Dungeon: A Case Study of Feature-Oriented Programming with Virtual Classes

Vaidas Gasiunas Ivica Aracic
Technische Universität Darmstadt, Germany
{gasiunas,aracic}@informatik.tu-darmstadt.de

Abstract

A feature is a logically cohesive piece of functionality and is present in all phases of software development. Thus, it is natural to expect that modularization of software into features can provide a lot advantages. This paper presents a Dungeon case study, which evaluates feature-oriented programming (FOP) using virtual classes and propagating mixin composition that are available in the CAESARJ programming language. We describe the techniques for feature separation with virtual classes, the process of deriving feature-oriented design from requirements and a gradual refactoring of object-oriented programs to a feature oriented design. Besides, reconfirming the already known advantages of FOP for extensibility and reuse, we also evaluate if FOP in general leads to a better design than OOP and how well it is supported by CAESARJ tools.

1. Introduction

Feature-oriented programming (FOP) emerged from the work of Prehofer [14] and the work on GenVoca [7]. The main idea of FOP is that by modularizing software into features we can achieve better extensibility and reusability. FOP is especially useful in the context of the product line development [16, 17, 5], because it allows to separate the features that are common for most of products of a product line from the features that vary from product to product.

Typical implementations of the GenVoca model [15, 6] are based on a layered design: each feature corresponds to a layer, which consists of a set of partial definitions of classes that implement the feature. Concrete programs are then generated by composing such layers. Using mixin layers it is also possible to decompose methods: slices of the implementation of a method can be distributed into multiple layers and combined using the semantics of super calls. The name clashes that occur during merging are resolved by the composition order of the layers, i.e. the methods of a layer override the corresponding methods of the layers that are further in the composition. In this way, it can be achieved that more specific features override the functionality of more general features.

An advantage of the AHEAD system is that it is not bound to a particular programming language and can even be applied to decompose non-code artifacts into features. For example, FeatureC++[2, 1] combines AHEAD with an aspect-oriented language and in this way enables expression of behavioral crosscutting in the feature modules.

Layered decomposition is also possible in languages supporting virtual classes and mixin composition, e.g. gbeta [10], CAESARJ [3]. The composition semantics of these approaches also supports overriding and composition of method implementations. Virtual classes provide an appropriate type system to express relationships between layers and their refinements. This enables modular type checking of the layers. Abstract virtual classes of CAESARJ enable definition of interfaces for feature modules. In this way it is possible to separate the implementations of different features at the level of segregated interfaces. Besides, virtual classes support coexistence of multiple different combinations of feature modules within the same program and their dynamic instantiation.

This paper presents the Dungeon game case study, the goal of which was to evaluate feature-oriented programming with CAESARJ using virtual classes and propagating mixin composition. Besides confirming the already known advantages of FOP for extensibility and reuse in the context of product lines, we also wanted to evaluate if FOP leads to a better modularization than OOP in general.

There are multiple reasons why this should be the case. Since features are units of end-value of software to the user, features are present in most of stages of software development: analysis, development, testing and maintenance [18]. Therefore, feature-oriented modularization of software can reduce the mismatch between specification and implementation and provide a better support for independent development and testing. Since software is usually extended by adding new features or enhancing the existing ones, the possibility to define features in separate modules can provide an advantage for software maintenance. Besides, feature-driven development [13] emphasizes advantages of organizing software development around features. However, for implementation it uses conventional object-oriented programming lan-

guages. This leads to a mismatch between the code modules and project's organizational units.

It must also be noted that product lines are seldom developed from scratch. Instead, they usually emerge from separate projects and products during the continuous process of reuse maximization. It is also difficult to draw a strict line between the cases when we speak about a product line and when we speak about simple reuse of components and libraries. Thus, it may be useful to start using FOP from the very beginning for development of invidual projects and products in order to support continuous transition to a product line.

The remainder of the paper is organized as follows. In Sec. 2 we give an introduction to the example of the case study. Sec. 3 explains the techniques of feature separation with virtual classes and propagating mixin composition. Sec. 4 describes an approach of feature identification from a requirements specification and a process of a gradual refactoring from an object-oriented to a feature-oriented design. We evaluate the results of the case study in Sec. 5 and conclude in Sec. 6.

2. Dungeon Case Study

Dungeon is a typical 2D real-time game with a map filled with moving creatures, collectable items and traps. One of the creatures is controlled by the player; the others are controlled by the artificial intelligence (AI). There are multiple AI strategies for movement and battle. Creatures may carry various items, such as weapons, armor, potions and gold. The player can manage and use its items in the inventory window. Items can also be traded to the merchant.

