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) [12] 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, 14].

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 [9], 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 [6, 5, 13, 11] [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 [11, 5] and a feature-oriented implementation in CaesarJ [2]. This case study covers different kinds of variability [13].

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`

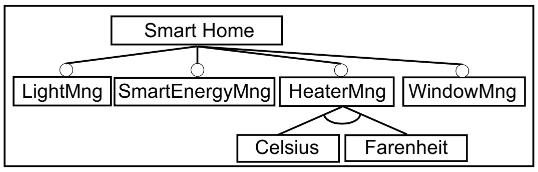

Figure 1: Smart Home feature model

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 5 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] [6, 5, 13, 11]. 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 [13].

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 [13, 11]. 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.

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 [12] 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 [10, 8, 4, 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 [14, 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

[3]`http://personales.unican.es/sanchezbp/CaseStrudies/SmartHome`
[4]`http://www.ample-project.net`

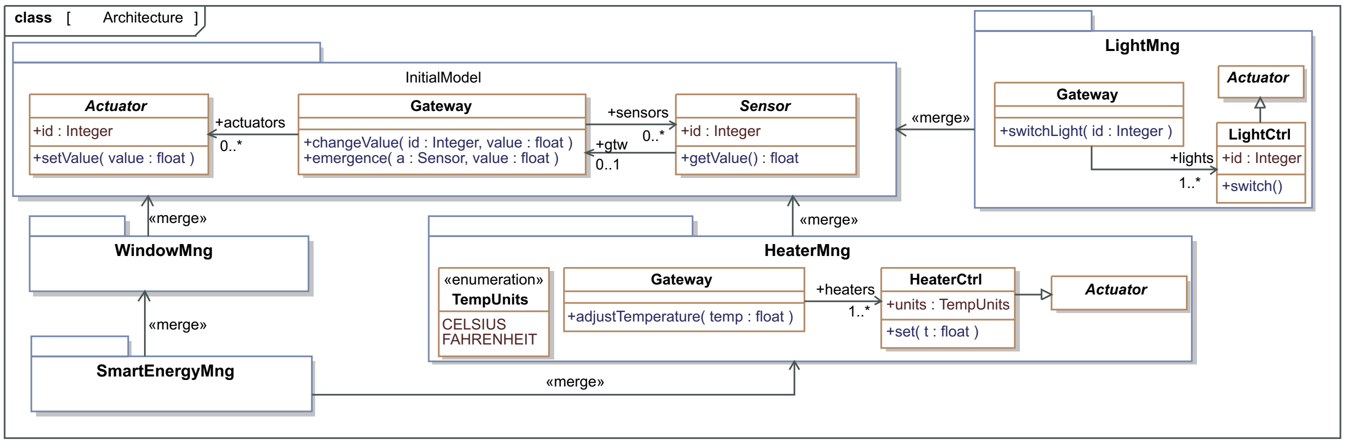

Figure 2: Smart Home design

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 [7] are able to automatically update these dependencies due to the usage of an advanced type system based on virtual classes 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

This section describes the results of our experiment about using C# partial classes to implement the feature-oriented design of the Smart Home case study. Before describing them, we provide some background on how C# partial classes work.

4.1 C# partial classes

C# partial classes allow developers to split the implementation of a class into several files, each one containing an slice

```
File InitialModel/Gateway.cs
--------------------------------------------------------
00 namespace SmartHome {
01  public partial class Gateway {
02    protected List<Sensor> sensors;
03    protected List<Actuator> actuators;
04
05    public void emergence(Sensor s, double value) {...}
06    public bool changeValue(int id, double value) {...}
07  } // Gateway
08 }// namespace

File LightMng/Gateway.cs
--------------------------------------------------------
09 namespace SmartHome
10 {
11    public partial class Gateway {
12        protected List<LightCtrl> lights;
13
14        public bool switchLight(int id) {...}
15    } // Gateway
16 } // namespace

File SmartHome.csproj
--------------------------------------------------------
17 </Project>...<ItemGroup>
18    <Compile Include="InitialModel\Gateway.cs" />
19    <Compile Include="LightMng\Gateway.cs" />
20 <!--  <Compile Include="SmartEnergyMng\Gateway.cs" />
21      <Compile Include="WindowMng\Gateway.cs" /> -->
22 ...
23 </ItemGroup></Project>
```

Figure 3: Gateway **implementation using partial classes**

of the global functionality of a class. All these slices are combined at compilation time to create a single class, containing all the functionality specified in the partial classes.To be combined, all partial classes need to belong to same *namespace*, have the same visibility and have been declared as partial by means of using the partial keyword. A *namespace* is simply used to group related classes and avoid name collisions.

