have been used in order to create sequences of sets of keywords, corresponding to the descriptions of the problems encountered by the users. The sequences are created as follows: given a document, we construct a distribution of the terms in this document and then each element of the k -element sequence is a sample from this distribution, represented as an n -dimensional tuple. Here $k \geq 1$ is the number of searches performed in a simulation, and each query in the sequence consists of $n \geq 1$ keywords.

To determine the patterns that are marked as “solution to the problem” and are placed in the user profile, the following approach is adopted. We represent each document in the repository as ‘a bag of words’ [3]. Then we calculate the similarity between the document used to create the profile and the rest of the repository. The cosine similarity metric [3] is used. The m document(s) with the highest

²The repository of Security Patterns contains 59 patterns.

similarity (excluding the documents used to create profiles) are added to the profile as “solutions to the problem”. We set $m = 2$ and created five user profiles using patterns that are given in Table 2. Please note that there is a partial overlap in the profiles (e.g. u01 and u03), so the transfer of knowledge takes place.

The index of the repository of patterns has been built using the Lucene library. This library allows for boosting some fields of the document when searching the repository. In our experiment we boosted the fields `Pattern.Name` and `Pattern.Summary`.

In the experiment we set $n = 3$, so user queries consisted of three keywords, and we ran simulations for $k=3,6,9,12,15$, and 18, measuring the precision, recall, and F-measure of the recommendations after completing each k -query sequence. At the end of each k -query sequence, the database of observations is deleted in order to have the *IC-Service* producing recommendations from scratch. We repeated simulations 10 times and averaged the precision, recall, and F-measure to control the effect of the order and keywords of queries.

The results contain the precision, recall and F-measure of the patterns retrieved from the Lucene pattern repository, recommended by the SICS module, and by the system (both repository results and recommendations). Figure 5 shows the precision of the recommendations produced by the five personal agents for five developers. Analogous results have been obtained for the recall (Figure 6) and F-measure (Figure 7). The curves marked as “LUCENE” correspond to the performance of the system without the SICS module.

The results show that the recommendations of the system maintain a certain level of quality even for a small number of searches. The precision of the SICS’s recommendations is almost always higher than the precision of patterns obtained from the Lucene repository. Contrary to precision, the Lucene results are better in terms of the recall, although the gap is not so big as for the precision curves. This is