The domain of computer games was selected, because it is a typical domain for object-orientation. Most of the application state and functionality is organized around game objects: they are manipulated, simulated and rendered. Numerous variations, such as item types and battle strategies, are typical cases for inheritance and polymorphism. It was interesting to investigate if CAESARJ can further improve the design of such typical object-oriented applications.

Before starting with CAESARJ design, first we implemented the game functionality in pure Java. According to the idea of object-orientation the code of the game was modularized around object types: various game objects (e.g. world, creature, room, map, item), user interface elements (e.g. game window, inventory window), algorithms (e.g. path finding, enemy selection). The object-oriented design did not, however, achieve complete separation features and lead to large sets of interdependent modules, because of following reasons. First, a lot of the game world classes are shared by different features. For example, almost every feature adds some new state and operations to the classes Creature and World. Thus, in order to separate the features from each other, we need static crosscutting [12]. Besides, we also found that there is a set of points in program execution, where the behavior of multiple features is tangled. The typical join points are execution of the simulation steps of the world objects that are shared by multiple features, rendering of these objects, resource loading and reaction to the user input.

3. Modularization of Features with Virtual Classes

We address the problem of feature separation by employing CAESARJ's virtual classes and mixin composition [3]. The main idea behind virtual classes is that overriding and late binding should also apply to inner classes. This means that, similarly to a late-bound (virtual) method, the behavior of a virtual class can be redefined in any subclass of the enclosing class.

In CAESARJ each feature is modeled by a top-level class, while domain objects are modeled by their virtual classes. We will refer to such top-level classes as feature classes. Dependencies between features are modeled by inheritance relationships between the corresponding feature classes. Since virtual classes can be overridden in subclasses, a feature class can extend and override functionality of the domain objects inherited from other feature classes.

Figure 1 demonstrates an example of feature modularization with virtual classes. Class FCreature[1] implements the base functionality of game creatures. It defines that the game world consists of multiple creatures, each creature is located at certain coordinates and that update of the game world involves update of all its creatures. It introduces class Creature and refines class World, which is inherited from the feature class FWorld. The method Creature.update is empty and intended to be extended by more specific feature classes.

Classes FMovement, FAttack implement correspondingly the movement and the attack functionality for creatures. They inherit from FCreature and refine Creature with new attributes and operations that are necessary to implement movement and attack. The class FMovementAnimation refines the feature FMovement further with the functionality to animate movement cycles.

In general a feature class inherits from another feature class in order to use or to override its functionality. Since a feature may need functionality of multiple features it is important to have some form of multiple inheritance in the language. For example, feature class FChase, which implements the functionality to chase the target enemy, requires both the movement and the attack functionality, so it must inherit from both FMovement and FAttack. The inherited classes are composed by so called propagating mixin composition: it is a kind of multiple inheritance mechanism, which recursively combines all members of participating hierarchies. Its implementation is based on the work in [11], which uses

[1] We prefix the names of all feature classes with the letter F and use keyword cclass to declare classes with CAESARJ specific semantics. Keyword class is reserved for standard Java classes.

```
cclass FWorld {
  cclass World {
    void update(float dTime) { }
  }
}
cclass FCreature extends FWorld {
  cclass Creature {
    World world;
    float x, y; int heading;
    void update(float dTime) { }
    ...
  }
  cclass World {
    Player player;
    Collection creatures = new LinkedList();
    void update(float dTime) {
      super.update(dTime);
      updateCreatures(dTime);
    }
    void updateCreatures(float dTime) {
      for(Iterator it = creatures(); it.hasNext(); ) {
        Creature creat = (Creature) it.next();
        creat.update(dTime);
      }
    }
    ...
  }
}
cclass FMovement extends FCreature & FPathFinding {
  cclass Creature {
    boolean moving;
    float targetX, targetY;
    MovementStrategy movementStrategy;
    void updateMovement(float dTime) { ... }
    void update(float dTime) {
      super.update(dTime);
      updateMovement(dTime);
    }
    ...
  }
  abstract cclass MovementStrategy { ... }
  ...
}
cclass FAttack extends FCreature {
  cclass Creature {
    Creature enemy;
    void update(float dTime) {
      super.update(dTime);
      updateAttack();
    }
    ...
  }
}
cclass FMovementAnimation extends FMovement {
  cclass Creature {
    int walkCycles; float walkCycleSpeed;
    void update(float dTime) {
      super.update(dTime);
      if (moving) { updateWalkCycle(); }
    }
    ...
  }
}
cclass FChase extends FAttack & FMovement { ... }
```

Figure 1. Modularization of features with virtual classes

mixins [8] and C3 inheritance linearization [4] as the underlying concept.

