From Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring

Roberto E. Lopez-Herrejon
Systems Engineering and Automation
Johannes Kepler University Linz, Austria
roberto.lopez@jku.at

Leticia Montalvillo-Mendizabal
Universitat Politecnica de Catalunya
Barcelona, Spain
leticia.montalvillo@est.fib.upc.edu

Alexander Egyed
Systems Engineering and Automation
Johannes Kepler University Linz, Austria
alexander.egyed@jku.at

***Abstract*—More and more frequently successful software systems need to evolve into families of systems, known as Software Product Lines (SPLs), to be able to cater to the different functionality requirements demanded by different customers while at the same time aiming to exploit as much common functionality as possible. As a first step, this evolution demands a clear understanding of how the functional requirements map into the features of the original system. Using this knowledge, features can be refactored so that they are reused for building the new systems of the evolved SPL. In this paper we present our experience in refactoring features based on the requirements specifications of a small and a medium size systems. Our work identified eight refactoring patterns that describe how to extract the elements of features which were subsequently implemented using Feature Oriented Software Development (FOSD) – a novel modularization paradigm whose driving goal is to effectively modularize features for the development of variable systems. We argue that the identification of refactoring patterns are a stepping stone towards automating Feature-Oriented Refactoring, and present some open issues that should be addressed to that avail.**

***Keywords*-Software Product Lines; Feature Orientation; Product Line Evolution; Feature Oriented Refactoring**

I. Introduction

Today, software systems are more frequently being built as families of systems also known as *Software Product Lines (SPLs)* [1]–[3]. In a SPL, each member product implements a different combination of *features* – increments in program functionality [4]. The success of a SPL lies at the effective management and realization of its *variability*, defined as the capacity of software artifacts to vary [5][44]. Extensive research and practice have been documented that corroborates the signifcant benefits of applying SPL practices both in academia and industry [2], [3], [6].

In this paper, we present our experience in refactoring features from the requirements specifications of two single software systems (i.e. without variability). From these systems, we manually extracted the features using a three-steps refactoring process and created a SPL for each of them. This work identified eight refactoring patterns that describe how to extract the code fragments that realize the features which were subsequently implemented with an advanced modularization paradigm. Furtheremore, a priority-based scheme is proposed to disambiguate the refactoring of code elements that are shared by more than one feature. We analyze the issues encountered and present a set of open issues that must be addressed towards automating feature refactoring. Next we shortly present the foundations of the modularization approach we used for our study.

II. Feature Oriented Software Development

Feature Oriented Software Development (FOSD) provides formalisms, methods, languages and tools for building variable, customizable and extensible software [7]. FOSD has been successfully used in several case studies [8], [9]. FOSD advocates modularizing *features*, increments in program functionality [4], as the systems building blocks. At the heart of FOSD is a feature algebra that drives the (de)composition of software artifacts [1], [10]–[12]. A *feature module* contains all the software artifacts, or parts thereof, required for implementing the corresponding feature.

In FOSD features are composed hierarchically starting from the root element of the corresponding models. Elements that have the same name and type at the same hierarchical level are composed together, elements that do not have a corresponding matching element are copied along hierarchically. FEATUREHOUSE [13] is a general architecture of software composition supported by a framework and tool chain and is a descendant of Batory's AHEAD program generator [1]. FEATUREHOUSE provides facilities for feature composition based on a language-independent model of software artifacts and an automatic plug-in mechanism for the integration of new artifact languages.

Figure 1 sketches how feature composition works in FEATUREHOUSE. Lines 2-9 shows the contents of class `C` in feature `A`, two field variables and a method. Lines 11-21 show a class refinement that has one field variable, a refinement to method `m` that uses keyword `original`, and a new method `m2`. The composition yields a class `C` that has the three fields from both features (lines 24-26). It also creates a method `m` from the version of feature `B` (lines 27-32) but substituting the keyword `original` with the statements of the method it is refining[1], namely `stmt1` and `stmt2` (lines 29-30). Finally method `m2` from feature `B` is copied along to the composed class (lines 33-35). For further details on FOSD and supporting tools please refer to [1], [13].

[1]The actual composition result is more elaborate but its functionality is comparable to the code snippet shown in the figure.

```
// Feature A
public class C {
 int f1;
 double f2;
 public void m() {
     stmt1;
     stmt2;
 }
}
// Feature B
public class C {
 float f3;
 public void m() {
     stmt3;
     original();
     stmt4;
 }
 public void m2() {
   stmt5;
 }
}
// Composed class C
public class C {
 int f1;            // feature A
 double f2;         // feature A
 float f3;          // feature B
 public void m() {
     stmt3;
     stmt1;   // original
     stmt2;   // original
     stmt4;
 }
 public void m2() { // feature B
   stmt5;
 }
}
```

Fig. 1: FEATUREHOUSE composition example

III. Feature Oriented Refactoring Process

Feature Oriented Refactoring (FOR) is the process of decomposing a program into features and modularizing them using advanced modularization paradigms [14]. In this section, we describe the process followed to refactor the two projects presented in Section IV. For our study we used FEATUREHOUSE as implementation platform because it supports FOSD for current versions of Java language [13]. We start by describing the modularization steps we followed, and then present our strategy to deal with cases where a program element was shared by more than one feature.

