Leszek A. Maciaszek, Bruc Lee Liong

Macquarie University, Sydney, Australia

MINING OF KNOWLEDGE IN OBJECT-ORIENTED SOFTWARE

(invited paper)

1. Introduction

Enterprise information systems (EIS) are focal points of all functional, operational and managerial areas of any enterprise. They are a critical ***resource*** for everyday business activities and for maintaining a competitive position in business. EIS-s represent the ***business and organizational knowledge***.

Knowledge is usually defined as "***know-how***" – an intellectual capital accumulated through experience. As noted by Rus and Lindvall (2002, p.26) – "The major problem with intellectual capital is that it has legs and walks home every day. At the same rate experience walks out the door, inexperience walks in the door. Whether or not many software organizations admit it, they face the challenge of sustaining the level of competence needed to win contracts and fulfil undertakings."

To sustain intellectual capital present in an EIS, the know-how has to be managed. Effectively, ***knowledge management*** has to be engaged to help organizations discover, organize, distribute and apply the knowledge encoded in information systems. ***Data mining*** is an area of knowledge management concerned with exploratory data analysis to discover relationships and patterns (perhaps previously unknown or forgotten) that can (re-)discover knowledge and can support decision making.

The main objectives of data mining are (Oz, 2004, p.343):

- *Path analysis* – finding patterns where one event leads to another later event.
- *Classification* – finding if certain facts fall into predefined groups.
- *Clustering* – finding groups of facts not previously known.

- *Forecasting* – discovering patterns in data that can lead to reasonable predictions.

When applied to the know-how present in EIS-s, data mining is the process of exploring, modeling, and evaluating software code (rather than large data sources). The ***exploration*** stage aims at discovering how the code is structured and how its components collaborate. The ***modeling*** stage aims at representing the code in an abstract language that can facilitate the understanding and maintenance of discovered knowledge. The ***evaluation*** stage aims at assessing the overall supportability aspect of discovered knowledge and at refactoring the code to achieve the required supportability level.

The point of *supportability* is critical. It sets the target for the mining of knowledge in the software and for the successive management of that knowledge. Supportability in software engineering combines three software characteristics – understandability, maintainability, and scalability (Maciaszek and Liong, 2005). Only supportable software is a viable information system and an interesting knowledge resource.

A necessary condition for a supportable EIS is its conformance to a *layered architectural design* that reduces system complexity by minimizing component *dependencies*. A supportable system enables knowledge transfer and keeping intellectual capital "in the door".

The remainder of the paper is structured as follows. Next section addresses the knowledge exploration phase. The input to exploration is any Java program. The output is the discovery if and how that Java program conforms to a supportable architectural design. Section 3 addresses the knowledge modeling phase. It shows how code visualization facilitates program understanding and how it reveals additional code dependencies. Section 4 explains how knowledge evaluation phase can improve the supportability quality of software. The Summary and Conclusion section explains once more why mining of knowledge present in object-oriented software is of paramount importance for businesses and why the very existence of many organizations depend on capturing and transferring knowledge encoded in EIS-s.

2. Knowledge Exploration

Mining of software knowledge begins with, what is called here, the code exploration stage. In fact, code exploration is where the real mining is done. The next two stages model and evaluate the "mined" knowledge.

Code exploration draws its techniques from the related areas of reverse engineering, code inspection and code profiling. However, code exploration, as addressed in this paper, is a "value-added" technique in the sense that it aims at analysing the code with regard to its supportability requirement.

Code exploration is given as input a supportable architectural meta-model and its mining activities are architecture-aware. Consequently, the research presented in this paper is only interested in mining software knowledge in information systems originally built to satisfy a given ***meta-architecture*** (Maciaszek, 2005).

Because virtually all new enterprise information system are ***object-oriented***, code exploration targets the object-oriented code and the object-oriented meta-architecture. Mining legacy systems is not considered in this paper.

Unlike procedural code in Cobol-style legacy systems, the object oriented code is significantly more difficult to explore because of its *different structural and execution models*. That is, the static compile-time code is not sufficient to understand all object interactions. Some object interactions "happen" dynamically at run-time and are not visible in the static code. Moreover, the dynamic (run-time) view of OO code changes for different runs.

