A Methodological Approach to Domain Engineering for Software Variability Enhancement

Alexandre Bragança[1,2] and Ricardo J. Machado[3]

[1] Dep. I&D, I2S Informática – Sistemas e Serviços SA, Porto, Portugal,
alexandre.braganca@i2s.pt
[2] Dep. Eng. Informática, ISEP, IPP, Porto, Portugal,
alex@dei.isep.ipp.pt
[3] Dep. Sistemas de Informação, Universidade do Minho, Guimarães, Portugal,
rmac@dsi.uminho.pt

Abstract. Flexibility is one of the major quality aspects that are required for today's applications. Variability realization techniques provide a mean to achieve this flexibility. Some variability realization techniques that provide higher levels of flexibility also imply more complex engineering processes. This is the case of more recent techniques like reflection and run-time code generation. This paper addresses the problem of the increasing of complexity in the engineering process caused by the adoption of very flexible variability realization techniques. We present an approach that is based on the adoption of domain engineering methods within the context of single application development. These methods, when applied in parallel with the application engineering methods, can provide support to manage the complexity of variability realization techniques.

1 Introduction

Flexibility of behaviour has become a need in today's complex software systems. This flexibility can be achieved by introducing variability points in software. These variability points can be introduced at several stages of the software development cycle and can adopt diverse realization techniques [1]. In order to achieve higher levels of flexibility, run-time variability realization techniques are emerging and starting to be adopted by the industry. Industry standards for software components like COM, XPCOM, Java2, CORBA and .Net support this kind of variability. Software platforms like Java2 and .Net also start to support run-time generation and deployment of software components. Examples of such techniques are reflection and run-time compilation. Applications such as JBoss already apply such run-time techniques [2].

The adoption of variability realization techniques, particularly run-time techniques, promises to promote higher levels of flexibility and extensibility. Despite such promises, to achieve the real advantages of variability realization techniques, its simple adoption isn't enough. In fact, the adoption of variability realization techniques also

raises the level of complexity of an application engineering process and, as such, can increase the maintenance cost and difficulty.

Domain engineering methods have been applied to promote reuse within organizations, particularly in the case of product-line approaches [3]. They have been successfully applied to enable the development of diverse applications within a domain from a common base of artefacts. As such, domain engineering main concern is the support for variability, namely the more recent, and also complex, techniques presented above.

This paper fosters a new approach to the application design by adopting domain engineering methods within the development process in order to manage application variability points and thus achieve flexibility at the application and development levels. Our position is that the problems faced by single applications that apply high flexible variability mechanisms are similar to the ones in domain engineering but at a smaller scale. Being so, we advocate the adoption of domain engineering methods, within the context of single application engineering, to support the development of high flexible variability points.

This paper is structured in two parts. In the first part (section 2 and 3), we discuss variability realization techniques at code level in order to better understand the issues they can raise at the development process level. Section 2 presents an overview of common adopted variability realization techniques and the concept of binding time. We argue that the more common techniques, although they seem to integrate well in the development process, they don't provide higher levels of flexibility because their binding times usually occur in phases of the development process prior to deployment. In section 3, we present and briefly discuss post-deployment variability realization techniques and their advantages regarding the support for flexibility. In the second part of the paper, in section 4, we discuss the issues raised by the adoption of variability realization techniques (particularly the ones with post-deployment binding times) in the context of the development process. In section 5, we present our approach to deal with these issues that is based on applying domain engineering methods within the application development process. In section 6, we conclude the paper.

2 Common Adopted Variability Realization Techniques

When dealing with variability, one important concept is the binding time, i.e., the time in which one variant is selected from a variation point. This normally means that from this point on the software will execute that variant in the variation point. It is possible to bind a variation point to a variant at several stages in the development cycle: product architecture derivation, compilation, linking and runtime. Nevertheless, it is possible to envision other binding times like, for instance, debug time.

Regarding variability realization techniques, it is very useful to make a distinction between techniques with pre- or post-deployment binding times. This is a very important distinction because it defines a line that separates the traditional software development life cycle, where it is possible to change almost everything, and the installation and running of the software in the customer, where change and variability are almost impossible to introduce.