What files must be included in a compilation unit is specified in C# using a XML document which contain information about the project and it specifies which files must be included. Therefore, we can manipulate this file to include partial classes as desired and to produce composed classes with the desired functionality.

Figure 3 shows an example of usage of partial classes where the implementation of the Gateway class for the InitialModel and LightMng feature has been split into two separate files with the same name, but placed in different directories (Figure 3 lines 00-08 and lines 09-16). The Gateway class for the InitialModel feature (Figure 3 lines 00-08) contains collections for sensors and actuators, as well as the emergence and changeValue methods (see Figure 2). The Gateway class for the LightMng feature (Figure 3 lines 09-16) adds to the previous Gateway class the collections for lights as well as the switchLight method.

Figure **??** shows a possible build file which specifies the Gateway partial classes for the InitialModel and the LightMng features must be included in the compilation; but the corresponding partial classes for the WindowMng or SmartEnergyMng features, oppositely, must be excluded. Therefore, the compiler will produce a Gateway class with functionality

```
File InitialModel/Gateway.cs
--------------------------------------------------------
01 public partial class Gateway {
02   ...
03   public Gateway() {
04     this.actuators = new List<Actuator>();
05     this.sensors = new List<Sensor>();
06   }
07 }
File LightMng/Gateway.cs
--------------------------------------------------------
08 public partial class Gateway {
09   protected List<LightCtrl> lights;
10   public Gateway() {
11       this.lights = new List<LightCtrl>();
12   }
13 }
```

Figure 4: Gateway **implementation using partial classes**

to manage lights, but not to manage windows, heaters or smart energy management.

Motivated by these results, some authors, such as Laguna et al [9] proposed to used C# partial classes as a mechanisms to implement feature-oriented design. Next section will evaluate strengthens and weaknesses of this approach using the elements presented in Section 3.

4.2 Evaluation

Feature Encapsulation and Extensibility.

C# partial classes does not seem to provide any mechanism to group an encapsulate features. We might use namespaces for this purpose. We might opt for placing all classes related to the InitialModel and LightMng features in the SmartHome.InitialM and SmartHome.LightMng namespaces respectively. Nevertheless, in this case the Gateway partial classes belonging to separate features will be not combined. Thus, partial classes force us to place all classes in a same namespace, so we can not use this mechanisms to group feature-related code.

Any other workaround, such as creating one coarse-grained class per feature and managing feature-specific classes as internal classes of the coarse-grained one will not work either, since the outer class serves as namespace for the internal classes. Thus, each internal class, although declared as partial, it is placed in a different namespace and they are not merge.

Regarding extensibility, partial classes allows us to add new attributes and methods to existing partial classes, but they do not allow us to extend or override existing ones. For instance, in the LightMng feature, we have added a new attribute lights to the Gateway class. So, we should extend the Gateway constructor to appropriately initialize this attribute. So, we write the code depicted in Figure 4. This excerpt of code contains the constructor for the most basic Gateway class (Figure 4, lines 03-06), which is extended with new functionality in the LightMng feature (Figure 4, lines 03-06). Nevertheless, this is not possible, since the compiler reports an error because the method Gateway(), i.e. the class constructor, is duplicated. This means we can not split the implementation of a method into several files, since we can not have methods with the same signature into two separate files of a partial class. This reduces extensibility to the addition of new methods and attributes (as shown in Figure 3), but we cannot add functionality to existing methods. It would be interesting if C# supported *partial methods* in addition to partial classes.