It is one of the tasks of a meta-architecture to ensure that dynamic object collaborations are legitimised (as much as possible) in the static code by means of establishing explicit associations between collaborating objects (so called Explicit Association Principle (EAP) – Maciaszek and Liong, 2005). The EAP is reminiscent of what in knowledge management is known as the transformation of *tacit knowledge* into an *explicit (codified) knowledge*.

A related difficulty of exploring an object-oriented code is the code ***delocalisation*** (Dunsmore, Roper and Wood, 2003). Delocalisation is a code characteristic that a closely related functionality is widely distributed throughout the code. Accordingly, the code exploration requires following the trail of method invocations and complicated traversals through inheritance hierarchies, polymorphic invocations, provided and realized interfaces, class libraries, etc.

To fully explore the code, there is a need to perform careful parsing of its *tacit knowledge*. This is performed in multiple stages by our mining tool called Design Quantifier, DQ (Maciaszek and Liong, 2003). DQ is a kind of ***software inspection tool*** and uses typical code analysis and inspection techniques (Devanbu, 1999; Anderson et al., 2003).

However, DQ is calibrated for discovering if the software conforms to architectural design and for measuring the supportability of the software. In this sense DQ relates to the work on architectural localization and extraction (Jerding and Rugaber, 2000), on architecture-level dependence analysis (Stafford and Wolf, 2001), and on object-oriented design patterns recovery (Antoniol *et al.*, 2001; Huang *et al.*, 2005).

DQ accepts the definition of a meta-architecture as its input and it assumes that the meta-architecture enforces supportability principles and patterns (such as the PCMEF meta-architecture described in Maciaszek and Liong (2005)). DQ can then analyze the code to determine the components into which the system is broken and the ways in which components communicate. Communication paths define dependencies between components. Dependencies that violate the meta-

architecture are pinpointed as leading to unsupportable code. As the aim is to minimize dependencies in software, the dependencies are measured according to various metrics.

Figure 1 shows the phases of code analysis and inspection performed by DQ (Maciaszek and Liong, 2003). Parsing of class declarations discovers explicit knowledge of associations between classes. Parsing of method signatures and local variables further discovers the implicit intention of associations. The analysis of function calls establishes the real dependencies. It indicates any runtime dependencies not legitimized in static program structures. It also points to delocalisation in the code.

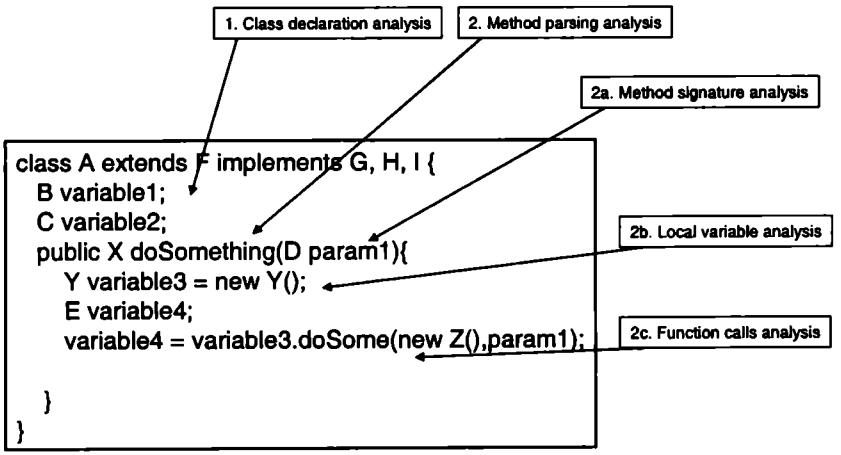

Fig. 1. Phases in object exploration

Parsing is performed via customized byte code engineering tool, ***BCEL*** (BCEL, 2005). Statistics as well as information regarding the class and its relationships with other classes are collected in various parsing phases. Each parsing phase unravels different information about the class and its role in the system.

