GENERIC FEATURE MODULES: TWO-STAGED PROGRAM CUSTOMIZATION

Sven Apel, Martin Kuhlemann and Thomas Leich
Otto-von-Guericke-Universität Magdeburg
P.O. Box 4120, Magdeburg, Germany

Keywords: Feature-oriented programming, generics, program customization, software reuse, software product lines.

Abstract: With feature-oriented programming (FOP) and generics programmers have proper means for structuring software so that its elements can be reused and extended. This paper addresses the issue whether both approaches are equivalent. While FOP targets at large-scale building blocks and compositional programming, generics provide fine-grained customization at type-level. We contribute an analysis that reveals the individual capabilities of both approaches with respect to program customization. Therefrom, we extract guidelines for programmers in what situations which approach suffices. Furthermore, we present a fully implemented language proposal that integrates FOP and generics in order to combine their strengths. Our approach facilitates two-staged program customization: (1) selecting sets of features; (2) parameterizing features subsequently. This allows a broader spectrum of code reuse to be covered – reflected by proper language level mechanisms. We underpin our proposal by means of a case study.

1 INTRODUCTION

Feature-oriented programming (FOP) aims at feature modularity in software product lines (Batory et al., 2004). *Features* are increments in program functionality and reflect stakeholder requirements. The key idea of FOP is to map features one-to-one to *feature modules*. A feature module encapsulates all software artifacts that contribute to a feature in a cohesive unit. FOP targets mainly at large-scale components and compositional programming. Hence, program customization takes place at the level of feature modules, i.e., by selecting and composing a set of desired modules. It is not obvious how this scales down to fine-grained customization needs.

Fine-grained program customization is exactly the aim of an alternative approach, *generic and parameterized programming (GPP)* (Goguen, 1989; Austern, 1998). The key idea of GPP is to implement program structures as generic as possible and to use these in different contexts by parameterization. This approach is known as very fine-grained since it enables adjusting the types of program elements; but it is not well explored whether GPP is capable for program customization and reuse at a larger scale.

In this paper we examine the differences of FOP and GPP and how do they influence their program customization capabilities. Thereof we derive a set of guidelines in what situations to use which paradigm. Furthermore, we propose a language-driven approach of integrating FOP and GPP to cover a broad spectrum of scales of customization, which we call *generic feature modules*. Generic feature modules support *two-staged program customization*: (1) the desired features of a program are selected; (2) the corresponding feature modules are parameterized for fine-grained customization. Besides customizability, this promotes reuse of feature modules and offers potential for reasoning about explicitly represented configuration knowledge in form of parameters.

To underpin our proposal, we present a fully functional compiler on top of FEATUREC++[1]. We use our compiler to apply generic feature modules to a case study.

In this paper we make the following contributions:

- We compare FOP and GPP with respect to program customization; we infer guidelines in what situations which paradigm suffices.
- We propose an integrated language approach that integrates FOP and GPP.
- We contribute a fully functional compiler that implements our language proposal.
- We underpin our proposal by means of a case study.

[1]http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc

Apel S., Kuhlemann M. and Leich T. (2006).
GENERIC FEATURE MODULES: TWO-STAGED PROGRAM CUSTOMIZATION.
In *Proceedings of the First International Conference on Software and Data Technologies*, pages 127-132
DOI: 10.5220/0001311301270132

2 BACKGROUND

2.1 Feature-Oriented Programming

FOP studies the modularity of features in product lines (Batory et al., 2004). The idea of FOP is to build software (individual programs) by composing features that are first-class entities in design and implementation. Features refine other features incrementally. Hence, the term *refinement* refers to the changes a feature applies to others. This *step-wise refinement* leads to conceptually layered software designs.

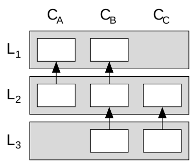

Figure 1: Mixin layers.

Feature modules. Feature modules implement features. *Mixin layers* is one implementation technique that aims at source code artifacts (Smaragdakis and Batory, 2002).[2] Typically, features are not implemented by single classes; often, a whole set of *collaborating* classes contributes to a feature. Classes play different *roles* in different *collaborations*. A mixin layer is a static component encapsulating fragments of several different classes (roles) so that all fragments are composed consistently. Figure 1 depicts a stack of three mixin layers ($L_1 - L_3$) in top down order. Mixin layers crosscut multiple classes ($C_A - C_C$). White boxes represent mixins and gray boxes feature modules.

