Synthesizing Objects

Krzysztof Czarnecki[1] and Ulrich W. Eisenecker[2]

[1]DaimlerChrysler AG Research and Technology, Ulm, Germany
czarnecki@acm.org

[2]University of Applied Sciences Heidelberg, Germany
ulrich.eisenecker@t-online.de

Abstract. This paper argues that the current OO technology does not support reuse and configurability in an effective way. This problem can be addressed by augmenting OO analysis and design with feature modeling and by applying generative implementation techniques. Feature modeling allows capturing the variability of domain concepts. Concrete concept instances can then be synthesized from abstract specifications.

Using a simple example of a configurable list component, we demonstrate the application of feature modeling and how to implement a feature model as a generator. We introduce the concepts of configuration repositories and configuration generators and show how to implement them using object-oriented, generic, and generative language mechanisms. The configuration generator utilizes C++ template metaprogramming, which enables its execution at compile-time.

1 Introduction

In the early days of OO, there used to be the belief that objects are reusable by their very nature and that reusable OO software simply “falls out” as a byproduct of application development. Today, the OO community widely recognizes that nothing could be further from the truth. Reusable OO software has to be carefully engineered and engineering *for reuse* requires a substantial investment. One of the weaknesses of current OO Analysis and Design (OOA/D) is the inadequate support for variability modeling. As suggested by the work in the reuse community (e.g. [CN98, GFA98, Cza98]), this problem can be addressed by augmenting OOA/D with *feature modeling*. Feature modeling was originally introduced in the Feature-Oriented Domain Analysis (FODA) method [KCH+90] as a technique for modeling commonalities and variabilities within a domain. Since reusable models may contain a significant amount of variability, rigid class hierarchies are inappropriate for implementing such models. They are more adequately implemented as flexible, highly parameterized component classes. Concrete component configurations can be synthesized from abstract descriptions by a *configuration generator*. An important aspect of such designs is the separation between the components and the configuration knowledge, which is achieved using a *configuration repository*. We demonstrate these concepts using a concrete example and also show how to implement them using object-oriented and generic language mechanisms in C++. The configuration generator utilizes C++ template metaprogramming, which enables its execution at compile-time.

The rest of the paper is organized as follows. Section 2 discusses the weaknesses of OOA/D in the context of reuse and configurability. Section 3 introduces the concept of feature models. Section 4 makes the connection between feature models and object synthesis. Section 5 contains a concrete example starting with a feature diagram and concluding with a configuration generator. Section 6 lists extensions not included in the example due to space limitations. Section 7 discusses two libraries implemented using the techniques from this paper. Section 8 discusses related work. Section 9 concludes by making the connection to active libraries and Generative Programming.

2 Problems of Object Technology in the Context of Software Reuse

Two important areas of OO technology addressing reuse are frameworks and design patterns. A framework embodies an abstract design for a *family* of related systems in the form of collaborating classes. Similarly, design patterns provide reusable solutions to recurring design problems across different systems. Patterns, as a documentation form, also proved useful in capturing reusable solutions in other areas such as analysis, architecture, and organizational issues. Unfortunately, only very few OOA/D methods provide any support for the development of frameworks.[1] Similarly, there is little systematic support in both finding and applying patterns.

Most OOA/D methods focus on developing single systems rather than families of systems. Given this goal, these methods are inadequate for developing reusable software, which requires focusing on classes of systems rather than single systems. A comparison between Domain Engineering methods (e.g. ODM [SCK+96] or FODA [KCH+90]), which are designed for engineering system families, and OOA/D methods reveals the following deficiencies of the latter:

- *No distinction between engineering for reuse and engineering with reuse*: Taking reuse into account requires splitting the OO software engineering process into engineering *for reuse* (i.e. Domain Engineering) and engineering *with reuse* (i.e. Application Engineering). OOA/D methods come closest to Application Engineering, with the difference that Application Engineering focuses on reusing available assets produced during Domain Engineering.

[1] As of writing, OOram [Ree96] is the only OOA/D method known to the authors which truly recognizes the need for a specialized engineering process for reuse. The method includes a domain scoping activity involving the analysis of different classes of consumers. However, the method does not incorporate feature modeling.

- *No domain scoping phase*: Since OOA/D methods focus on engineering single systems, they lack a domain scoping phase, where the target class of systems is selected. Also, OOA/D focuses on satisfying “the customer” of a single system rather than analyzing and satisfying *stakeholders* (including potential customers) of a class of systems.
- *Inadequate modeling of variability*: The only kind of variability modeled in current OOA/D is intra-application variability, e.g. variability of certain objects over time and the use of different variants of an object at different locations within an application. Domain Engineering, on the other hand, focuses on variability across different systems in a domain for different users and usage contexts. Since modeling variability is fundamental to Domain Engineering, Domain Engineering methods provide specialized notations for expressing variability.

Thus, a general problem of all OOA/D methods is inadequate modeling of variability. Although the various modeling techniques used in OOA/D methods support variability mechanisms (e.g. inheritance, aggregation, and static parameterization), OOA/D methods do not include an abstract and concise model of commonality, variability, and dependencies. There are several reasons for providing such a model:

- Since the same variability may be implemented using different variability mechanisms in different models, we need a more abstract representation of variability (cf. Section 3).
- The user of reusable software needs an explicit and concise representation of available features and variability.
- The developer of reusable software needs to be able to answer the question: why is a certain feature or variation point included in the reusable software?

The lack of domain scoping and explicit variability modeling may cause two serious problems:

- relevant features and variation points are missing; and
- many features and variation points are included but never used; this causes unnecessary complexity and cost (both development and maintenance cost).

Covering the right features and variation points requires a careful balancing between current and future needs. Thus, we need an explicit model that summarizes the features and the variation points and includes the rationale and the stakeholders for each of them. In Domain Engineering, this role is played by a *feature model*. A feature model captures the reusability and configurability aspect of reusable software.

3 Feature Models

Domain concepts that we try to model in reusable software are inherently complex. Even the implementation of a simple container object requires a multitude of design decisions, e.g. type of elements, what kind of iterators it provides, ownership (i.e. whether the container keeps the original elements or their copies and whether it is responsible for deallocating the elements or not), memory allocation (on the stack, on the heap, in a persistent store, etc.), memory management (e.g. whether growing is possible or not), error detection (e.g. whether bounds checking is done or not), synchronization of concurrent access, etc. If the container is to be reusable, many of these decisions have to be changeable since different usage contexts will have different requirements. The changeable design decisions span a design space containing *variation points* [JGJ98] at which different design alternatives and options may be selected.

In Domain Engineering, each of the alternative design decisions is represented by a *feature*.[2] Following the conceptual modeling perspective, a feature is defined as an important property of a concept. For example, the features of a *list* may include *ordered*, *singly linked*, *keeps track of its size*, *can contain elements of different types*, etc. In the context of Domain Engineering, features represent reusable, configurable requirements and each feature has to make a difference to someone, e.g. a stakeholder or a client program. For example, when we build an order processing system, one of the features of the pricing component could be *aging pricing strategy*, e.g. you pay less for older merchandise. This pricing strategy might be particularly interesting to companies selling perishable goods.