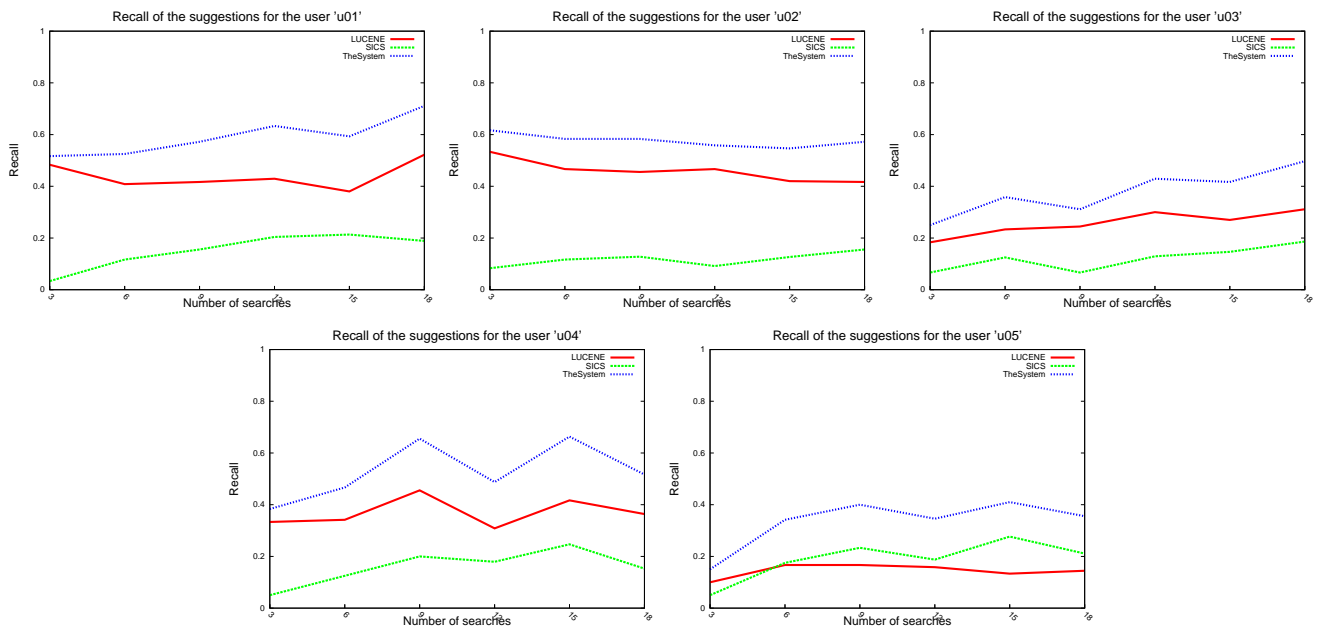


Figure 6: The recall of suggestions in the experiment

consistent with the fact that the number of the patterns retrieved from the Lucene repository is limited only by the number of documents relevant to the query, while the number of recommendations from the SICS is fixed and equal to one in the experiment. The F-measure of suggestions produced by the system as a whole, in the most cases is higher than the F-measure of the suggestions produced by the Lucene or the SICS alone. This suggests that (1) the system with the SICS module outperforms the system without this module, (2) the approach of complementing results from the pattern repository with recommendations of the SICS proves to be useful.

6. RELATED WORK

The paper by Kung et al. [11] represents the most related work to our approach. The authors propose a methodology for constructing expert systems which suggest design patterns to solve problems faced by developers. They also present a prototype, the Expert System for Suggesting Design Patterns (ESSDP) which implements the methodology. ESSDP selects a design pattern based on the user's requirements. A user interacts with the system in a question-answer manner, which helps to narrow down the selection process. At the end of the interaction, a suitable design pattern is offered to the user. There are several significant differences between our approach and ESSDP. Firstly, ESSDP assumes the knowledge acquisition as the primary step of the methodology. In this step human experts must fill in the knowledge base with some pre-defined rules. Differently, in our system the SICS learns from the interactions with users, without any initial knowledge base, allowing for continuous improvement of suggestions. Moreover, we exploit interactions with inexperienced users as well, offering to novices patterns that have been chosen in similar situations not only by experts but also by other novices. Thus we support sharing users' experience with others. Secondly,

our architecture is not restricted to the use of a rule-based knowledge base assuming that different learning techniques can be adopted.

There is also a number of tools that are dealing with the refactoring of an old code using design patterns [1, 13]. In most of the cases it is supposed that the choice of the pattern to use for refactoring is made by a developer. Adding the functionalities manifested by our system would enable a means for facilitating the selection of the patterns in these tools. It would also take the place of the explicit indication about the uses of patterns.

7. CONCLUSION AND FUTURE WORK

We have presented a multi-agent system that facilitates the process of the selection of patterns suitable for a given problem. The system is based on the implicit culture framework that uses the history of user-system interactions to provide recommendations on patterns. The recommendations are supplemented with the results obtained from the pattern repository and the viability of our approach has been proven by the experimental results.

Future work includes the implementation and evaluation of more complex recommendation scenarios such as recommending sequences of patterns, or recommending core patterns in a given group of patterns (for training).

The system described in the paper is available as an open-source project. Only the implementation with limited functionality is available from the project site now, but future extensions will be made public through this venue.

8. ACKNOWLEDGEMENTS

This work is funded by research projects EU SERENITY "System Engineering for Security and Dependability", COFIN "Artificial Intelligence Techniques for the Retrieval of High Quality Information on the Web" and by Fondo

