Refining Feature Driven Development - A methodology for early aspects

Jianxiong Pang   Lynne Blair

Computing Department

Lancaster University, Bailrigg

Lancaster, LA1 4YR, U.K.

j.pang@lancaster.ac.uk   *lb@comp.lancs.ac.uk*

Abstract

This position paper focuses on refining an agile processes approach named FDD to make it more aspect-oriented, hence a natural candidate for early aspects.

We show that only a slight refinement is needed to adapt FDD to aspect-oriented development. Within the refinement, all requirements, be they concerns (architectural, non-functional and functional) or properties or rules, are described by using the feature template in FDD. Features as development units are first class entities throughout the whole development period. Since the later stages of FDD are also enhanced with aspect-oriented technique, this makes the transition from requirements to design and implementation much easier and smoother.

Keywords: early aspects, feature driven development, agile process, feature template, aspect-oriented, requirement engineering.

1. Introduction

A *feature*, as a term, is used for describing a small piece of particular valuable (sometimes also attractive) capability/functionality. Within telecommunication systems, it is defined as "a unit of functionality existing in a system and usually perceived as having a self-contained functional role" [2].

A feature is a concept mostly belonging to the problem domain rather than the solution domain [3], hence is highly relevant to requirement analysis. This has been reflected by the fact that the "feature engineering" is categorised as a branch of requirement engineering [4]. A feature is suitable for behaviour modularity, which in turn, supports change of system's behaviour.

Feature Driven Development (FDD) is a lightweight and model-driven software development process tailored to the delivery of frequent, tangible, working results. The lightweight characteristics make these processes easy-to-follow and agile. Another remarkable characteristic of FDD is that it gives a slight variation to the definition of "feature", with a feature being referred to as "a client-valued functionality that can be implemented in two weeks or less". By that, it includes a "timing factor", which is believed to have subtly combined the technical factors (e.g. agility) and social psycho factors (e.g. encouraging value) in the software development activity. Under FDD, features as incremental units are planned and developed one by one, making tangible and solid results. This makes quality control more manageable. Furthermore, a feature is carefully tailored, so as to be implementable in a relatively comfortable time. This gives confidence, encouragement, and incentive to developers.

General feature driven methodology has been proven effective in modern software systems. In *product line development*, a company produces a range of similar software products; the product can be structured in such a way that common units of development (e.g. feature or feature sets) are shared. Such development has been shown to significantly reduce development costs, and benefits end-users in terms of flexibility, in being able to choose a customized combination of features for their product [5]. *The telecommunication industry* has particularly benefited from a development centred on features. The introduction of Intelligent Network (IN) brought in a generic model where a basic call could be updated by

adding features implemented as discrete components (Service Logic Programs). As a result, the telecommunication industry has a tradition of organizing development projects, people, and even marketing by features [7]. Microsoft has also apparently followed some feature-centric processes in their software product line for a number of years [8].

Building on the object-oriented paradigm and reflective programming, aspect-oriented programming is emerging as a technique that supports more advanced separation of concerns. Recently, as this technique has become more broadly recognized, more techniques have been merged under the umbrella of aspect-oriented software development (AOSD). The worthiness of aspect-oriented techniques being combined with FDD lies in that aspect-oriented technology is capable of flexible behaviour modification being carried out on an existing, or even, running system. Although FDD has existed for quite a long time now, it is not easy for a traditional OO technique to implement features in an entirely modular way. It is with the emergence of aspect-oriented techniques that the development of features in a neat and clean way becomes a reality. As feature modularisation and localization is dramatically improved, a chance certainly exists to refine FDD into an aspect-oriented process.

2. Related work

As has happened in object-oriented programming, researchers have applied aspect-oriented ideas to higher levels of the software lifecycle, e.g. requirement analysis and design.

Works that are most relevant are those about *aspect-oriented component based software development* [9] and *aspect-oriented requirement engineering* [10].

