TOWARDS A CLASSIFICATION SCHEME IN ORTHOGONAL DIMENSIONS OF REUSABILITY

Markus Aulkemeier, Jürgen Heine, Emilio G. Roselló, Jacinto G. Dacosta
Departmento Informática, Edificio Fundición, Universidad de Vigo, Campus Lagoas-Marcosende, 36310 Vigo, Spain

J. Baltasar García Perez-Scholfield
Departamento de Informática, E.S.E.I., Universidad de Vigo, Campus As Lagoas, 32004 Ourense, Spain

Keywords: Classification, reuse, independence, contract, composition.

Abstract: The reuse of existing bits of code has emerged as a habitual practice in software engineering. Despite the lively interest that has been directed towards this field, the major part of existent literature and publications is based on concrete aspects and models of reuse what provides a fragmented and compartmentalized vision of this domain. No holistic and unifying proposal exists that sorts the reuse domain as a conceptual software characteristic in a comprehensive way. Related to this context, the present work contributes a three-dimensional sorting model for reusable software artefacts. The three dimensions are independence, contract specification and composition, identified as fundamental dimensions of reusable software artefacts.

1 INTRODUCTION

Code reuse appears, rather than architecture or design reuse, reuse through evolution of programming languages or any other type of reuse like data reuse or code generators, as one of the principle forms of reuse (Krueger, 1992; Prieto-Díaz, 1993). This category implies the reuse of software pieces or artefacts that contain some kind of code, without changing their internal implementation (e.g. the reuse of libraries of functions, classes or components). Thus, this article addresses the reuse of black-boxes, widely accepted as the most important and efficient form of reuse (Szyperski, 2002). We chose the term reusable software artefact, or simply artefact, as concept to designate in general any kind of software black-boxes aimed to be reused.

Since the initial proposal of McIlroy (1968) that anticipated the component based software engineering (CBSE) (Heineman and Council, 2001), the reuse of pre-existent code artefacts turned into a common practice in software engineering. It aims to facilitate the development of growing complexity with higher efficiency, lower costs, in less time and with better quality. Despite the high attention attracted by this field, bibliography about software reusability mainly bases on concrete aspects and models of reuse, presenting thus a fragmented and compartmentalized vision of this domain. Even those works that propose a wider perspective (e.g. Sametinger, 1997; Schäfer, Prieto-Díaz and Matsumoto, 1994) lack a holistic and unified vision, sorting the domain of reusability comprehensively as a conceptual characteristic of software.

Due to the fact, that classification of a domain is a fundamental step for its global understanding, for the systematization of associated processes, and in the development of evaluation and cataloguing models, the interest in defining a global model of software artefact reusability is evident. One of the aspects that should be analyzed within this model is the technological support since it is an indispensable element to equip software artefacts with reusability, attaining so a model for systematic reutilization that produces real and quantifiable benefits. Aiming towards this goal, this work proposes a model of classification based on the separation of orthogonal characteristics of reusable software artefacts from the point of view of the underlying technological support. Those orthogonal characteristics, fundamental in the artefact categorization, are presented like the three dimensions of the reusability

Aulkemeier M., Heine J., G. Roselló E., G. Dacosta J. and Baltasar García Perez-Scholfield J. (2008).
TOWARDS A CLASSIFICATION SCHEME IN ORTHOGONAL DIMENSIONS OF REUSABILITY.
In *Proceedings of the Third International Conference on Software and Data Technologies - SE/GSDCA/MUSE*, pages 122-126
DOI: 10.5220/0001875101220126

space, namely the independence, the contract specification and composition.

In the remaining parts of the work the orthogonal dimensions independence, contract and composition are described respectively. Finally, the conclusions are presented.

2 INDEPENDENCE DIMENSION

Before reusing an existing software artefact, it is necessary to isolate it from its context and its original environment, preferably by the support of the underlying platform technology. This ranges from the simple possibility to embed the code in a function, class or module and this again in a compiled library, to some further options like the realization of remote system calls via a network. Nevertheless, some dependencies generally remain existent in the isolated artefact. Sametinger (1997) calls them platform dependencies, using the term platform in a very wide sense. He denotes, the less the platform dependencies, the better the possibility for reuse. The most important types of platform independence that an artefact may exhibit are exposed in the following subsections.

