APPLYING SOFTWARE FACTORIES TO PERVASIVE SYSTEMS: A PLATFORM SPECIFIC FRAMEWORK

Javier Muñoz
Technical University of Valencia
Camí de Vera s/n, E-46022, Spain

Vicente Pelechano
Technical University of Valencia
Camí de Vera s/n, E-46022, Spain

Keywords: Pervasive systems, frameworks, MDA, software factories, templates, model transformation.

Abstract: The raise of the number and complexity of pervasive systems is a fact. This kind of systems involves the integration of physical devices and software components in order to provide services to the inhabitants of an environment. Current techniques for developing pervasive systems provide low-level abstraction primitives which makes difficult the construction of large systems. Software Factories and the Model Driven Architecture (MDA) are two important trends in the software engineering field that can provide sensible benefits in the development of pervasive systems. In this paper, we present an approach for building a Software Factory for pervasive systems, focusing in the definition of a product line for this kind of systems. We introduce a software architecture for pervasive systems, which is supported by a software framework implemented using the OSGi technology. Then, we integrate the framework into the MDA standard defining the framework metamodel and providing tool support for the automatic code generation.

1 INTRODUCTION

Pervasive systems try to build environments where computation elements disappear from the user point of view but their functionality is still provided. This vision was initially described by Weiser (Weiser, 1991) in the early 90s and it is based on the construction of computing-saturated environments properly integrated with human users.

The application of model driven approaches to this filed can provide many benefits (Fernandes et al., 2004). Software Factories (Greenfield et al., 2004) and the Model Driven Architecture (MDA) (Object Management Group, 2003) provide strategies for raising the abstraction level in the software development process and making affordable the development of complex systems. The application of the guidelines defined in these approaches to pervasive systems development can help to build better systems in an easier way than applying traditional methods.

This paper presents an approach for building a Software Factory for pervasive systems development, focusing in the definition of a product line for this kind of systems. The structure of the paper is the following: Section 2 briefly introduces our application of Software Factories and MDA to Pervasive Systems Development. Section 3 describes Pervasive Systems main characteristics and it presents our point of view for developing this kind of systems. Based on the previous analysis, section 4 proposes an architecture for implementing pervasive systems. This architecture is supported by a framework which is introduced in section 5. In order to integrate this framework into an MDA environment, section 6 shows the framework metamodel and templates for generating code from the metamodel. All these contributions constitute a practical application of the Software Factories and MDA approaches to a specific domain: the pervasive systems. Finally, section 7 includes some conclusions and further work.

2 A SOFTWARE FACTORY FOR PERVASIVE SYSTEMS

The development of a pervasive system implies the use of many different technologies in order to satisfy all users' requirements. Usually these technologies provide low abstraction level constructs to the developer. Therefore, applying a MDA approach to pervasive systems supposes jumping a very wide abstraction gap that must deal with the heterogeneity

Muñoz J. and Pelechano V. (2006).
APPLYING SOFTWARE FACTORIES TO PERVASIVE SYSTEMS: A PLATFORM SPECIFIC FRAMEWORK.
In *Proceedings of the Eighth International Conference on Enterprise Information Systems - ISAS*, pages 337-342
DOI: 10.5220/0002457603370342

of the technology. Fig. 1 describes three strategies for filling this abstraction gap: (Fig.1.1) generating a large amount of code, (Fig.1.2) building a framework that raises the abstraction level of the target technology and then generating a minimum amount of code, (Fig. 1.3) manually refining the model in order to decrease their abstraction level until achieve the abstraction level of the target technology and then generating a minimum amount of code.

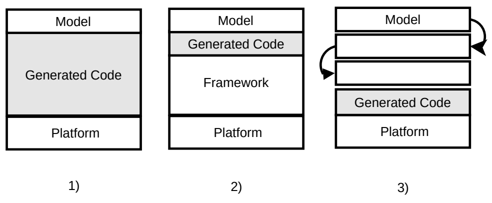

Figure 1: Strategies for filling the abstraction gap. Extracted from (Greenfield et al., 2004).

The Software Factories approach follows the second strategy. A framework for pervasive systems should be developed applying domain engineering principles. This framework raises the abstraction level of the target platform and, therefore, the amount of code is sensibley reduced. Thus our proposed methodological approach to pervasive systems development, which was presented in (Muñoz and Pelechano, 2005), is based on:

- the construction of a domain specific language for the description of pervasive systems.
- the construction of a framework that raises the abstraction level by providing similar constructs to those defined by the domain specific language.
- the definition of rules for the transformation of models, that are built using the domain specific language, to code that fulfills the defined framework.