BCEL has an advantage over standard methods, such as Java ***reflection***, which reveal only class metadata such as class signature, method signature, parameters expected for the method, and class variables. Reflection does not reveal local variables in methods or method invocations performed in each method.

BCEL parses the byte code to recover the class information including the information recoverable by Java reflection and, furthermore, it recovers method invocations and local variables. Ability to analyze method invocations is important for discovering certain features or weaknesses of the system, in particular, hidden dependencies.

Knowledge exploration starts with ***class declaration analysis***. The class declaration shows dependencies between the current class and its inherited classes

and implemented interfaces. Moreover, variables declared as part of the class reveal dependencies of the current class to those declared variable types.

Consider the class declaration in Figure 2. There are a number of different types of dependencies derivable from that declaration, such as extend dependency, implementation dependency, and instantiation dependency. Class D extends A and it, therefore, depends on A (it needs A because it inherits from it). Class D depends on interfaces B and C because it makes the promise to its clients that it will implement both interfaces (D can be used anywhere in the program to substitute for B or C). Finally D depends on both E and F as these are declared as the types of variables needed for class D to be fully functional. Note that all this information can be gathered from Java reflection as well as by BCEL.

```
public class D extends A implements B, C{
E varE;
F varF;
}
```

Fig. 2. Class declaration

The phase of ***method parsing analysis*** examines the declaration and content of each method in the class. This phase is performed in three sub-phases. The first is to analyse the *method's signature*. Information gathered in this sub-phase adds extra classes to the list of dependencies with the following information: method parameter types, method return type, method's declared exception type. This subphase can be analysed via Java reflection as well as by BCEL.

The next sub-phase is to analyse *local variable* declarations. Similarly to class variables, local variables indicate dependencies from the current class to the declared classes. The program can use the local variables to establish hidden dependencies in delocalised classes. This usually signifies a breach of the meta-architecture. Java reflection mechanism is unable to reveal such hidden dependencies.

The third sub-phase of method parsing involves *function calls*. A method call may reveal object types not previously registered as the class dependencies. This is another category of hidden dependencies, which – in extreme cases – can even create a class without revealing any dependency in the static code.

Another purpose of function calls analysis is to discover execution paths that lead to runtime behavior. This reveals further information such as method delegation, method composition, abidance to de Morgan's law, etc. Function calls analysis will also reveal information important for enforcing architectural conformity.

3. Knowledge Modeling

Tools such as DQ or SA4J (SA4J, 2005) provide means to model the mined knowledge. UML has been used to model such knowledge as it provides rich set of

abstractions and notations. For example, dependencies discovered in the ***class declaration*** in Figure 2 can be visually modeled as in Figure 3.

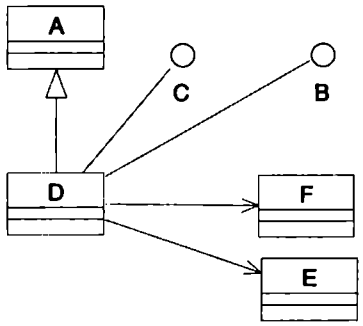

Fig. 3. Class dependencies

Figure 4 is an example showing how ***method parsing*** analysis can lead to a visual model represented as a sequence diagram. The model shows how `function1()` contains invocation to `function2()` from `paramB` and supplying two parameters. The first parameter is an object `D` from the result of invoking `getD()` method from `paramC`. The second parameter is its own class. The `getA()` method is further invoked from the result of `function2()`, expected to be of type `A` as indicated by the return type of `function1()`.

```
A function1(B paramB, C paramC){
  returnparamB.function2(paramC.getD(),this).getA();
}
```

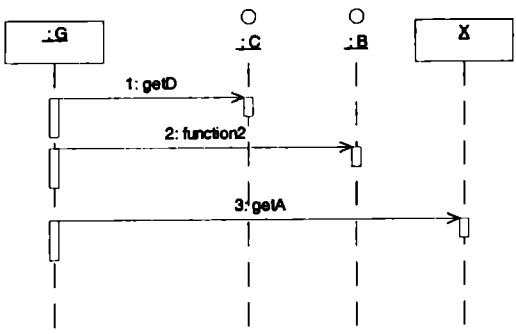

Fig. 4. Method dependencies

The implementation of `function1()` reveals that the class `A` depends on `B` and `C`. The success of `function1()` is determined by the existence, result, some knowledge of `function2()`, `getD()`, and `getA()` methods. In particular, it

depends on `function2()` returning an object with a method `getA()`, which will return an object of class A. It depends on A merely for A's existence since it does not seem to use A's functionalities at all.