In [9], an approach called "aspect-oriented component requirement engineering process" is proposed "to address some difficult issues of component requirement engineering by analysing and characterising components based on different aspects of the overall application a component addresses". This approach applies aspects to categorized components with properties, and provides facilities for the requirement changes (e.g. the change of stakeholders or running context). Our analysis reveals that it is not clear, in this approach, the relationship of aspects with the component architecture, and other aspects. Aspects in this approach are concepts rather than first class entities that later are mapped to some design and implementation artefacts (i.e. aspects are still identifiable). The lack of simple and unified concept of aspects and their associated base systems contributes to making this software process relatively 'heavy'. We believe that keeping it lightweight and agile is vital for today's software, especially for service-oriented systems, which rapidly become pervasive.

In [10], a model for "aspect-oriented requirement engineering" is proposed that supports "the reconciliation of *separation of concern* with the need to satisfy broadly scoped requirements and constraints". This model is built on an existing approach called PREView [11] that already supports separation of crosscutting properties but lacks guidelines on avoiding inconsistencies between concerns, and also lacks the mapping or influence of crosscutting properties on artefacts at later development stages. Therefore, the model can be viewed as a refinement of PREView, aiming to overcome the shortcomings mentioned above. The model uses "concerns" to represent aspects at the requirements level. We believe that these "concerns" are close to "features" in meaning, except that there is no concern template equivalent to the feature template, therefore the description of a concern seems ad-hoc. Furthermore, the model expressed in [11] does not require building an overall model (the backbone) as in FDD. Therefore, it is not clear how to make a smooth transition to the design and implementation stages given there is no placeholder for concerns in the requirement stage, or advice on how to deal with multi views if there are multiple models.

3. Comparing FDD and aspect-orientation

There are 5 processes within FDD (taken from [12]):

1. *Develop an overall model (focussing more on shape than on content)*

2. *Build a detailed, prioritised feature list*

3. *Plan by feature*

4. *Design by feature (focussing more on content than on shape)*

5. *Build by feature*

   Note: process 4-and 5 are iterated

Further information on FDD can be found in [12],[13] and [14].

It can be seen that, among the 5 process steps, the first three steps belong to the *requirement engineering*. The *feature*, as a development unit, is emphasised throughout these processes. Note that before any feature is developed, there has already been an *overall model* produced, typically represented as a class diagram. Thus, all features have to be composed or integrated into the overall model later. This arrangement makes it comparable to some important AOP techniques (e.g. AspectJ [15]), i.e. a system can be developed by separating a base system and a number of advice modules that later are woven to the base.

The determination of features is largely guided by user's value rather than by the class structure of the overall model[1], such that if, during the integration/composition of features into the overall model, a feature looks unsuitable to fit in a class, then it should be spread into several classes.

At a more detailed level, every feature in the feature list is described by using a feature template. Such a template is as follows:

<action> the <result><by | for | of | to>a(n)<object>

e.g. *calculate* the *total* of a *sale*, *invoke* the *spell-checker* for a *document*, *upgrade* the *quality* of a *service*, etc. Among these features, some can be naturally included into one class, such as the first and the third one; some cannot, like the second one (involving at least two classes the *spell-checker* and the *document*).

This has led us to conclude that FDD closely resembles aspect-oriented development in nature if purely examined from a perspective of the high levels (e.g. requirement and design levels), because it has satisfied two key aspect-oriented conditions:

1. *Crosscutting*: features do crosscut the overall model. (See the "spell-checker" example)

2. *Modular*: features are semantically complete and self-contained entities.

On the other hand, FDD is not completely an aspect-oriented technique because it lacks explicit guidelines on how to:

1. incorporate crosscutting concerns into features.

2. seemingly map features to design and implementation artefacts in a aspect-oriented programming environments at later stage.

Therefore, it is necessary to refine the FDD processes so as to complement it with respect to aspect-oriented development.