According to our experience, the variability realization techniques with pre-deployment binding times used by the industry are based on:

- preprocessing of source code (e.g. C preprocessor directives);
- linking directives (e.g. linking diverse objects and libraries);
- parameterization (e.g. C++ templates);
- inheritance;
- code composition.

To our knowledge, code composition techniques are not widely used in the industry. Most of these techniques are based on recent academia research and therefore, in the major cases, they are not used by the industry. Examples of these techniques are aspect oriented programming [4], subject oriented programming [5], generative programming [6] and GenVoca [7].

Code composition techniques are based on the assumption that the object-oriented programming model has limitations regarding class composition such as the handling of crosscutting concerns [8]. In GenVoca, for instance, it is possible to compose the code of several classes as a unity, or layer. This combination of layers, so as to achieve one certain behavior, has some similarities with the inheritance mechanism of the object-oriented model. In the object-oriented model, inheritance is used to add new behavior to a class by specializing its original features. We can see GenVoca layers in a similar manner to inheritance but with a larger scope. These approaches to component composition are usually based on program transformation techniques [9].

The development of software using GenVoca is based on the composition of components through layers of data types. These layers of data types represent features of the component. Thus, layer compositions can be described by expressions. Here is an example of a GenVoca expression for a component based on [7]:

```
Jak = Sm ( Templ ( Java ) )
```

This expression is defining a dialect of Java that extends the Java language with the state machines and templates features. A layer data type represents each feature. Each layer contributes to the component composition with classes, attributes and methods that implement the corresponding feature (or features). By stacking the different layers we can combine the different features that the component needs.

All the presented techniques, even the more academic ones, do not impose much complexity in the development process because they are applied in phases prior to deployment. Nonetheless, they usually don't provide the degree of flexibility that is possible to achieve with post-deployment variability realization techniques. The next section discusses run-time variability realization techniques, the advantage they provide in terms of application flexibility and also the complexity they can impose to the development process.

3 Run-time Variability Realization Techniques

One very frequent technique of run-time variability, which is used by almost every application, is the conditional execution of code based on the value of a variable or expression. In this situation, all the variants of the variation point are built into the application and it is not possible to extend the set of possible variants in the variant point after the build of the application. Other techniques like the use of function pointers or the use of the template method pattern [10] are very similar in effect because all the possible variants are defined at compilation-time. One way to extend the variants of a variation point, that uses the referenced techniques, is to replace the module with the variation point with a new compatible module that has the new variants. This technique is normally used when a company needs to correct some problem with a module of an application; it replaces the binary of the module with a new version that corrects the problem. In the case of run-time variability we are not correcting a problem, like a "bug", but extending a variation point with new variants. There are other more flexible run-time variability techniques in which the variant point is open to new variants after build-time. According to our experience, the variability implementation techniques with post-deployment binding times used by the industry and opened to new variants at run-time are:

- dynamic library loading (dynamic binding);
- component infrastructure;
- scripting languages;
- reflection.

One way to extend the set of variants at run-time is to use dynamic library loading. This technique is widely used by the industry and one well-known example is the ODBC driver architecture. The possibility of dynamically loading a library at run-time and using the library entry points enables the distribution of only the libraries that implement the variants that are needed for that variability point. With this technique it is also possible to add new variants after application deployment.

The dynamic binding technique is one of the foundations of software component infrastructures like COM or XPCOM. These infrastructures implement the concept of software component based on dynamic link libraries and on the notion of interfaces to access the services provided by the components. Since applications access the components via interfaces, it is possible to change the component that the application is using if the new component supports the interface the application needs. Apart from this basic functionality, component infrastructures can offer functionalities like remote invocation or object pooling. Component infrastructures are a well-known run-time variability realization technique used in the industry, such as the Mozzila project and the Microsoft Office suite of applications. These two examples also use another variability technique: scripting languages. Scripting languages enable a very high degree of variability and flexibility because the functionality of an application can be totally changed at run-time. Nevertheless, to our knowledge, they are mostly used to extend and customize applications. The great advantage of scripting languages is that they don't need the heavy building steps of general programming languages (e.g., compilation and linking). However, their use is still limited because, as we know,

scripting languages still require some training from the end-user. This is a very important factor in run-time realization techniques because their post-deployment binding time characteristic can, in certain cases, imply the end-user involvement. This is usually the case of scripting languages.

