An Evaluation of the NetBeans Module System as a Product line Implementation Technology

Mærsk-Møller, Hans Martin; Jørgensen, Bo Nørregaard

Published in:

Publication date: 2011

Document version
Peer reviewed version

Citation for published version (APA):
AN EVALUATION OF THE NETBEANS MODULE SYSTEM AS A PRODUCT LINE IMPLEMENTATION TECHNOLOGY

HANS MARTIN MÆRSK-MOLLER
The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Odense, Denmark

BO NØRREGAARD JØRGENSEN
The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Odense, Denmark

ABSTRACT
Variability is one of the key concepts of software reuse in software product line engineering (SPLE). Rich client platforms (RCPs) have been designed to facilitate development of modern client applications. They facilitate this with concepts, structures and pre-made functionality which support the development and conceptualization of flexible and modular applications. These characteristics make them interesting in the context of SPLE. Despite the fact that modularity, extensibility and flexibility are characteristics, which in general are even more important to software product lines (SPLs) than to standalone applications, these platforms and their facilities have not been investigated scientifically from a SPLE perspective. This paper remedies the situation by providing an evaluation and a discussion of the module system provided by NetBeans Platform (a RCP). The paper contributes to the catalog of SPL implementation technologies, it introduces SPLE practitioners and theoreticians to RCP technologies, and sheds light on some of the concepts, which can help to make a more informed decision when it comes to choosing implementation technology for development of SPLs.

KEY WORDS
Component-Based Software Engineering, Software Reuse and Metrics, Computer and Software Engineering, Design Patterns and Frameworks, Software Architecture and Design.

1. INTRODUCTION
The software developers of today are met with many demands, such as reducing labor, cost and time-to-market, while at the same time increasing the quality, manage increasing complexity, and creating, managing and evolving continuously growing and diversifying products. These demands have been recognized by the software industry and many organizations are planning to adopt or are currently running software product lines to target these demands. Typically, these organizations are developing and maintaining a set of highly similar but separate products, which makes development and maintenance activities both redundant and labor-intensive. SPLE has been proven to meet the demands, while reducing these activities, through strategic reuse of development artifacts. The strategic reuse consists of explicitly managing the variabilities between the products to exploit their commonalities.

Significant work has been done on various topics in the SPLE research community, but implementation-specific handling of variability has acquired very little attention. This paper remedies this by using the general evaluation schema for product line implementation technology provided by Anastasopoulos...
& Muthig [1] to evaluate the NetBeans module system (NMS) as a variability implementation technology on the component level of granularity.

The case study is our SPL called GreenComponents, which is an SPL for creating tools for energy and cost-efficient greenhouse production. It currently has three SPL members, DynaLight Web, DynaLight Desktop with control capabilities, and DynaLight Desktop without control capabilities. The products are deployed in the industry and have previously been described in [2] [3].

The SPL is implemented as a NetBeans Module Suite, which contains the core assets (modules) for the three SPL members. The project is primarily written in Java and uses the modular architecture provided by the NetBeans Platform. The project includes the following technologies: Java Server Pages, Java Servlets, HTML, Perl, Ant, Derby DB and MySQL DB. The DynaLight Desktop application, which we will exclusively focus on in this paper, is a NetBeans Platform Application. Other variability mechanisms than the NMS are used in the SPL, but these are outside the scope of this paper.

The structure of the paper is as follows. Section 2 introduces the NetBeans Module System and shows implementation details on module variability. Section 3 introduces the evaluation schema. Section 4 contains the evaluation based on our case study experiences. Section 5 describes the related work, and finally section 6 concludes the paper.

2. THE NETBEANS MODULE SYSTEM

The NetBeans Platform provides a reliable and flexible modular architecture to application developers. We define the NMS as the module-related functionality provided by NetBeans Runtime Container (NRC) [4]. NetBeans Modules (NBMs) are specialized software components and the basic building blocks of NetBeans Platform Applications. NBMs encapsulate functionally-related classes, and describe their exposed interfaces and their module dependencies.