4. Refining the FDD processes

The proposed refinement is made with the following goals in mind:
- Facilitate the separation of concerns.
- Assist the detection and avoidance of inconsistency between features
- Maintain a smooth transition to the next stage of the development process

To facilitate the separation of concerns, the basic arrangement of the system structure is preserved, i.e. an overall model plus a set of features. Features can be grouped into a feature set, according to their characteristics of functionality, which might result in a hierarchical structure of features.

To assist the detection and avoidance of inconsistency between features, we have proposed a method called *boundary condition exploration* [16] based on our study of a large number of feature interaction cases within and beyond telecommunication systems [2][17]. This method is built on the fact that most of the subversions or conflicts between features happened across the boundary condition of features.

To maintain smooth transitions between process stages, we keep a sharp decomposition based on features throughout the whole development period. The requirement is represented by a *feature list*, a development plan is made for each *feature*, the software is designed by *features*, and finally the software is built by *features*. By using aspect-oriented programming technology, a feature can even be localized within its own module, becoming a truly separate entity [16][18].

The refinement of FDD is briefly showed in Table 1.

Table 1: a refinement to FDD

Num	Activity	Refinements
#1	Build an overall model	Preserve without refinement
#2	Work out a feature list	Preserve all the principles of identification of features. All the "rules", "policy", "non-functional and functional concerns", "architectural concerns", "properties" are uniformly represented as "features". Use the "boundary condition exploration" technique to detect inconsistency/conflicts/subversion between features
#3	Plan by features	Preserve all the principles of planning. And the plan should consider the influence of adopting aspect-oriented development.
#4	Design by features	Adopting an aspect-oriented design principle and notation. Features and conflict resolutions are designed as "aspects".
#5	Build by features	Adopting an aspect-oriented programming language. Features are implemented as "aspects".

Here we use some examples to illustrate how different forms of requirement information can be represented as "features". We use the feature template of section 3 to describe all the features. For example, in an E-learning system we are currently developing, there is a pedagogical rule described as follows:

---

[1] It is also guided by development time, as can be seen from the FDD description. The time factor has been omitted here for simplicity.

If a learner answers a question wrong in a Quiz *then* guide him/her to a corresponding study unit.

This rule can be transformed to a feature style in FDD as follows:

Direct *learner* to *a particular study unit* for a wrong answer to *a question* in a *Quiz*.

  - where words in italics are potential classes *of object*.

Regarding the early aspects work of [19] and [10], all aspectual requirements are described as "concerns" with each concern being defined in a XML document. In fact, each concern corresponds to a feature set in FDD. For example, one such concern is described as follows:

```
<?xml version = "1.0" ?>
<Concern name="Compatibility">
 <Requirement id="1">
    The system must be compatible with systems used to
      <Requirement id="1.1">
          read the gizmo identifier;
      </Requirement>
    <Requirement id="1.2">
        deal with infraction incidents;
      </Requirement>
    <Requirement id="1.3">
        charge for usage;
      </Requirement>
 </Requirement>
</Concern>
```

Note that each sub concern looks very similar to the FDD feature template.

Furthermore, most requirement examples in [9] are typically architectural features by nature. For example:

```
Persistency Aspects
    +save and load function
    - file manager
    - remote data manager
```
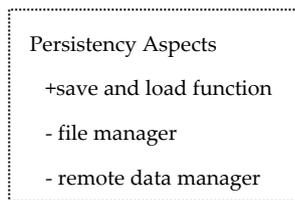
Where: "+" denote functionality is *provided* by components, while "-" means the functionality is *required* from some other components

Figure 1. An aspectual requirement in [9]

While this is quite different to the FDD way of describing features, it can still be represented naturally by feature templates, in which, a feature set named "Persistency" is created, with the following feature included in the feature set:

-Save&load *object/components* from/to *the storage*.

-Request to *file manager* for *storage services*

-Request to *remote data manager* for *data*.

A more complete example is from a *building control system*. A feature list of the subsystem "*Room light control* " derived from [1] is given as follow.