Reflection is also a technique that can be used to implement run-time variability. Development platforms like Java2 or .Net support this technique. Since they lead the industry adopted development platforms, it is very probable that reflection will become also a very used run-time variability realization technique. At the very base, reflection techniques provide a way to dynamically load types, instantiate objects and execute methods. More advanced reflection functionalities, like code generation, can be provided. These reflection characteristics are very powerful because they can bring almost all the possibilities of pre-deployment stages of software development to post-deployment time. This is also one of the reasons why these techniques impose much more complexity to the development process.

4 Variability in the Software Development Process

All the presented variability realization techniques can be used to increase the degree of variability of an application. There are two core concepts regarding variability and the software lifecycle: introduction time and binding time. Introduction time regards the moment in the software lifecycle when a variability point is introduced. For instance, in the requirements phase it is possible to introduce a variability point regarding a database feature of the application specifying that it must support object-oriented or relational database engines. In the introduction time, we open the variability point to possible variants. In a later stage of the application lifecycle, the variability point is closed by selecting one of the possible variants. This moment is called the binding time. In our example, in a later stage, we choose to use an object-oriented or relational database engine. Some authors extend these simple variability concepts by introducing the concept of binding site, which differs from the binding time because it encompasses not only the time dimension but also the space dimension [6]. Basically, the binding site represents when and where the variability point is bound to a variant. For simplicity we will use the concept of binding time during this paper. A variability point can also remain open after the binding time. This means that the variability point can be rebound to another variant.

Another important characteristic regarding a variability point is the number of variants it supports. For instance, in our previous example the database variability point only supported two variants.

Figure 1 presents the possible introduction and binding times for the database example. In this case, the introduction (IT) and binding times (BT) regard two consecutive stages of the software lifecycle. This means that only from analysis to design the variability point remains open. After the design phase the variability point is bound to one of the two variations so, from this phase on, regarding database engine support, the degree of variability of the application is null. This doesn't mean the application doesn't have database variability. In fact it has variability because of the database variation point but its introduction and binding times occur at earlier phases of the

software lifecycle. If the database variability point remains open until a later phase, for instance compilation, then the degree of variability of the application is greater. This is due to the fact that if we delay the introduction and binding moments we are increasing the variability. The later we apply variability, the more variable the application becomes. The adoption of variability at later phases of the development process provide a quicker changing of variants because there is no need to restart the process from earlier development phases. In the case of figure 1, if we need to change from an object-oriented database engine to a relational one, we need to do so in the design phase. So, the figure 1 case has a lower degree of variability than a similar case with a binding time in the compilation phase.

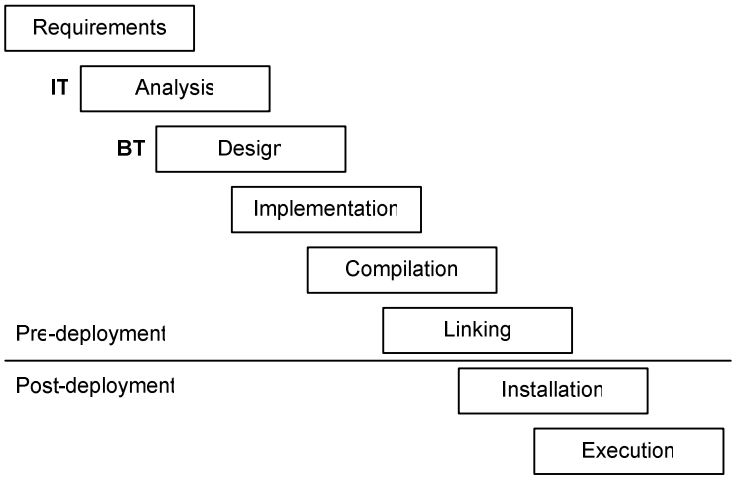

Fig. 1. Introduction time (IT) and binding time (BT) for the database variability point.