A. Modularization steps

Our ultimate goal was to modularize systems using FOSD. Our starting point was a list of systems requirements described in prose and the source code which in most of the cases provided little or no additional documentation information. We identified three steps to modularize a program into feature modules that are illustrated in Figure 2. The first two: (1) identifying features, and (2) making decisions are iterative processes, meaning that once a feature fragment is identified in source code we might have to make a decision and continue with the identification process for other feature fragments. These three steps are:

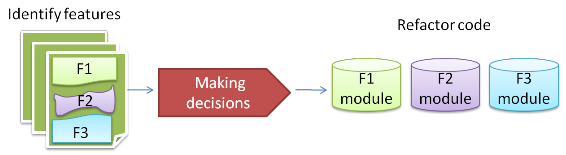

Fig. 2: Modularization steps diagram

1) *Identify features in code*. In this step we create conceptual traces between the requirements given and the source code provided. In our study we consider the requirements as the features that should be modularized. For this identification we followed as guidance: the naming patterns of the program elements, the relationship between requirements and GUI elements, and code inspections.
2) *Making decisions*. A piece of code may be identified to belong to more than one feature. A common reason for this is because such piece of code is the manifestation of an interaction among the features involved. At this moment a decision has to be made for the modularization to proceed. In Section III-B we describe the approach we followed.
3) *Refactor source code*. At this point, we know exactly which code implements each feature. Therefore, we implement it according to a modularization paradigm, which in our case was FEATUREHOUSE. In performing this step, we collected eight refactoring patterns that try to maintain the original implementation structure as much as possible while implementing the required functionality in our developing platform.

B. Feature Module Disambiguation

In several instances, deciding in which features to modularize a code fragment is not straightforward process. A code fragment can effectively implement parts of one or more features. We say that those fragments are *shared* by the features and reflect their interaction. Nonetheless, a decision on where to modularize a code fragment has to be taken. We applied a priority-based scheme where each feature that shares a give piece of code is assigned a priority; the feature with the highest priority is selected to modularize it. For the systems we study, the priority scheme was as follows: *i)* high priority root feature, *ii)* medium priority mandatory features, and *iii)* low priority optional features.

IV. Feature oriented refactoring study

In order to help us gauge the applicability of our FOR process, we use two distinct system projects that are publicly available as single systems, that is, without any consideration whatsoever of variability. In this section we describe the main characteristics of these two systems, their structure and the refactoring patterns that we found in them. We selected these two systems because of their size (small and medium) and availability of documentation and other related program analysis information which we will exploit as part of our future work.

A. *Video-On-Demand Player (VODPlayer) Project*

`VODPlayer` stands for a Video On Demand Player. This project is a program which allows users connected to a server to watch short movie clips.

1) System description: In this system, there are 13 requirements considered as shown in Table I. As indicated in the previous section, for the purpose of our study we equated the term functional requirement with feature. Thus, this table shows both the name of the requirement (e.g. `R4`) and its corresponding implementing feature (e.g. `Pause`). Notice that in the second column the description and functionality of each feature is provided at a conceptual level. But this is not enough, it is also important to define which elements at code level actually implement a feature. The last column of Table I shows the criteria used for selecting feature code. Requirements `R5`, `R6` and `R7` are non-functional requirements and as such they do not have a corresponding refactoring criterion. Based on the feature descriptions, we developed a SPL by making some of the features optional and considering feature combinations that are meaningful for this domain. The feature model we elaborated for `VODPlayer` is shown in Figure 3.

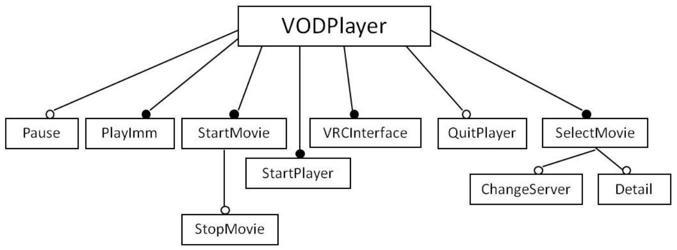

Fig. 3: Feature Diagram of VODPlayer

`VODPLayer` is implemented in Java with a total of 3.6KLOC, across 42 classes of which 12 are listeners for the GUI elements. There are four main classes which are the core frames of the application. We refer to these classes when we illustrate our refactoring patterns. These classes are:

- *`VODClient`*: is the main frame of the application and appears when users connect to the application.
- *`ListFrame`*: this frame is displayed after selecting movies (clicking the `Movies` button) in `VODClient` frame. From this frame, users can connect to other servers, select a movie, display its details or close the frame.
- *`Detail`*: this frame is displayed after clicking `Detail` button on `ListFrame` frame and displays information about the movie selected.
- *`ChangeServer`*: this frame is displayed after clicking `Servers` button on `ListFrame` frame and allows users to connect to another server to get a new movie list.

2) Refactoring Patterns: In this subsection we present and illustrate the five refactoring patterns we found in `VODPLayer`. For simplicity, we follow a similar notation to Alves et al. [15]. We use the term `C` to refer to a class, `fs` for a set of class fields, `ms` a set of methods in a class, `T` for a

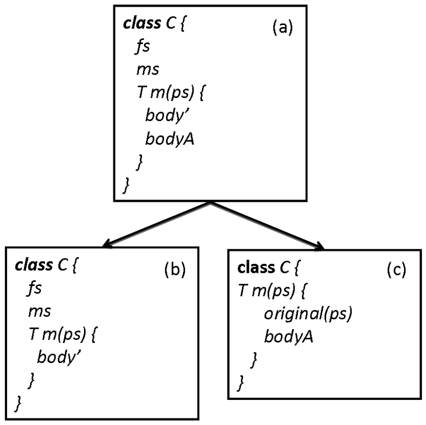

Fig. 4: Addition at end of method: (a) original, (b) refactored and (c) refining classes

```
 1  // Original VODClient
 2  public final void stopmovie() {        // T m(ps)
 3    if (curmovie == null) return;        // body'
 4    buttonControl2.setLabel("PLAY");     // bodyA
 5  }
 6  // Refactored VODCLIENT  Feature StartMovie
 7  public final void stopmovie() {        // T m(ps)
 8    if (curmovie == null) return;        // body'
 9  }
10  // Refining class VODClient Feature StopMovie
11  public final void stopmovie(){         // T m(ps)
12     original();                      //original(ps)
13     buttonControl2.setLabel("PLAY");      // bodyA
14  }
```

Fig. 5: Example addition at end of method

type, `ps` a method parameter set, and terms with prefix `body` to denote sets of statements that appear in a method.