Table 2: A feature list in a building control system

ID	Name	Description
f01	Illuminance	Turn off the lamplight of a room for saving energy
f02	EnergySaving_Blind	Control a list of blinds for energy saving.
f03	EnergySaving_Lamp	Control a list of lamplight for energy saving
f04	GlarePrevention_Blind	Control a list of blinds for glare prevention
f05	LightTurnOn_Request	Turn light on or off on request
f06	BlindOnOff_Request	Open or close blind on request
f07	RoomStatus	Determine and report if a room is being occupied
f08	RadiatorControl	Control room temperature by using the radiator
f09	RadiatorValve_Request	Open or close radiator valve on request
f10	TemperatureStatus	Determine and report current temperature

The features and feature set are organised into hierarchical structure and prioritised, weighted in the process "working out a feature list".

With the completion of a feature list, a "plan by feature" process can be carried out with the purpose of "making high-payoff results" in mind.

In the process of "design by feature", a feature's functionality specification is viewed as "the core business logic" or "hard logic", which is designed into one or more classes in an OO paradigm. For example, the core business logic of the feature "Illuminance" can be designed as in Figure 2 (in pseudo java code).

For the interaction, composition and connection between features, modules called "resolutions" or "soft logic" are designed to guarantee the inter-working of features. A resolution module can be designed in a pseudo AspectJ [15] as showed in Figure 3.

```
public class Illuminance {
 ArrayList rooms;
 public void peopleArrive(in roomNumber){
  turnOn(roomNumber);
 }
 public void turnOn(int roomNumber){
//code used for turning on the light in a
//room with a known number
 }
 public void peopleDepart(int roomNumber){
  turnOff(roomNumber);
 }
 public void turnOff(int roomNumber) {
//code used for turning off the light in a
//room with a known number
 }
}
```

Figure 2. Illuminance's hard logic that implements exactly the specification of the feature

```
aspect  ResolutionForIllmnce_EnergySaving
{
 void around(int roomNumber):
      call(void Illuminance.turnOn(int)) &&
                            args(roomNumber)
  {
   if (isDayLightSuitable(roomNumber))
      new EnergySaving().open(roomNumber);
   else proceed(roomNumber);
  }
 boolean isDayLightSuitable(int roomNumber){
   .......;
  }
 void around(int roomNumber):
    call(void Illuminance.turnoff(int)) &&
                            args(roomNumber)
  {
   if (isDayLightSuitable(roomNumber))
     new EnergySaving().close(roomNumber);
   else proceed(roomNumber);
  }
}
```

Figure 3 Resolving Illuminance vs. EnergySavingFeature Interactions

The resolution module ensures that "whenever there is suitable daylight, use the energy saving feature instead the normal illuminance feature".

In the process of "build by feature", all features are carefully examined, unit tested and deployed by using an AOP implementation language.

5. Brief evaluation of FDD refinement

The above examples have highlighted interesting overlaps between FDD and the early stage of the software lifecycle. The proposed refinement to FDD (table 1) attempts to integrate AOSD techniques into FDD. More specifically, the refinements preserve all the good properties of FDD, and at the same time, introduce the benefit of aspect-oriented techniques. The improved separation of concerns in all the processes helps control the complexity of software development, which in turn, helps to improve the maintenance and the capability of evolution (feature modification, replacement, addition and deletion, etc.). The assistance of inconsistency detection and avoidance facilitates secure and high quality software development, and supports the integration of new features in the future. Furthermore, the smooth transitions from requirement analysis to design and then to the implementation stage make the implementation more aligned to the requirements, which is highly helpful for the maintainability and the application of generative programming.

The refinement of FDD by adopting aspect-oriented techniques has so far had a positive impact on the later stages, and has allowed us to elegantly design and implement features, that are also faithful to the feature specification. For further information on the design and implementation of features in an aspect-oriented technique, we refer the interested reader to [16] and [18].

6. Conclusions and Future Work

In summary, FDD is an excellent agile process, and its development strategy bears similarities to that of aspect-oriented techniques. However, because it is not originally designed for aspect-orientation, it still lacks some properties to be a complete aspect-oriented solution to software development. A refinement based on an aspect-oriented technique can make the first three FDD processes good candidates for early aspects, namely aspect-oriented requirement analysis.