Another not easily noticeable fact is that there is a local variable declared implicitly as the result of `function2()` call. This variable is a placeholder for `getA()` function call. This means that the class also depends on this particular placeholder class, whatever the class is, since a method `getA()` is assumed from it.

All in all, there is lots of interesting information that can be discovered from the code snippet in Figure 4. The fact that C has a method called `getD()`, presumably of type D, indicates that C also depends on D. Knowledge like that is critical for assessing the impact of propagation of changes in the system. Figure 5 is the knowledge model obtained from the method parsing analysis applied to `function1()` in Figure 4.

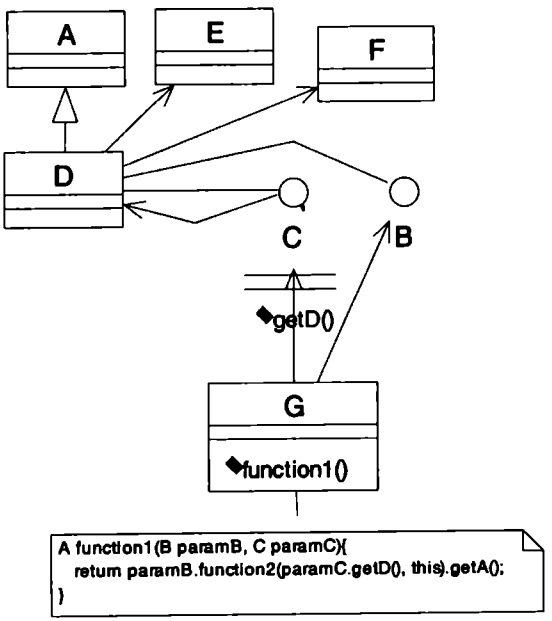

Fig. 5: Knowledge model corresponding to method dependencies in Figure 4

4. Knowledge Evaluation

It is the task of developers to ensure that the developed software follows the architectural design, which in turn conforms to a chosen meta-architecture. However, the ***evaluation*** of the explored knowledge represented in knowledge models is likely to reveal that most software does not follow the intended meta-architecture.

With regard to the knowledge model in Figure 5, method parsing analysis clearly revealed some suspect dependencies. For example, there is a circular dependency in the implemented code – interface C depends on class D and yet class D implements C. *Refactoring* is advisable on such circular dependencies as discussed in Maciaszek and Liong (2005).

Knowledge models for software allow better understanding of software properties and its functionality, thus leading to refactorings that result in supportable systems. For example, it may help to realize that the code as presented in Figure 4 could further be improved by considering implementation of façade pattern (Maciaszek and Liong, 2005).

Knowledge evaluation is about determining the *quality* of the software. The quality addressed in this paper is system supportability. *Metrics* have been proposed to measure system supportability. Supportability metrics have been discussed in Maciaszek and Liong (2003) and this work is continuing.

DQ supports knowledge evaluation by generating UML notes that contain metric values (Maciaszek and Liong, 2003; Maciaszek, 2005). These notes are attached to corresponding classes in the diagram, as shown in Figure 6. The model shows (in UML notes) the cumulative metrics for the classes. The metrics are: CCD – cumulative class dependency; CMD – cumulative message dependency, and CED – cumulative event dependency.