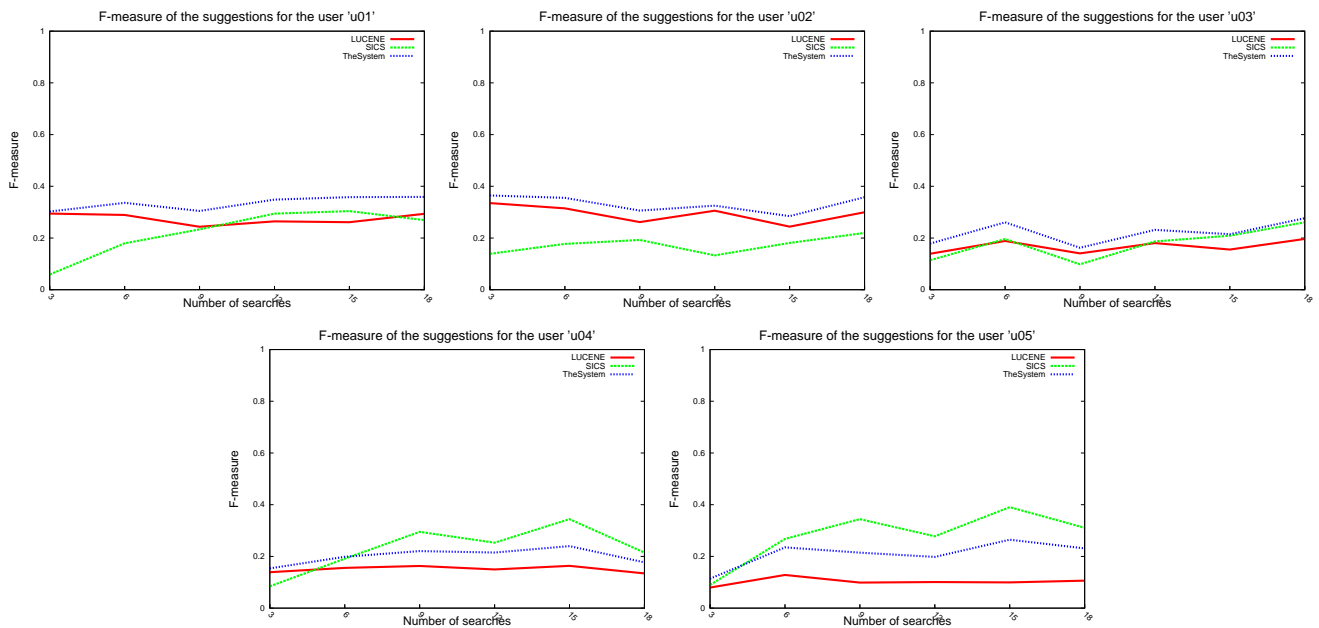


Figure 7: The F-measure of suggestions in the experiment

Progetti PAT, MOSTRO "Modeling Security and Trust Relationships within Organizations" and QUIEW (Quality-based indexing of the Web), art. 9, Legge Provinciale 3/2000, DGP n. 1587 dd. 09/07/04.

9. REFERENCES

- [1] H. Albin-Amiot, P. Cointe, Y.-G. Gueheneuc, and N. Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *16th Annual International Conference on Automated Software Engineering*, pages 166 – 173, 2001.
- [2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A pattern language*. Oxford University Press, 1977.
- [3] P. Baldi, P. Frasconi, and P. Smyth. *Modeling the Internet and the Web: Probabilistic Methods and Algorithms*. Wiley, 2003.
- [4] A. Birukou, E. Blanzieri, V. D'Andrea, P. Giorgini, N. Kokash, and A. Modena. IC-Service: A service-oriented approach to the development of recommendation systems. In *Proceedings of ACM Symposium on Applied Computing. Special Track on Web Technologies*, 2007.
- [5] E. Blanzieri, P. Giorgini, P. Massa, and S. Recla. Implicit culture for multi-agent interaction support. In *Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 27–39, 2001.
- [6] H. L. Dreyfus and S. E. Dreyfus. *Mind over machine: the power of human intuition and expertise in the era of the computer*. The Free Press, 2000.
- [7] S. Fincher. Plml: Pattern language markup language report of workshop held at CHI, Interfaces, 56 (pp. 26-28). Technical report, 2003.
- [8] S. Fox, K. Karnawat, M. Mydland, S. Dumais, and T. White. Evaluating implicit measures to improve web search. *ACM Trans. Inf. Syst.*, 23(2):147–168, 2005.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [10] M. Hafiz and R. E. Johnson. Security patterns and their classification schemes. Technical report, 2006.
- [11] D. C. Kung, H. Bhambhani, R. Shah, and G. Pancholi. An expert system for suggesting design patterns: a methodology and a prototype. In T. M. Khoshgoftaar, editor, *Software Engineering With Computational Intelligence*, Series in Engineering and Computer Science. Kluwer International, 2003.
- [12] J. Noble. Classifying relationships between object-oriented design patterns. In *Proceedings of the Australian Software Engineering Conference*, pages 98–107. IEEE Computer Society Press, 1998.
- [13] M. O'Cinnide and P. Nixon. Automated software evolution towards design patterns. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 162–165. ACM Press, 2001.
- [14] Removed_for_blind_review. Choosing the right desing pattern: the implicit culture approach. In *Proceedings of the fourth Industrial Simulation Conference 2006 (ISC-2006)*, pages 55–57. EUROESIS, June 2006.
- [15] L. Rising. *The Pattern Almanac*. Addison-Wesley Longman Publishing Co., Inc., 2000.
- [16] M. Schumacher. *Security Engineering with Patterns Origins, Theoretical Model, and New Applications*. Number 2754 in LNCS. Springer, 2003.
- [17] I. Sommerville. *Software engineering (7th ed.)*. Addison-Wesley, Boston, MA, USA, 2004.
- [18] L. A. Suchman. *Plans and Situated Action*. Cambridge University Press, 1987.