Components and Generative Programming

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 is about a paradigm shift from the current practice of manually searching for and adapting components and their manual assembly to Generative Programming, which is the automatic selection and assembly of components on demand. First, we argue that the current OO technology does not support reuse and configurability in an effective way. Then we show how a system family approach can aid in defining reusable components. Finally, we describe how to automate the assembly of components based on configuration knowledge. We compare this paradigm shift to the introduction of interchangeable parts and automated assembly lines in the automobile industry.

We also illustrate the steps necessary to develop a product line using a simple example of a car product line. We present the feature model of the product line, develop a layered architecture for it, and automate the assembly of the components using a generator. We also discuss some design issues, applicability of the approach, and future development.

1 From Handcrafting to an Automated Assembly Line

This paper is about a paradigm shift from the current practice of manually searching for and adapting components and their manual assembly to Generative Programming, which is the automatic selection and assembly of components on demand. This paradigm shift takes two steps. First, we need to move our focus from engineering single systems to engineering families of systems—this will allow us to come up with the "right" implementation components. Second, we can automate the assembly of the implementation components using generators.

Let us explain this idea using a metaphor: Suppose that you are buying a car and instead of getting a read-to-use car, you get all the parts necessary to assemble the car yourself. Actually, not quite. Some of the parts are not a one-hundred-percent fit and you have to do some cutting and filing to make them fit (i.e. adapt them). This is the current practice in component-based software engineering. Brad Cox compares this situation to the one at the brink of the industrial revolution, when it took 25 years of unsuccessful attempts, such as Ely Whitney's pioneering effort, until John Hall finally succeeded to manufacture muskets from interchangeable parts in 1822 (see [Cox90, Wil97]). Then it took several decades before this groundbreaking idea of mass-manufacturing from interchangeable parts spread to other sectors.

Even if you use a library of designed-to-fit, elementary components (such as the C++ Standard Template Library [MS96]), you still have to assemble them manually and there is a lot of detail to care about. In other words, even if you don't have to do the

Reference:
K. Czarnecki and U. Eisenecker. Components and Generative Programming. Invited talk, in *Proceedings of the 7th European Software Engineering Conference, held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '99), Toulouse, France, September 1999*, O. Nierstrasz and M. Lemoine (Eds.), LNCS 1687, Springer-Verlag, 1999, pp. 2-19, http://nero.prakinf.tu-ilmenau.de/~czarn/esec99

cutting and filing, you still have to assemble your car yourself (that's the "lego principle").

Surely you rather want to be able to order your car by describing it in abstract terms, saying only as much as you really care to, e.g. "get me a Mercedes-Benz S-Class with all the extras" or "a C-Class customized for racing, with a high-performance V8 engine, four-wheel vented disc brakes, and a roll cage", and get the ready-to-drive car. And that's what Generative Programming means for the application programmer: The programmer states what he wants in abstract terms and the generator produces the desired system or component.

This magic works if you (1) design the implementation components to fit a common product-line architecture, (2) model the configuration knowledge stating how to translate abstract requirements into specific constellations of components, and (3) implement the configuration knowledge using generators. This is similar to what happened in car building: the principle of interchangeable parts was the prerequisite for the introduction of the assembly line by Ransome Olds in 1901, which was further refined and popularized by Henry Ford in 1913, and finally automated using industrial robots in the early 1980s.[1, 2]

The rest of this paper is structured as follows. We first explain the necessary transition from one-of-a-kind development to the system family approach (Sections 2-3). Then, we describe the idea of automating component assembly based on configuration knowledge (Section 3). Next, we will demonstrate the previous two steps using a simple example: We will first come up with the components for a product line (Sections 5.1-5.4) and then develop the generator for their automatic assembly (Sections 5.5-5.6). Finally, we'll give you some real-world examples (Section 6), describe the idea of active libraries (Section 7), and present the conclusions (Section 8).

---

[1] The world's first industrial robot was installed in 1961 at a General Motors factory in New Jersey, USA, but it was the advance of the microchip in the 1970s that made possible the enormous advances in robotics.