Based on the previous principles, it is possible to try to quantify the degree of variability of a variability point. As we have pointed out, the major variability concepts are the introduction time, the binding time and the number of variants. For a given variability point, the later it is introduced or bound the more variable it is. The variability also increases with the increase of the number of variants. So, if increasing numbers are given to the software lifecycle phases (i.e., 1: Requirements; 2: Analysis; …; 8: Execution), we can achieve a very simple measure of variability by the following expression:

```
IT * BT * nV
```

Where `IT` represents the introduction time, `BT` represents the binding time and `nV` represents the number of variants. The degree of variability for the figure 1 case becomes 2*3*2=12. If we change the binding time to the compilation phase we get a variability degree of 2*5*2=20. This approach to measure the variability degree of a variability point is further discussed and a normalized representation of variability is

proposed in [11]. In this paper we will use our simplified measure of variability because it is clear and more suited for the aim of the paper.

It is clear that a maximum degree of variability can be achieved if variability is adopted at later software lifecycle phases, particularly in post-deployment phases. This is one of the reasons why run-time variability techniques, such as the ones described in the previous section, are becoming more and more widely adopted in the industry.

Since the degree of variability measures the capability to support changes in an application, it is also seen as a measure of the flexibility of the application. In our example, as we have a database variability point that enables us to choose between two database engines, our application is more flexible than one that doesn't explicitly have that possibility (variation point) in the software lifecycle. This means that changing the database engine of the later application will require more effort than in the former, i.e., the former application is more flexible. Thus, the introduction of variation points in the software lifecycle will enhance the flexibility of the application. One topic of interest is how to identify possible variability points. We will not focus our attention on this concern in this paper. Nevertheless, variability is one of the core topics of product lines and product families, and many authors have propose methods and techniques regarding variability identification and representation [12-14]. When the aim is to develop a single application system, the objective of variability is not to support the possibility to generate diverse applications from a common base but to provide support for flexibility and extensibility in a single application, i.e., to facilitate the possibility of changes in the application.

In general, if an application feature is taken into account as possibly being an aspect which realization can change in the future, then a variability realization technique should be adopted. This kind of decision should be taken with care. If, for instance, the realization of such feature already implies some kind of variability, then it is probably best to adopt a variability realization technique that supports the ease extension of variants (to support the possible future inclusion of new variants). In this way, it is possible to increase the flexibility of the application with little development effort, since a variability mechanism was already needed. For instance, in the previous database example, instead of adopting a pre-deployment technique, like the use of pre-processing compiler directives to support the two database engines, it was possible to adopt a more easily extended variability realization technique. Two possibilities were the adoption of a factory like design pattern [10] with run-time binding time for variations or even a variability realization technique based on dynamic linking libraries, which support the introduction of new variants at application run-time.

As we have already mentioned, the identification of variability points is not in the scope of this paper. Nevertheless, if a variation point is identified, this means that at least two variants exist for that variability point. In this context, and if the aim is to develop a flexible application, care should be taken with regard to the adopted variability realization technique. We have presented a simple expression that gives a hint on the degree of variability that we obtain by adopting a variability realization technique. Using this simple approach, a software engineer can more easily predict the application flexibility implications of his choices. Resulting from our experience, we can say that, with the possible exception of reflection and code generation techniques, the effort and complexity added by adopting variability realization techniques at later

stages of the software lifecycle has no major cost implications for medium and large projects [15].

As we have seen in this section, the adoption of run-time variability realization techniques provides a mean to increase the flexibility of applications. With these techniques, the introduction of new variants in applications becomes easier and manageable. This is true because run-time variability realization techniques, like dynamic linking libraries, make it possible to introduce new variants without requiring changes in the application. One well-known example of such an approach is the ODBC architecture that supports changing the database engine of an application at run-time.

