C# Partial Classes as a Mechanisms to Implement Feature-Oriented Designs: An Exploratory Study*

Elio López
Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga (Spain)
ealopezs@gmail.com

Pablo Sánchez
Dpto. Matemáticas, Estadística y Computación
Universidad de Cantabria (Spain)
p.sanchez@unican.es

Lidia Fuentes
Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga (Spain)
lff@lcc.uma.es

ABSTRACT

C# partial classes allows developers to divide the implementation of a class into several slices where each slice contains an increment of functionality as compared to the other slices. Thus, combining different set of slices, we can get classes with a variable range of functionality. With this description, C# partial classes seems to be, as also pointed out by other authors, a suitable mechanism for implementing feature-oriented designs. This paper explores this idea, by systematically applying C# to a feature-oriented decomposition based on an industrial case study and comparing the results we previously obtained using the feature-oriented language CaesarJ. As main contributions, (1) we identify benefits and pitfalls of C# partial classes for implementing feature-oriented decompositions; and (2) we outline potential solutions to alleviate these pitfalls.

Categories and Subject Descriptors

D2.3 [**Software-Engineering**]: Coding Tools and Techniques

General Terms

Experimentation, Languages

Keywords

Partial Classes, Feature-Oriented Programming, Software-Product Line, C#

1. INTRODUCTION

Feature-Oriented Programming (FOP) [8] is a relatively recent paradigm for programming software systems by composing software modules, which are called *features*. A feature is considered as an increment on functionality, usually with a coherent purpose, of a software system [3, 10].

C# partial classes [1] allows developers to split the implementation of a class into a set of of files, each one containing a slice of, or an increment on, the class functionality. Therefore, as already identified by other authors [6], C# partial classes seems a suitable mechanism to implement feature-oriented designs keeping feature implementations well-modularized and appropriately separated. Partial classes can also be found in other modern programming languages, such as Ruby.

Nevertheless, although this idea seems initially promising, it has not been explored in depth. As a consequence, the community lacks of empirical evidence about the strengthens and weaknesses of partial classes as a mechanism to achieve feature-orientation at the code level.

This paper provides an exploratory study on this topic, where C# partial classes are applied to the implementation of a feature-oriented design of an industrial case study, more specifically, to the design of a Smart Home Software Product Line [5, 4, 9, 7] [1].

We have had used this case study during 3 years in the context of the AMPLE project and we have already produced a feature-oriented design, using UML packages and merge relationships [7, 4] and a feature-oriented implementation in CaesarJ [2]. This case study covers different kinds of variability [9].

In our experiment, we have tried to derive a feature-oriented implementation from the same feature-oriented design we have used previously. Then, we have compared the obtained result with the feature-oriented implementation in CaesarJ we already had. Finally, by comparing both implementations, we have identified strengthens and weaknesses of C# partial classes as mechanism to implement feature-oriented designs.

To check the results obtained in this experiment are meaningful and valid, we have also used C# partial classes to a second industrial case study, called Sales Scenario [2], which is in charge of sales processing in a Customer Relationship Management (CRM) system. We have checked the strengthens and weaknesses identified for the SmartHome case study also appear in the Sales Scenario case study.

After this introduction, this paper is structured as fol-

*This work has been supported by the Spanish Ministry Project TIN2008-01942/TIN, the EC STREP Project AMPLE IST-033710, and the Junta de Andalucía regional project FarmWare TIC-5231.

[1] `http://personales.unican.es/sanchezbp/CaseStrudies/SmartHome`
[2] `http://www.feasiple.de/description/bsp_ss_en.html`

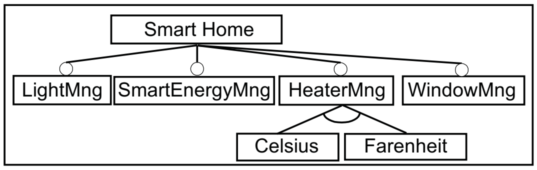

Figure 1: Smart Home feature model

lows: Section 2 introduces briefly the Smart Home case study. Section 3 identified desirable language facilities which have found useful when implementing feature.oriented designs. Section 4 describes the results obtained from our exploratory study using C# partial classes. Finally, Section 6 outlines some conclusions and future work.

2. CASE STUDY: A SMART HOME SOFTWARE PRODUCT LINE

This section presents the Smart Home Software Product Line case study [3], which will be used through the paper to analyze if C# partial classes are a suitable mechanism to achieve feature-oriented software development. This case study was provided by Siemens in the context of the AMPLE project[4] [5, 4, 9, 7]. We have selected it because it has demonstrated during the AMPLE project to be an excellent benchmark for Software Product Line Engineering; since it contains a wide range of variations of different kind and nature. Thus, it can be used to analyze if a new research contribution works properly in a wide range of potential situations.

A Smart Home software aims to improve comfort and security of the inhabitants of a house or a building, as well as to make a more efficient use of energy and resources. To achieve this goal, the software is in charge of controlling and coordinating a set of devices, such as doors, lights, heaters, windows and so forth. Figure 1 depicts a feature model for our Smart Home case study. It has been simplified for the sake of brevity. A complete version can be found in Sánchez et al [9].