Features are organized into *feature diagrams* [KCH+90], which reveal the kinds of variability contained in the design space. An example of a feature diagram is shown in Fig. 1. It describes a simple model of a car. The root of the diagram represents the concept *car*. The remaining nodes are features. The diagram contains four kinds of features (see [Cza98] for other kinds of features and a full description of the feature diagram notation):[3]

- *Mandatory features*: Mandatory features are pointed to by simple edges ending with a filled circle. The features *car body*, *transmission*, and *engine* are mandatory and thus part of any car.
- *Optional features*: Optional features are pointed to by simple edges ending with an empty circle, e.g. *pulls trailer*. A car may pull a trailer or not.

[2] Features and feature modeling have been propagated by the Feature-Oriented Domain-Analysis (FODA) method [KCH+90].

[3] Or-features are not part of the notation in [KCH+90]. They were introduced in [Cza98].

- *Alternative features*: Alternative features are pointed to by edges connected by an arc, e.g. *automatic* and *manual*. Thus, a car may have an automatic or manual transmission.
- *Or-features*: Or-features are pointed to by edges connected by a filled arc, e.g. electric and gasoline. Thus, a car may have an electric engine, a gasoline engine, or both.

A feature diagram is usually accompanied by additional information, such as short semantic description of each feature, rationale for each feature, stakeholders and client programs interested in each feature, examples of systems with a given feature, constraints between features (e.g. which feature combinations are illegal and which features imply the selection of which other features), default dependency rules (i.e. which feature suggests the selection of which other features), availability sites (i.e. where, when, and to whom a feature is available), binding modes (e.g. whether a feature is bound dynamically or statically), open/closed attributes (i.e. whether new subfeatures are expected), and priorities (i.e. how important is a feature). All this information together constitutes a *feature model* (see [Cza98] for a detailed description).

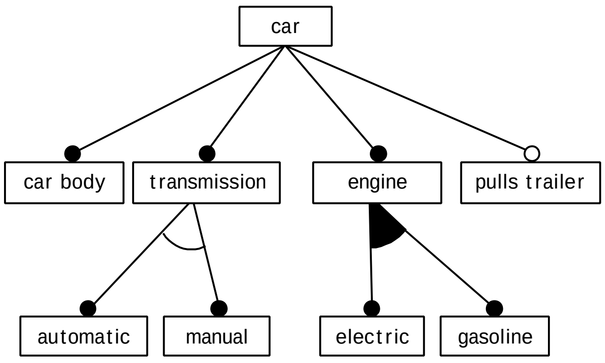

Fig. 1. A sample feature diagram of a car

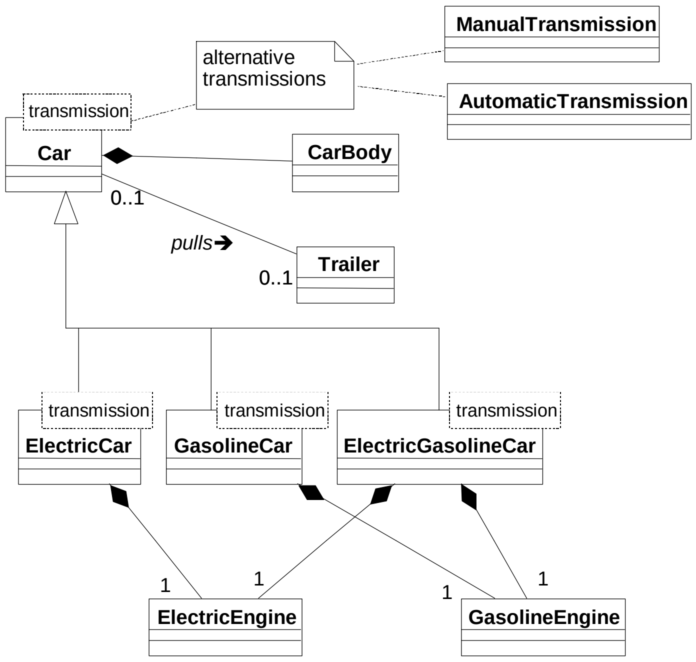

Fig. 2. One possible implementation of the simple car from Fig. 1 using a UML class diagram

A feature model describes the configurability aspect of a concept at a high level of abstraction. Indeed, feature models are more abstract than object diagrams. Fig. 2 shows *one possible implementation* of the car feature diagram in UML. In order to be able to represent our car as an UML class diagram, we had to make a number of concrete decisions about which variability mechanisms to use. The implementation in Fig. 2 uses static parameterization for the transmission, inheritance for the engine, and an association with the cardinality 0..1 for the trailer. Of course, other implementation choices are also possible. For example, we could also use inheritance or dynamic parameterization for the transmission.

The fundamental flaw of the OOA/D methodology in the context of software reuse is to start modeling in terms of variability implementation mechanisms such as inheritance, dynamic parameterization, and static parameterization rather than more abstract variability representations.

For this reason, as suggested by various researchers [CN98, GFA98, Cza98], we use feature models in addition to other OO models.

4 Feature Models and Object Synthesis

We can implement the functionality represented by a feature model using a set of *implementation components*. The idea is that the implementation components can be configured to yield the different systems covered by a feature model. We can utilize different technologies for implementing the components. One possibility is to use objects.

As stated, object-oriented designs support variability using different mechanisms and techniques. Nearly every design pattern listed in [GHJV95] is about making some part of a design variable, e.g. *bridge* lets you vary the implementation of an object; *strategy* turns an algorithm into a parameter; *state* allows you to vary behavior depending on the state; *template method* provides a way to vary computation steps while keeping the algorithm structure constant. Most of the standard implementations of these patterns utilize dynamic parameterization allowing parameters to vary at runtime. However, reusable models are also full of static variation points, i.e. ones that vary from application to application rather than within one application at runtime. Such variation points are better implemented using static parameterization. We will see concrete examples in Section 5.2.

What is the relationship between features and the implementation components? In this context, we differentiate between three kinds of features:

- *Concrete features*: A concrete feature is directly implemented by one component, e.g. sorting can be directly implemented by a sorting component.
- *Aspect features*: An aspect feature is implemented as an *aspect* in the sense of Aspect-Oriented Programming [KLM+97]. An *aspect* is a kind of modularity which affects many other components, e.g. a declarative description of the synchronization of a number of components. Aspects are usually implemented using an aspect weaver, i.e. a language processor which automatically implements the aspect by coordinating (e.g. merging or interpretatively scheduling) the component code and the aspect code.
- *Abstract features*: Abstract features do not have direct implementations whatsoever. They are implemented by an appropriate combination of components and aspects. Examples of abstract features are performance requirements, such as optimize for speed or space or accuracy.