Following this strategy, first we have defined Perv-ML (Pervasive Modeling Language), a Domain Specific Language (DSL) for specifying pervasive systems in a technology independent fashion (Muñoz et al., 2004). Perv-ML promotes the separation of roles where developers can be categorized as analysts and architects.
Systems analysts capture system requirements and describe the pervasive system at a high level of abstraction using the service metaphor as the main conceptual primitive. Analysts build three graphical models that constitute what we call the **Analyst View**. In these models the analyst describes (1) the kinds of services (by means of their interfaces, their relationships, their triggers and a Protocol State Machine for specifying the behavior of each service), (2) the components that are going to provide the defined services and (3) how these components interact to each other.
On the other hand, system architects specify what COTS devices and/or existing software systems implement the system services. We call *binding providers* to the elements that are responsible of binding the software system with its physical and logical environment. For instance, a lighting sensor is in charge of measuring a physical feature of the environment, whereas an e-mail server allows sending information to agents that are out of the scope of the system. Architects build other three models that constitute what we call the **Architect View**. We need to build a detailed specification of the lower level artifacts that realize system services in order to have a complete and operative pervasive system description. In order to achieve this goal, the architect describes (1) every kind of binding provider (their interfaces and their relationships), (2) the binding providers which are used by each system component, and (3) which actions should be executed when a component operation is invoked.
The development process starts from a Perv-ML specification which is transformed using our model-to-model transformation tool (we are currently working with the AGG[1] graph grammars engine) into a model (PSM) that is built using implementation concepts. This paper is focused in the PSM building block. In the context of MDA, the implementation framework for pervasive systems can be considered a platform, since it provides an implementation environment for our platform independent language. The construction of the framework has followed the Software Factories guidelines. This approach proposes a product line like strategy, which is composed, in short, by the following steps (Greenfield et al., 2004, chapter 11):

1. **Product Line Analysis**. Its purpose is to decide what kind of systems the product line will develop. In order to achieve that goal, the scope of the systems to be developed should be specified.
2. **Product Line Design**. Its purpose is to decide how the product line will develop the software products. In order to achieve that goal, an architecture for the systems to be developed should be specified.
3. **Product Line Implementation**. Its purpose is to supply the implementation assets required by the product line architecture. In our case, we are going to implement a framework for supporting the specified architecture.

Next sections give an overview of each step.

3 PRODUCT LINE ANALYSIS

Requirements for current and future pervasive systems involve a great diversity of types of services.

[1]http://tfs.cs.tu.berlin.de/agg/

Such different services as multimedia, communication or automation services need hardware devices that different manufacturers provide and external software systems. These elements live in several networks running on different technological platforms, but they can not satisfy isolatedly all system requirements. In this context, our approach to pervasive system development consists of:

- **The selection of the suitable COTS devices or external software systems.** These elements should provide the services that users require either isolatedly or interacting with other elements.
- **The development of the software system that integrates the external elements in order to provide the services that users require.** The development of that software may imply the use of different technologies but some gateway technology should exist.

In order to provide a software architecture that correctly fits the requirements of this kind of systems, we should determine the scope of our applications and we should identify some basic non functional requirements that must be satisfied by the assets (the software architecture and the implementation framework) that are produced for supporting the product line.

- **Support to the conceptual primitives that are provided by the modeling language**. As introduced in section 2, a key goal of our framework is to rise the abstraction level of the implementation technology in order to make easier the code generation from our DSL. Therefore, supporting the PervML conceptual primitives is a very important issue.
- **Integration with external software systems**. Services provided by pervasive systems can be supplied by physical devices and also by existing software systems (multimedia servers, contacts management software, etc.). Therefore, the integration with external software systems should be supported by our framework
- **Isolation of the manufacturer-dependent components**. As outlined above, we consider that a pervasive system is built from several COTS elements. But, on the other hand, our framework should support the DSL for pervasive systems. Therefore, in order to integrate these two requirements, the framework should clearly isolate the manufacturer-dependent parts from the parts that can be automatically generated.
- **Support to multiple user interfaces**. Pervasive systems emphasize new ways of Human-Computer Interaction (HCI). Different kind of devices and platforms could be used. Therefore, our systems should be ready to provide support to several kinds of user interfaces.

4 PRODUCT LINE DESIGN: DEFINITION OF A SOFTWARE ARCHITECTURE

Our proposed architecture for pervasive systems has been designed in order to support the requirements introduced in the above section. We apply the Layers and Model-View-Controller (MVC) architectural patterns (Buschmann et al., 1996) for providing a multitier architecture for the pervasive systems (see Fig. 2).