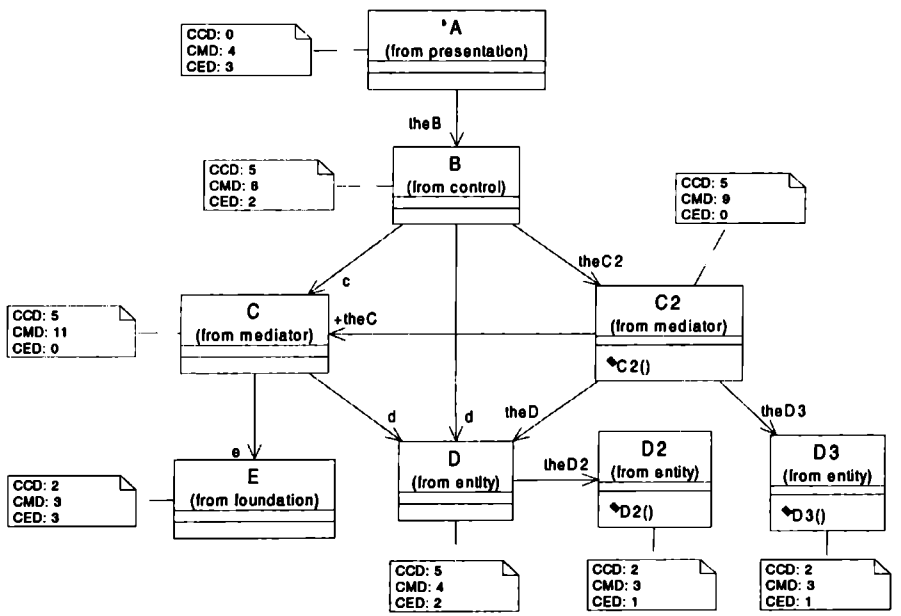

Fig. 6. Knowledge model with upportability metrics

Although not currently supported by DQ, tools like DQ should be able to visualize dependencies by producing *call graphs*. Ideally, a call graph could be a variant of a UML sequence diagram. A call graph can be used for the change impact analysis and to answer "what-if" questions such as "which methods are affected if a particular method is modified?". Figure 7 shows the SA4J's "what-if" analysis performed on the same Java program that was mined in order to obtain the knowledge model in Figure 6.

Knowledge evaluation supports *learning and improvement*. As discussed by Rus and Lindvall (2002), evaluation based on metrics can lead to predictive models that can help to guide decision making for future projects based on past experiences. Software evaluation with metrics defines a knowledge management strategy. Applying induction, generalization and abstraction on this knowledge can generate new knowledge.

5. Summary and Conclusion

This paper started with the conjecture that information systems embody most of *business and organizational knowledge* of any enterprise. Much of the knowledge that underpins everyday business processes and decides about competitive advantages and the overall future of an enterprise is hidden in the code rather than present in the minds of employees. The consequence must be that many large enterprises have no sufficient "*know-how*" of how their business is done.

Humans have no option but to search for the "know-how' in the programming code, frequently legacy code. However, this paper is about mining of knowledge in *modern object-oriented software*. Our point is that newly developed object-oriented software can be as unsupportable as any thirty years old COBOL program. Worse – the source code of a COBOL program explains how it works and what it does; the source code of a C# or Java program may say little about the runtime behaviour of the code. Hence, this paper is also about how to make the object-oriented code "*mine-able*"

To be mine-able, the object-oriented code must be written with *supportability* (understandability, maintainability, and scalability) in mind. To be supportable, the *architectural design* of the code must conform to the supportability-enforcing meta-architecture. Hence, an overriding objective of the ***knowledge exploration*** stage is to determine if the code supports the architectural design and is, therefore, supportable. Mining unsupportable code is a misguided effort.

The aim of software mining is to make the implicit implementation knowledge into an explicit subject matter knowledge that humans can communicate to each other. Communications requires a language and a set of abstractions that permits communications at a desired level of detail. The UML delivers necessary abstractions and communication ability. During the ***knowledge modeling*** stage, the explored knowledge is modelled in UML to formalize knowledge of the subject matter. The models represent the business and organizational "memory"

implemented in software. They also define the current (implemented) architectural design for the system.

