UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

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

CHOOSING THE RIGHT DESIGN PATTERN: AN IMPLICIT CULTURE APPROACH.

Aliaksandr Birukou, Enrico Blanzieri and Paolo Giorgini

February 2006

Technical Report # DIT-06-007

CHOOSING THE RIGHT DESIGN PATTERN: AN IMPLICIT CULTURE APPROACH

Aliaksandr Birukou
Enrico Blanzieri
Paolo Giorgini
Department of Information and Communication Technology
University of Trento
via Sommarive 14, 38050 Povo (Trento), Italy
e-mail: {aliaksandr.birukou, enrico.blanzieri, paolo.giorgini}@dit.unitn.it

KEYWORDS

Design patterns, implicit culture, design pattern selection, multi-agent system, information agents

ABSTRACT

Design patterns represent solutions to problems that have been proved to be useful in different contexts and can be reused. An experienced programmer can choose a suitable pattern for a given problem effectively. However, for an inexperienced programmer this is a very hard task. We propose a multi-agent system that supports programmers in choosing the design pattern suitable for the given problem. Personal agents in our system produce knowledge transfer among users, allowing for the reuse of experience in choosing design patterns.

INTRODUCTION

"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." This sentence is taken from (Gamma et al., 1995) and shows that selecting a design pattern was a problem even ten years ago. Since the number of proposed design patterns is continuously increasing, the selection problem becomes more and more difficult. For instance, Walter F. Tichy's catalogue of design patterns (Tichy, 2006) contains over 100 patterns. In particular, for an inexperienced programmer the problem of choosing the right design pattern is very hard and tools assisting in this process become of utmost importance.

Unfortunately, there is still a lack of systems that guide a programmer in the selection of design patterns. To the best of our knowledge, only one example of such a system was reported in the literature (Kung et al., 2003). Most of the current approaches, dealing with the patterns suppose that it is the programmer who makes the choice of the pattern (Albin-Amiot et al., 2001; Ó Cinnéide and Nixon, 2001).

In this paper, we address the problem of pattern selection and we propose an approach that considers the problem from a social point of view. We propose to use a multi-agent system based on the Implicit Culture framework to help programmers in selecting patterns. To help a person to make a decision about the pattern selection, getting suggestions from the group is important. In our system the problem faced by the programmer is compared with those faced previously by colleagues and suggestions about the most suitable design patterns are provided.

IMPLICIT CULTURE

This section presents an overview of the general idea of Implicit Culture and Systems for Implicit Culture Support (SICS). For a more thorough description we refer the reader to our previous work (Blanzieri and Giorgini, 2000; Blanzieri et al., 2001).

"Only experienced software engineers who have a deep knowledge of patterns can use them effectively. 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 pattern or need to develop a special-purpose solution." (Sommerville, 2004). The difference between two programmers is that the experienced programmer uses the implicit knowledge (in particularly that which is referred to as his/her experience) about the problem. Knowledge is called *implicit* when it is embodied in the capabilities and the abilities of the community members. It is *explicit* when it is possible to describe and share it through documents and/or information bases. In learning how to select suitable design patterns the inexperienced programmer faces the problem of acquiring the implicit knowledge of more experienced programmers.

We argue that it is possible to shift the behavior of inexperienced programmers in design patterns selection towards the behavior of experienced programmers by means of suggesting them patterns which are more suitable for the current design task. We call the behavior of experienced programmers related to the pattern se-

lection a *community culture*. When inexperienced programmers start behaving similarly to the community culture we can speak about knowledge transfer. In our architecture it is a SICS which performs this knowledge transfer. The relationship characterized by this knowledge transfer is called *"Implicit Culture"*.
For example, let us consider a programmer that needs to define an interface for creating an object, but let subclasses decide which class to instantiate. Let us suppose that for an experienced programmer the use of Factory Method design pattern in this case is obvious. If the system is able to use previous history to suggest the novice to use Factory Method pattern and he/she actually uses it, then it is possible to say that he/she behaves in accordance with the community culture and that the Implicit Culture relation is established.
The general architecture of SICS (Blanzieri et al., 2001) consists of the following three components:

- The *observer* is the part of SICS that stores in a database of observations information about actions executed by the user;
- The *inductive module* analyzes the stored observations and implements data mining techniques to infer a theory about actions executed in different situations;
- The *composer* exploits the information collected by the observer and analyzed by the inductive module in order to suggest actions in a given situation.