We say that features make up the *problem space*, whereas the implementation components and aspects constitute the *solution space*. Both spaces have different structures: abstract features have no directly corresponding components or aspects in the solution space, and there may be some “implementation-detail” components and aspects that have no direct correspondence in the problem space. Both spaces are also driven by different, usually conflicting goals: The problem space consists of high-level concepts and features which application programmers would like to work with, while the components in the solution space are designed

- as elementary components combinable in as many ways as possible,
- to avoid any code duplication, and possibly
- to be reusable across many product lines.

The separation between problem and solution space allows us to satisfy both the problem space goals and the solution space goals. It also promotes software evolution since both spaces may be modified (to a certain degree) independently.

The mapping between the problem and the solution space is facilitated by *configuration knowledge*, which consists of

- *constraints* specifying which feature combinations are illegal and which features require the selection of which other features,
- *default dependency rules* specifying which feature suggests the selection of which other features, and
- *mapping rules* specifying which feature combinations require which combinations of components and aspects.

Since some of the variation points in the problem space are static, we need a way to evaluate the configuration knowledge and compose the appropriate components and aspects at compile time. What does this mean if we use objects to implement the solution space? We need a metaprogramming facility which synthesizes objects according to abstract featural descriptions at compile time. As with any new programming concept, direct support in a programming language is desirable. Surprisingly, the concepts outlined above can be implemented utilizing the OO and generic features of C++. We demonstrate the necessary techniques in the following section.

5 Example: Synthesizing a List Container

5.1 Feature Model

Suppose we want to develop a reusable singly linked list. First, we need to analyze the requirements different applications may have for a list in areas such as type of elements, element traversal, storage layout, ownership, memory allocation and memory management, error detection, synchronization of concurrent access, etc. We document the variable features in different areas using feature diagrams. Finally, we should prioritize the features according to project goals (e.g. target customers and applications). For the purpose of this presentation, we show the implementation of a list covering the features shown in Fig. 3. `ElementType` is the type of the elements stored in the list and is a free parameter (i.e. any type can be substituted for `ElementType`), as indicated by the square brackets. `Ownership` indicates whether the list keeps references to the original elements and is not responsible for element deallocation (i.e. `external reference`), or keeps references and is responsible for element deallocation (i.e. `owned reference`), or keeps copies of the original elements and is responsible for their allocation and deallocation (i.e. `copy`). `Morphology` describes whether the list is `monomorphic` (i.e. all elements have the same type) or `polymorphic` (i.e. can contain elements of different types). Each list element may also contain a length counter allowing for a length operation of a constant time complexity. `LengthType` is the type of the counter. Finally, the list may optionally trace its operation, e.g. by logging operation calls to the console. (Of course, this diagram could be extended with further features.)

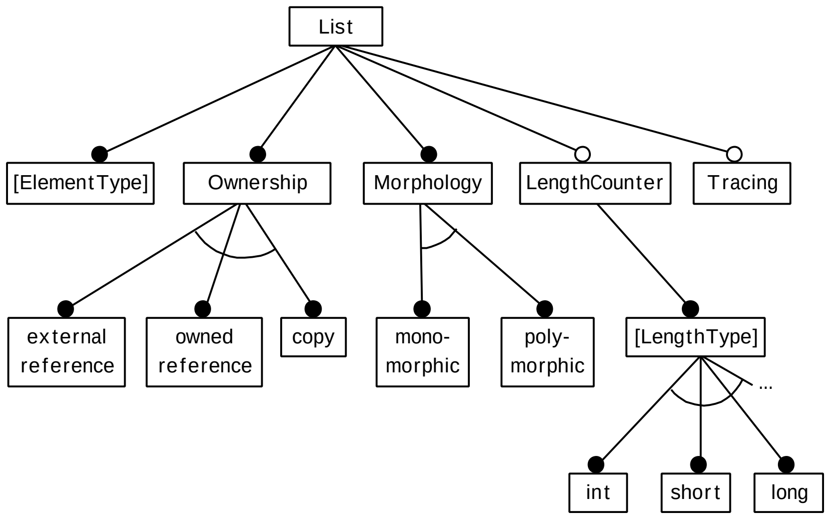

Fig. 3 Feature diagram of a simple list container

5.2 Implementation Components

As stated, we can apply different OO techniques to implement the variability contained in the list feature diagram. Here we show an implementation using static parameterization (including parameterized inheritance). Our implementation consists of the following components:

- `PtrList`, which implements the basic list functionality including the accessing operations for the head (`head()` and `setHead()`) and tail (`tail()` and `setTail()`);

- `LenList`, which is a wrapper for adding a length counter to a list;
- `TracedList`, which is a wrapper for adding tracing to a list.

Additionally, there are three sets of small components for implementing ownership and morphology:

- *destroyers*, which deallocate memory;
- *type checkers*, which check the type of the elements added to the list;
- *copiers*, which copy the elements.

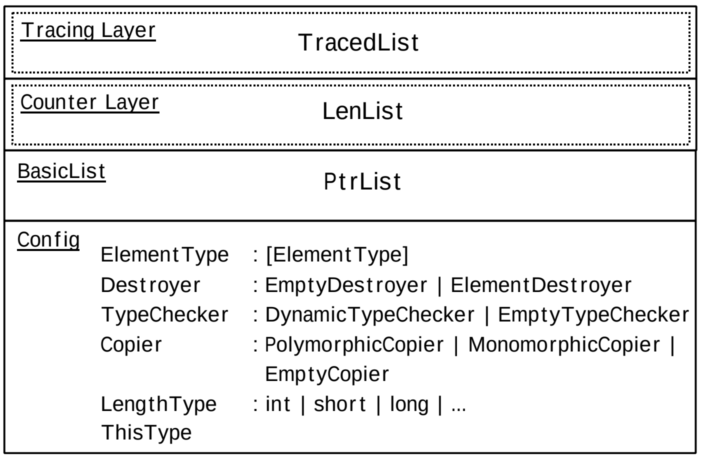
Tracing Layer
TracedList
Counter Layer
LenList
BasicList
PtrList
Config
ElementType : [ElementType]
Destroyer : EmptyDestroyer | ElementDestroyer
TypeChecker : DynamicTypeChecker | EmptyTypeChecker
Copier : PolymorphicCopier | MonomorphicCopier | EmptyCopier
LengthType : int | short | long | ...
ThisType

Fig. 4 Target architecture of the list container

```
List              : TracedList[ OptCounterList] |  OptCounterList
OptCounterList    : LenList[ BasicList] |  BasicList
BasicList         : PtrList[Config]
Config            :
  ElementType     : [ElementType]
  Destroyer       : EmptyDestroyer | ElementDestroyer
  TypeChecker     : DynamicT ypeChecker | EmptyTypeChecker
  Copier          : PolymorphicCopier | MonomorphicCopier | EmptyCopier
  LengthType      : int | short | long | ...
  ReturnType //the final list type
```

Fig. 5 GenVoca grammar of the list container

The implementation components constitute a layered architecture (or a GenVoca architecture [BO92]) shown in Fig. 4. Each rectangle represents a layer. A layer drawn on top of another layer refines the latter by adding new functionality. In general, a layer may contain more than one alternative component. A component from one layer takes another component from the layer below as its parameter, e.g. `LenList` may take

PtrList as its parameter. A layer containing a dashed rectangle is optional (see [Cza98] for details of this notation). The bottom layer is referred to as a *configuration repository* (abbr. Config). A configuration repository provides types needed by the other layers under standardized names. It works as a kind of registry, which components may use to retrieve configuration information from and also to exchange such information among each other. A configuration repository allows us to separate the configuration knowledge from the components. It minimizes dependencies between components since the components do not exchange types and constants directly, but through the repository. As we show later, we implement a configuration repository as a so-called traits class [Mye95]. The use of trait classes as configuration repositories was first proposed in [Eis96].