Feature modules in C++. FEATUREC++ is an extension to C++ that supports FOP (Apel et al., 2005). Feature modules contain classes and refinements to classes, which are declared by *refines*. Figure 2 depicts a class that implements a simple list (Lines 1-4) and a refinement that determines its size (Lines 5-8). Refinements may introduce new attributes and methods or may extend methods of their parent classes. In our example, the refinement introduces a *size* field (Line 6), a *getSize* method (Line 6), and extends the *put* method by code for item counting (Line 7). To access the extended method the *super* keyword is used (Line 7). The class *List* and its refinement belong to two distinct development steps implementing two individual features: a basic list feature ($list$) and a feature for determining its size ($size$).

[2]Feature modules may contain manifold types of software artifacts, not only source code (Batory et al., 2004).

```
1 class List {
2     Item *head;
3     void put(Item *i) { i->next = head; head = i;   }
4 };
5 refines class List {
6     int size;   int getSize() { return size; }
7     void put(Item *i) { super::put(i); size++; }
8 };
```

Figure 2: Refining a list with code for determining the size.

2.2 Generic and Parameterized Programming

GPP is about generalizing software components so that they can be easily reused in a wide variety of situations (Goguen, 1989; Czarnecki and Eisenecker, 2000). It serves the need for customizing components to specific requirements. In this paper we use C++ templates as a representative approach of GPP (Austern, 1998).

They key idea of GPP is that software components often do not rely on specific data types, but can operate with arbitrary types. In order to reuse these components for all possible kinds of types, they are implemented against type parameters that act as placeholders for different concrete data types. To use a generic component in a specific context, it has to be instantiated by passing a concrete type to the component.

Figure 3 depicts a standard GPP example: a generic list implementation (Lines 1-5). It is generic because it relies on a template parameter *_ItemT* (Line 1). Therefore, it can be used polymorphically in different contexts. When instantiating a list object a programmer has to pass a concrete data type to the list, e.g. *Item* or *Thread* (Line 6).

```
1 template <typename _ItemT> class List {
2     typedef _ItemT ItemT;
3   ItemT *head;
4   void put(ItemT *i) { i->next = head; head = i; }
5 }; ...
6 int main() { List<Item> il; List<Thread> tl; }
```

Figure 3: A generic list and two concrete variants.

3 ANALYSIS OF FOP AND GPP

In this section, we analyze the different capabilities of FOP and GPP to implement customizable software by means of an example.

3.1 A Product Line of Linked Lists

As example, we choose a standard problem: a product line of linked lists (*list product line – LPL*), adopted from (Czarnecki and Eisenecker, 2000). Figure 4 depicts the feature model of LPL.

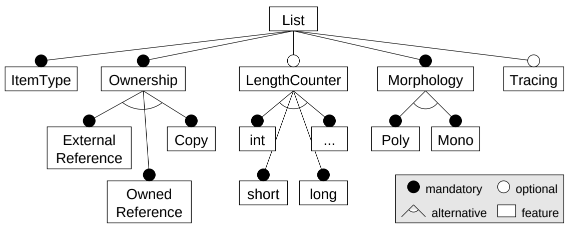

Figure 4: Feature model of LPL.

The feature *ItemType* abstracts over possible data types of items that are stored in the list; it is mandatory. Lists can have different ownership relationships to their stored items: (1) references to items are maintained externally; (2) lists own references to items and therefore they are responsible for releasing the allocated memory when items are removed; (3) lists copy items that are stored; they are responsible for cleaning up when items are removed. These features are *alternative* – only one variant can be selected for a concrete list. Furthermore, lists have different morphologies: (1) a list stores items of the same type only (monomorphic) or (2) of different types (polymorphic). Only one feature variant can be selected. Lists may have an *optional* length counter feature that counts the number of stored elements. The counter itself can be implemented using different data types (*short*, *int*, ...). Finally, lists have an optional feature for tracing the operations on list objects.