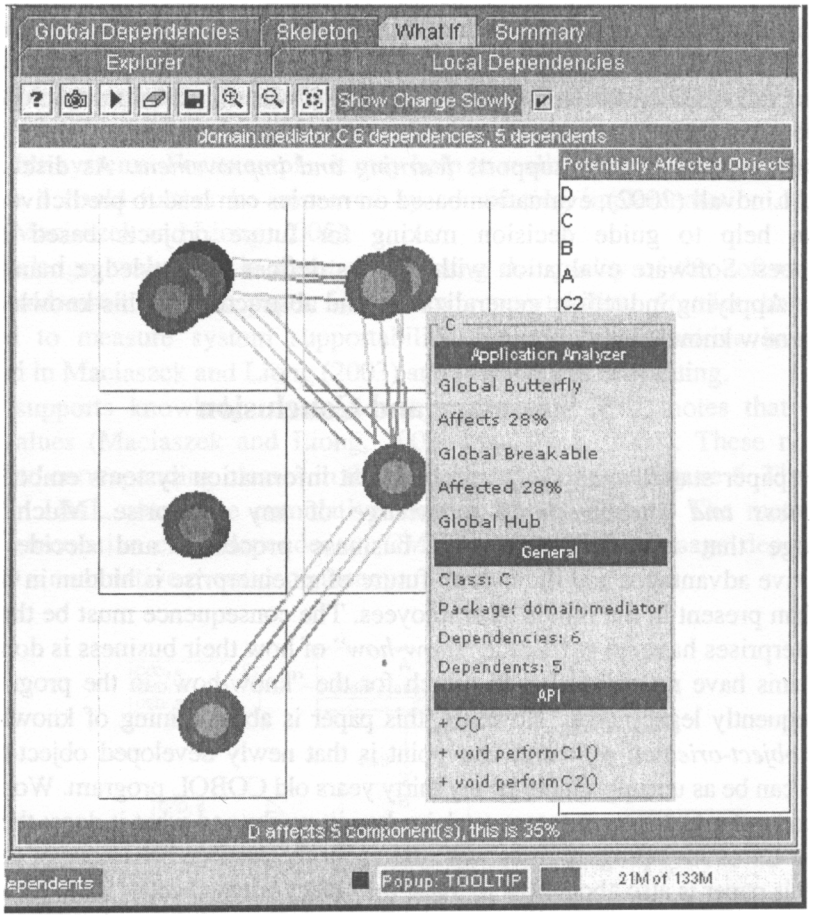

Fig. 7. „What-if" visualization

Being about software mining, this paper starts with knowledge exploration in the existing code and it accepts that the original developments models and the code get out of sync. So, it is not just sufficient to create models from the implementation, but these models must be evaluated for supportability. Hence, during the ***knowledge-evaluation*** stage, the discovered models (and the current architectural design) are checked if they conform to the meta-architecture. If they do then the system is still supportable. But because there are different levels of supportability, the knowledge evaluation stage attempts also to measure the supportability. To re-introduce supportability to a system found to be unsupportable, or to improve supportability level, various refactorings can be applied on the models.

Like in the case of *data mining*, the focus of *software mining* is to reveal information that is unknown, hidden or unexpected but that is actionable and can be used for business decisions. In case of software mining, the obtained knowledge serves the business decisions with relation to the understanding, maintenance and scalability of EIS-s. Such decisions are of fundamental importance in today's information and knowledge society.

Note that the approach presented in this paper creates models that become assets, not just costs (as it is the case in most contemporary software development). Interestingly, this "***models-as-assets***" approach underpins another interesting trend in software production – the trend standardized as MDA (Model Driven Architecture) (Mellor *et al.*, 2004). The MDA raises the abstraction bar even higher. In the MDA, all knowledge of an information system is formalized in the modeling language and the models are software-independent. A model compiler is used to generate all the implementation source code, which is then normally compiled to the executable code. Under the MDA philosophy, the knowledge exploration, modeling, and evaluation stages would all apply to the models, never to the code.

Literatura

[1] Anderson P., Reps T. and Teitelbaum T.: Design and Implementation of a Fine-Grained Software Inspection Tool, "IEEE Trans. on Soft. Eng.", 2003, Vol. 29, No.8, pp.721-733.