Alternatively, the list architecture can be described as a so-called GenVoca grammar (see Fig. 5). The vertical bar separates alternatives and parameters are enclosed in square brackets, e.g. List is either TracedList parameterized by OptCounterList or OptCounterList. The names exported by the configuration repository Config are listed below it. ReturnType is the final type of a configured list.

We can implement PtrList in C++ as follows:[4]

```
template<class  Config_>
class PtrList
{
public:
   //make Config available as a member type
   typedef Config_ Config;

private:
   //retrieve needed types from the configuration repository
   typedef  typename  Config::ElementType ElementType;
   typedef  typename  Config::SetHeadElementType SetHeadElementType;
   typedef  typename  Config::ReturnType  ReturnType;

   typedef  typename  Config::Destroyer Destroyer;
   typedef  typename  Config::TypeChecker  TypeChecker;
   typedef  typename  Config::Copier Copier;

public:
   PtrList( SetHeadElementType& h,  ReturnType *t = 0) :
      head_(0), tail_(t)
   {  setHead(h); }

   ~PtrList()
   {  Destroyer::destroy(head_); }

   void  setHead( SetHeadElementType& h)
   {  TypeChecker::check(h);
```

[4] The keyword typename is required by ANSI C++ to tell the compiler that a member of a template parameter is expected to be a type.

```
      head_ = Copier::copy(h);
   }

   ElementType& head()
   {  return *head_; }

   void setTail(ReturnType *t)
   {  tail_ = t; }

   ReturnType *tail()  const
   {  return tail_; }

private:
   ElementType*  head_;
   ReturnType *tail_;
};
```

PtrList has two instance variables: head_ and tail_. head_ points to the head element and tail_ points to the rest of the list. Please note that the type of tail_ is ReturnType, which is the final type of the list. We cannot use PtrList as the type of tail_ since we will derive a list with counter and a list with tracing from PtrList. Whenever we derive classes from PtrList and want to create instances of the *most refined type*, tail_ has to be of the most refined type. Since this type is unknown in PtrList, PtrList retrieves it from the configuration repository (which is passed to PtrList as the parameter Config_).[5]

The next interesting point about PtrList is that methods setting the head (i.e. the constructor and setHead()) use the type SetHeadElementType&. This type should be either ElementType& or const ElementType&, depending whether the list stores references to elements or copies of elements. Since this is unknown in PtrList, PtrList retrieves SetHeadElementType from the configuration repository.

Finally, PtrList delegates some of its work to other components: The destructor delegates its job to the type name Destroyer, which is retrieved from the configuration repository. Similarly, setHead() delegates type checking and copying to TypeChecker and Copier, respectively. The type names Destroyer, TypeChecker, and Copier may point to different components, as specified in Fig. 5.

We have two destroyer components: ElementDestroyer and EmptyDestroyer. ElementDestroyer deletes an element and is used if a list keeps element copies or owned references. It is implemented as a struct rather than a class since it defines only one public operation and struct members are public by default:

```
template<class ElementType>
struct ElementDestroyer
{  static void destroy(ElementType *e)
```

[5] Our example demonstrates how configuration repositories can help in typing recursive classes, i.e. classes that are used directly or indirectly in their own definition (e.g. a list).

```
   {  delete e; }
};
```
`EmptyDestroyer` is used if a list keeps external references to the original elements. `EmptyDestroyer` does nothing. Since its `destroy()` method is implemented inline, an optimizing compiler will remove any calls to this method.

```
template<class ElementType>
struct EmptyDestroyer
{  static void destroy(ElementType *e)
   {} //do nothing
};
```
`DynamicTypeChecker` is used to assure that a monomorphic list contains elements of one type only:

```
template<class ElementType>
struct DynamicTypeChecker
{  static void check( const ElementType& e)
   {  assert( typeid(e)== typeid(ElementType)); }
};
```
`EmptyTypeChecker` is used in a polymorphic list and it does nothing:

```
template<class ElementType>
struct EmptyTypeChecker
{  static void check( const ElementType& e)
   {}
};
```
`PolymorphicCopier` copies an element by calling the virtual function `clone()`:

```
template<class ElementType>
struct PolymorphicCopier
{  static ElementType* copy( const ElementType& e)
   {  return e.clone(); } //call a virtual clone()
};
```
`MonomorphicCopier` copies an element by calling its copy constructor:

```
template<class ElementType>
struct MonomorphicCopier
{  static ElementType* copy( const ElementType& e)
   {  return new ElementType(e); } //call copy constructor
};
```
Finally, `EmptyCopier` simply returns the original element:

```
template<class ElementType>
struct EmptyCopier
{  static ElementType* copy( ElementType& e) //pass by non- const
                                              //reference!
   {  return &e; } //simply return the original
};
```

Next, we need the two wrappers `LenList` and `TracedList` implementing length counter and tracing, respectively. `LenList` is implemented as an *inheritance-based* *wrapper*, i.e. a template class derived from its parameter.[6] It overrides the method `setTail()` to keep track of the list length and adds the method `length()`. Please note that the component retrieves the configuration repository from its parameter:

```
template<class BaseList>
class LenList : public BaseList
{
public:
   //retrieve the configuration repository
   typedef typename BaseList::Config Config;

private:
   //retrieve the necessary types from the repository
   typedef typename Config::ElementType ElementType;
   typedef typename Config::SetHeadElementType SetHeadElementType;
   typedef typename Config::ReturnType ReturnType;
   typedef typename Config::LengthType LengthType;

public:
   LenList( SetHeadElementType& h, ReturnType *t = 0) :
      BaseList( h,t), length_( computedLength())
   {}

   void setTail( ReturnType *t)
   { BaseList::setTail(t);
      length_ = computedLength();
   }

   const LengthType& length() const
   { return length_; }

private:
   LengthType computedLength() const
   { return tail() ? tail()->length()+1
                    : 1;
   }

   LengthType length_;
};
```