With the three feature representation examples, we conclude that the idea "early aspects can all be uniformly described with a feature template, then planned, designed and implemented later with an aspect-oriented programming technique" is viable.

While this theory came from our software development practices, it is vital to use this theory back to the real world development. The future work is to study the usefulness of this methodology in a variety of domains, such as server-side development, Grid/web service, P2P, Internet telephony and reactive control systems etc. It is also important to collect cases of feature (or aspect) conflict/subversion, and abstract the resolution pattern for the interworking of features.

References

[1] Andreas Metzger, Christian Webel, "Feature Interaction Detection in Building Control Systems", in [20], pp60-66, June 2003.

[2] L.Blair. & J.Pang., "Feature Interactions - Life Beyond Traditional Telephony", In [6], pp 83-93, 2000.

[3] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. Journal of Systems and Software, 49(1):3--15, Dec. 1999

[4] P.Zave, "Requirements for evolving systems: A telecommunications perspective", in Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, pages 2-9, IEEE Computer Society, 2001.

[5] Clements P., Northrop L., Software Product Line Practice Patterns. Addison-Wesley, 2001.

[6] M.Calder, E.Magill , editors, "Feature Interactions in Telecommunications and Software Systems VI", Glasgow, Scotland, IOS Press(Amsterdam), 2000.

[7] P.Zave, " FAQ Sheet on Feature Interactions ", AT&T, 2001, Available via: http://www.research.att.com-/~pamela/faq.html.

[8] M.A.Cusumano and R.W.Selby, "Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People", Simon & Schuster, 1998. ISBN: 0684855313.

[9] J. Gundy, Aspect-Oriented Requirements Engineering for Component-based Software Systems", 4th IEEE International Symposium on RE, 1999, IEEE Computer Society Press, pp. 84-91.

[10] Rashid, A., P. Sawyer, A. Moreira and J. Araujo."Early Aspects: A Model for Aspect-Oriented Requirements Engineering". IEEE Joint International Conference on Requirements Engineering. IEEE Computer Society Press. Pages 199-202.2002

[11] I.Sommerville and P.Sawyer, Requirements Engineering – A Good Practice Guide: John Wiley and Sons, 1997.

[12] P.Coad, J.de Luca, E.Lefebrve, *Java Monitoring in Color with UML*. Chapter 6:Feature Driven Development, Prentice Hall, 1999.

[13] P.Coad and S.Palmer, Feature-Driven Development, TogetherSoft Corporation 2002, Available via http://www.nebulon.com/articles/fdd/latestprocesses.html.

[14] S.Palmer, Feature-Driven Development and Extreme Programming, informIT, 2002, Available via http://www.informit.com.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, Getting started with ASPECTJ , Communication of the ACM, Vol. 44, Issue 10, ACM Press (New York), pp59-65, October 2001.

[16] J. Pang, L. Blair, "Separating Interaction Concerns from Distributed Feature Components", Proceedings Software Composition workshop (SC 2003), held in conjunction with ETAPS 2003, Warsaw, Poland, Electronic Notes in Theoretical Science, Vol. 82, Issue 5, Elsevier, April 2003.

[17] E.Jane Cameron, Nancy D.Griffeth, Y. Lin, M.Nilson, W.Schnure, and H.Velthuijsen. A feature-interaction benchmark for IN and beyond. IEEE Communications XXXI(3):64-69, March 1993.

[18] Blair L., Pang J., "Aspect-Oriented Solutions to Feature Interaction Concerns using AspectJ", in [20], pp87-104, June 2003.

[19] Rashid, A. Moreira, and J. Araujo."Modularisation and Composition of Aspectual Requirements". 2nd International Conference on Aspect-Oriented Software Development. ACM. Pages 11-20.2003

[20] D. Amyot, L. Logrippo (eds), "Proceedings 7th International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'03), Ottawa, Canada, Amsterdam: IOS Press, June 2003.