There are two ways modules can interact, the first way is with static module dependencies (similar to static library dependencies) and the second is dynamic module dependencies using the NetBeans Platform as a Service-Oriented Component Platform. In the latter, which is the one we will focus on in this paper, modules can provide services by implementing service-provider interfaces, and are then discoverable and usable by service-consumer modules.

The NetBeans Platform facilitates service-provider registration and discovery by using the Lookup Pattern [5]. The realization of the pattern has several characteristics, which makes it particularly useful for module variability. One of these characteristic is the design choice of the search term used for finding objects in the NetBeans Lookup (hereon referred to as lookup). We will return to this later in this section.

The NetBeans Platform provides a premade optional mechanism for plug-in support, which can be added to all NetBeans Platform Applications. Plug-ins are modules that can be loaded and managed at runtime. The support for plug-ins is only available in distributions where it is explicitly selected during composition.
Providing Service Modules

Before a module is available to other modules it needs to register. There are different ways of registering modules in the lookup, but we will only show the method using annotation, which is illustrated in Tab.1. The annotation takes a parameter to specify the service-provider interface (SPI), which the implementation module is service provider for.

Tab.1 - Code Excerpt of Lookup Registration

```java
@ServiceProvider(service=aServiceProviderInterface.class)
public class aServiceProvider implements aServiceProviderInterface{
    //implementation
}
```

At startup of any NetBeans Platform Application all registered modules are found and used to build an internal registry containing them (the global lookup), thereby becoming discoverable by others.

Service Consumers

When an object is put into the lookup e.g. by the application startup, it can be looked up using all of its implemented interfaces and its class inheritance. The lookup implementation thereby gives good possibilities for module variability using generalization and decomposition, i.e. a module can depend on looking up a general interface or class object which precisely fulfills its requirements.

An example is shown with the class diagram and code in

```java
Lookup lookup = Lookup.getDefault();
PriceDataService aInstance = lookup.lookup(PriceDataService.class);
aInstance.doStuff();
```

Fig.1. The consumer only depends on the interface called Price Data Service, not directly on the implementation, and this enables variation between implementations of service providers, i.e. modules. The NRC finds the modules at startup, loads them and adds them to the global lookup. The consumer depends on known interfaces and uses these class objects to find implementations in the lookup. It can thereafter use the services without knowing the concrete implementation.

The module system thereby facilitates selecting module(s) for a distribution at composition time or, with the optional plug-in facility, during runtime.

```
Lookup lookup = Lookup.getDefault();
PriceDataService aInstance = lookup.lookup(PriceDataService.class);
aInstance.doStuff();
```

Fig.1 – Price System UML Class Diagram and Code Excerpt
Shared Objects

We use the construction shown in Fig.2 for optional variability. It involves using the lookup for a different purpose. Modules make interfaces they want others to be able to use publicly available. They place an object implementing the interface in the lookup. Other modules can look it up and use the object. The lookup implementation also provides an event-mechanism on content changes.

![Shared Object UML Diagram](image)

Fig.2 – Shared Object UML Diagram

3. EVALUATION SCHEMA

Anastasopoulos & Muthig [1] provide the following table for evaluating software product line implementation technologies (Tab.2).

<table>
<thead>
<tr>
<th>Activity</th>
<th>Effort</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework Engineering</td>
<td>Implementing reusable code Effort for making code reusable across the product line (development for reuse)</td>
<td>Reuse techniques</td>
</tr>
<tr>
<td></td>
<td>Effort for testing reusable code</td>
<td>Variation types</td>
</tr>
<tr>
<td></td>
<td>Reacting to evolutionary change Effort for integrating system-specific code into the product line</td>
<td>Granularity levels</td>
</tr>
<tr>
<td></td>
<td>Effort for adding or removing variation (variability management)</td>
<td>Integration impact</td>
</tr>
<tr>
<td></td>
<td>Maintenance effort</td>
<td>Automation</td>
</tr>
<tr>
<td>Application Engineering</td>
<td>Reusing code Effort for reusing code to derive a concrete product (development with reuse)</td>
<td>Reuse techniques</td>
</tr>
<tr>
<td></td>
<td>Resolving variations Effort for creating a concrete product line member</td>
<td>Binding time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Automation</td>
</tr>
</tbody>
</table>