In terms of our problem domain, the observer saves information about the problem, which patterns were proposed as solution and which pattern was actually selected. The inductive module discovers problem-solution pairs by analyzing the history of users' interaction with the system. A set of problem-solution pairs is a *theory* and it shows which patterns are selected for what problems. The goal of the composer is to compare a new problem faced by the programmers with the problem part of the theory mined by the inductive module and to suggest the corresponding solution part. If this step fails, the composer just tries to match the problem with the solution by calculating similarity between its description and the descriptions of patterns.

PATTERN SELECTION

In this section we formalize the problem faced by a community member when choosing a design pattern satisfying some pre-specified requirements. In order to formalize the problem of pattern selections for a community of programmers we need to answer the following questions: How to describe patterns? How to describe requirements? How to match requirements with a pattern?

General Model

(Gamma et al., 1995) define the following four parts of a design pattern:

- The *name* serves as a reference to the pattern and allows for higher-level design.
- The *problem* describes where and when to apply the pattern. It can contain a list of conditions which should be satisfied to apply pattern.
- The *solution* contains the description of the design elements, their relations and functions.
- The *consequences* summarize the previous applications of the pattern and possible compromises.

The most interesting part for us is the *problem*, since it contains the description of the task solved by the pattern. The system should match this description with a description of the problem to be solved. We assume there exists semiformal or formal description of the problem faced by the programmer and of the problem solved by the pattern. In terms of Implicit Culture these problems constitute *situations*. The goal of SICS is to find the most similar situations, based on a similarity function. The *similarity function* takes two problem descriptions as arguments and returns a similarity value (let us assume it ranges from 0 — completely dissimilar to 1 — completely similar). In the next subsection we propose a concrete way of describing problems and a similarity function which we have chosen for the forthcoming implementation.
(Gamma et al., 1995) propose to describe a pattern using several sections. "Intent" section can be used in our problem description. According to (Gamma et al., 1995) "Intent contains a short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?"
Besides abstractions of situations we need to introduce several terms. We consider programmers as *agents* which execute *actions* on *objects*. We suppose actions have *attributes*, which are features that can be useful for the analysis of the actions. In Implicit Culture framework, actions are assumed to be executed in situations, therefore we can speak of *situated actions*. In our application, the SICS analyzes the actions presented in Table 1. Since all the actions are executed by programmers, we omit agents in the table. Every action is recorded being executed in a context of a certain project, characterized by an optional attribute *project_name*. We introduce this attribute, because project that has some specific requirements can influence the choice of patterns.
We explain the information contained in the table in detail. A programmer *requests* the system to find patterns that are suitable for implementation of his/her

Table 1: The actions that can be observed by the system

action	objects	attributes
request	problem_description	project_name
apply	pattern, problem_description	project_name
reject	pattern, problem_description	project_name

task. The request contains a description of the problem faced by the programmer.

A programmer *applies* the pattern when he/she implements it in the code. To observe this action we can either assume explicit feedback from the programmer or the existence of a case tool which serves for selection of the patterns and their subsequent (semi)automatic implementation. As an example of this tool we can mention the system that refactors existing code using a design pattern selected by a designer (Ó Cinnéide and Nixon, 2001).

The system should be able to observe that some patterns were offered to the user, but not selected. These patterns therefore can be considered as *rejected* as unsuitable (or not very suitable) for the current task. Alternatively, we can provide an opportunity to specify inapplicability of a pattern to the task explicitly, marking them as rejected only in this case.