If a run-time variability realization approach is adopted in order to enhance the flexibility, we get the possibility of adding new variants as the requirements regarding the variability point change. But this is only easy if the structure (or interface) of the variability point doesn't change, i.e., we can add a new ODBC driver to support a new database engine only if the new driver fits in the actual ODBC interface. If this isn't true, then the ODBC interface needs to change to accommodate the new driver and this may have many implications, like the updating of the old drivers or the support for two running versions. This implies that we must return to early software development phases, where the variability point was introduced, in order to change the structure of the variability point. When this occurs the software development process can become more complex and software development problems start to appear. Because this kind of problem can occur with some frequency, in the next section we propose an approach based on domain engineering which goal is to deal with the level of complexity that variability realization techniques can impose in the software development process.

5 Domain Engineering for Variability Support

Product lines and product families aim at promoting reusability between applications [3]. To achieve reuse between the diverse applications of a product line the applications must share characteristics or features. This is usually only achieved if they share some domain. For instance, it is possible to achieve reuse between a word processor and a presentation application because they share the domain of office applications.

Domain engineering is a process that aims at identifying, representing and implementing reusable artefacts [16]. Examples of methods that apply domain engineering principles are FODA [12] and RSEB [13]. Traditionally these methods only have been applied when multiple applications share the same domain, like the case of product lines. In these cases, domain engineering provides adequate methods and techniques to support the development of reusable artefacts, like software architectures, design models and software components. Figure 2 shows the possible interactions between domain engineering and application engineering. The artefacts produced by domain engineering can be reused in application engineering. The arrows from application engineering to domain engineering show that application engineering can supply input to domain engineering, usually in the form of domain knowledge. Each new application that is built within the domain will gain from reusing domain artefacts but will also provide knowledge to refine the domain artefacts or build

new ones. The artefacts that result from the domain engineering process consist on common components to be used in applications but also on components implementing the variants of a variability point that will be used only in specific applications. This is also what is needed when building flexible and extensible applications based on variability. So, instead of using domain engineering as a process to support the development of diverse applications in a domain, it may be used to develop the variants of variability points in single applications. What we advocate is that domain engineering methods can be adopted to support the design of variability points in single application development processes. In fact, if we consider each variability point a domain, then domain engineering methods can be applied for each variability point. Since domain engineering methods require extra effort in the engineering process, care must be taken in the selection of variability points that should be designed using domain engineering methods.

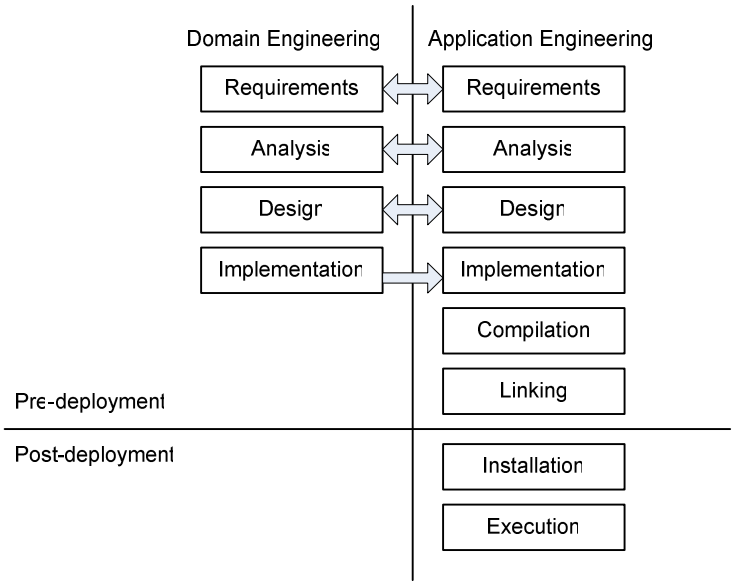

Fig. 2. Domain Engineering vs. Application Engineering.