`TracedList` is also implemented as an inheritance-based wrapper:

```
template<class BaseList>
class TracedList : public BaseList
{
public:
   typedef typename BaseList::Config Config;

private:
   typedef typename Config::ElementType ElementType;
   typedef typename Config::SetHeadElementType SetHeadElementType;
   typedef typename Config::ReturnType ReturnType;
```

[6] Inheritance-based wrappers are useful whenever we only want to override few methods and inherit the remaining ones. Otherwise, we can use *aggregation-based wrappers*.

```
public:
   TracedList( SetHeadElementType& h,  ReturnType *t = 0) :
      BaseList( h,t)
   { }

   void setHead( SetHeadElementType& h)
   {  cout << " setHead(" << h << ")"<<  endl;
      BaseList::setHead(h);
   }

   ElementType& head()
   {  cout << "head()"<<  endl;
      return BaseList::head();
   }

   void setTail( ReturnType *t)
   {  cout << " setTail(t)"<<  endl;
      BaseList::setTail(t);
   }
};
```

5.3 Composing List Components

Now that we have created all components the question is how to produce a concrete configuration, e.g. a monomorphic list keeping copies of elements of type `Person` and providing a length counter and tracing. For this purpose we have to define a configuration repository with the necessary configuration information (see Fig. 6).

```
struct TracedCopyMonoBaseLenListConfig
{
   //provide the different type names required by the components
   typedef Person                              ElementType;
   typedef const ElementType                   SetHeadElementType;
   typedef int                                 LengthType;
   typedef ElementDestroyer<ElementType>       Destroyer;
   typedef DynamicTypeChecker<ElementType>     TypeChecker;
   typedef MonomorphicCopier<ElementType>      Copier;

   //wrap PtrList into  LenList and  TracedList
   typedef
      TracedList<
       LenList<
         PtrList< TracedCopyMonoBaseLenListConfig  > > >  ReturnType ;
};
//define a short name for our list
typedef TracedCopyMonoBaseLenListConfig::ReturnType MyList;
```

Fig. 6 Sample configuration repository

`TracedCopyMonoBaseLenListConfig` is implemented as a so-called *traits class* [Mye95], i.e. a class aggregating a number of types and constants to be passed to a template as a parameter. `MyList` contains the type we want to produce. The gray boxes and arrows visualize the flow of types from the configuration repository to `PtrList` to `LenList` and, finally, to `TracedList`. Among those types is `ReturnType`, which represents the final list type. The list components can retrieve it from the repository. It is interesting to note the circularity involving `ReturnType`: The components are actually composed in the configuration repository and the repository is passed to the components.

It is also worth noting that, since all variation points were static, we used static parameterization and inlining to avoid the unnecessary cost of virtual function calls and the cost of function calls for small functions. Templates and inlining allow composing code fragments without any runtime cost. As a result, the composed types are as efficient as manually coded concrete variants.

Without counting `ElementType` and `LengthType`, the feature diagram in Fig. 3 defines 24 different list configurations. We can define a configuration repository for each of them, e.g. a polymorphic list keeping external references to original elements of type `Person` or derived types:

```
struct RefPolyBaseListConfig
{
   typedef Person                               ElementType;
   typedef ElementType                          SetHeadElementType;
   typedef EmptyDestroyer<ElementType>          Destroyer;
   typedef EmptyTypeChecker<ElementType>        TypeChecker;
   typedef EmptyCopier<ElementType>             Copier;

   typedef PtrList<RefPolyBaseListConfig>       ReturnType;
};
typedef RefPolyBaseListConfig::ReturnType  RefPolyBaseList;
```

Writing down the configuration repositories for all 24 list variants (we would parameterize `ElementType` and `LengthType` for this purpose) is rather tedious. The situation is worse if we have concepts with more variable features. For example, we implemented a configurable matrix component, which covers 1840 different matrix variants (see Section 7). Writing down all configuration repositories in this case is impracticable. An alternative would be to let the application programmer write the configuration repositories for the types he or she needs. This approach has its drawbacks. Writing the lengthy configuration repositories is error prone and tedious. Furthermore, configuration repositories mention implementation detail which is not relevant at the level of the feature diagram in Fig. 3. For example, we have to explicitly define `SetHeadElementType`, although it is not part of the feature diagram. Similarly, selecting the appropriate destroyer, type checker, and copier is an implementation detail. These choices, although they automatically follow from the selected abstract features, have to be programmed manually! A much better solution is to generate the configuration repositories from abstract specifications. This is described in the following section.

5.4 Generating Lists from Abstract Specifications

Our goal now is to generate list configurations from abstract specifications, i.e. sets of features defined in Fig. 3. This requires writing some code which is executed at compile time. We can use *template metaprogramming* for this purpose [Vel95a, CE98, Cza98]. Template metaprograms consist of class templates operating on numbers and/or types as data.[7] Algorithms are expressed using template recursion as a looping construct and class template specialization as a conditional construct. Template recursion involves the direct or indirect use of a class template in the construction of its own member type or member constant. In other words, C++ templates constitute a Turing-complete sublanguage of C++, which is interpreted by the compiler at compile time. As an example, consider a template which computes the factorial of a non-negative integral number:

```
template<int n>
struct Factorial
{  enum { RET = Factorial<n-1>::RET * n };
};

//the following template specialization terminates the recursion
template<>
struct Factorial<0>
{  enum { RET = 1 };
};
```

We can use this class template as follows:

```
void main()
{  cout << Factorial<7>::RET <<  endl; //prints 5040
}
```

The important point about this program is that `Factorial<7>` is instantiated at compile time. During the instantiation, the compiler also determines the value of `Factorial<7>::RET` (`RET` is an abbreviation for `RETURN`). Thus, the code generated for this `main()` function by the C++ compiler is the same as the code generated for the following `main()`:

```
void main()
{  cout << 5040 <<  endl; //prints 5040
}
```

We can regard `Factorial<>` as a *metafunction* which is evaluated at compile time. `Factorial<>` is a metafunction since, at compilation time, it computes constant data of a program which has not been generated yet. Template metafunctions can also take types

[7] The ability to use the template instantiation process to perform compile-time computations was observed by Erwin Unruh. He wrote a small C++ program generating prime numbers at compile time, which the compiler would output as a sequence of warnings. This program [Unr94] was circulated as a “curiosity” during an ANSI C++ standardization committee meeting in 1994.

as parameters and return types. The following metafunction takes a Boolean and two types as its parameters and returns a type:[8]

```
template<bool  cond, class  ThenType, class  ElseType>
struct IF
{  typedef  ThenType RET;
};

template<class  ThenType, class  ElseType>
struct IF<false,  ThenType,  ElseType>
{  typedef  ElseType RET;
};
```

This function corresponds to an if statement: it has a condition parameter, a “then” parameter, and an “else” parameter. If the condition is `true`, it returns `ThenType` in `RET`. This is encoded in the base definition of the template. If the condition is `false`, it returns `ElseType` in `RET`. Thus, this metafunction can be viewed as a meta-control statement. Implementing other meta-control statements such as switch and looping constructs is also possible [CE98]. Indeed, even a simple Lisp was implemented using these techniques [CE98, Cza98].