2.1 Application Independence

The application independence is the minimum level of achievable independence, meaning that an artefact can be used in some application different from the one it was originally designed for, that is, the minimum requirement to consider an artefact reusable. Normally, application independence is achieved by placing code within a library or a module. The general difference between both is that normally a module is used as a whole, whereas a library may contain a collection of potentially independent functions and classes from which applications make just partial use.

In practice, various types of libraries have been proposed which aim to provide this kind of independence. We first point out the source code libraries that may contain collections of definitions of values, functions, classes, generic parts, etc. in readable and editable source code form, sometimes referred to as glass box type of reuse (Goldberg and Rubin, 1995). The code is simply copied into the new application thanks any kind of support mechanism, like the include directive in C/C++. The STL of C++ is an example for this kind of library.

A second type of library is the static library, containing compiled code that can be reused by including it into a new executable application through a binder or linker. In Windows, those library files normally have the ending .lib, or in Unix .a.

Dynamic libraries, as the third type of library, also contain precompiled code that is, in contrast to static libraries, connected with the executable program during runtime. It is the loaders responsibility to find the demanded library and load it into the memory at the adequate time. This is the case with Windows DLLs or shared objects in Unix.

2.2 Programming Language Independence

The programming language independence can be explained as the option to reuse a reusable artefact in different programming languages and not only in the one used to create it. In some cases, even existing and precompiled binary files can be reused from a language different from the one that was used to create that file (for example an object file created in C, reused by a program written in Smalltalk). This represents a form of programming language independent reuse, even though differences between languages concerning their data types, procedures calls and parameter passing often complicate its application.

In order to obtain real language independence, these problems have to be solved, normally by establishing some language independent, binary compatibility norm. The Component Object Model (COM) (Box, 1997), for example, guarantees the reusability of programming language independent artefacts by describing their types and interfaces with an independent interface definition language (IDL) and the adoption of a specific format for procedure calls and parameter passing. The underlying platform cares for the calling process and the type conversion via the so-called marshalling process or later by the Common Type System (CTS) that is used by all .Net compatible languages to define their own set of types.

2.3 OS/Location Independence

This type of independence implies the possibility to reuse a software artefact in different operating systems and, in a further step, from different locations in a network. The first case corresponds to the portability of an artefact what may appear in different ways: 1) The portability of source code, which implies that compilers for different operating systems are available to compile the same programming language, often complemented by the option to let the source code include instructions that handle differences between different operating systems, e.g. by processor directives in C/C++. 2) Packages that contain different executable versions for distinct operating systems, choosing automatically the appropriate one. Examples are fat binaries of programs like OpenStep, and fat or universal binaries for different Mac OS versions. 3) Virtual Machines that abstract the underlying operating system, like the Java VM or CIL for .Net, allowing source code to be executed on any system a compatible virtual machine exists for.

The location independence goes beyond OS independence by entering in the area of distributed systems. It can be distinguished between a homogeneous location independence where both the origin and the target platform have to be identical, e.g. DCOM, and a heterogeneous independence, where the communication between different components placed on different platforms and machines is possible, e.g. CORBA, or Web Services (WS). This level of independence can be achieved through some kind of middleware or platform that allows a remote call to a remotely installed artefact.

3 CONTRACT DIMENSION

An isolated reusable artefact needs to come with a description about itself that is as precise as possible in order to integrate it properly with another application and avoid any unforeseen complications. In fact, the description should not only explain what the artefact exactly does, but rather guarantee its functionality. Meyer (1992) called that a contract between the client and the artefact.

Due to the focus on the technological dimension of reusability, we do not refer to informal or human readable contract but rather on some kind of explicit and formal contract that can be automatically verified. This concern is crucial for reuse, and orthogonal to both independence and composition, as it doesn’t specify how the artefact will be integrated or connected with other ones, but only the usable functionality or behaviour of this artefact. In practice, the contract dimension can be broke down in various levels of strictness, as exposed in the following.

3.1 No Contract

In the most basic case, a reusable artefact does not provide any contract at all. The artefact is not forced to expose any information, not even parameters. Examples for this are the Windows DLLs that cannot contain contracts. Normally, if they are dedicated to developers rather than to end-users, they are delivered with some documentation or files that contain all prototypes of the classes and functions of the DLL, acting somehow as contract. Other examples are applications for Unix shell. The developer of shell scripts have to use help files or test the output of the different shell commands to correctly compose them, due to the missing formal contract description that would allow for a automatic verification.