3.2 Implementation of Lpl

Ideally, when implementing this product line the programmer implements each feature via one feature module. This is in line with the methodology and principles of FOP. Applying this methodology to our example, we would have to implement at least 11 feature modules, assuming one basic list, three different length counter types, and one item type.

FOP implementation. Figure 5 shows a basic list and a tracing feature implemented in FEATUREC++. It can be seen that both features are implemented as mixins, encapsulated in feature modules (not shown). Up to here FOP works fine. But implementing other features reveals the weaknesses of FOP. For example, it is not obvious how to implement different variants of the item type feature, i.e., different types of items. One can implement for each type a distinct basic list, e.g., a list of item objects and a list of thread objects. Unfortunately, this results in replicated code (one feature module per type) for the base feature and all subsequent added features!

```
class List {
  Item *head;
  void put(Item *i) { i->next = head; head = i; }
};
refines class List {
  void put(Item *i) { super::put(i); trace(i); }
};
```

Figure 5: Refining a list with a tracing feature.

Another approach would be introducing an abstract base class of all items. Hence, the list implementation becomes invariant with respect to the item type. New item types inherit from the abstract base class. This solution imposes performance penalties and a higher resource consumption due to dynamic binding; it may demand for dynamic type checking.

Similar problems occur when implementing other features that come in slightly different variants, e.g., the length counter type, an allocator or iterator type.

GPP implementation. Readers familiar with GPP may notice that the problematic features could be easily implemented using template parameters or alternative mechanisms for generics. Figure 6 depicts a generic list that expects an item type and a type for a length counter, as well as two concrete variants, an item object list with integer counter and a thread list with short integer counter. With GPP, a programmer can define a concrete variant at compile time in a type safe manner. However, one has always to anticipate a potential variation point when implementing a feature. Moreover, it is not obvious how GPP can implement larger program features, e.g., implementing the size feature.

```
template <typename _ItemT, typename _SizeT>
class List {
    typedef _ItemT ItemT; typedef _SizeT SizeT;
  ItemT *head; SizeT size;
  void put(ItemT *i){i->next=head; head=i; size++;}
}; ...
int main() {
    List<Item,int> iil; List<Thread,short> tsl;
}
```

Figure 6: A generic list implementation.

3.3 A Comparison of Fop and Gpp

It seems that both, FOP and GPP, are necessary to implement a highly customizable software. While feature modules encapsulate large-scale features, GPP allows for fine-grained tuning. The questions that arise are what approach is useful under which circumstances? Are both approaches equally expressible? How to integrate both in a consistent way? In this section we shed more light on these issues and provide guidelines for the efficient use of FOP and GPP.

FOP. Feature modules usually contain a set of classes. These classes are introductions or refinements to existing classes. Thus, feature modules implement mainly increments to a program's functionality. A refinement may extend existing methods by executing code around a method execution. Although refinements rely on a reasonable structure of the base program, they do not expect explicitly represented variation points (e.g. *hooks*) for being applicable. A feature module binds to the natural structure of the base program and may apply unanticipated changes. A consequence of its encapsulation property is that a feature module can implement a variant that concerns multiple variation points, e.g., a synchronization feature extends simultaneously a list *and* its iterator.

However, the fact that feature modules rely on given structural abstractions defines the minimal granularity of customization of software built of feature modules. It is not possible to refine a base feature at statement level to change existing types, etc. However, achieving even so customizability at type level (1) imposes performance penalties due to dynamic binding, i.e., implementing a feature against an abstract class, and (2) it results in redundant code, i.e., for each type a distinct feature module. That is, FOP imposes a complexity overhead at small scales.

GPP. GPP supports program customizability at a smaller scale than FOP. For example, templates enable the programmer to customize program structures down at type level by parameterizing types of used variables and arguments. This methodology implies that programmers have to anticipate changes to and variants of a program. Variation points are explicit and fixed; they are an inherent part of the referring modules. n variation points demand for n parameters. This in-language approach to customization facilitates static type-safety.

Although templates can be used to implement entire feature modules (Smaragdakis and Batory, 2002), they are mainly suited for fine-grained customization. This is because the overhead of complexity to maintain the template expressions grows considerably for large-scale features.

Table 1 summarizes our observation of the properties of FOP and GPP achieving customizability and reusability. It is intended to serve as guideline for programmers to decide when to use which technique.