We can use `IF<>` to implement a metafunction which takes a number of flags representing the features from Fig. 3 and returns a ready-to-use list type. First, we define the flags representing the list features:

```
enum Ownership { ext_ref,  own_ref,  cp};
enum Morphology {mono,   poly};
enum  CounterFlag{ with_counter,  no_counter};
enum  TracingFlag{ with_tracing,  no_tracing};
```

Next, we present the metafunction `LIST_GENERATOR<>` implementing a *configuration generator*, which takes an abstract description of a list and generates a ready-to-use list type. Using `LIST_GENERATOR<>` , we can declare a monomorphic list keeping copies of elements of type `Person` and providing a length counter and tracing as follows:

```
LIST_GENERATOR<Person,  cp, mono,  with_counter,  with_tracing>::RET list1;
```

A polymorphic list keeping external references to original elements of type `Person` or derived types can be declared as follows:

```
LIST_GENERATOR<Person,  ext_ref,  poly>::RET list2;
```

Here is the definition of the metafunction:

```
template<
   class ElementType_ =  int,
   Ownership  ownership =  cp,
   Morphology  morphology = mono,
   CounterFlag  counterFlag =  no_counter,
   TracingFlag  tracingFlag =  no_tracing,
   class LengthType_ =  int
>
```

[8] A different `IF<>` implementation which does not require partial template specialization is described in [Cza98].

```
class LIST_GENERATOR
{
public:
   typedef LIST_GENERATOR<
      ElementType_,
      ownership,
      morphology,
      counterFlag,
      tracingFlag,
      LengthType_> Generator;

private:
   enum {
      isCopy      = ownership==cp,
      isOwnRef    = ownership==own_ref,
      isMono      = morphology==mono,
      hasCounter  = counterFlag==with_counter,
      doesTracing = tracingFlag==with_tracing };

   typedef
      IF<isCopy ||  isOwnRef,
            ElementDestroyer< ElementType_>,
            EmptyDestroyer< ElementType_>
      >::RET Destroyer_;

   typedef
      IF<isMono,
            DynamicTypeChecker<ElementType_>,
            EmptyTypeChecker< ElementType_>
      >::RET TypeChecker_;

   typedef
      IF<isCopy,
            IF<isMono,
                  MonomorphicCopier< ElementType_>,
                  PolymorphicCopier< ElementType_> >::RET,
            EmptyCopier< ElementType_>
      >::RET Copier_;

   typedef
      IF<isCopy,
            const ElementType_,
            ElementType_
      >::RET SetHeadElementType_;

   typedef PtrList<Generator> List;

   typedef
      IF<hasCounter,
            LenList<List>,
            List
      >::RET List_with_counter_or_not;

   typedef
      IF<doesTracing,
            TracedList< List_with_counter_or_not>,
            List_with_counter_or_not
      >::RET  List_with_tracing_or_not;
public:
   typedef  List_with_tracing_or_not RET;

   struct Config
   {
      typedef ElementType_           ElementType;
      typedef SetHeadElementType_    SetHeadElementType;
      typedef Destroyer_             Destroyer;
      typedef  TypeChecker_          TypeChecker;
      typedef Copier_                Copier;
      typedef LengthType_            LengthType;
      typedef RET                    ReturnType;
   };
};
```

`LIST_GENERATOR<>` evaluates the input flags, computes the types for the configuration repository, wraps `PtrList` (if necessary), and returns the final list type in `RET`. The last part of `LIST_GENERATOR<>` is the configuration repository. Please note that we pass `Generator` to `PtrList` as parameter. Since `Config` is a member of `Generator`, `PtrList` can retrieve `Config` from `Generator`. For this reason, we need to slightly modify two lines of `PtrList` (the modifications are highlighted):

```
template<class  Generator >
class PtrList
{
public:
   typedef  typename  Generator::Config  Config;
//the rest as previously
//...
```

An important aspect of the separation between the problem-space-oriented feature description and the implementation components is that we can make useful extensions to the components (e.g. modify the component structure or add new components) without the need to change existing client code. This is certainly possible as long as the abstract feature space can still be mapped on the new components. Moreover, we can even make certain extensions to the feature space without the need to change existing client code. For example, we could append new parameters, e.g. memory allocation, to the parameter list expected by the generator. By choosing appropriate defaults for the new parameters, existing calls to the generator will still work properly. Similarly, if we model all features as template structs, we can also add new nested features (cf. the following section).

6 Extensions

The previous section demonstrated the basic techniques using a very simple example. Applying these techniques to larger problems requires several extensions:

- *Nested features*: Our sample generator expects a flat list of features although feature diagrams are trees. This was acceptable for this small example, but in general configuration generators accept tree-like structures. We can represent tree-like feature structures in C++ using types and templates. For example, we could model `counterFlag` as a type parameter with the values `no_counter` and `with_counter<>`. `no_counter` would be a struct and `with_counter<>` a template struct expecting `LengthType` as its parameter.

- *Multistage configuration generators*: Large feature models may contain many constraints and default dependency rules. In this case, a configuration generator consists of several stages: specification completion stage (computes defaults for the unspecified features based on default dependency rules and constraints), feature combination checking stage (checks whether the feature combinations satisfy the constraints), and component assembly stage (assembles components into the final type). The dependency rules and constraints are specified using a kind of decision tables. For this reason, we implemented a table evaluation metafunction, which allows us to directly type in the tables in the C++ source code [Cza98, Kna98]. This function utilizes both `IF<>` and template recursion.
- *Nested configuration repositories*: Avoiding name clashes in the configuration repository may require introducing separate name scopes within the repository. For example, two different components retrieve the type name `ElementType`, but each of them should be supplied a different type. This can be resolved by providing a separate name scope for each of these components in the repository. We can model such nested name scopes in C++ as nested classes.
- *Metafunctions as part of configuration repositories*: Sometimes one component is used more than once in a configuration and each instance needs different configuration parameters. In this case, the component does not retrieve the required type from the configuration repository directly, but it retrieves a metafunction. Each instance can then supply a different parameter to the metafunction to compute the needed type. In C++, class templates can be defined as members of other classes. This way it is possible to pass around a metafunction as a type, which corresponds to the idea of higher-order metafunctions.
- *Configuration repository as a part of the generated type*: Each component exports the configuration repository under the name `Config`. Thus, we can also retrieve `Config` from the final type. e.g. `MyList::Config`. This feature can be used by other generators, e.g. for generating customized algorithms operating on the generated types. An algorithm generator can retrieve the properties of a type from its `Config` (e.g. `ElementType`, `Ownership`, etc.) and use this information to generate optimized algorithms. For example, we have used the configuration repository of a matrix type in order to generate optimized matrix operation code using expression templates [Cza98].
- *Traits templates for encoding metainformation*: Metainformation about types can be encoded as *traits templates* [Mye95], which basically correspond to metafunctions

taking types as parameter and returning their properties. For example, our polymorphic copier in Section 5.2 assumes that the element type provides the virtual `clone()` method. If a particular element type does not provide this method, we can use an adapter. In this case, we could use a traits template on element type to retrieve the appropriate (user-provided) adapter. Furthermore, we could use a traits template to make sure that `DynamicTypeChecker` is used only for types having a virtual function table.