Concrete Implementation

To specify the problem faced by the programmer and to describe design patterns, we propose to use a notion of a *precursor*: "[...]a precursor is a design structure that expresses the intent of a design pattern in a simple way, but that would not be regarded as an example of poor design. This is not a formal definition,[..]" (Ó Cinnéide and Nixon, 2001). In other words, we assume that there exists a textual description of the problem faced by the programmer and a textual description of the design pattern. The following precursor is proposed for the Factory Method pattern (Ó Cinnéide and Nixon, 2001): the Creator class must create an instance of the Product class. In case precursor is too specific we can specify the description more extensively, using e.g. *intent* of the pattern. As for Factory Method pattern, the intent is as follows: "Define an interface for creating an object, but let subclasses decide which class to instantiate. It lets a class defer instantiation to subclasses." (Gamma et al., 1995).

To illustrate the use of the textual description of a design problem we can imagine a programmer realizing that in some place in the code he/she needs to create an instance of a class without knowing its exact type. Thus he/she can specify his/her need like "a set of subclasses deal with their instantiation" and to request the system to find a suitable pattern.

In order to relate two descriptions we propose to use *"bag of words"* approach, also referred to as the *vector-space model* (Baldi et al., 2003). More precisely, a tex-

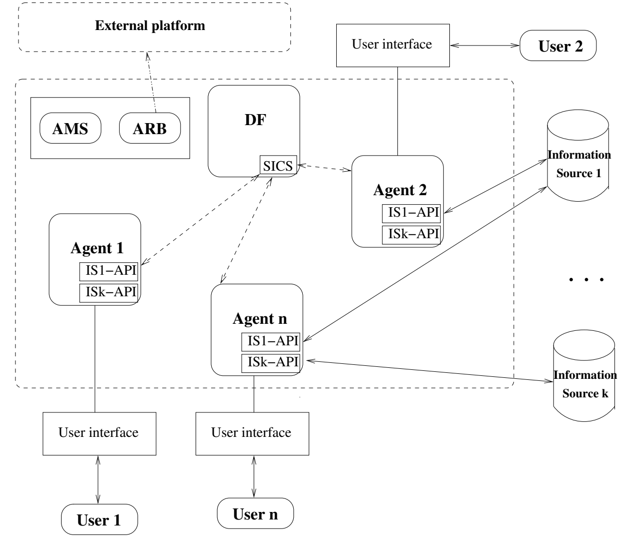

Figure 1: The architecture of the general Implicit Culture Multi-Agent platform.

tual description is represented as a sequence of terms, $d = (t_1, t_2, \ldots, t_n)$, where n is the length of the description and $t_i \in T,\ i = \overline{1,n}$. T is a vocabulary of terms, containing all the terms. The simplest vector-space representation is Boolean, i.e. x is $|T|$-dimensional vector which components $x_j \in \{0, 1\}$ indicate correspondingly the absence or the presence of the j-th vocabulary term in the description.

Having a vector-space representation of the problem description it is possible to use *cosine similarity* (Baldi et al., 2003) as a similarity function. This measure is simply the cosine of the angle formed by two vectors x and x' corresponding to two descriptions d and d'. It is calculated as follows:

$$\mathrm{sim}(d, d') = \frac{x^T x'}{||x|| \cdot ||x'||}.$$

APPLYING IMPLICIT CULTURE MULTI-AGENT PLATFORM TO PATTERN SELECTION

In this section we describe the architecture of the multi-agent system for design pattern selection.

We start from the description of a general Implicit Culture Multi-Agent platform. This platform was previously applied to the problem of web link recommendation for a community of users (Birukov et al., 2005) and to the scientific publication search (Birukou et al., 2006). The platform is implemented using Java Agent DEvelopment framework (JADE) (Bellifemine et al., 2001) which is a FIPA-compliant (FIPA, 2006) and widely accepted framework for the development of multi-agent systems.