Inheritance linearization is a common mechanism to reduce a multiple-inheritance graph to an ordered list, so that the order of the elements in the list determines the behavior in a case of ambiguity. The C3 algorithm is a topological sort of a multi-inheritance graph with additional rules that make the linearization unambiguous and enforce some desirable properties of the result (see [4]). E.g., the C3 linearization of FAttack & FMovement would produce the linear list [FAttack, FMovement, FCreature, FPathFinding, FWorld]. Note that each class is inherited only once, even if it is inherited over different inheritance paths. The linearization defines the overriding order of inherited methods and the order of composition of the virtual classes.

An application is defined as a feature class that inherits from the classes of all other features that must be available in the application.

Mixin composition semantics enables definition of composable methods. For example, Creature in FCreature defines method update, which is refined in FMovement, FAttack and FMovementAnimation. Each refinement contributes some new functionality and calls the super version of update. When the classes are composed and linearized, the method overriding as well as super calls follow the linearization order, so when Creature.update is called somewhere in the application all available refinements of the method are called in the order determined by the linearization.

4. Process of Feature-Oriented Design

Although we had a clear understanding how to modularize features using virtual classes and mixin composition, it was still not clear what features we should modularize and how we could efficiently refactor an object-oriented design to a feature-oriented one.

Since there is a correspondence between features and requirements, it was natural to look for features in the functional specification of the game. For this purpose, we modularized the requirements by the criteria similar to the ones formulated in [9].[2] The requirements that bring useful functionality only together were identified as features, and the specification itself was restructured so that features were described in separate sections. The sections were further grouped into chapters, which then represented the major feature groups.

Further, we identified logical dependencies between features, by studying their descriptions and looking for references to the concepts or facts that were introduced in the descriptions other features. Undesired dependencies were eliminated by decomposing requirement statements into smaller ones and moving one from one feature to another. It is important to note that we only restructured the specification, but did not change its meaning in any way. The dependency analysis was performed hierarchically. First, we determined dependencies between the feature groups, represented by chapters and then analyzed dependencies between the sections inside chapter as well as their external dependencies.

Once we established the desired modularization of the requirements, it served as a plan for the refactoring to the

[2] Our concept of feature is close to the concept of requirements module in this paper.

feature-oriented design. The goal was to modularize the game so that the feature classes and their grouping into packages correspond to the structure of the functional requirements (i.e. the sections and subsections of the specification) as much as possible. A dependency between two feature modules should exist only if there is a logical dependency between the requirements describing these features.

Another challenge was organization of the refactoring process. Our goal was to divide the refactoring process into small steps, so that after each step we have a program that can be compiled and tested. Such process not only significantly reduces the complexity of the task, but also allows to organize refactoring as a process of continuous improvement, what is very important in product development. So we divided the refactoring process into following steps:

- In the first step the Java classes are transformed to CAESARJ classes. This transformation step is rather mechanical, but the biggest problem here is that CAESARJ classes cannot be defined as subclasses of regular Java classes, thus inheritance from the Java classes that are outside the refactoring scope must be replaced by delegation.
- Then for each package in the refactoring scope a CAESARJ class is created and the classes of that package are inserted as inner classes of the class representing the package. The dependences between packages are modeled as inheritance relationships between the corresponding classes. In this step cyclic dependences between packages must be broken. This can be achieved by inserting abstract declarations. For example, if classes of package A use classes of package B, and also the other way around, then we can declare class B as subclass of class A, and include in class A abstract declarations of all classes and operations from package B that are necessary for A.
- Define an application (or the refactored part of it) as a mixin composition of the package classes that are leaves of the inheritance hierarchy.
- Extract one by one all the identified features, starting from the root nodes of the feature dependency graph and gradually going down.

It can be difficult to extract a feature, because its code is often scattered in the application. Furthermore, there is a lot of code, which does not implement any externally visible effects of the feature, but rather contributes to the implementation of the feature indirectly. So we elaborated a process, which employs the compiler and a source control system in order to achieve a clean separation of a feature:

- Use a source control system to save the version of the application before feature extraction.
- Locate all the code that belongs to the feature, and remove it. This can be done by starting with the core feature code and subsequently removing code that causes compilation errors due to the missing feature.
- Test the application without the feature. Watch for functionality that does not make sense without the feature and remove it too.
- Now compare the current version (without the feature) to the latest repository version. This comparison shows the differences between both versions, which is exactly the code of our feature.
- Create a new CAESARJ class, dedicated to the feature, and reimplement the feature by copying code from the repository into this class. This class like package classes consists of multiple virtual classes implementing the feature.
- If the feature requires functionality of other features (or unrefactored packages), corresponding inheritance relationships must be added between the class of the feature and the classes of the features it depends upon.
- Combine the feature class with other feature (and package) classes and test the combined application.
- Analyze if the feature class depends only on more general features. If it is not the case, define an abstract class that captures the interface of the dependency and inherit from this abstract class instead of inheriting from the concrete feature.