We used the above extensions in the implementation of two applications described in the following section.

The generator approach described here can be also used to synthesize frameworks. Frameworks can be modeled as compositions of collaborations and the latter can be implemented as *mixin layers* [SB98]. In C++, a mixin layer may be implemented as a class containing a number of nested classes. Each of the nested classes implements a particular role. The nested classes can inherit from their parameters, so that they can be used to extend other classes. A mixin layer takes another layer as its parameter, accesses the classes nested in the parameter and uses them as superclasses of some of its own nested classes (see [SB98] for details). Just as we used our configuration generator to compose parameterized classes, we can also use it to compose mixin layers.

7 Applications

The techniques described in previous sections were use to develop two medium size libraries demonstrating their applicability to generating efficient abstract data types and algorithms:

- *Generative Matrix Computation Library* (GMCL) [GMCL, Cza98, Neu98] contains a matrix generator (in the style of our list generator from Section 5.4) able to generate matrices with a selected combination of features such as element types (real numbers), density (dense and sparse), storage formats (row- and column-wise, several sparse formats), memory allocation (dynamic and static), error checking (bounds, compatibility, memory allocation), and operations (addition, subtraction, multiplication). GMCL contains another generator for generating efficient implementations of matrix expressions (e.g. “(A+B)*(C+D)”). The expression generator is based on expression templates [Vel95b]. It reads out the properties of the operands from their configuration repositories in order to generate optimized code. The C++ implementation of the matrix component comprises 7500 lines of C++ code (6000 lines for the configuration generator and the matrix components and 1500 lines for the operations). The matrix configuration feature diagram covers more than 1840 different kinds of matrices. Despite the large number of provided matrix variants, the performance of the generated code is comparable with the performance of manually coded variants. This is achieved by the exclusive use of static binding, which is often combined with inlining.

- *Generative matrix factorization library* [Kna98] contains a generator synthesizing different instances of the LU factorization algorithm family (e.g. Gauss, Cholesky, LDL^T) with different pivoting strategies (e.g. partial, full, symmetric, diagonal) and for different matrix shapes. The different parts of the algorithms are implemented as methods of class templates organized in a layered architecture. The templates are configured by a configuration generator.

8 Related Work

Variability modeling The need for variability modeling in framework design has been recognized in the form of “hot spots” [Pre95]. Hot spots represent variation points. Unfortunately, they are not supported in current OOA/D methods. Furthermore, they make no distinction between different kinds of variation points (e.g. dimensions, dimensions with optional features, extension points, etc. [Cza98]) and do not model the information contained in feature models. Reuse-Driven Software Engineering Business (RSEB) [JGJ98] extends UML with the concept of variation points and defines a reuse-driven development process. However, it still lacks feature modeling. Feature modeling is the corner stone of Domain Engineering (DE) and efforts aimed at integrating OO and DE methods [CN98, GFA98, Cza98] augment OO modeling techniques with feature modeling.

Layered Designs and Fragment-Based Component Models GenVoca is a layered architecture model based on parameterized layers of refinement [BO92]. Recent work by Smaragdakis and Batory [SB98] views GenVoca layers as so-called *mixin layers*, i.e. layers containing classes whose superclasses are parameters. Parameterized inheritance has also been used to express collaboration-based designs [VHN96]. The technique of exchanging types between components at compile time is extensively used in the Standard Template Library (STL) [MS96]. Fragment-based designs have been studied in the context of OO, among others, in [Pre97, Mez97, ML98]. Our work extends these approaches with configuration repositories, which among others provide an effective approach for typing synthesized recursive classes. Furthermore, our configuration generator is capable of synthesizing layered and fragment-based designs from abstract specifications.

Metaprogramming There is a large body of work on static metaprogramming for code composition. Most of this work is has been done in the context of procedural languages such as C (e.g. [SG97, Eng97]). There are also examples of static metaprogramming systems for C++, e.g. Open C++ [Chi95] and MPC++ [IHS+96]. All of these systems require special language extensions. Our approach, on the other hand, uses standard C++ language features and thus is widely available. Template metaprogramming has been used to develop a number of libraries including Blitz++ [Vel97], POOMA [POOMA], and MTL [SL98]. Unfortunately, it lacks debugging and error-reporting facilities. An example of a commercial metaprogramming environment supporting the full development cycle is Intentional Programming (IP) [Sim96]. IP is currently under development at Microsoft Research.

9 Conclusions and Outlook

The development of truly reusable software requires the parameterization of a multitude of design decisions. We showed that modeling the variability of domain concepts is inadequately supported by current OO modeling notations. We also showed that this problem can be addressed by using feature models in addition to OO models. Furthermore, we demonstrated that the implementation of feature models requires three ingredients: domain-level interface (e.g. a domain-specific language, a domain-specific GUI, etc.) allowing the application programmer to describe concepts at the abstract domain level, configuration knowledge mapping between abstract descriptions and concrete component configurations, and elementary implementation components, which can be configured in a vast number of ways. We showed that the design of the problem and the solution space starts with feature modeling and the solution space is structured according to some appropriate architecture (in our example, we used a layered architecture). We also found it important to have a direct programming language support for these concepts. We showed concrete examples in C++; however, the concepts are not limited to C++. Indeed, the examples demonstrate the importance of parameterized inheritance in avoiding rigid inheritance hierarchies and the value of generic, STL-style techniques for implementing efficient and highly configurable components. The newly introduced concept of configuration repositories allows the separation between the components and the configuration knowledge and also facilitates an efficient approach to typing recursive classes. Finally, the built-in metaprogramming capabilities of C++ allow the configuration generators to be part of the same library as the implementation components. The presented example concentrated on static configuration and binding, but similar designs based on dynamic configuration and binding can be implemented in C++ and in other languages (e.g. Smalltalk, CLOS, Java[9]). Indeed, we want to parameterize configuration time and binding time. The latter is easily done in C++ [Eis97], but the former requires the ability to write metacode which can be executed both by the compiler and at runtime – a feature not supported by current languages. One of the conclusions of our work is the need for industrial strength metaprogramming environments. Template metaprogramming (TM) has the important advantage that is readily available to users as a built-in part of C++. But since TM is a child of accident

[9] Java does not support parameterized inheritance. Therefore, whenever we need a parameterized relationship in Java, we have to use dynamic parameterization.