4 GENERIC FEATURE MODULES

As our analysis revealed, both, GPP and FOP, have strengths at different scales of customization. Consequently, we propose the notion of *generic feature modules* that integrates mechanisms of GPP into FOP.

Table 1: Comparison of FOP and GPP.

	FOP	**GPP**
scale	large	small
granularity	methods, classes	statements, types
var. points	implicit	explicit
extensions	unanticipated	anticipated
locality	multiple points	single point

Templates. Mixins within feature modules can declare a list of template parameters (Fig. 7). In contrast to traditional classes, subsequently applied refinements to a class may extend its (possibly empty) template parameter list. This is useful because in this way the set of parameterizable types has not to be anticipated up front. Figure 8 depicts a refinement to the basic list that implements the size feature. The type of the counter is passed via template parameter; for that, the template list is extended (Line 1).

```
template <typename _ItemT> class List {
    typedef _ItemT ItemT;
  ItemT *head;
  void put(ItemT *i) {  i->next = head; head = i; }
};
```

Figure 7: A list template.

```
1 template <typename _ItemT, typename _SizeT>
2 refines class List {
3     typedef _ItemT ItemT;   typedef _SizeT SizeT;
4   SizeT size;
5   void put(ItemT *i) { super::put(i); size++; }
6 };
```

Figure 8: Extending the parameter list in a refinement.

However, extending the parameter list implies that clients have two provide a set of expected parameters, which may vary depending on the current feature selection. We address this issue later.

Parameterizing feature modules. Templates can be used to parameterize feature modules statically by a set of types. Figure 9 depicts a stack of generic feature modules. Each feature module extends existing structures, but also the template parameter list. This enables the individual features to declare new parameters that are intended for customizing themselves. Composing a concrete program out of this set of features allows the final program to be parameterized with concrete types.

This example illustrates the two-staged nature of the configuration process imposed by generic feature modules: (1) a subset of features is selected; (2) the selected features are parameterized.

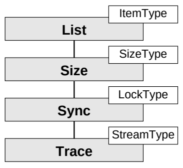

Figure 9: Generic feature modules.

```
class ListConfig {
  typedef Item ItemT;
  typedef short LengthT;
  typedef Array ArrayT;
  typedef ostream StreamT;
};
```

Figure 10: Example of a configuration repository.

Configuration repositories. Since each refinement may potentially extend the template parameter list, the programmer may easily get lost in the mass of parameters and values (constants and types). In order to simplify the parameterization and to improve code readability without constraining its expressibility and flexibility, we adopt the notion of *configuration repositories* (Czarnecki and Eisenecker, 2000). A configuration repository encapsulates the overall configuration knowledge that is passed via parameters (Fig. 10).

The key benefit of using configuration repositories in generic feature modules is that now classes and refinements expect only one parameter, namely a repository. Each refinement takes out only this configuration information that it depends on. That solves the problem that each client has to know what set of parameters are expected by the current selection of feature modules. Instead, the configuration repository can be defined in one location; clients do not need to know about its overall structure.

Figure 11 depicts a reimplementation of the basic list and the size feature. Both use different subsets of this repository.

```
template <typename _Config> class List {
    typedef _Config Config;
    typedef typename Config::ItemT ItemT;
    ItemT *head;
    void put(ItemT *i) { i->next = head; head = i; }
};
template <typename _Config> refines class List {
    typedef _Config Config;
    typedef typename Config::ItemT ItemT;
    typedef typename Config::SizeT SizeT;
    SizeT size;
    void put(ItemT *i) { super::put(i); size++; }
};
```

Figure 11: Customizing features via repositories.

5 CASE STUDY

For demonstrating the practical applicability of our integrated language approach, we extended the LPL by several new features, all implemented in FEATUREC++: LPL consists of 20 features and 11 parameters. Each feature expects on average 3 parameters, up to 6 parameters in maximum. Table 2 summarizes all features of LPL and the number of the parameters used. Note that some parameters are used by more than one feature. This study demonstrates that in highly configurable libraries, such as LPL, there is a demand for parameterizing features. The parameters increase the possible solution space of LPL significantly.

Table 2: Features and numbers of their parameters.