Addition at end of method. This pattern allows adding a piece of code at the end of a method. Figure 4(a) shows the method `m` in class `C` to be refactored. The result of the identification process is a piece of code that must be refactored into its associated feature (e.g. feature `A`), this is denoted with term `bodyA`. The term `body'` denotes the fragment of method `m` that executes before `bodyA`. Figure 4(b) depicts the refactored class, now only with `body'` in method `m`, while Figure 4(c) shows the refining class with the refinement of method `m`. Notice here that the refining class calls special method `original`. Note also that, `bodyA` cannot use any variable defined for `body'`, it can only use variables defined in itself, `ps` parameter(s) and `fs` variables of class `C`.

Figure 5 illustrates this pattern applied to method `stopmovie`. Notice that the original method body has two elements a begin (`body'`) and an end (`bodyA`). According to our pattern, this method is refactored in two pieces. In this case the refactored class is contained in `StartMovie` feature (lines 6-9), and the refining class in `StopMovie` feature (lines 10-14).

Addition anywhere with a hook method. This type of refactoring pattern allows to add pieces of code in any state-

TABLE I: VOD Player Requirements, Features, and Refactoring Criteria

Requirement/Feature	Requirement description	Refactoring Criteria
R1/SelectMovie	Display a list of movies and select one	Code that allows getting & displaying a list of movies and selecting one
R2/PlayImm	Play movie immediately after selection	Code enabling playing a movie immediately after it's selected
R3/Detail	Display textual movie information	Every code line, button, attribute that displays a movie detail frame
R4/Pause	Pause a movie	Code and buttons that enables a movie to be paused
R5/-	3 seconds max to load movie list	*Non-functional requirement*
R6/-	3 seconds max to load movie textual	*Non-functional requirement*
R7/-	1 second max to start playing a movie	*Non-functional requirement*
R8/VRCInterface	Provide VCR-like user interface	General graphic player UI. Dimensions of the frame, general buttons & labels
R9/StopMovie	Stop a movie	Piece of code & buttons that enables a movie to be stopped
R10/StartMovie	Start a movie	Code making possible watching a movie. Load movie
R11/ChangeServer	Change server	Code making possible a change of the server
R12/QuitPlayer	Exit the player	Code lines that enable closing and quitting the VOD Player
R13/StartPlayer	Start the movie player	The implementation concerning the running of the VOD Player

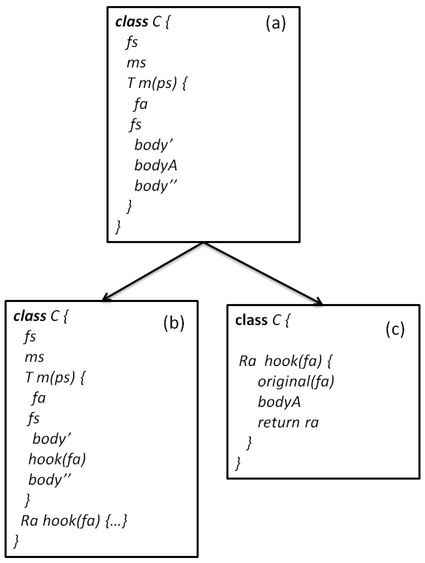

Fig. 6: Addition anywhere with a hook method: (a) original, (b) refactored and (c) refining classes

```
 1  // Original VODClient bc2AP
 2  void bc2AP(ActionEvent actionEvent)   //T m(ps)
 3  {
 4   if (bc2.getLabel().equals("PLAY")){  //body'
 5     if (curmovie == null)              //body'
 6         return;                        //body'
 7     vthread = new Thread(video);       //body'
 8     vthread.start();                   //body'
 9    bc2.setLabel("PAUSE");              //bodyA
10   }
11     ...                                //body''
12  }
13  //Refactored class VODClient feature StartMovie
14  void bc2AP(ActionEvent actionEvent){   // T m(ps)
15   if (bc2.getLabel().equals("PLAY")){   //body'
16     if (curmovie == null)               //body'
17         return;                         //body'
18     vthread = new Thread(video);        //body'
19     vthread.start();                    //body'
20     setLabelPause();                    //hook(fa)
21     }
22      ...                                //body''
23  }
24  void setLabelPause(){}                 //hook(fa)
25  // Refining class VODClient feature Pause
26  void setLabelPause(){             // Ra hook(fa)
27   original();                      //original(fa)
28   bc2.setLabel("PAUSE");           //bodyA
29  }
```

Fig. 7: Example addition anywhere with hook method

ment where it is needed. Consider the method depicted in Figure 6(a). In method `m`, the piece of code denoted with `bodyA` belongs to feature `A` and consequently must be refactored into that feature. Therefore this class must be decomposed in the refactored class shown in Figure 6(b) and the refining class of Figure 6(c). Please notice that `bodyA` is between `body'` (the code of the method which is executed before) and `body"` (the code which is executed after). This pattern replaces `bodyA` with a call to a new method `hook(fa)` which can be subsequently refined by feature `A` to execute the original functionality of `bodyA`. Note that this method has a parameter `fa` because it is common that hook methods make reference to some variables local to method `m`. Similarly, sometimes it is also necessary for these methods to return a value, denoted in `ra` of type `Ra`. Figure 6(b) shows this replacement and the new method, and Figure 6(c) illustrates the refinement to method `hook(fa)` done by feature `A` and expressed with the `original` keyword.