[2] Antoniol G., Casazza G., Di Penta M. and Fiutem R.: Object-Oriented Design Patterns Recovery, "The J. of Syst. and Soft.", 2001, 59, pp.181-196.

[3] BCEL: Byte Code Engineering library, http://jakarta.apache.org/bcel/ (last accessed Feb. 2005)

[4] Devanpu P.: GENOA – A Customizable, Front-End-Retargetable Source Code Analysis Framework, "ACM Trans. on Soft. Eng. and Methodology", 1999, Vol. 8, No.2, pp.177-212.

[5] Dunsmore A., Roper M. and Wood M.: Practical Code Inspection techniques for Object-Oriented Systems: An Experimental Comparison, "IEEE Soft.", 2003, July/August, pp.21-29.

[6] Huang H., Zhang S., Cao J. and Duan Y.: A Practical Pattern Recovery Approach Based on both Structural and Behavioral Analysis, "The J. of Syst. and Soft.", 2005, 75, pp.68-87.

[7] Jerding D. and Rugaber S.: Using Visualization for Architectural Localization and Extraction, "Science of Comp. Programming", 2000, 36, pp.267-284.

[8] Maciaszek L. and Liong B.L.: Designing Measurably-Supportable Systems, "Advanced Information Technologies for Management", Research Papers No 986, ed. by E.,Niedzielska, H. Dudycz, M. Dyczkowski, Wroclaw University of Economics, 2003, pp.120-149.

[9] Maciaszek L. and Liong B.L.: "Practical Software Engineering. A Case Study Approach", Addison-Wesley, 2005, 829p.

[10] Maciaszek L.A.: Roundtrip Architectural Modeling, "Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005)", Newcastle, Australia, eds. S. Hartmann and M.Stumper, Australian Computer Science Communications, 2005, Vol. 27, No. 6, pp.17-23. (invited paper)

[11] Mellor S., Scott K., Uhl A. and Weise, D.: "MDA Distilled. Principles of Model-Driven Architecture", Addison-Wesley 2004, 150p.

[12] Oz E.: Management Information Systems, 4th ed., Thomson, 2004, 756p.

[13] Rus I. and Lindvall M.: Knowledge Management in Software Engineering, "IEEE Soft.", 2002, May/June, pp.26-38.
[14] SA4J: Structural Analysis for Java, http://www.alphaworks.ibm.com/tech/sa4j (last accessed Fe. 2005)
[15] Stafford J. and Wolf A.: Architecture-Level Dependence Analysis for Software Systems, "Int. J. on Soft. Eng. and Knowledge Eng.", 2001, Vol. 11, No. 4, pp.431-451..

DRĄŻENIE WIEDZY W OPROGRAMOWANIU OBIEKTOWO-ZORIENTOWANYM

Streszczenie

Punktem wyjściowym artykułu jest domniemanie, że systemy informacyjne zawierają większość *organizacyjnej i biznesowej wiedzy* przedsiębiorstwa. Przeważająca część tej wiedzy, wykorzystywanej w codziennie realizowanych procesach biznesowych i decydującej o przewadze danej firmy nad konkurencją i jej przyszłości, jest raczej ukryta w oprogramowaniu niż w umysłach jej pracowników. Częściowo jest za to odpowiedzialny współczesny dynamiczny rynek pracy, w którym pracownicy, zmieniając pracodawców, pozbawiają ich kapitału intelektualnego zdobytego przez lata pracy. W konsekwencji wiele dużych firm nie ma wystarczającej wiedzy o tym w jaki sposób one funkcjonują.

Nie ma zatem wyboru: wiedzy tej należy szukać w kodach programowych, najczęściej w systemach informatycznych. Artykuł ten jest jednakowoż o drążeniu wiedzy we *współczesnym oprogramowaniu obiektowo-zorientowanym*. Naszym zdaniem, współczesne oprogramowanie (także obiektowo-zorientowane) może być w takim samym stopniu „nieznośne" jak 30 letnie programy pisane w COBOLu. Nawet gorzej – kody źródłowe w COBOLu wyjaśniały sposób działania podczas gdy programy w C# lub Javie mogą niewiele mówić o metodach zachowania się kodu. Zatem artykuł ten jest o tym jak sprawić, żeby programy obiektowo-zorientowane uczynić podatne na wydobywanie z nich wiedzy (*mine-able*).