rather than the result of conscious language design, it suffers from several deficiencies in the areas of debugging and error-reporting, code readability, long compilation times, various compiler limits, and portability [Cza98, Neu98, Kna98]. Currently, the size of template metaprograms is limited by compiler limits, compilation times, and debugging problems. However, compiler limits and portability problems will decrease as more and more compiler vendors adopt the new C++ ISO standard. Adequate metaprogramming support opens new possibilities to raise “the level” of programming using domain-specific abstractions. For example, domain-specific abstractions in Intentional Programming [Sim96] are active at any time (including programming and compilation time) and generate efficient, optimized code. This perspective forces us to redefine the conventional interaction between compilers, libraries, and applications and to acknowledge the need for *active libraries* [CEG+98], which “are not passive collections of routines or objects, as are traditional libraries, but take an active role in generating code. Active libraries provide abstractions and can optimize those abstractions themselves. They may generate components, specialize algorithms, optimize code, automatically configure and tune themselves for a target machine, and check source code for correctness. They may also describe themselves to tools such as profilers and debuggers in an intelligible way.” The effective application of metaprogramming in software engineering requires new analysis and design approaches. The integration of modeling and implementation technologies based on domain-specific abstractions and metaprogramming into a coherent paradigm is the goal of the emerging area of Generative Programming [CE99, CEG+98, Eis97].

Note: The source code for the list example is available at http://nero.prakinf.tu-ilmenau.de/~czarn/ecoop99

References

[AM97] M. Aksit and S. Matsuoka, (Eds.). *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP '97)*, Springer-Verlag 1997

[BO92] D. Batory and S. O’Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. In *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, October 1992, pp. 355-398

[CE98] K. Czarnecki and U. Eisenecker. Template-Metaprogramming, http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm

[CE99] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. To appear, Addison-Wesley, 1999

[CEG+98] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative Programming and Active Libraries. Submitted for publication, 1998

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications*

(OOPSLA'95), ACM SIGPLAN Notices, vol. 30, no. 10, 1995, pp. 285-299, http://www.softlab.is.tsukuba.ac.jp/~chiba/openc++.html

[CN98] S. Cohen and L. M. Northrop. Object-Oriented Technology and Domain Analysis. In [DP98], pp. 86-93

[Cza98] K. Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Ph.D. thesis, Technische Universität Ilmenau, Germany, 1998, see http://nero.prakinf.tu-ilmenau.de/~czarn/

[DP98] P. Devanbu and J. Poulin, (Eds.). *Proceedings of the Fifth International Conference on Software Reuse (Victoria, Canada, June 1998).* IEEE Computer Society Press, 1998

[Eis96] U. Eisenecker. Generatives Programmieren mit C++. In *OBJEKTspektrum,* No. 6, November/December 1996, pp. 79-84

[Eis97] U. Eisenecker. Generative Programming (GP) with C++. In *Proceedings of Modular Programming Languages (JMLC'97, Linz, Austria, March 1997),* H. Mössenböck, (Ed.), Springer-Verlag, Heidelberg 1997, pp. 351-365

[Eng97] D. R. Engler. Incorporating application semantics and control into compilation. In *Proceedings USENIX Conference on Domain-Specific Languages (DSL'97),* 1997

[GFA98] M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In [DP98], pp. 76-85, see http://www.intecs.it

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[GMCL] Homepage of the Generative Matrix Computation Library at http://nero.prakinf.tu-ilmenau.de/~czarn/gmcl/

[Gog96] J. A. Goguen. Parameterized Programming and Software Architecture. In *Proceedings of the Fourth International Conference on Software Reuse*, April 23-26, Orlando, Florida. IEEE Computer Society Press, Los Alamitos, California, 1996, pp. 2-10

[IHS+96] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and Implementation of Metalevel Architecture in C++ – MPC++ approach. In *Proceedings of Reflection'96*, 1996

[JGJ98] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Longman, May 1997

[Jul98] E. Jul, (Ed.). *Proceedings of the 12th European Conference Object-Oriented Programming (ECOOP'98)*, LNCS 1445, Springer-Verlag, 1998

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In [AM97], pp. 220-242

[Kna98] J. Knaupp. Algorithm Generators: A First Experience, see http://nero.prakinf.tu-ilmenau.de/~czarn/generate/stja98/knaupp.zip

[Mez97] M. Mezini. Dynamic Object Evolution Without Name Collisions. In [AM97], pp. 190-219

[ML98] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proceedings of the Conference on Object-Oriented Programming Languages and Applications (OOPSLA '98)*, 1998

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, Massachusetts, 1996

[Mye95] N.C. Myers. Traits: a new and useful template technique. In *C++ Report*, June 1995, see http://www.cantrip.org/traits.html

[Neu98] T. Neubert. Anwendung von generativen Programmiertechniken am Beispiel der Matrixalgebra. Diplomarbeit, Technische Universität Chemnitz, 1998, also see [GMCL]

[POOMA] POOMA: Parallel Object-Oriented Methods and Applications. A framework for scientific computing applications on parallel computers. Available at http://www.acl.lanl.gov/pooma

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995

[Pre97] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In [AM97], pp. 419-443

[Ree96] T. Reenskaug with P. Wold and O.A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning, 1996

[SB98] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In [Jul98], pp. 550-570

[SCK+96] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang. Organization Domain Modeling (ODM) Guidebook, Version 2.0. Informal Technical Report for STARS, STARS-VC-A025/001/00, June 14, 1996, see http://direct.asset.com

[SG97] J. Stichnoth and T. Gross. Code composition as an implementation language for compilers. In *Proceedings USENIX Conference on Domain-Specific Languages (DSL'97)*, 1997

[Sim96] C. Simonyi. Intentional Programming — Innovation in the Legacy Age. Position paper presented at IFIP WG 2.1 meeting, June 4, 1996, see http://www.research.microsoft.com/research/ip/

[SL98] J. G. Siek and A. Lumsdaine. A Rational Approach to Portable High Performance: The Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (FAST) Library. In Proceedings of the ECOOP'98 Workshop on Parallel Object-Oriented Computing (POOSC'98), 1998, see http://www.lsc.nd.edu/

[Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994

[Unr94] E. Unruh. Prime number computation. ANSI X3J16-94-0075/ISO WG21-462, 1994

[Vel95a] T. Veldhuizen. Using C++ template metaprograms. In *C++ Report*, vol. 7, no. 4, May 1995, pp. 36-43, see http://monet.uwaterloo.ca/blitz/

[Vel95b] T. Veldhuizen. Expression Templates. In *C++ Report*, vol. 7 no. 5, June 1995, pp. 26-31, see http://monet.uwaterloo.ca/blitz/

[Vel97] T. Veldhuizen. Scientific Computing: C++ versus Fortran. In *Dr. Dobb's Journal*, November 1997, pp. 34-41, see http://monet.uwaterloo.ca/blitz/

[VHN96] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, 1996, pp. 359-369