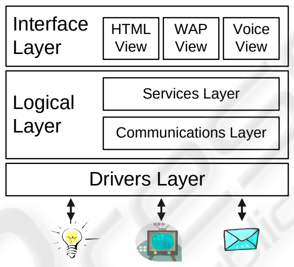

Figure 2: Overview of the proposed architecture.

The **Drivers Layer** is the lowest layer in the architecture. It is in charge of managing the access to the devices and to the external software services. In order to achieve the goals of this layer, drivers should be manually developed for dealing with manufacturer-dependent issues. Following this strategy, the drivers adapt the specific mechanisms for using the binding providers (the drivers or APIs supplied by the manufacturers), so a common interface is provided for every kind of binding provider. This means that, for instance, all the lamp devices must be adapted to a generic interface.
The **Communications Layer** provides a representation of the binding providers that can be used by the Services Layer, so it provides a bridge between these two layers. There is a one-to-one relationship between the elements in the communication layer and the elements in the Drivers Layer. Concretely, this layer holds the manufacturer-independent part of the binding providers whereas the Drivers Layer holds the manufacturer-dependent issues. For instance, if there is a driver in the Drivers Layer for accessing a light sensor which is located in a particular technology control network (like EIB or LonWorks in home-automation systems), there will be too a light sensor element in the Communications Layer. The driver would be in charge of dealing with the specific issues of the control technology, whereas their representation in the Communications Layer would be in charge of logging the operations calls, updating an icon image representing the state of the device, etc.
The **Services Layer** provides the functionality as it is required by the users of the system. The components

that implement the services make use of the elements in the communications layer or other services in the same layer. Moreover, interactions between services which are triggered by some condition can occur.
Finally, the **Interface Layer** manages the access to the system by human or software users. We apply the Model-View-Controller pattern in this layer. Following this strategy, the components of the Services Layer could be seen as the model whereas specific controller and viewers for every supported interface should be implemented.
Other architectures for pervasive systems have been proposed (Grimm et al., 2004; Kirby et al., 2003). Some of them are focused on providing very interesting capabilities like dynamic services discovery or high robustness. We are very interested in those advanced features and we plan to extend in the future our software factory for adding new characteristics, but they are out of the scope of this version.

5 PRODUCT LINE IMPLEMENTATION: BUILDING AN IMPLEMENTATION FRAMEWORK

In order to provide support to the architecture that has been introduced in the above section, we have developed an implementation framework. We have selected the middleware OSGi (The Open Services Gateway Iniatite, 2003) as the implementation technology, since it has bridges to many of the technologies used in pervasive systems and provides high-level implementation constructs. This middleware help us notably for filling the abstraction gap between the domain specific language and the target implementation technology.

5.1 An Implementation Framework for Pervasive Systems

This section briefly describes the implementation framework for pervasive systems that has been developed in order to support the proposed architecture. We do not implement the Drivers Layer because its software components should be manually developed in order to deal with manufacturer dependencies.

5.1.1 The Logic Layer

Fig. 3 shows the design classes diagram that represents the framework classes of the system logic layer. Classes in this layer can be classified in three functional groups:

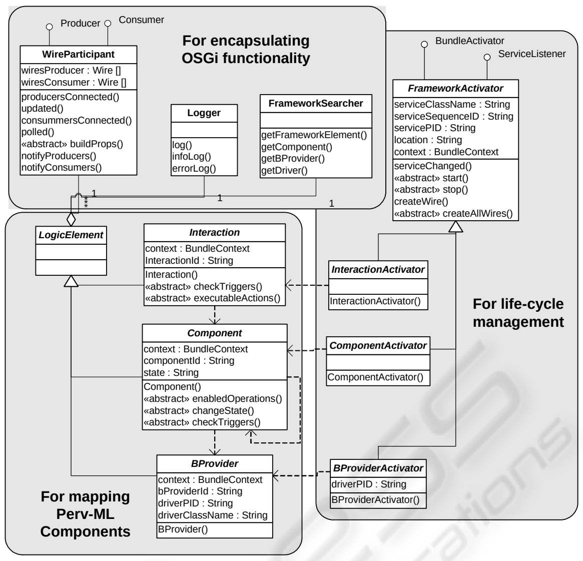

Figure 3: Design classes for the system logic layer of the framework.