[2] Some people think that the main purpose of an assembly line is to produce masses of the same good, which in software corresponds to copying CDs. Nothing could be further from the truth. For example, the Mercedes-Benz assembly line in Sindelfingen, Germany, produces hundreds of thousands of variants of the C-, E-, and S-Class (there are about 8000 cockpit variants and 10000 variants of seats alone for the E-Class). There are almost no two equal cars rolling from the assembly line the same day. That means that the right kind of engine or any other component has to be available at the right place and time at the assembly line. Furthermore, the suppliers have to provide the right parts at the right time (to minimize storage costs). And the whole process starts when different people order different cars at car dealerships. The fulfillment of the customer orders requires an enormous logistic and organizational effort involving a lot of configuration knowledge (in fact, they use product configurators based on configuration rules). Thus, the analogy between the automobile industry and building customized software solutions using a gallery of standard components (e.g. SAP's R3 modules) is not that far-fetched after all.

2 "One-of-a-kind" Development

Most OOA/D methods focus on developing single systems rather than families of systems. Therefore, they do not adequately support software reuse. More specifically, they have the following deficiencies [CE99a]: [3]

- *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* and engineering *with reuse*. The scope of engineering for reuse is not a single system but a system family. This enables the production of *reusable* components. The process of engineering with reuse has to be designed to take advantage of the reusable assets produced during engineering for reuse. Current OOA/D processes lack any of these properties.
- *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 the *stakeholders* (including potential customers) of a class of systems.
- *No differentiation between modeling variability within one application and between several applications*: Current OO notations make no distinction between intra-application variability, e.g. variability of objects over time and the use of different variants of an object at different locations in an application, and variability between applications, i.e. variability across different applications for different users and usage contexts. Furthermore, OO implementation mechanisms for implementing intra-application variability (e.g. dynamic polymorphism) are also used for inter-application variability. This results in "fat" components or frameworks ending up in "fat" applications.
- *No implementation-independent means of variability modeling*: Furthermore, current OO notations do not support variability modeling in an implementation-independent way, i.e. the moment you draw a UML class diagram, you have to decide whether to use inheritance, aggregation, class parameterization, or some other implementation mechanism to represent a given variation point.

Patterns and frameworks represent an extremely valuable contribution of the OO technology to software reuse. However, they still need to be accompanied by a systematic approach to engineering for and with reuse.

3 System Family Approach

In order to come up with reusable components, we have to move our focus from single systems to system families.[4] The first thing to do is to distinguish between the

[3] 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.

development *for reuse* and *with reuse*. In the software reuse community, development for reuse is referred to as *Domain Engineering*.[5] Development with reuse, on the other hand, is referred to as *Application Engineering*. Let us take a closer look at the steps of Domain Engineering:

- *Domain Analysis*: Domain Analysis involves domain scoping and feature modeling. *Domain scoping* determines which systems and features belong to the domain and which not. This process is driven not only by technical but also marketing and economic aspects (i.e. there is an economic analysis as in the case of any investment) and involves all the stakeholders of the domain. For this reason, the resulting domain is often referred to as a *product line*. *Feature modeling* identifies the common and variable features of the domain concepts and the dependencies between the variable features. Refining the semantic contents of the features usually requires several other modeling techniques such as modeling relationships and interactions between objects (e.g. using UML).
- *Domain Design*: The purpose of domain design is to develop a common architecture for the system family.
- *Domain Implementation*: Finally, we need to implement the components, generators, and the reuse infrastructure (dissemination, feedback loop from application engineering, quality control, etc.).

There are many Domain Engineering methods available, e.g. FODA [KCH+90] and ODM [SCK+96] (e.g. see surveys in [Cza98, Arr94]). However, most of the methods incorporate the above-listed steps in some form. Examples of methods combining Domain Engineering and OO concepts are RSEB [JGJ98 and GFA98], DEMRAL [Cza98], and the work presented in [CN98].

4 Problem vs. Solution Space and Configuration Knowledge

Once we have the "right" components, the next step is to provide means of mapping abstract requirements onto appropriate configurations of components, i.e. automate the component assembly. The key to this automation is the *configuration knowledge*, which maps between the *problem space* and the *solution space* (Fig. 1).