Detailed descriptions of the factors and the rationale behind the evaluation schema can be found in [1]. However, we will briefly introduce these factors:

- **Reuse techniques**: This factor is divided into two: Reuse between SPL members, where the technology is evaluated on how variability can be separated from commonalities, and how variants can be selected for the specific members. Reuse over time, where it is evaluated on support for introduction of unexpected features and variability during software evolution.
- **Variation types**: Is evaluation on how the technology handles positive and negative variability. Positive variability is when functionality is added for creating an SPL member and negative is when functionality is removed.
- **Granularity**: The level of granularity that can be handled by a technology is an important characteristic of its applicability. The granularity level can be everything from single lines of code to entire frameworks.
- **Testability**: Is evaluation of the support for tests, when the technology is used.
• **Integration impact:** It covers the support for integration of member specific assets and externally-developed assets into common assets of the SPL.

• **Binding time:** The binding time is the point in time, when a variant can be bound. It is an important characteristic as it may influence application engineering and may limit the suitability of the implementation technology.

• **Automation:** Evaluates the support for automation, when using the technology. Automation refers to technologies that can support management and creation of SPL members, and supports resolution of variation.

4. EVALUATION OF MECHANISM

**Reuse across product line members:** The modular architecture promotes decomposition and generalization of the applications. The extent to which an application is decomposable depends on its nature. In some cases features are cross-cutting and thereby can be difficult to modularize using NBMs. Then again, other technologies can be applied to target this deficiency, e.g. AspectJ.

The module system is good at facilitating variability on the granularity level of modules or clusters of modules, in other words coarse-grained variability. However, implementation technologies exclusively focused on this level of granularity may not be adequate for realizing the required variability between SPL members. This problem can be solved by applying additional SPL implementation techniques.

Some features can be developed as completely independent structures and be untraceable in the final SPL member if it is not part of its module composition. A case study example is the SuperLink4 Logger (SL4). It is responsible for writing production set points from a database to an environmental climate computer inside a greenhouse. A common feature of DynaLight Desktop writes the set points to a database for persistence and later analysis. However, the database is also accessible to SL4, and thereby works as the enterprise integration pattern called Shared Database [6]. This makes SL4 a completely optional feature. It can be present as a module, not be present at all or even exist as a separate external running program. This shows that even features which are conceptually integrated parts of systems can be completely optional between the SPL members without side effects.

**Reuse over time:** We consider the evolution effect on framework engineering, and not evolution on specific SPL members. The evolution of SPLs can be categorized into proactive, which is dealing with anticipated changes, and reactive that is dealing with unanticipated changes [7].

We evaluate the technology to have good support for modifying cardinality of variants for existing variation points at composition time and even at runtime. Generally, this is regarded an anticipated form of change, but if the amount of variants were frozen in the initial design, this is an unanticipated change. The ability to incorporate unforeseen changes due to evolution is difficult to assess, but we evaluate the technology to proactively support reactive changes because of its modular infrastructure, as changes become easier to manage and structure.

**Variation types:** The NMS technology supports both types of variability, both negative and positive. An example of negative variability is the creation of SPL members from GreenComponents module suite. Here the creation consists of
omitting modules which are not to be part of a specific SPL member. This is analogous to carve out the final form from a whole piece of wood.

An example of positive variability is adding plug-ins, which are not part of GreenComponents, to a SPL member. This could for example be a database management tool plug-in in an already running application.

**Granularity:** The technology only deals with variability on the granularity level of modules. However, it is possible to create modules without content, and modules that contain packages, classes, interfaces, data types and external libraries. Granularity levels below java classes are not supported by this technology. Using the NMS for isolated single classes is considered excessive.

**Testability:** The testability of modular applications depends on the design of the implementation. Testability is increased if the lookup can be substituted with alternative implementations. This allows an easy way to provide mocks and stubs for testing purposes. JUnit Tests and Continuous Builds are used in the case study and we consider the testability and its support to be very good.