Figure 7 illustrates the refactoring for method `bc2AP`[2] in `VODClient` class (lines 1-12). At the identification step we found the code belonging to feature `Pause` in line 9. This line of code is refactored by introducing `setLabelPause` hook method in the refactored class, in our example in feature `StartMovie` (lines 13-24). Note that it also holds the implementation of the empty hook method (line 24). Finally, the refining class is contained in feature `Pause` (lines 25-29). Notice here the use of the `original` keyword in line 27 that denotes a method refinement.

Move entire method. This pattern refactors a method in a class by removing it from the refactored class and adding a copy to the refining class. Figure 8(a) depicts class `C`

[2]The methods names are shorten for simplicity. For instance, the name of method `mA` is `buttonControl2_actionPerformed`.

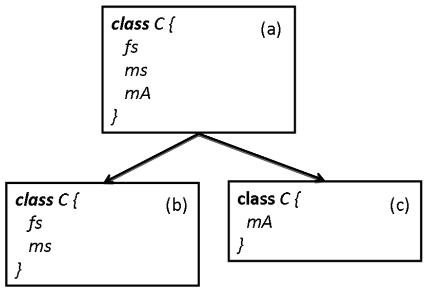

Fig. 8: Move entire method: (a) original, (b) refactored and (c) refining classes

```
 1  // Original class
 2  public class VODClient extends Frame {
 3   boolean isStandalone;     // fs
 4   BorderPanel bevelPanel2;
 5   ...
 6   public VODClient() { ... }
 7   void bc1AP(ActionEvent actionEvent){...}  // ms
 8   void bc3AP(ActionEvent actionEvent){...}
 9   void bc4AP(ActionEvent actionEvent){...}  // mA
10   ...
11  }
12  // Refactored class
13  public class VODClient extends Frame {
14   boolean isStandalone;     // fs
15   BorderPanel bevelPanel2;
16   ...
17   public VODClient() { ... }
18   void bc1AP(ActionEvent actionEvent){...}  // ms
19   void bc3AP(ActionEvent actionEvent){...}
20   ...
21  }
22  // Refining class in feature QuitPlayer
23  public class VODClient extends Frame {
24   void bc4AP(ActionEvent actionEvent){...}  // mA
25  }
```

Fig. 9: Example move entire method

with fields `fs`, methods `ms`, and method `mA` which has been identified as belonging to feature `A`. The original class turns into: *i)* refactored class containing both fields `fs` and methods `ms` as shown in Figure 8(b), and *ii)* refining class with method `mA` as depicted in Figure 8(c).

At this moment, it is important to draw a parallel with standard object-oriented refactoring in particular with the *move method* pattern [16]. The crucial difference is that in our case the method moves across features. In other words, the method stays in the same class `C` but it moves from its containing feature (or legacy code like in our study) to a new feature such as `A`.

An example of this pattern is method `bc4AP`in class `VODClient` of the original system shown in lines 1-11 of Figure 9. This method was identified as being in charge of quitting the VOD player when button `Quit` is hit, and consequently refactored to feature `QuitPlayer` as illustrated in lines 22-25 of Figure 9.

Move field. This pattern refactors a field from a class and puts it into the refinement of the given class. For example,

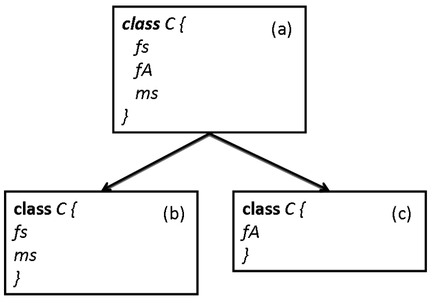

Fig. 10: Move field: (a) original, (b) refactored and (c) refining classes

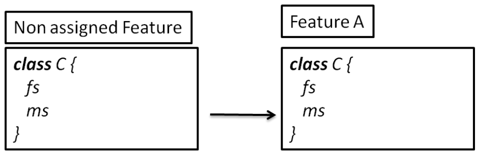

Fig. 11: Move entire class

class `C` depicted in Figure 10(a) describes the class `C` with its fields `fs`, methods `ms`, and field `fA` which has been identified to belong to feature `A`. Again, the original class is divided into two classes: refactored class depicted in Figure 10(b) and the refining class depicted in Figure 10(c) with field `fA`. The use of this pattern is frequently accompanied with instances of the previous pattern that moves entire methods.

Examples of this pattern are the refactorings of fields in class `ListFrame` that correspond to the visual elements (e.g. buttons and labels) of this frame class. Because these variables form part of the GUI they were moved to feature `VCRInterface`.

Move entire class. This pattern refactors a class from original code to a feature as described in Figure 11. On the left we have the original class with its methods and field, and on the right the same class with its elements but now in a new containing feature `A`. It should be noted though that in order to perform this refactoring, both methods `ms` and fields `ms` must all be identified to belong in feature `A`.

Examples of this refactoring pattern are the classes that implement the distinct listeners required to react and respond to events in the GUI. For instance, `DetailListener1` that corresponds to the listener for the `Detail` frame which is refactored to feature `Detail`.

B. Gantt Project

Gantt Project is a cross-platform desktop tool for project scheduling and management [17]. It is free and open source. The main objective for this project was to validate the applicability of our work to systems of larger size. We found many instances of the patterns identified in `VODPlayer`, some of which allowed us to extend and generalize these patterns. But also this project provided examples of three new patterns.

1) System Description: The system is implemented in Java in a total of approximately 41 KLOC with 479 classes modularized in 43 packages. Among other functionality, Grant-

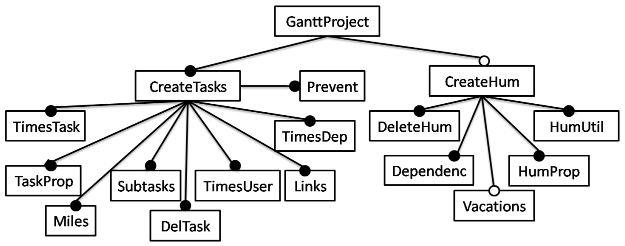

Fig. 12: Gantt Project feature model

Project allows users to: *i)* create work breakdown structure, draw dependencies and define milestones, *ii)* manage human resources and their work on tasks, *iii)* manage PERT charts, *iv)* export and import charts in several file formats (e.g. PDF and HTML), *v)* share projects using WebDAV. Our study considered 16 requirements which are summarized in Table II. As before, based on the feature descriptions we developed a SPL by making some of the feature optional and considering feature combinations that are meaningful for this domain. The feature model we obtained for this system is shown in Figure 12.