The solution space consists of the implementation components with all their possible combinations. The implementation components are designed to maximize their combinability (i.e. you should be able to combine them in as many ways as possible),

---

[4] Parnas defined a system family as follows [Par76, p. 1]: "We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members."

[5] In the software reuse community, a *domain* is defined as a well-scoped family of existing and potential systems including the expertise required to build these systems. So, in our context, we can use the terms "domain" and "system family" interchangeably.

minimize redundancy (i.e. minimize code duplication), and maximize reuse.[6] The problem space, on the other hand, consists of the application-oriented concepts and features that application programmers would like to use to express their needs. The configuration knowledge consists of illegal feature combinations (certain combinations of features may be not allowed), default settings (if the application does not specify certain features, some reasonable defaults are assumed), default dependencies (some defaults may be computed based on some other features), and construction rules (combinations of certain features translate into certain combinations of implementation components).

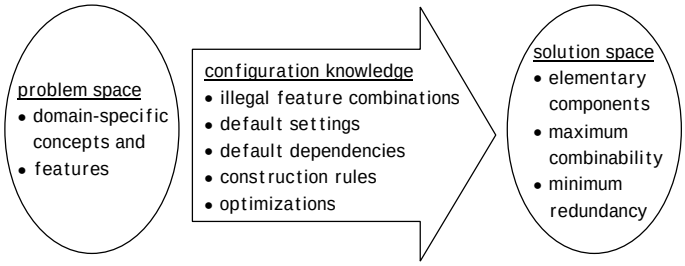

Fig. 1. Problem and solution space

This kind of separation between problem and solution space and the configuration knowledge allow us to enjoy the same convenience in requesting concrete systems or components as in ordering cars: You don't have to enumerate all the concrete parts, e.g. suspension, carburetor, battery, and all the nuts and bolts, but rather specify the class (e.g. C- or E-Class), the line (e.g. Classic-, Elegance- or SportLine), and the options (e.g. side airbags, trailer coupling, air conditioning, etc.) and get a finished car. Please note that the features in the problem space may be inherently abstract, e.g. SportLine. In other words, there is no single component that makes a car to be a sports car, but it is rather a particular combination of carefully selected parts that achieve this quality (in computer science, think of the features "optimized for speed" or "optimized for space"). This also makes Generative Programming different from Generic Programming: While Generative Programming allows you to specify abstract features, the parameters you supply in Generic Programming represent concrete components.

It is important that you can order a car by specifying only as much as you want. For example, there are people who have a lot of money and no time. They could say "give me a Mercedes S-class with all the extras". On the other hand, there could be a customer interested in car racing, who needs to be very specific. He might want to specify details about particular car parts, e.g. "an aluminum block with cast-in Nikasil sleeves and twin-spark plug, three-valve cylinder heads".

[6] These are also the properties usually required from generic components. Thus, the principles of Generic Programming apply to the solution space. An even higher level of genericity of the implementation components can be achieved using Aspect-Oriented Programming [KLM+97].

The same spectrum of specificity should be supported by a library of reusable components [KLL+97]. As an application programmer, when you request a component, you should be able to specify only as much as necessary. You should not be *forced* to specify too much detail: this would make you unnecessarily dependent on the implementation of the library. However, if necessary, you should be *able to* specify details, or even supply your own implementations for some aspects. Defaults, default dependencies, and hard constraints (i.e. illegal feature combinations) make this kind of flexibility possible.

Finally, the configuration knowledge is implemented using generators. Depending on the complexity of the configuration space, the configuration process may be an algorithmic one (for simple configuration spaces) or search-based (for more complex configuration spaces).

5 Example: Designing a Product Line

Now we'll illustrate the steps necessary to develop a product line and automate the component assembly. We will use a pedagogical toy example: a simple C++ model of a car. Real applications of these techniques are described in Section 6.

5.1 Domain Analysis

Domain analysis involves domain scoping and feature analysis of the concepts in the domain. Feature analysis allows you to discover the commonalities and the variabilities in a domain. Reusable models usually contain large amounts of variability. For example, a bank account can have different types (savings, checking, or investment), different kinds of ownership (personal or business), different currencies, different bank statement periods, different interest rates, service fees, overdraft policies, etc.