The general architecture of Implicit Culture Multi-Agent platform is depicted in Figure 1 and it consists of

the following components:

- A *user* accesses the system using an interface on the client side. He/she submits requests to the system, specifying the description of the problem requiring the use of a design pattern.

- A *personal agent* is a software agent running on the server side. The task of the personal agent is to assist the user in choosing a suitable design pattern. It uses dedicated API to access several information sources, which contain information about design patterns. The information in these sources is selected according to the request submitted by the user. The agent contacts directory facilitator to receive recommendations about previous experience of the users, namely about actions that were taken in similar situations. Information about users' actions is also sent to the directory facilitator.

- A *Directory Facilitator (DF)* holds the list of services offered by agents and provides a set of agents that offer a specific service. In our case, it simply provides agents with the IDs of other personal agents. Moreover, in our system the DF also contains SICS module. This module receives users' requests redirected to the DF by personal agents and it sends back suggestions about patterns which seem to be suitable for a given problems, containing in the request. The SICS module processes past actions of the users in order to find a suitable pattern. The SICS module also receives information about users' actions, namely about their requests and (not) accepted patterns.

- By an *information source* we mean any database containing information about design patterns and offering some kind of API to access it.

- An *Agent Management System (AMS)* exerts a supervisory control over the platform. It provides the registration, the search, the deletion of agents and other services. It is an internal JADE agent running on every platform.

- An *Agent Resource Broker (ARB)* provides a link between the platform and other platforms. Using this link, the agents can propagate requests of their users to different platforms.

There is a difference in our platform with respect to the general Implicit Culture Multi-Agent platform: in our case SICS module is moved to the DF, while in the general architecture it is included in each personal agent. We redirected recommendation creation process to the DF because we do not see the need for SICS being distributed in the case all the users are working in the same company and the tasks of selecting a suitable pattern does not depend on a particular user. Rather, we assume that all the programmers are willing to share their experience and to store it in a common database.
A typical usage scenario is as follows: a user submits a request, expressing his/her design need. The personal agent contacts the DF. The DF stores the request action in the database of observations and uses SICS module in order to find similar descriptions of problems (situations) faced previously. The suggestions about patterns chosen in similar situations are sent to the personal agents. This list of patterns is shown to the user. Moreover, it is also possible to show descriptions of the discovered similar design problems, because it can help the user to make a decision. Finally, being able to see a list of similar problems faced by his/her colleagues, the programmer can just have a short talk with them in order to share the experience.

EVALUATION

Since we do not expect that specified requirements will match exactly one design pattern, it is necessary to show to the user as many potentially suitable patterns as possible (perhaps, even including some unsuitable ones). In information retrieval there exists the measure called *recall* which shows which fraction of the relevant items we retrieve in response to an information need (Baldi et al., 2003). In terms of choosing a design pattern it means that we show not only patterns which have exactly the same behavior as requested (which are not always available) but also the patterns which have comparable behavior. Moreover, having an alternative in the choice allows the programmer to make more qualitative decision, since looking at several patterns he/she can suddenly recognize that a pattern meets his/her requirements, including implicit ones, which were not expressed in the query.
We are currently working on the implementation of information sources which uses Apache Lucene (Lucene, 2006). Apache Lucene is a full-featured text search engine library. It is an open source java project. In our system we use it to create index of the patterns repository and to provide search of pattern descriptions using this index. Currently we have built a repository of 23 design patterns from (Gamma et al., 1995). Personal agents use API to access Lucene searching capabilities.
We have not performed the system evaluation with real users yet. However, in Birukov et al. (2005) we presented numerical results obtained using simulator developed for the application of Implicit Culture Multi-Agent Platform to web search. The aim of the experiment was to understand how adding a new user affects the relevance, in terms of precision and recall, of the links that were produced for users by SICS. In this experiment, the interaction between agents and users was replaced with the interaction between agents and user models that contain user profiles. User profile determined search keywords sequence and acceptance of the

results. The recall was among the measures we used to evaluate the quality of the suggestions.
The results have shown that the increase of the number of users causes the increase of the recall of the suggestions produced by personal agents. We think that the problem we are dealing with in this paper is much related to the problem of selecting web links relevant to keywords and that our approach will prove to be useful also for the selection of design patterns.