2) Refactoring Patterns: In this subsection we describe the three new patterns our work identified.

Addition at beginning of method. Figure 13 shows the representation of this pattern. It follows the same idea that addition at the end of the method, but in this case instead of adding code at the end, it adds at the beginning. Figure 13(a) shows the method `m` in class `C` to be refactored, with the piece of code `bodyA` that belongs to feature `A`. The term `body″` denotes the fragment of method `m` that executes after `bodyA`. Figure 13(b) depicts the refactored class, now only with `body′` in method `m`, while Figure 13(c) shows the refining class with the refinement of method `m`. Notice here that the refining class calls special method `original`. Note also that `bodyA` can only use variables defined in itself, `ps` parameter(s) and `fs` variables of class `C`, and that `bodyA` cannot define any variable that `body″` uses. This is because if feature `A` is not included in the composition then `body″` will be referring to a non existing variable.

Figure 14 shows an example of the refactoring pattern applied to method `changeLanguageOfMenu()` (lines 1-12). The original method has element `bodyA` (lines 3-5) , which goes at the beginning of the method, and the rest of the body belongs to element `body″` (lines 6-11). The refactoring process divides the original method into two pieces. The refactored class is contained in feature `GanttProject` (lines 13-21) while the refining class is in feature `CreateTask` (lines 22-29). Notice the position of keyword `original` at the end of the refining method.

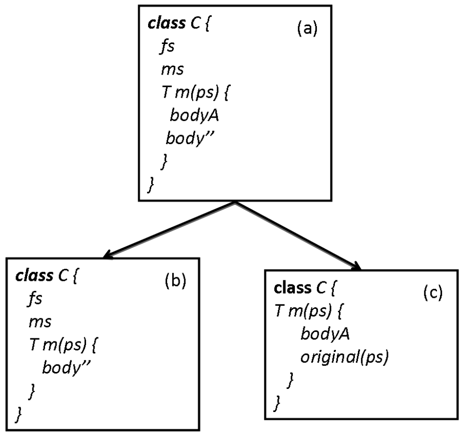

Fig. 13: Refactoring addition at beginning of method: (a) original, (b) refactored and (c) refining classes

```
1  // Original GanttProject
2  void changeLanguageOfMenu() {
3   bNewTask.setToolTipText(getToolTip      //bodyA
4      (correctLabel(
5         language.getText("createTask"))));
6   ...
7   getTabs().setTitleAt(1,                  //body''
8      correctLabel(
9           language.getText("human")));
10  setButtonText();                         //body''
11  toolBar.updateButtonsLook();             //body''
12 }
13 // Refactored Feature GanttProject
14 void changeLanguageOfMenu(){    // T m(ps)
15  ...
16  getTabs().setTitleAt(1,                  //body''
17     correctLabel(
18          language.getText("human")));
19  setButtonText();                         //body''
20  toolBar.updateButtonsLook();             //body''
21 }
22 // Refining Feature CreateTask
23 void changeLanguageOfMenu(){        // T m(ps)
24  bNewTask.setToolTipText(getToolTip     //bodyA
25     (correctLabel(
26        language.getText("createTask"))));
27  ...
28  original();                          //original(ps)
29 }
```

Fig. 14: Example addition at beginning

Overwrite method. For instance, we have a class `C` depicted in Figure 15 (a) which has a method `m` with parameters `ps` and return type `T`. This pattern refactors class `C` into two classes, that have both the refactored method `m`. However, they define different method bodies and return expressions, Figure 15(a) and Figure 15(b). Notice that in none of them appears the keyword `original`. This means that when both classes are composed together, one method overwrites the meaning of the other depending on the order in which they are composed [7].

Figure 16 illustrates an example of the overwrite refactoring pattern in `getTaskManager` method (lines 1-4). Note that for this method, we just have the element return `v` in the original class (line 3), thus there is no body for this example. This method is also refactored in two pieces. One method is contained in feature `GanttProject` feature (lines 5-8), and the second method is contained in feature `TaskProp`

TABLE II: Gantt Project Requirements, Features, and Refactoring Criteria

Requirement/ Features	Requirement description	Refactoring Criteria
R1/CreateTask	Create Tasks	Code enabling tasks creation
R2/DeleteTask	Delete Tasks	Code enabling tasks deletion
R3/TaskProp	Maintain Task Properties	Code that maintains task properties
R4/Subtask	Add/Remove Tasks as Subtasks	Code that creates and deletes tasks as subtasks
R5/Miles	Handle Milestones	Code enabling milestones handling
R6/CreateHum	Create Resources (person)	Code that creates human resources
R7/DeleteHum	Delete Resources (person)	Code that deletes human resources
R8/HumProp	Maintain Resource Properties	Code for maintaining resource properties
R9/Links	Add/Remove Task Links	Code that adds and removes task links
R10/Dependenc	Add/Remove Resources to Tasks Dependencies	Codes that adds/removes resources to task dependencies
R11/TimesUser	Change Task Begin/End Times manually with user changes	Code that allows user to change task times
R12/TimesDep	Change Task Begin/End Times automatically with dependency changes	Code that changes task times by dependency changes
R13/TimesTask	Change Task Begin/End Times automatically with subtask changes	Code that implements task times by subtask changes
R14/Prevent	Prevent Circular Dependencies	Code for preventing circular dependencies
R15/Vacations	Add/Remove Holidays and Vacation Days	Code for adding and deleting vacation days
R16/HumUtil	Show Resource Utilization (underused or overused person)	Code for showing utilization of a resource