- **Classes for mapping the Perv-ML conceptual primitives**. This functional group is composed by the classes `Component`, `BProvider` and `Interaction`. These classes define several abstract methods which should be fulfilled for providing support to the Perv-ML execution strategy.
- **Classes for encapsulating OSGi-related functionality**. The goal of this group is to isolate some OSGi-related functionality that is inherited by the classes in the previous functional group. Classes in this group provide facilities for logging events (`Logger`), for searching services in the OSGi framework (`FrameworkSearcher`) and for participating in the event notification mechanism supplied by OSGi (`WireParticipant`).
- **Classes for dealing with the system life-cycle**. This functional group is composed by the classes `ComponentActivator`, `BProviderActivator` and `InteractionActivator`. An *activator* is an OSGi concept which describes the class that is in charge of registering and unregistering the services in the OSGi framework when a bundle in started or stopped.

5.1.2 The Interface Layer

In order to provide support to interfaces in multiple devices, the *Abstract Factory* design pattern has been applied. Moreover, abstract classes have been included for making easy the fulfilment of two critical user tasks:

- The `ServiceListing` class provides mechanisms for accessing the services. Users can index the services by the kind of service that they provide, by their location or ordered by their last use.
- The `ServiceUI` class supports the creation of interfaces for the management of a kind of service. This interface shows (1) general information about the service, like their location or their last use; (2) specific information about that kind of service (eg. the current state of a multimedia service); and (2) the mechanism (buttons, textboxex, etc.) for using the functionality provided by that kind of service.

6 INTEGRATING THE FRAMEWORK INTO MDA

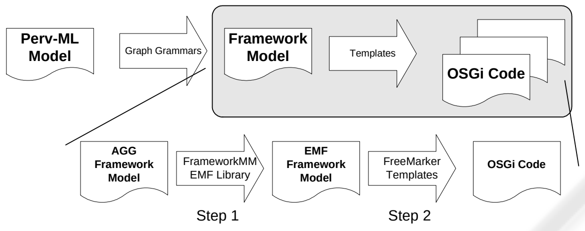

Figure 4: Integrating the framework into MDA. Tools and Techniques.

Fig. 4 shows the techniques and tools provided in order to integrate the framework into MDA. The initial step for developing an application using the complete Software Factory is the specification of the platform independent model using Perv-ML. Then, our model-to-model transformation tool converts a XML/XMI representation of a Perv-ML model into a XML file which represents a model built using concepts of the implementation framework. Then, the generated XML file is loaded into a Java data structure which is used as input to FreeMarker, the templates engine. As result of the application of the templates, the final source code is obtained. Next subsections describe the techniques related to the integration of the implementation framework in this process.

6.1 Platform Metamodel

The MDA standard proposes the construction of metamodels for defining (the abstract syntax of) new modeling languages. Therefore, we have built a metamodel of our implementation framework for pervasive systems, which is shown in Fig. 5.

The models generated by our model-to-model transformation tool conforms to this metamodel. The model-to-model transformation tool performs the

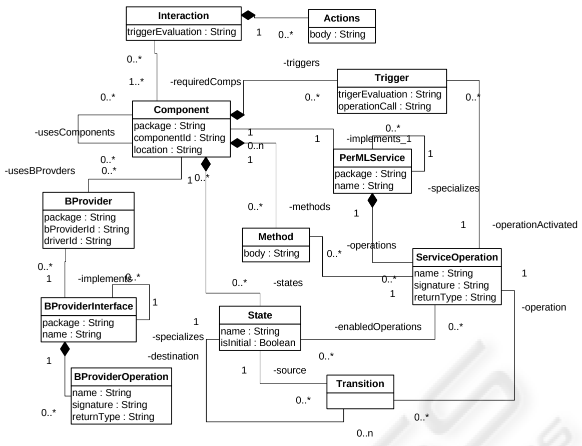

Figure 5: Framework metamodel.

jump of the abstraction gap between the DSL and the implementation framework.

6.2 Templates for Code Generation

Using the metamodel introduced above we have a graph-like representation of the pervasive systems but, in order to automatically obtain the source code of the final application, we need to transform that representation into Java files (since our aim is to produce OSGi code) and other textual resources (manifest files, build files, etc.).

In order to put in practice our product line, we have used the FreeMarker[2] template engine. FreeMarker is a free software engine that works on Java data structures, and provides a powerful syntax for specifying templates. We have specified a set of templates for the main metamodel elements. These templates navigates through the metamodel structure (which has been implemented using the EMF plugin for Eclipse[3]) in order to obtain the needed data for filling the gaps.