RELATED WORK AND DISCUSSION

The closest work is presented in the paper of (Kung et al., 2003). The authors propose a methodology for constructing expert systems which suggest design patterns to solve problems faced by designers. 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 suggested to the user. There are several significant differences between our approach and ESSDP. At first, ESSDP assumes the knowledge acquisition as the primary step of the methodology. In this step human experts fill in the system knowledge base with some pre-defined rules. Differently, in our system the SICS learns from the interaction with users, without any initial knowledge base. It allows for continuous improvements of suggestions. Moreover, we exploit interactions with inexperienced users as well, offering to novices patterns that were chosen in similar situations not only by experts but also by other novices. Thus we support sharing users' experience with others. At second, our architecture is not restricted to the use of rule-based knowledge base assuming that different learning techniques can be adopted in suggesting suitable patterns.
Several approaches were proposed to automate transformation of the old code to the new one which implements patterns. For instance, (Ó Cinnéide and Nixon, 2001) report on a methodology that allows changing the design of the program so as to make it amenable to the new requirements, without changing the behavior of the program. The automated tool support is provided to apply a selected pattern to the old code. However, the choice of the design pattern is still remaining with the designer. The same assumption is used in the PatternsBox tool presented in (Albin-Amiot et al., 2001). This tool assists in choosing patterns only by providing access to the design pattern repository where each pattern is annotated with a shortcut.
Our approach can be used in tools similar to the two mentioned at the stage of selecting the pattern: a programmer can be provided with suggestions about patterns used in similar situations previously.
For the description of design problems/design patterns in our system it could be useful to adopt a formal framework, such as non-functional requirements (NRM) framework (Gross and Yu, 2001). A semiformal code representation of the design problem, e.g. class diagrams, activity diagrams, etc. could also be adopted. We also think that the problem we address is related to the problem of web service composition, where in order to solve a sub-problem it is necessary to find a web service which suits some design need (Lazovik et al., 2006). The comparison of the requirements with the service description seems to be similar to the comparison of the requirements with the pattern description.

CONCLUSION

The system that helps programmers in choosing design patterns suitable for a given task is considered. The system takes into account the social part of the problem, providing users with suggestions from other community members about patterns that were used to solve similar problems.
As future work we would like to conduct several experiments with real users.

REFERENCES

Albin-Amiot, H., P. Cointe, Y.-G. Gueheneuc, and N. Jussien. 2001, November. Instantiating and detecting design patterns: Putting bits and pieces together. In *16th Annual International Conference on Automated Software Engineering (ASE)*, 166 – 173.

Baldi, P., P. Frasconi, and P. Smyth. 2003. *Modeling the internet and the web: Probabilistic methods and algorithms*. Wiley.

Bellifemine, F., A. Poggi, and G. Rimassa. 2001. Developing multi-agent systems with a fipa-compliant agent framework. *Software - Practice and Experience* 31 (2): 103–128.

Birukou, A., E. Blanzieri, and P. Giorgini. 2006. A multi-agent system that facilitates scientific publications search. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems. To appear.*

Birukov, A., E. Blanzieri, and P. Giorgini. 2005. Implicit: An agent-based recommendation system for web search. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, 618–624: ACM Press.

Blanzieri, E., and P. Giorgini. 2000. From collaborative filtering to implicit culture: a general agent-based framework. In *Proceedings of the Workshop on Agents and Recommender Systems*. Barcellona.