Aby z takich programów można było wydobyć wiedzę – powinny one być projektowane z takim zamysłem (tzn. muszą być zrozumiałe, łatwe do utrzymania i skalowalne). W tym celu *projekt architektury* programu musi być zgodny z pewną wymuszoną meta-architekturą. Zatem, nadrzędnym celem w fazie ***odkrywania wiedzy*** jest ustalenie czy kod programowy wspomaga taki projekt architektury i czy jest w konsekwencji podatny na wydobywanie wiedzy. W przeciwnym razie drążenie jest chybionym zabiegiem.

Celem drążenia oprogramowania jest przekształcenie wiedzy ukrytej zawartej w kodach programowych na wiedzę przedmiotową, która może być przekazywana do użytkowników. Komunikacja międzyludzka wymaga odpowiedniego języka i zbioru pojęć, pozwalających na wymianę informacji na pożądanym poziomie szczegółowości. Takim językiem jest UML, który dostarcza pojęć i niezbędnych metod (zdolności) komunikacyjnych. W czasie fazy ***modelowania wiedzy,*** odkrywana wiedza jest formalizowana za pośrednictwem języka UML przybierając postać odpowiednich modeli wiedzy dziedzinowej. Modele reprezentują biznesową i organizacyjną "pamięć" zaimplementowaną w oprogramowaniu. Zarazem stanowią projekty architektury docelowego systemu.

Reprezentując tematykę drążenia oprogramowania, artykuł zaczyna od odkrywania wiedzy w ramach istniejących kodów, akceptując zarazem oryginalne modele budowy i otrzymywane z nich programy. Zatem nie jest wystarczające tworzenie modeli z opracowanych wcześniej ale powinny być one oceniane pod kątem ich przydatności z punktu widzenia drążenia wiedzy. W fazie ***oceny***

wiedzy odkrywane modele (i projekty architektury zarazem) są sprawdzane pod kątem ich zgodności z meta-architekturą. Jeśli są zgodne to system jest uznawany za podatny na odkrywanie wiedzy. Jednakże z uwagi na różne poziomy „podatności" faza oceny wiedzy wymaga wprowadzenia miar podatności. Aby uczynić z systemu wersję „podatną" na odkrywanie wiedzy lub podnieść poziom „podatności" do istniejących modeli można wprowadzić różne zabiegi typowe dla re-factoringu.

Podobnie jak w przypadku *drążenia danych* – celem *drążenia oprogramowania* jest odsłonięcie informacji nieznanych, ukrytych i niespodziewanych ale zarazem użytecznych w decyzyjnych procesach biznesowych. Uzyskiwana w ten sposób wiedza obsługuje decyzje biznesowe odnoszące się do zrozumienia, utrzymania i skalowalności systemów typu EIS (Executive Information Systems).Takie decyzje mają fundamentalne znaczenie we współczesnym społeczeństwie informacji i wiedzy.

Należy podkreślić, że w podejściu prezentowanym w tym artykule wykorzystuje się modele, które stają się wartością a nie tylko wywołują koszty (jak w przypadku większości tworzonego współcześnie oprogramowania). Interesujące jest, że podejście ***„modele jako wartości"*** wspiera inną ciekawą tendencję produkcji oprogramowania – trend MDA (*Model Driven Architecture)* zaliczony do standardów (Mellor *et al.*, 2004). Model MDA jest traktowany jako koncepcja wyżej. W modelu MDA cała wiedza systemu informacyjnego jest formalizowana w języku modelowania a powstające modele są programowo niezależne. Kompilator modelu jest używany do generowania wszystkich implementowanych kodów źródłowych, które dalej są kompilowane do postaci wykonywalnej. Według filozofii MDA poszczególne fazy odkrywania, modelowania i oceny wiedzy powinny być do modeli a nie do kodów programowych.