Next, a FreeMarker template for generating a `Component` element of the framework is introduced. This template receives a `Component` metamodel element in the `comp` variable.

```
<#assign packageName =
"${comp.getPackage()}.${comp.getImplements().getName()}
${comp.getComponentId()}">

package ${comp.getPackageName()};
//Imports here

public class Component  extends
       org.oomethod.framework.Component
            implements ${getImplements().getName()} {

  <#list comp.getUsesBProviders() as bProv>
  public static String bProvider${bProv.getBProviderId()}
  PID ="${bProv.getImplements().getName()}$
       {bProv.getBProviderId()}";
```

[2]http://freemarker.sf.net

[3]http://www.eclipse.org/emf/

```
</#list>

<#list comp.getMethod() as meth>
   public ${meth.getServiceOperation().getReturnType()}
      ${meth.getServiceOperation().getName()}
      ${meth.getServiceOperation().getSignature()} {

  //Searching the BindingProviders
  <#list comp.getUsesBProviders() as bProv>
     ${bProv.getBProviderInterface().getName()}
             bProvider${bProv.getBProviderId()};

    bProvider${bProv.getBProviderId()} =
          (${bProv.getBProviderInterface().getName()})
    this.getBProvider(
    ${bProv.getBProviderInterface().getName()}.
            class.getName(),
                    bProvider${bProv.getBProviderId()}PID
    );
  </#list>

  <#if meth.getServiceOperation().getReturnType()!="void">
  ${meth.getServiceOperation().getReturnType()} result;
  </#if>

  ${meth.getBody()}

  this.changeState("${meth.getServiceOperation().
       getName()}");
  this.log("Operation
          '${meth.getServiceOperation().getName()}'
            invoked on Component ${getPackageName()}");


  <#if meth.getServiceOperation().getReturnType()!="void">
  return result;
  <#else>
  this.notifyConsumers();
  </#if>
}
</#list>
//constructor, checkTriggers, changeState,
enabledOperations and buildProps definition here. }
```

Templates like the one presented above have been developed for the Interaction and BProvider elements. Moreover, templates for generating their corresponding activators have been developed too, using the data of the main elements (the `ComponentActivator` using the `Component` data, etc.).

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented and approach for building a Software Factory for pervasive systems, focusing in the definition of a product line for this kind of systems. We have previously experimented the benefits of the application of similar approaches. Our research group have developed a model driven development method (called OO-Method (Pastor et al., 2001)) with full code generation capabilities that has been implemented in the OlivaNova Model Execution System [4]. Our aim is to apply these successful ideas to pervasive systems development. Concretely, we are working with a specific kind of pervasive systems: home automation systems.
Currently our future work is focused on four important tasks: (1) specifying and implementing the rules for automatically transforming the Perv-ML models into models built using the metamodel introduced in this work, (2) extending the architecture and the framework in order to support advanced capabilities like dynamic evolution or high robustness, (3) providing new interfaces for accessing the system by means of new devices, like PDAs, or directly using natural language, and (4) providing industrial tool support for the application of the software factory.

[4]http://www.care-t.com/

REFERENCES

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture*, volume 1: A System of Patterns. Wiley.

Fernandes, J. E., Machado, R. J., and ao Álvaro Carvalho, J. (2004). Model-Driven Methodologies for Pervasive Information Systems Development. In *I International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, pages 15 – 23. Turku Centre for Computer Science.

Greenfield, J., Short, K., Cook, S., and Kent, S. (2004). *Software Factories*. Wiley Publising Inc.

Grimm, R., Davis, J., Lemar, E., MacBeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., and Wetherall, D. (2004). System Support for Pervasive Applications. *ACM Transactions on Computer Systems*, 22(4):421–486.

Kirby, G., Dearle, A., Morrison, R., Dunlop, M., Connor, R., and Nixon, P. (2003). Active architecture for pervasive contextual services. In *International Workshop on Middleware for Pervasive and Ad-hoc Computing (MPAC 2003)*. Rio de Janeiro.

Muñoz, J. and Pelechano, V. (2005). Building a Software Factory for Pervasive Systems Development. In Oscar Pastor and Joo Falco e Cunha, editor, *Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17*, volume 3520 of *Lecture Notes in Computer Science*, pages 329–343. Springer-Verlag GmbH.

Muñoz, J., Pelechano, V., and Fons, J. (2004). Model Driven Development of Pervasive Systems. In *I International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, pages 3 – 14. Turku Centre for Computer Science.

Object Management Group (2003). Model Driven Architecture Guide.

Pastor, O., Gómez, J., Insfrán, E., and Pelechano, V. (2001). The OO-Method Approach for Information Systems Modelling: From Object-Oriented Conceptual Modeling to Automated Programming. *Information Systems*, 26(7):507–534.

The Open Services Gateway Iniatite (2003). *OSGi Service Platform (Release 3)*. IOS Press.

Weiser, M. (1991). The Computer for the 21st Century. *Scientific American*, 265(3):94–104.