It specifies a Smart Home can optionally manage lights, windows and heaters. Moreover, there is a smart energy option which coordinates windows and heaters for save energy. For instance, before heating a room, if outside temperature is colder the software close the windows to avoid wasting energy. Obviously, this function required the heater and window function has been selected, which is specified by means of a required relationship.

Figure 2 shows a UML model depicting a design supporting the variations specified for the previous feature model. Each coarse-grained feature, such as LightMng is encapsulated in a UML package, as established by the feature-oriented design guidelines elaborated in the AMPLE project [9, 7]. These packages are combined using UML merge relationships. UML merge relationships, roughly speaking, merge those elements of the same kind which match by name. For instance, Gateway classes will be merged to produce a combined version. Fine-grained variations are accommodated using other techniques, such as parametrization. So, variation in heaters measurement units is achieved by setting appropriately up an attribute in the HeaterCtrl class.

[3]`http://personales.unican.es/sanchezbp/CaseStrudies/SmartHome`

[4]`http://www.ample-project.net`

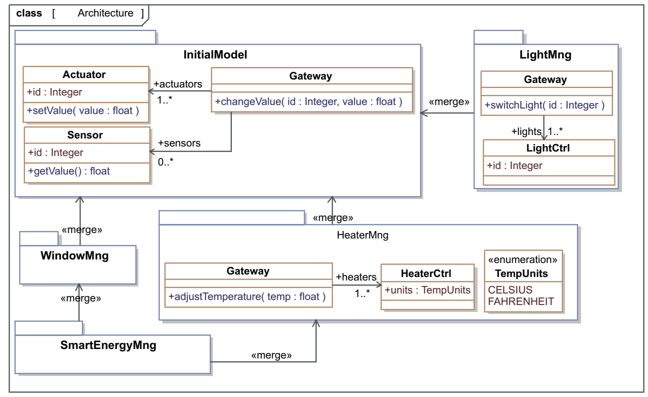

Figure 2: Smart Home design

In the following sections, we will try to create an implementation of this feature-oriented design in C# using partial classes. Strengthens and weaknesses of this technique will be identified.

3. WHAT WE WANT TO FIND IN A FEATURE-ORIENTED PROGRAMMING LANGUAGE

This section describes which characteristics are desirable to find in a language to support feature-oriented programming.

Feature-Oriented Programming [8] aims to encapsulates coherent slices of the functionality provided by an application into independent and composable modules called *features*. Therefore, different versions of a same application should be easily obtained by simply combining different set of features. Obviously, not all feature combinations lead to right final applications, so feature-oriented languages should try to ensure the result of composing a set of features produces a safe and correct application. For instance, for the previous

Wit these goals on mind, and based on [**?**, **?**, **?**, 2], we have identified several desirable characteristics we would like to find in a language for supporting feature-oriented programming. We comment on each one of them.

Feature Encapsulation and Extensibility.

A feature is often considered as an increment on program functionality [10, **?**]. For instance, in the SmartHome case study, LightMng, WindowMng, and so forth, are increments on functionality for monitoring and controlling new devices. Thus, feature-oriented languages must provide mechanism for adding new functionality to the existing one. In object-oriented languages, extension is achieved by inheritance. Inheritance is adequate when we need to extend a single class, but commonly, features need to extend several classes at the same time. For instance, the LightMng feature needs to add a new class for the light controller (LightCtrl) and it needs to add to the Gateway class methods for switching on and off lights. So, a language with feature-oriented support must provide extension mechanisms.

Extension is not always achieved by addition, sometimes *substitution* is required. For instance, in the case of the SmartEnergyMng feature, it is necessary, for instance, to override the implementation of the method adjustTemperature for checking is additional operations are required. For instance, if the command is to cool a room with the windows closed and the outside temperature is lower than the indoor one, windows will be open in addition to switching off heaters. This kind of substitution is achieved in object-oriented languages by method overriding.

All the extensions belonging to a certain feature must be added in a atomic way, i.e. either all extensions are added or no extension is carried out at all. For instance, the Gateway class must not be extended with the swicthLight method if the class Light is not added at all. Therefore, featured-oriented languages should provide mechanisms to group and ideally encapsulate elements belonging to a same feature in well-identified modules. Moreover, these modules should be compilable separately. In Figure 2, we use UML packages with this purpose.

Finally, extensions might have unavoidable ripple effects. For instance, sensors might have a reference to the Gateway class to send messages with urgent or critic situations are detected, such as fire, for instance. The class Gateway is refined through each feature, thus, several versions of the Gateway class are created, e.g. LightMng::Gateway, HeaterMng::Gateway, HeaterMng::Gateway and so forth. Thus, depending upon the selected features the customer want to include in the final product, a specific subclass, or combination of subclasses, should be selected. Let us suppose we select the LightMng feature. In that case, the version of the Gateway to be included in the final product is LightMng::Gateway. But, the Sensor class references InitialModel::Gateway. So, we should update constructors, getters, setters and so forth to reference the subclass, i.e. LightMng::Gateway, and get rid of castings and other problems related to the type system. Usually, this need to be done manually. Nevertheless, some languages, such as CeasarJ [2] or ObjectTeams [**?**] are able to automatically update these dependencies due to the usage of an advanced type system based on virtual clases and mixin composition. So, automatic dependency management is also a desirable characteristic in feature-oriented languages.