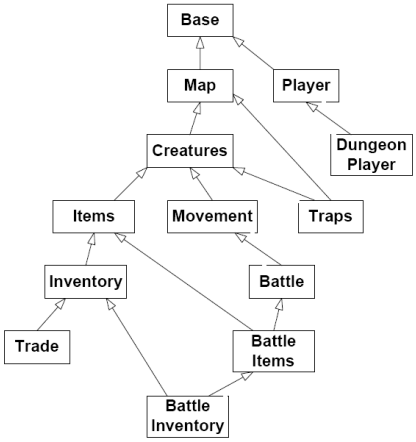

Figure 2. Dependencies between top-level feature groups.

5. Results

The diagram in Fig. 2 presents the top level view on the identified Dungeon features. The blocks in the diagram correspond to top level feature groups and dependencies between them. These groups consist of multiple features that correspond to more fine-grained pieces of functionality. When analyzing the shared functionality between observable features, we identified that most of the internal features fall into four categories: data model features, simulation features,

rendering features and control features. The dependencies between the features of these categories follow a consistent pattern: simulation depends on data model, while rendering and control depends on data model and simulation. These four categories formed the second level of feature grouping inside the top level feature groups in Fig. 2.

In the CAESARJ implementation of the game, the package structure reflects the described feature grouping structure. The classes in the packages correspond to the identified features, and the actual game objects are described by virtual classes of the feature classes.[3]

Reuse. In order to test the quality of our modularization, we combined the features into multiple different versions of the game. For example, for each feature group in Fig. 2 we could derive a combination that consisted of this feature group and the feature groups it depends upon. In this way we could build a game with battle, but without items and traps, with items, but without inventory management and trade and so on. Besides, the possibility to switch off certain features makes it possible to replace them with alternative ones.

Extensibility. We have tested the extensibility of the new design by trying to extend the game with new functionality. The Traps feature group was created as a result of this experiment. Traps can be placed on the map, if a creature runs over it, then it takes damage. This extension has been implemented in a modular way by providing new feature classes that depend on the functionality of the feature groups Map and Creatures. The extension could be added without any change to the existing modules. The class representing a room was overridden to keep a list of the traps in the room. Rendering and simulation functionality was extended with trap rendering and simulation by overriding corresponding methods.

Size. In order to compare the size and complexity of the object-oriented and feature-oriented versions of the game, we computed the lines of code of the source code without the blank and comment lines. We also excluded the code for testing, which is much larger in feature-oriented design, because we tested various variants of the game. The result was 3,984 lines for the OO version and 4,989 for the FO version.[4] So, refactoring to the feature-oriented design produced an increase in size of about 25%. This increase was mainly caused by the finer granularity of the decomposition into classes and methods in the FO design.

Complexity. The OO version of the game contains 99 classes and interfaces, while in the FO version we counted 60 feature classes and 175 virtual classes. So by the size and the number of classes FO design seems to be more complicated. However, in order to evaluate complexity we should also analyze the dependencies between modules. In the object-oriented design most of classes are interdependent, i.e. each of these classes directly or indirectly depends on all the others. The cause of these cyclic dependencies is that most of the relationships of the game objects are bidirectional, and according the object-oriented design principles most of functionality is located in the classes of the game objects. In the FO design the picture is much better, because dependencies between features are strictly acyclic, which means that if feature A depends on feature B, there is no dependency in the opposite direction, neither directly nor indirectly. The acyclic design is in fact enforced, because we model feature dependencies by class inheritance relations. Besides, the dependencies are acyclic also at the level of packages, as can be seen in Fig. 2.

Understandability. The high level design has a clear structure, which corresponds to the structure of the specification. Because of this it is easier to understand the software. For example, if we want to understand how creature movement is implemented, we can at first read about it in specification and then locate all related functionality in the movement package. The classes of this package describing domain objects contain only the attributes and methods that are relevant to this feature. In order to understand how the feature works we need to study only the feature and the features it depends upon. We can even combine these features into a working application and experiment with their behavior in isolation from other features. However, we found that it is not intuitive that feature classes can use the functionality of transitively inherited features.