Feature	No.	Feature	No.	Feature	No.
Base	6	Alloc	3	Contain	3
Length	3	Bounded	3	Connect	3
Trace	3	Sync.	4	Delete	2
DblLink	4	Array	4	Stack	3
Iter	3	Compare	3	Queue	3
Sorted	3	Clone	2	Set	2
Insert	3	Persist	3	Map	4

Discussion. An alternative version of LPL that omits templates could be implemented using abstract classes. Each parameter would be passed as object reference via the list's constructor. Different parameter settings would be implemented by subclassing abstract classes that serve for representing parameters.

Besides the mentioned penalties imposed by abstract classes, it is not obvious how to bundle parameters in repositories without type definitions and templates. Nevertheless, many design and customization decisions are made upfront. In contrast to using FOP standalone, generic feature modules provide a well-aligned symbiosis between GPP and FOP that supports customizable large-scale components with configuration support and static type safety.

6 RELATED WORK

GenVoca is an architectural model for large-scale components and collaboration-based designs (Batory and O'Malley, 1992). Principally, GenVoca distinguishes between horizontal and vertical parameters. The vertical parameters are instrumental in defining the vertical refinement hierarchies of layers, whereas the horizontal parameters provide for variability within a single layer (Goguen, 1996). Mapping this to our approach, concrete configuration repositories *encapsulate* horizontal parameters; vertical parameters are the classes that will be refined. Interestingly, we integrate the configuration repositories into the GenVoca layers themselves, enabling subsequent refinement.

Our notion of configuration repositories builds on an earlier proposal: They are implemented as *trait classes*, i.e., classes that aggregate a set of types and constants to be passed to a template as a parameter (Myers, 1995). Additionally, we provide means for integrating configuration repositories into feature modules.

Consul is an integrated approach to manage variabilities and customization (Beuche et al., 2004). It provides a proprietary component model and a logic-based representation of configuration knowledge. The component model lacks the flexibility of mixin composition and compositional reasoning; the logic-based approach of customization is powerful, but relies on a complex program transformation approach. Issues as type-safety are not discussed.

Making configuration knowledge and management explicit is a kind of meta-programming. A comprehensive overview of static meta-programming in C++ is given in (Czarnecki and Eisenecker, 2000). There it is shown how configuration repositories can be further processed to automatically determine parameter settings on the basis of partially specified configurations.

7 CONCLUSION

In this paper, we examined the capabilities of FOP and GPP for implementing reusable software: FOP performs well for implementing composable large-scale building blocks, but it imposes a complexity overhead when implementing fine-grained customizable features; GPP focuses mainly on reuse in the small by providing proper means for fine-grained program customization, but lacks abstraction and composition capabilities for programming in the large. Consequently, we proposed an integrated language-level approach for supporting both kinds of customization and reuse. Generic feature modules impose a two-staged program customization: After selecting and composing features, they can be parameterized to adapt them to a specific application context. A distinguishing feature of our approach is that we integrate the configuration knowledge into the associated feature modules to improve the encapsulation properties. For implementation, C++ templates are only a first attempt to demonstrate our ideas. Exploring sophisticated mechanisms for representing and reasoning about configuration knowledge is part of further work.

ACKNOWLEDGEMENTS

We thank Don Batory and Christian Kästner for useful comments on earlier drafts of this paper. This research is sponsored in parts by the German Research Foundation (DFG), project number SA 465/31-1, as well as by the German Academic Exchange Service (DAAD), PKZ D/05/44809.

REFERENCES

Apel, S. et al. (2005). FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of International Conference on Generative Programming and Component Engineering (GPCE)*.

Austern, M. (1998). *Generic Programming and the STL*. Addison-Wesley.

Batory, D. and O'Malley, S. (1992). The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4).

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6).

Beuche, D., Papajewski, H., and Schröder-Preikschat, W. (2004). Variability Management with Feature Models. *Science of Computer Programming*, 53(3).

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

Goguen, J. (1996). Parameterized Programming and Software Architecture. In *Proceedings of International Conference on Software Reuse (ICSR)*.

Goguen, J. A. (1989). Principles of Parameterized Programming. In *Software Reusability: Vol. 1, Concepts and Models*. ACM Press.

Myers, N. (1995). Traits: A New and Useful Template Technique. *C++ Report*.

Smaragdakis, Y. and Batory, D. (2002). Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2).