Blanzieri, E., P. Giorgini, P. Massa, and S. Recla. 2001. Implicit culture for multi-agent interaction support. In *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, ed. C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, Volume 2172 of *Lecture Notes in Computer Science (LNCS)*, 27–39. London, UK: Springer-Verlag.

FIPA 2006. Foundation for intelligent physical agents. `http://www.fipa.org/`.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Gross, D., and E. S. K. Yu. 2001. From non-functional requirements to design through patterns. *Requirements Engineering* 6 (1): 18–36.

Kung, D. C., H. Bhambhani, R. Shah, and G. Pancholi. 2003. An expert system for suggesting design patterns: a methodology and a prototype. In *Software Engineering With Computational Intelligence*, ed. T. M. Khoshgoftaar, Volume 731 of *The International Series in Engineering and Computer Science*, 376. Kluwer International.

Lazovik, A., M. Aiello, and M. Papazoglou. 2006. Planning and monitoring the execution of web service requests. *Journal on Digital Libraries*. To appear.

Lucene 2006. The apache lucene project. `http://lucene.apache.org/`.

Ó Cinnéide, M., and P. Nixon. 2001. Automated software evolution towards design patterns. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, 162–165. New York, NY, USA: ACM Press.

Sommerville, I. 2004. *Software engineering (7th ed.)*. Boston, MA, USA: Addison-Wesley.

Tichy, W. F. 2006. Essential software design patterns. `http://wwwipd.ira.uka.de/tichy/patterns/overview.html`.

BIOGRAPHY

ALIAKSANDR BIRUKOU is currently a PhD candidate at the University of Trento, Italy. He received degree with distinction in Applied Mathematics and Computer Science from the Belarusian State University, Minsk, Belarus in 2002. His current research interests are in Data Mining, Multi-Agent Systems and Recommendation Systems. Previously, he worked and carried out research on Queueing Systems publishing about 10 scientific papers on the topic.

ENRICO BLANZIERI is currently Assistant Professor at the University of Trento, Italy where, since 2002 he is within the Faculty of Engineering. Since 2004 he is the coordinator of the Data Mining and Learning Systems research program at the Department of Information and Communication Technology. Between 2000 - 2002 he worked as researcher at the Faculty of Psychology of the University of Turin. In 1997 - 2000 he was researcher at ITC-IRST of Trento. He received a laurea con lode in Electronic Engineering from the University of Bologna, Italy in 1992 and a PhD in Cognitive Science in a joint program between Polytechnic and University of Turin, Italy in 1998. His present major scientific interests are Data Mining, Machine Learning and Bioinformatics, and in the past he devoted attention to Statistical Reasoning and Cognitive Science. His research focuses on Instance-Based Learning techniques such as Nearest Neighbour, Radial Basis Function Networks and more recently SVMs. He also contributes to the application of Machine Learning techniques to various fields. He published more than 40 scientific publications in journals and referred conferences. He is a member of the scientific committee at the International and European conferences of Case-Based Reasoning.

PAOLO GIORGINI is researcher at University of Trento. He received his Ph.D. degree from Computer Science Institute of University of Ancona (Italy) in 1998. Between March and October 1998 he worked at University of Macerata and University of Ancona as research assistant. In November 1998 he joined the Mechanized Reasoning Group (MRG) at University of Trento as posdoc researcher. In December 1998 he was researcher visiting at the Computer Science Department of University of Toronto (Canada) and more recently he was visiting professor at the Software Engineering Department of University of Technology in Sydney. He has worked on the development of requirements and design languages for agent-based systems, and the application of knowledge representation techniques to software repositories and software development. He is one of the founder of Tropos, an agent-based oriented software engineering methodology. His publication list includes more than 100 refereed journal and conference proceedings papers and five edited books. He has contributed to the organization of international conferences as chair and program committee member, such as CoopIS, ER, CAiSE, AAMAS, EUMAS, AOSE, AOIS, and ISWC and he is Co-editor in Chief of the International Journal of Agent-Oriented Software Engineering (IJAOSE).