Testability. The advantage of the CAESARJ design for testing is that features can be tested independently from other unrelated features. For each feature group, such as Items and Battle, we defined a minimal application that consists only of the features of this group and their dependencies and used this application to test these features. In object-oriented applications test cases are often organized by use cases, which by their nature are similar to features, but such test cases usually depend upon much more source code than they actually need to test the corresponding functionality.

Maintainability. The improved traceability of requirements in the implementation also helped for bug fixing, because it was easier to locate the cause of the bug. A big advantage for maintainability is also the clear acyclic dependencies between features, because it is easy to identify what features can be evolved independently from each other and what is potential impact of the change in a certain feature. However, our experience with maintainability was mainly limited to the stabilization of the game and small improvements.

Tool Support. Tool support is very important for efficient software development. At the time when the case study was developed we provided very limited tool support: integration of CAESARJ compiler and source editor in the Eclipse platform, outline and inheritance hierarchy views, as well as a rudimentary support for debugging. The modular type

[3] The implementation of the game with CAESARJ is available for download at http://caesarj.org/index.php/Caesar/Examples

[4] The total numbers of lines are correspondingly 7,956 and 12,504.

checking provided by compiler and the debugging support have proved to be essential for enabling FOP with CAESARJ. However, for efficient programming there was also a strong need for navigation and search in the structure of the source code, code completion and automatic generation of imports. Another problem, was that CAESARJ compiler did not support incremental compilation, so even after a small change it took about a minute to recompile the project.

6. Conclusions

The results of the case study presented in the paper have shown that feature-oriented modularization not only increases reusability and extensibility of software, but also brings the implementation closer to the requirements and can improve the structure of dependencies in the program. As a consequence, the software becomes easier to understand, to test and to maintain. However, an efficient FOP requires an appropriate tool support: a compiler that supports incremental compilation and modular type checking, a debugger and a rich IDE support.

During development of the case study we elaborated methodology for feature modularization with virtual classes and mixin composition, for identification and modularization of features at the level of requirements and for refactoring from an object-oriented to a feature-oriented design in CAESARJ.

An important advantage of virtual classes is that their semantics support modular type checking and incremental compilation. The explicit relationships between the feature classes help to undestand the dependencies in the software, and the acyclicity of the inheritance graph enforce acyclic module structure. The mixin linearization semantics, in which classes inherited over multiple paths are shared, correspond to the intuition of dependencies between features. The main limitation of feature modularization with virtual classes and mixin composition is that it is a static variation mechanism, and thus it is not possible to turn on and off separate features at runtime.

7. Acknowledgments

We would like to thank the students who implemented the case study: Sacha Droste, Jörg Meyer, Lukas Pruschke and Carsten Schoger. The work was supported by the projects TopPrax (01|SC04A) and AOSD-Europe NoE (IST-2-004349).

References

[1] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *Proceedings of ICSE'06*, pages 122–131, New York, NY, USA, 2006. ACM Press.

[2] S. Apel, M. Rosenmueller, T. Leich, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of GPCE'05*, 2005.

[3] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of CaesarJ. *Transactions on AOSD I, LNCS*, 3880:135 – 173, 2006.

[4] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A monotonic superclass linearization for dylan. In *Proceedings of OOPSLA'96*, pages 69–82. ACM Press, 1996.

[5] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of SPLC'05*, pages 7–20, 2005.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. In *Proceedings of ICSE '03*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[7] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. *IEEE Software*, 11(5):89–94, 1994.

[8] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90. ACM SIGPLAN Notices 25(10)*, pages 303–311. ACM, 1990.

[9] J. Bredereke. On feature orientation and on requirements encapsulation using families of requirements. In *Objects, Agents, and Features*, pages 26–44, 2003.

[10] E. Ernst. *gbeta - a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[11] E. Ernst. Propagating class and method combination. In *Proceedings ECOOP'99*, LNCS 1628, pages 67–91, Lisboa, Portugal, June 1999. Springer-Verlag.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP '01*, 2001.

[13] S. R. Palmer and M. Felsing. *A Practical Guide to Feature-Driven Development*. Pearson Education, 2001.

[14] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of ECOOP'97*, pages 419–443, 1997.

[15] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proceedings of ECOOP'98*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[16] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of GPCE'07*, 2007.

[17] S. Trujillo, D. Batory, and O. Diaz. Feature refactoring a multi-representation program into a product line. In *Proceedings of GPCE '06*, pages 191–200, New York, NY, USA, 2006. ACM Press.

[18] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *J. Syst. Softw.*, 49(1):3–15, 1999.