Usually, only variability points which provide high degree of variability should be designed using domain engineering processes. This is normally the case of variability points which use run-time variability realization techniques. The database example, introduced in the previous section, is one potential variability point for which a domain engineering method could be adopted. If the degree of variability is higher, for instance, because different database engines will be supported by dynamic linking, then domain engineering should be adopted for the variability point, even if the support for different database engines will only be used for one single application. In such a case, the support for the variability point should be promoted to a component to be reused in the application. In this way, a domain engineering method could be

used to engineer the component that supports the variability point and also to engineer the diverse variants of the variability point.

This approach to the engineering of variability points, that combines domain engineering and application engineering, has many advantages over simple application engineering. In our opinion, one of the major advantages is that structural changes in the variability point become more easily managed. Since the support for the variability point becomes a domain engineered component, a structural change in such a component becomes a change in the domain of the variability point. That kind of change is then managed within the domain engineering process adopted and doesn't imply more complexity in the application development process.

6 Conclusions

The adoption of variability realization techniques, in particular post-deployment techniques, is increasing. The major reason for this is the flexibility they provide for the applications. Nonetheless, their adoption implies also an increase in the complexity of the development process. Variability related issues have been managed through domain engineering methods in the context of product line approaches [3]. Based on these facts, the paper presents a new approach regarding the adoption of domain engineering methods. We propose that domain engineering can be adopted as a successful method to support the introduction and development of variability points in single applications. We present a simple approach to measure the degree of flexibility of application's variability points. Using this simple measure we discuss the conditions that justify the adoption of domain engineering methods within application development.

Our proposal is based on our own experience with the adoption of variability realization techniques in application engineering as a mean to increase application flexibility [15]. The results so far seem promising but it is also clear that further research regarding domain engineering method adoption within application development process is needed. In our future work we plan on extending our present work with these concerns.

References

1. van Gurp, J.: On the Design & Preservation of Software Systems. PhD Thesis. Computer Science Department, University of Groningen, Groningen (2003)
2. Fleury, M., Reverbel F.: The JBoss Extensible Server. In: Middleware 2003 - ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro. Lecture Notes in Computer Science, Vol. 2672, Springer-Verlag (2003) 344-373
3. Bosch, J.: Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach. Addison-Wesley (2000)
4. Kiczales, G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M., Irwin J.: Aspect-Oriented Programming. In: European Conference on Object-Oriented Programming (ECOOP), Finland. Lecture Notes in Computer Science, Vol. 1241, Springer-Verlag (1997)

5. Ossher, H., Harrison, W., Budinsky, F., Simmonds, I.: Subject-Oriented Programming: Supporting Decentralized Development of Objects. In: 7th IBM Conference on Object-Oriented Technology (1994)
6. Czarnecki, K.: Generative Programming Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. PhD Thesis. Department of Computer Science and Automation, Technical University of Ilmenau (1998)
7. Batory, D., Lopez-Herrejon, R.E., Martin, J.-P.: Generating Product-Lines of Product-Families. In ASE'02, Edinburgh, Scotland. IEEE Computer Society (2002)
8. Lopes, C.: D: A Language Framework for Distributed Programming. PhD Thesis. College of Computer Science, Northeastern University (1997)
9. Neighbors, J.M.: The Draco approach to constructing software from reusable components. IEEE Transactions on Software Engineering, Vol. 10, Issue 5, IEEE (1984) 64-74
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
11. Jaring, M., Bosch, J.: Representing Variability in Software Product Lines: A case study. In: Second Software Product Line Conference (SPLC2). Springer-Verlag (2002)
12. Kang, K.C., Cohen, S.H., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study Technical Report. Software Engineering Institute, Carnegie Mellon University (1990)
13. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. Addison Wesley Longman (1997)
14. Griss, M.L., Favaro, J., d'Alessandro, M.: Integrating Feature Modeling with the RSEB. In: Fifth International Conference on Software Reuse, Victoria, Canada. IEEE Computer Society Press (1998)
15. Bragança, A., Machado, R.J.: Run-time Variability Issues in Software Product Lines. In: ICSR8 Workshop on Implementation of Software Product Lines and Reusable Components, Madrid (2004)
16. SEI: Domain Engineering: A Model-Based Approach. 2004. http://www.sei.cmu.edu/domain-engineering/domain_engineering.html