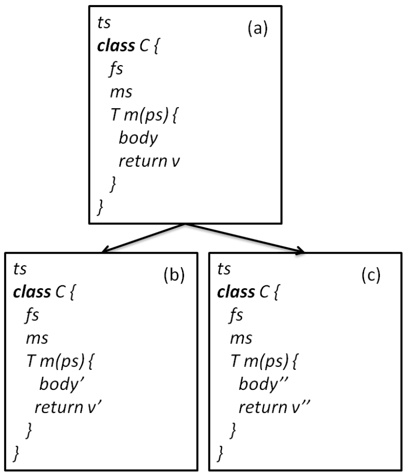

Fig. 15: Overwrite method: (a) original, (b) refactored and (c) refining classes

```
1   // Original class GanttProject
2   public TaskManager getTaskManager(){  //T m(ps)
3    return myTaskManager;                //return v
4   }
5   // Refactored in Feature GanttProject
6   public TaskManager getTaskManager(){  //T m(ps)
7    return null;                         //return v
8   }
9   // Refining in Feature TaskProp
10  public TaskManager getTaskManager(){  //T m(ps)
11   return myTaskManager;                //return v'
12  }
```

Fig. 16: Example overwrite method

(lines 9-12). In this example, please note that the first method is empty and because it has a return value we return null. When the refining method overwrites the method it returns `myTaskManager` instead.

Remove field modifiers declarations. This pattern just removes the modifiers declarations of a field. For instance, Figure 10 left depicts a class `C` with both methods ms and field `fs`. Field `fs` has modifiers declarations `D`. Due to refactoring process, the developer might need to remove this declaration of modifiers to the variable. Therefore, the solution is to remove directly this declaration as shown in Figure 17 on the right.

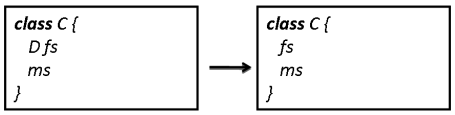

Fig. 17: Remove field modifiers declarations

For example, removing modifier `final` in the following statement:

```
private final HumanResourceManager myresourceManager;
```

Notice here that the removal of the modifiers is a consequence of the composition mechanism used by FEATUREHOUSE. Ways to address this limitation have been presented in Apel's et al. work [18].

V. Case Study Analysis

In this section we summarized the findings of our study and present the insights gained by our work that suggest possibilities for further research.

A. General statistics

We applied our refactoring approach to two systems. Refactoring `VODPlayer` (3.5 KLOC) took 4 days of work (approx. 32 hours). We found that each feature on average refines two distinct classes (minimum 1 and maximum 4) and adds two methods. For each method refined, there are on average 2-3 features involved (minimum 1 and maximum 10). We also noticed that the number of methods and the total LOC increase after refactoring. This is due to method refinements because of the wrapping that FEATUREHOUSE does for composing them. The total increase is a small 0.42%.

Refactoring Gantt Project (41 KLOC) proved a daunting task, because additional to the sheer size of the system there was no proper project documentation available. We had to rely on a lot of trial and error, and exhaustive code inspections. The total effort required 43 days of work, so an order of magnitude increase on the code size implied for us an order of magnitude increase in refactoring time. We found that each feature on average: makes 12 class refinements, has 82 base methods, makes 5 method refinements and hook methods. For this case study, the number of LOC and number of methods also increased after the refactoring process, again because of the composition done by FEATUREHOUSE. The increment was still quite negligible with a 0.47%.

Table III summarizes the number of occurrences found of each pattern. For `VODPlayer` the most frequent pattern was the *addition at end*. The second most frequent was the *move entire class* pattern, this was the case because there were listener classes completely related to particular features. For Gantt Project, the most frequent pattern was by far *move entire method*. We argue that these results bear a relation to both the granularity of our requirement descriptions and the size of the system. In `VODPlayer` the requirements – our features – were fine grained, when considering the small size of the system, and thus demanded more fine grain patterns. On the contrary, in the Gantt Project the requirements were more broad and coarse grained requiring more coarse grain patterns. Additionally, the underlying implementation technology may as well play an important role when selecting the granularity of features [19]. The exact relation between these three factors is an issue we plan to explore.

B. Patterns and their composition

In the refactoring of `VODPlayer` we identified 6 patterns. However, when we analyzed the Gantt Project we realized that two of them could be fused into a more general version of the hook method pattern which is the one we presented in section IV-A2. This finding suggests the possibility of expressing more complex patterns in terms of simpler ones. Thus, part of our future research is to investigate how to formally describe, represent and reason about the patterns we found, along the lines of the work by Bayley et al. [20].

C. Selection Criteria

Perhaps one of the most difficult parts of the refactoring experience was selecting the appropriate criteria to describe what is in a feature. For `VODPlayer` the criteria was simpler to obtain as it was easy to associate features with the visual elements of the GUI and trace the corresponding relations. However, the Gantt Project posed a set of additional challenges as these traces were sometimes either ambiguous or hard to obtain because of the size of the programs. In this regard, we believe that a more formal approach to express the selection criteria may help address this problem. An approach we will investigate is based on the work of Classen et al. [21].

D. Feature Module Disambiguation

Element sharing among more than one feature is a common occurrence. For example, in `VODPlayer` 9 variables were shared each of them involving (on average) between 2 and 3 features, and in the Gantt Project 27 variables were shared each involving (on average) between 3 and 4 features. In our study we followed a simple and straightforward strategy to disambiguate where a code fragment is modularized when more than one feature share it. The priority scheme we used favors first the root feature, followed by mandatory and optional features. In `VODPlayer`, 5 variables were modularized in the root, 3 in mandatory features and 1 in an optional feature. In the Gantt Project, 21 variables were moved to the root and 6 into mandatory features.