In the case of extensibility by substitution, the problem will be the same. Let us consider now the case of the SmartEnergyMng feature. The Gateway class for this feature must override, for instance, the adjustTemperature method to check if additional operations involving the windows must be executed before adjusting the temperature. Nevertheless, since we can not declared a method adjustTemperature in the partial class belonging to the SmartEnergyMng feature with the signature as the method declared for the HeaterMng feature, there is no way to override the latter one. Indeed, normal object-oriented overriding is somehow more complex than usual in C# because, as in C++, for a method being effectively overridden, we need to declared it as *virtual*. Thus, we need somehow to foresee a method is going to be overridden to declare it as virtual, or declare all methods as virtual, independently if we know they are going to be overridden or not. This last practice is the recommended one by authors, although it is tedious and verbose.

Regarding automatic management of dependencies between classes, C# partial classes do not present the problem described in Section 3. The problem was a class A, e.g. Sensor in Figure 2, has a reference to a class B, e.g. Gateway, which is refined by means of inheritance through different features. When a final product is created, this reference should be changed to the deepest child belonging to a selected feature in the inheritance tree to avoid castings and other problems related to the type system. This problem does not appear when using C# partial classes because we are not using subclassing by inheritance. We are simply merging slices of the same class, so the type of a class is not changed although a class is refined through several features. Moreover, when a instance of a class, e.g. Gateway, is created, this instance will be an instance of a class containing the functionality belonging to the all the selected features. So, regarding this particular point, C# partial classes seems to provide a performance similar to languages such as CaesarJ or ObjectTeams.

Feature Level Composition.

Since they are not specific mechanisms to group and encapsulate a bundle of classes related to a same feature, it is not possible either to compose products at the feature level. Composition of specific products is achieved by means of including/excluding (partial) classes from the compilation unit, such as shown Figure 3. This means we need to manage the elements belonging to a same feature individually, and take care of producing consistent configurations. For instance, we should check the LightCtrl class is included in the compilation unit if and only if the partial class of Gateway belonging to the LightMng feature is also included. So, several actions must be carried out each time a feature is selected and deselected.

In the same way, C# does not provide explicit mechanisms to guarantee a build file contains a valid selection of features. Nevertheless, if a feature uses classes and methods declared in another feature, the compiler will report an error about we are using some undefined elements. For instance, if the Gateway partial class belonging to the SmartEnergyMng feature use some classes called WindowCtrl and HeaterCtrl and we have not included them in the compilation unit, the compilation process will fail until we include these classes in the compilation unit. Nevertheless, we should not include only these classes, we should also include all the classes belonging to the LightMng and HeaterMng features. So, using C# partial classes we can detect dependencies between classes, but not dependencies at the feature level, which would be the ideal case. If errors are detected, the compiler reports it, but it need to be solved manually, i.e. if the feature SmartEnergyMng is included, the WindowMng and HeaterMng features need to be manually included. Finally, there is not support to detect and solve mutual exclusion constraints, i.e. situations where if a feature A is selected, a feature B must not be selected. Anyway, to the best of our knowledge, this kind of support is rarely found in feature-oriented languages.

4.3 Conclusions

According to the results shown in previous section, we must conclude C# partial classes are not a suitable mechanism to achieve feature-oriented programming. The main problems come from:

1. Lack of support for feature modules and feature encapsulation.
2. Lack of support for extending existing methods.
3. Lack of support for overriding existing methods.
4. Lack of support for composition at the feature level.

5. SUMMARY AND FUTURE WORK

6. 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] Wonseok Chae and Matthias Blume. Language Support for Feature-Oriented Poduct Line Engineering. In Sven Apel, William R. Cook, Krzysztof Czarnecki, Christian Kästner, Neil Loughran, and Oscar Nierstrasz, editors, *FOSD*, pages 3–10, Denver (Colorado, USA), October 2009.

[5] 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.

[6] 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.

[7] Stephan Herrmann. A precise model for contextual roles: The programming language objectteams/java. *Applied Ontology*, 2(2):181–207, 2007.

[8] Kyo Chul Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, July-August 2002.

[9] 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.

[10] Roberto E. Lopez-Herrejon, Don S. Batory, and William R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In Andrew P. Black, editor, *Proc. of the 19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 169–194, Glasgow (United Kingdom), July 2005.

[11] 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.

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

[13] 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.

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