3.2 Interface Signature

This contract level is about the most common format of contracts. The basic and minimal information is a formal syntactic description of the interface that can be verified automatically, usually by a compiler. In the case of functions and methods that description normally consists of the name, the parameters and the return values. Many technologies only use this kind of contract description, e.g. COM, CORBA, WS, as well as the most common programming languages (Java, C#, C++, etc.) This eases that reusable artefacts can be reused as black boxes on those technologies.

3.3 Pre-, Post-Conditions and Invariants

The pre-, post-conditions and invariants were notably introduced with the programming language Eiffel, at the same time when talking started about more or less formal contracts between clients and providers of a function. Pre-conditions, on the one hand, are used to describe the required state of a program or some other conditions which have to be fulfilled to make use of the function. The post-conditions, on the other hand, specify which predicates must always be true just after the execution of a function. Invariants describe conditions that maintain stable and without change throughout the whole interaction process with the application. Some recent languages have adapted this concept like for example ContractJ – an implementation of Design by Contract Paradigm based on Java 5 annotations –, AspectJ, plug-ins for Eclipse or Spec# (Barnett et al., 2004), an implementation of Design by Contract for C#.

3.4 Additional Implementation Details

One drawback of the previous contract model is its limited expressiveness as also demonstrated by

Szyperski (2002) or Büchi and Weck (1999). One example for that is the missing capacity to indicate the existence of callbacks. Another problem is the unnoticed violation of a contract that may occur when implementation details of a base class change that has dependency relation with subclasses. Then some methods of the subclasses that had worked before may fail if the contract won't be adapted.

Furthermore it is difficult to express side effects that a function may have with pre-, post-conditions and invariants, e.g. adding changing variables or member class variables to the function, changing input parameters, creating new objects within the function that outlast the working off of the function or to release some objects within the function. Thus, Büchi and Weck (1999) propose the idea of gray-box reuse that exposes just some important implementation details while others remain hidden.

Beside the information that refers to internal implementation details, some other, non-functional specifications might be important enough to form part of the contract, e.g. execution speed, precision of mathematical calculations, or behaviour in concurrent environments, etc.

3.5 Semantic Descriptions

One problem that all methods and syntactic description methods and languages share is their lower expressiveness compared to their semantic description. Therefore, the last level of the contract scale is the semantic description. But because the natural language is, due to its ambiguity, very difficult to be read by a machine, the only possibility to specify the semantic of an artefact is the use of formal languages, as it is realized with Semantic Web or Semantic WS.

Some existing approaches with better options to express semantic are Kind Description Language (KDL), Ontologies (Terzin and Nixon, 1999), Petri Nets (Puder and Markwitz, 1995), or the usage of agent languages like the Knowledge Interchange Format (KIF) and the Knowledge Query Manipulation Language (Terzin and Nixon, 1999).

4 COMPOSITION DIMENSION

In the context of software reuse, the term composition is mentioned mainly in relation with software components (Szyperski, 2002) and includes various software engineering processes like localization, understanding, modification and component connection. As the presented work does not limit software artefacts to components, but refers furthermore to any kind of technological units in its original form, the concept of composition will be restricted to the most associative field of composition, that is, to techniques that support the connection between artefacts. Those techniques differ between their strength of coupling.

In the following we present the identified levels of composition.

4.1 Substitution

The strongest form of composition is realized through text substitution of code fragments within one application file. A sequence of code is provided with placeholders at pre-defined positions. Usually those indicators will be substituted at compile-time through corresponding artefacts, sometimes even later at run-time. Once, the substitution process has been realized, it is not possible anymore to identify the composed artefacts as single elements or structures inside the resulting code. Examples for substitution are macros used by assembler languages.

4.2 Aggregation

A less tight level of coupling is aggregation. That is, artefacts are combined and encapsulated into newer and larger artefacts. Internally, artefacts may interact directly between each other to provide a more comprehensive functionality that can only be accessed by external users through interfaces provided by the encapsulating artefact. As the inner architecture of the composed artefacts consists of independent artefacts, later modifications or enhancements are still possible. Some examples for aggregation are nested functions from Algol, the module encapsulation of Modula or Ada, or the inheritance concept of object oriented programming languages.