Though in our example the priority scheme worked fine, this scheme may not yield optimal results in SPLs with more complex feature models. A strategy that we will explore to address this limitation relies on operations of feature models [22]. More concretely, on the *commonality* value which represents the percentage a configuration (one feature in our case) appears in all possible feature combinations expressed in the feature model. We believe that refactoring a shared code fragment into the sharing feature with higher commonality reduces the chances of configuration problems. This is a claim we are working to evaluate empirically.

E. Refactoring Identification

One of the prerequisites to automate the refactoring patterns presented in this paper is to be able to recover or identify them directly from source code. An approach we plan to investigate is based on the work of Rasool et al. [23]. This approach works by using annotations, regular expressions and database queries to match a refactoring pattern to the source code elements. Alternatively, we have ongoing research on using execution traces for tracing requirements to code [24]. We believe that this work can potentially help both to corroborate the refactoring instances obtained in our study and to shed some light for helping the automation of FOR.

F. Refactoring Application

Once an instance of a refactoring pattern has been identified, the next step is to perform a transformation to actually carry out the refactoring. We are considering two main alternatives for the implementation of the refactorings. The first alternative is using general purpose transformation languages, such as TXL which provides strong support for structural analysis and program transformation based on formal notations such as programming languages, specification languages, and structured documents [25]. The second option is specifying the refactorings by example, like it is performed in several model refactoring approaches such as [26]. In this alternative, refactorings are specified via abstract templates derived by computing the differences between a model before and after the refactoring is applied. The main challenge here is specifying and validating correctly and completely all the preconditions that should be met for a refactoring to be applied.

TABLE III: Summary of refactoring patterns occurrence, and percentages

Pattern Name	VODPlayer	Gantt	VODPlayer%	Gantt%
Addition at the beginning	0	5	0	1
Addition at the end of the method	22	41	28	9
Addition anywhere with a hook method	8	47	10	10
Overwrite method	0	25	0	5
Move entire method	12	192	16	41
Move field	17	35	22	8
Remove field modifiers declarations	0	10	0	2
Move entire class	19	110	24	24
TOTAL	78	465	100	100

G. Refactoring Validation

Considering that our two case studies do not yield a large number of feature configurations, we manually tested all the combinations and made sure they provide the expected functionality.

H. Study Limitations

Next we list the limitations we identified in our study and describe how we plan to address them in our future work:

- *Small number of systems analyzed.* The scope of our study consisted of two systems that we selected for their size, a small one to develop and fine tune our FOR process and a larger one to corroborate and extend the results obtained in the first system. Certainly to address this limitation we plan to apply our approach to more case studies not only of different sizes but also of varied domains.
- *Refactoring patterns tied to a language and supporting platform.* Our work focused on Java language and FEATUREHOUSE. Certainly FOSD is applicable to other languages [13], [27], and consequently refactoring patterns for other languages could also be identified. This is also a venue for further research.
- *Ambiguous interpretation of requirement specifications.* The requirements we worked on were provided as prose, consequently they may be open to more than one interpretation. In other words, the meaning of a feature may be different among different stakeholders. One way to reduce this ambiguity is to involve more people in the refactoring process and employ more precise specification approaches as mentioned above.

VI. Related Work

There is an extensive body of literature of related work. For sake of brevity we present only those works closest to ours.

The refactoring approach we followed was inspired by Liu et al. research [14]. In contrast to our work, they develop an algebraic theory of feature refactoring. This theory supports reasoning about features and their interactions. Work on FOR by Kästner et al. proposes a model for refactoring features that are virtually and physically separated [28], in contrast with our work that extracts features from an interpretation of requirements in natural language. Additionally, work by Kuhlemann et al. [29] proposes refactorings of feature modules to accomodate for mismatches with legacy applications instead of refactorings from standard systems as in our study.

Our refactoring work relates to three other main research topics. The first is *feature location* [30] which consists on identifying the parts of the source code that correspond to a specific functionality. It is one of the most common activities undertaken by developers when they must find the location in the code where the first change must be made when evolving a system. This can be done manually but tool support becomes essential when programs are large and complex. Three alternatives have been proposed: *dynamic* that relies on execution traces, *static* that uses text patterns on the source code, and *hybrid* that combines the benefits of both previous approaches. In contrast, our feature location process was totally manual, meaning that first we had to understand how the program worked and then try to assign each code element to a feature. In principle, existing automated tool support for feature location could be used for our purposes. However, special consideration should be taken as in this research area there is no consideration for the inherent variability that features can have in SPLs.

A second related topic is *program comprehension* which essentially means understanding the software systems being dealt with [30]. There are many approaches and tools that could be leveraged for our purposes [31]–[34], but again it is an open question how those approaches would need to be adapted (if at all) to consider feature variability.

The last related topic, the *optional feature problem*, describes a common mismatch between variability intended in the domain and dependencies in the implementation [35]. The problem of the optional feature arises when the implementation of two optional features is not independent. This means that some variants that are valid in the domain cannot be produced due to implementation issues. Several solutions are proposed to solve this problem but all of the approaches suffer some disadvantages such as variability reduction, development effort increase, program performance and binary size deterioration or code quality decrease. Two main approaches of handling the problem are proposed. The first is to keep the implementation dependency, which means that variability is reduced as some variants cannot be produced. And the second is to change the feature implementation, which has five different ways of eliminating the implementation dependency. In our study, we addressed a simplified version of this problem and removed the dependency to the feature with higher priority.