**Integration impact:** The integration impact is highly dependent on the SPL strategy applied. We have employed the strategy of adding member-specific modules to GreenComponents thereby letting the SPL members be derivable from the module suite. This approach supports the integration very well as the only thing, which is required to promote a variant feature to a common feature, is to select them for the remaining members during their composition. The NMS has also good integration facilities for COTS and external components, as there is support for creating wrapper modules, and the case study includes several.

**Binding time:** There are two different binding times, and these are composition time and runtime. Composition-time binding forces the variants to be defined and built before composition time, while the latter allows variants to be defined even after deployment.

**Automation:** The NetBeans IDE supports the module development with automation from module creation to composition. It provides wizards with guidance through the necessary steps. All of this can be done manually outside the IDE, but it is cumbersome and the automation provided by the NetBeans IDE is valuable.

Composition of an SPL member is simplified to checking boxes (Fig.3 a) and adding optional branding. The steps to create a new product are guided by a
wizard. The GreenComponents are thereafter added as an external cluster, and the individual modules are selected as mentioned.

Predefined modules from the NetBeans Platform can add functionality for plug-ins and automated updates. This enables users to install, update, activate/deactivate installed modules, download modules, etc. (Fig.3 b).

Mutual exclusiveness between modules is to the best of our knowledge not enforced by automation. Anastasopoulos and Muthig also encountered this absence when evaluating the implementation technology of Aspect-Oriented Programming [1] with Eclipse IDE, AspectJ and Ant and found one solution to prevent invalid compositions using Ant. Ant is used in NetBeans IDE and we expect the same prevention can be made here, however, this is left to future research. It should be noted that the NetBeans Platform is not designed as a platform for SPLE, but for accelerated development of independent applications.

5. RELATED WORK

SPL implementation technologies have received some attention by the research community. However, most covers the semantics (e.g. Overloading, Conditional Compilation, and Reflection) like Anastasopoulos & Gacek [8] and Svahnberg et Al [9] do, and is not technology specific. Fazal-e-Amin et Al [10], however, use Java as implementation technology in their case study.

To the best of our knowledge only a few have treated the topic of evaluating specific SPL implementation technologies. Some programming languages are evaluated by Patzke & Muthig [11], Aspect-Oriented Programming is evaluated by Anastasopoulos & Muthig [1], and different component technologies are evaluated by Kettemann et Al [12]. However, to the best of our knowledge no one has evaluated the NetBeans Platform’s module system or the NetBeans IDE’s support for developing SPLs using the platform.

The general idea of using components for implementing SPLs has been investigated both in scientific papers by Cirilo et Al [13] and Atkinson et Al [14] and treated in the book by Atkinson et Al [15]. However, their emphasis is on methodology where we consider implementation of varying assets.

Rich Client Platforms (e.g. Eclipse RCP and NetBeans Platform) have not received much attention by the SPLE research community as SPL implementation platforms, except as framework for another research platform by Cervantes & Charleston-Villalobos [16]. Plug-in component-based techniques have received attention by Caporuscio et Al [17] and Wolfinger et Al [18]. None of these works explicitly evaluate RCPs or their module systems as SPL implementation technologies.

6. CONCLUSION

We conclude based on our evaluation results that the module system implementation technology offered by the NetBeans Platform is very suitable for both composition time and runtime module variability. It is a valuable step forward for SPL implementation techniques, in particular because it is so well supported by automation included in the NetBeans IDE. The level of tool-support and automation is more complete than we have experienced in other projects using other SPL technologies so far. However, it is an implementation technology aimed at one particular granularity level, which is components, and
this is where it provides the most benefits and advantages. In other words, this is no silver bullet, but it is a good technology example for any SPL engineer, and one to remember when considering technologies for new or existing SPL.

We consider evaluations like this important for the impact of SPLE in the industry as it lowers the adoption barrier, which in turn can create synergies that will benefit the field of research. We add to the catalog of implementation technologies and contribute to the collection of experience and knowledge called for in the work by Anastasopoulos & Muthig [1] and others.

REFERENCES