4.3 Cooperation

To attain a composition with even lesser coupling, a model of cooperation can be used, where artefacts remain conceptually and sometimes physically independent, separated, identifiable and accessible from outside. They may interact momentarily with other artefacts without necessarily building new coarse grained artefacts. An example for mechanisms that allows for definition of cooperation models are high-order functions in functional programming languages. In that case, one function uses other function without further strong connection between both. The composition only lasts as long as the process is active. Other examples are mechanisms like the Unix pipe, composing two or more applications, software components or WS.

4.4 Intermediation

While in the earlier described levels of composition the artefacts are (in the latter case just temporarily) compound directly with each other, intermediation provides an indirect composition between two artefacts realized via a mediator. From the client point of view, this allows to a greater or lesser extent for a transparent substitution of one artefact by another. The basic form of intermediation is the option to define interfaces, supported by different programming languages, like Java or C#, which allows the interface implementation to change in a quite transparent form. A more sophisticated form of intermediation is supported by a trader, e.g. CORBA trader, that registers different implementations of a service specification and chooses a concrete implementation according to the requirements demanded by the client that later calls this trader.

5 CONCLUSIONS

The systematic and efficient reusability of software code is an activity that implies different aspects to be supported by technology platforms. A detailed study of those aspects should allow for achieving an identification and ordination of implied dimensions that later permits the construction of methods, processes and tools for evaluation, measurement and improvement of reusability. This kind of studies has been realized so far in a more fragmented way, focusing more on concrete reuse fields. To contribute a more complete and global vision of this field, it is necessary to tackle it from a wider and more holistic perspective. This work provides a contribution that wants to advance towards this aim, proposing a structured classification of reusability in three orthogonal dimensions, supported by the technology that allows establishing an order of increasing reusability grade. The increase of each dimension also implies a higher level of abstraction. Although this work is just a first introducing exposition, we think that the theoretical interest is obvious, contributing to the building of the general conceptual fundaments of reusability.

REFERENCES

Barnett, M., Rustan, K., Leino, M., Schulte, W., 2004. The spec# programming system: An overview, In *CASSIS 2004 post-proceedings*.

Biggerstaff, T.J., Richter, C., 1989. Reusability framework, assessment, and directions. In *Software reusability: vol. 1, concepts and models*. New York, NY, USA: ACM.

Box, D., 1997. *Essential COM*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Büchi, M., Weck, W., 1999. *The greybox approach: When blackbox specifications hide too much*. Technical report. Turku Centre for Computer Science.

Chang, H., Collet, P., 2007. Patterns for Integrating and Exploiting Some Non-Functional Properties in Hierarchical Software Components, In *14th Annual IEEE International Conference and Workshops on the ECBS '07, 83-92.*

Goldberg, A., Rubin, K. S., 1995. *Succeeding with Objects: Decision Frameworks for Project Management*. Reading, Mass., USA: Addison-Wesley Professionals.

Heineman, G.T., Councill, W.T., eds., 2001. *Component-based software engineering: putting the pieces together*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Kiniry, J.R., 1999. *Leading to a kind description language: Thoughts on component specification.*. Technical report. Pasadena, CA, USA: California Institute of Technology.

Krueger, C.W., 1992. Software reuse. *ACM Computer Survey*, 24(2), 131-183.

McIlroy, D., 1968. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering*, Garmisch Pattenkirchen, Germany, 88-98.

Meyer, B., 1992. *Eiffel: the language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Nierstrasz, O., 1995. Research topics in software composition. In *Proceedings, Langages et Modèles à Objets*, Nancy, 193-204.

Prieto-Díaz, R., 1993. Status report: Software reusability. *IEEE Software*, 10(3), 61-66.

Sametinger, J., 1997. *Software engineering with reusable components*. New York, NY, USA : Springer-Verlag New York, Inc.

Schäfer, W., Prieto-Díaz, R., Matsumoto, M., 1994. *Software reusability*. Upper Saddle River, NJ, USA: Ellis Horwood.

Szyperski, C., 2002. *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Terzis, S., Nixon, P., 1999. Component trading: the basis for a component-oriented development framework. In *Proceedings of the Workshop on Object-Oriented Technology*. London, UK: Springer-Verlag.