VII. Conclusions

Evolving single systems into SPLs is becoming a more pervasive demand in the software industry to cope with the increasing variability present in modern software systems. This evolution requires the ability to identify features in single systems and refactor them usually exploiting advanced modularization paradigms. In this paper, we present our experience in refactoring two systems, one of small size and a larger one publicly available as an open source project. Our work followed a three-step refactoring process. We identified 8 recurring refactoring patterns and proposed a simple scheme for disambiguating shared code elements. Finally, we discussed the main issues and alternatives towards automating our refactoring approach.

Acknowledgment

This research is partially sponsored by the Austrian FWF under agreement P21321-N15 and and Marie Curie Actions - Intra-European Fellowship (IEF) project number 254965. Leticia research was sponsored by the Erasmus exchange program and the government of the Basque Country Spain. Currently, she is partially sponsored by the Universitat Politecnica de Catalunya.

References

[1] D. S. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 355–371, 2004.

[2] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[3] K. Pohl, G. Bockle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[4] P. Zave, "Faq sheet on feature interaction," http://www.research.att.com/ pamela/faq.html.

[5] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques," *Softw., Pract. Exper.*, vol. 35, no. 8, pp. 705–754, 2005.

[6] F. J. van d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[7] D. Batory, "AHEAD Tool Suite," 2010, http://www.cs.utexas.edu/users/schwartz/ATS.html.

[8] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 4, pp. 355–398, 1992.

[9] S. Trujillo, D. S. Batory, and O. Díaz, "Feature oriented model driven development: A case study for portlets," in *ICSE*. IEEE Computer Society, 2007, pp. 44–53.

[10] D. S. Batory, R. E. Lopez-Herrejon, and J.-P. Martin, "Generating product-lines of product-families," in *ASE*. IEEE Computer Society, 2002, pp. 81–92.

[11] R. E. Lopez-Herrejon, D. S. Batory, and C. Lengauer, "A disciplined approach to aspect composition," in *PEPM*, J. Hatcliff and F. Tip, Eds. ACM, 2006, pp. 68–77.

[12] D. S. Batory, "Using modern mathematics as an fosd modeling language," in *GPCE*, Y. Smaragdakis and J. G. Siek, Eds. ACM, 2008, pp. 35–44.

[13] S. Apel, C. Kästner, and C. Lengauer, "Featurehouse: Language-independent, automated software composition," in *ICSE*. IEEE, 2009, pp. 221–231.

[14] J. Liu, D. S. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *ICSE*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 112–121.

[15] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho, "Extracting and evolving code in product lines with aspect-oriented programming," *T. Aspect-Oriented Software Development*, vol. 4, pp. 117–142, 2007.

[16] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[17] "Gantt Project," 2010, http://www.ganttproject.biz/.

[18] S. Apel, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich, "An orthogonal access modifier model for feature-oriented programming," in *FOSD*, ser. ACM International Conference Proceeding Series, S. Apel, W. R. Cook, K. Czarnecki, C. Kästner, N. Loughran, and O. Nierstrasz, Eds. ACM, 2009, pp. 27–33.

[19] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM, 2008, pp. 311–320.

[20] I. Bayley and H. Zhu, "On the composition of design patterns," in *QSIC '08: Proceedings of the 2008 The Eighth International Conference on Quality Software*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 27–36.

[21] A. Classen, P. Heymans, and P.-Y. Schobbens, "What's in a feature: A requirements engineering perspective," in *FASE*, ser. Lecture Notes in Computer Science, J. L. Fiadeiro and P. Inverardi, Eds., vol. 4961. Springer, 2008, pp. 16–30.

[22] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, 2010.

[23] G. Rasool, I. Philippow, and P. Mäder, "Design pattern recovery based on annotations," *Adv. Eng. Softw.*, vol. 41, no. 4, pp. 519–526, 2010.

[24] B. Burgstaller and A. Egyed, "Understanding where requirements are implemented," in *ICSM*. IEEE Computer Society, 2010, pp. 1–5.

[25] "TXL," 2010, http://www.txl.ca/.

[26] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, "An example is worth a thousand words: Composite operation modeling by-example," in *MoDELS*, ser. Lecture Notes in Computer Science, A. Schürr and B. Selic, Eds., vol. 5795. Springer, 2009, pp. 271–285.

[27] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Feature (de)composition in functional programming," in *Software Composition*, ser. Lecture Notes in Computer Science, A. Bergel and J. Fabry, Eds., vol. 5634. Springer, 2009, pp. 9–26.

[28] C. Kästner, S. Apel, and M. Kuhlemann, "A model of refactoring physically and virtually separated features," in *GPCE*, J. G. Siek and B. F. 0002, Eds. ACM, 2009, pp. 157–166.

[29] M. Kuhlemann, D. S. Batory, and S. Apel, "Refactoring feature modules," in *ICSR*, ser. Lecture Notes in Computer Science, S. H. Edwards and G. Kulczycki, Eds., vol. 5791. Springer, 2009, pp. 106–115.

[30] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Trans. Software Eng.*, vol. 35, no. 5, pp. 684–702, 2009.

[31] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995. [Online]. Available: http://dx.doi.org/10.1002/smr.4360070105

[32] W. E. Wong and S. Gokhale, "Static and dynamic distance metrics for feature-based code analysis," *Journal of Systems and Software*, vol. 74, no. 3, pp. 283 – 295, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0N-4C8PD97-1/2/1655dcfe96cbfb26c1b44899890b9240

[33] R. Koschke and J. Quante, "On dynamic feature location," in *ASE*, D. F. Redmiles, T. Ellman, and A. Zisman, Eds. ACM, 2005, pp. 86–95.

[34] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *ICPC*, R. L. Krikhaar, R. Lämmel, and C. Verhoef, Eds. IEEE Computer Society, 2008, pp. 53–62.

[35] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. S. Batory, and G. Saake, "On the impact of the optional feature problem: analysis and case studies," in *SPLC*, ser. ACM International Conference Proceeding Series, D. Muthig and J. D. McGregor, Eds., vol. 446. ACM, 2009, pp. 181–190.