Feature-Level Composition and Composition Consistency Checking.

Specific products, or configurations of a feature-oriented decomposition, are obtained by means of selecting a set of features. Thus, it would desirable a feature-oriented language would provide language constructs for specifying which specific features should be instantiated, or composed, when producing a specific product. This should be done at the feature-level, specifying which *feature modules* should be included, instead of having to specify individual elements of each feature. For instance, if the LightMng feature has been selected, we would like to specify the LightMng package should be included, according to Figure 2, instead of having to specify the LightMng::Gateway feature and the LightCtrl class must be added individually to the final product.

Moreover, neither all feature combinations can be composed safely nor all composition ordering are valid. For instance, if the SmartEnergyMng feature is selected, the WindowMng and HeaterMng features should also be selected. Moreover, WindowMng and HeaterMng must be composed before SmartEnergyMng is composed, since the latter one depends on the former ones. A feature-oriented language should be aware of these constraints, avoiding invalid product can be created and managing automatically dependencies whenever possible. For instance, when SmartEnergyMng feature is selected, the language should detect the WindowMng and HeaterMng should be also composed and, in addition, before the SmartEnergyMng feature.

So summarising, a feature-oriented language must provide:

1. Feature Extension by addition and substitution.
2. Modules to group and encapsulate feature elements.
3. Automatic dependency management at intra-feature level.
4. Composition at the feature level.
5. Composition consistency checking.
6. Automatic management of constraints at the feature level.

Next section will evaluate how C# partial classes are able to manage these issues.

4. IMPLEMENTING THE SMART HOME MODELS USING C# PARTIAL CLASSES

Describir como se implementan diferentes situaciones en el modelo de la SmartHome.

4.1 C# partial classes

4.2 Scenarios 1: ...

Descripción de como implementar algo tal como una feature que extiende de otra.

4.3 Scenarios 2: ...

Descripción de como implementar otra cosa, tal como una feature que extiende de dos.

4.4 Scenarios 3: ...

Otro caso diferente ...

4.5 Conclusions

Tabla o bullets comparando lo que querríamos tener y lo que tenemos con las clases parciales.

5. SKETCH OF SOLUTIONS FOR C# PARTIAL CLASSES PITFALLS

Breves comentarios sobre como solucionar estos problemas. (NOTA: No estoy muy seguro acerca de si poner o no esta sección).

6. SUMMARY AND FUTURE WORK

7. REFERENCES

[1] Joseph Albahari and Ben Albahari. *C# 3.0 in a Nutshell.* O'Reilly, 4 edition, January 2010.

[2] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of CaesarJ. 3880:135–173, 2006.

[3] Don S. Batory. Feature Models, Grammars, and Propositional Formulas. In J. Henk Obbink and Klaus Pohl, editors, *Proc. of the 9th Int. Conference on Software Product Lines (SPLC)*, volume 3714 of *LNCS*, pages 7–20, Rennes (France), September 2005.

[4] Lidia Fuentes, Carlos Nebrera, and Pablo Sánchez. Feature-Oriented Model-Driven Software Product Lines: The TENTE Approach. In Eric Yu, Johann Eder, and Colette Rolland, editors, *Proc. of the Forum of the 21st Int. Conference on Advanced Information Systems (CAiSE)*, volume 453 of *CEURS Workshops*, pages 67–72, Amsterdam (The Netherlands), June 2009.

[5] Iris Groher and Markus Völter. Aspect-Oriented Model-Driven Software Product Line Engineering. *Transacations on Aspect-Oriented Software Development (Special Issue on Aspects and Model-Driven Engineering)*, 6:111–152, 2009.

[6] Miguel A. Laguna, Bruno González-Baixauli, and José M. Marqués. Seamless Development of Software Product Lines. In *Proc. of the 6th Int. Conference on Generative Programming and Component Engineering (GPCE)*, pages 85–94, Salzburg (Austria), October 2007.

[7] Carlos Nebrera, Pablo Sánchez, Lidia Fuentes, Christa Schwanninger, Ludger Fiege, and Michael Jäger. Description of Model Transformations from Architecture to Design. Deliverable D2.3, AMPLE Project, September 2008.

[8] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects.

[9] Pablo Sánchez, Nadia Gámez, Lidia Fuentes, Neil Loughran, and Alessandro Garcia. A Metamodel for Designing Software Architectures of Aspect-Oriented Software Product Lines. Deliverable D2.2, AMPLE Project, September 2007.

[10] Pamela Zave. FAQ Sheet on Feature Interaction. AT&T, `http://www2.research.att.com/~pamela/faq.html`, 1999.