But let's start with the domain analysis for our car product line. Suppose that based on our market studies, we decided that the cars we are going to produce will provide the following features: automatic or manual transmission, electric or gasoline or hybrid engine, and an optional trailer coupling.

The results of feature analysis can be documented using *feature diagrams* [KCH+90], which reveal the kinds of variability contained in the design space. The feature diagram for our car is shown in Fig. 2. The root of the diagram represents the concept *car*. The remaining nodes are features. The diagram contains four kinds of features:[7]

- *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.

[7] Or-features are not part of the notation in [KCH+90]. They were introduced in [Cza98]. See [Cza98] for a full description of the feature diagram notation used here.

- *Optional features*: Optional features are pointed to by simple edges ending with an empty circle, e.g. *trailer coupling*. A car may have a trailer coupling or not.
- *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.

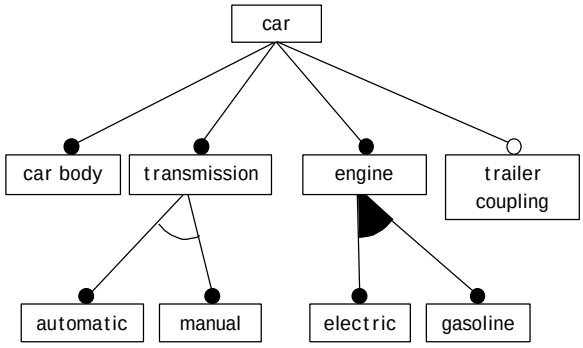

Fig. 2. A sample feature diagram of a car

The diagram in Fig. 2 describes twelve different car variants (two different transmissions, three kinds of engine, and an optional trailer coupling, i.e. 2·3·2=12). Constraints that cannot be expressed in a feature diagram have to be recorded separately. For example, let's assume that an electric or hybrid engine requires an automatic transmission. With this constraint, we have just eight valid feature combinations left. The feature diagram, the constraints, and other information (e.g. binding times, descriptions, etc.) constitute a *feature model*. Modeling the semantic contents of the features usually requires other kinds of diagrams (e.g. object diagrams or interaction diagrams).

The important property of a feature diagram is that it allows us to model variability without having to commit to a particular implementation mechanism such as inheritance, aggregation, templates, or `#ifdef`-directives.

5.2 Domain Design

Now we need to come up with the architecture for the product line. This involves answering questions such as what kinds of components are needed, how they will be connected, what kind of middleware or component model will be used, what interfaces the component categories will have, how they will accommodate the requirements, etc. Designing the architecture is an iterative process and it usually requires prototyping. Studying existing architecture styles and patterns greatly helps in this process (e.g. see [BMR+96]).

For our car example, we'll use a particular kind of a layered architecture, called a GenVoca architecture (see [BO92, SB98] and also [ML98]). This kind of architecture proved to be useful for a wide variety of systems (see Section 6). In our experience, designing a GenVoca architecture requires the following steps:

Identify the main functionalities in the feature diagrams from the Domain Analysis. The main functionalities for our sample car are car body, transmission, engine, and trailer coupling.

Enumerate component categories and components per category. The component categories correspond to the four main functionalities listed above. The component categories and the components per category for our sample car are shown in Fig. 3.

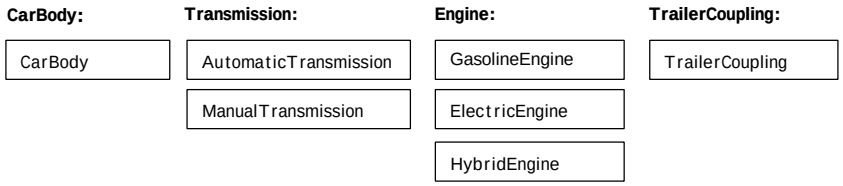

Fig. 3. Component categories for the car product line

Identify "uses" dependencies between component categories. For our example, let's assume that `CarBody` uses `Engine` (1) and `Transmission` (2), `Transmission` uses `Engine` (3) and `TrailerCoupling` uses `CarBody` (4) (see Fig. 4 a).

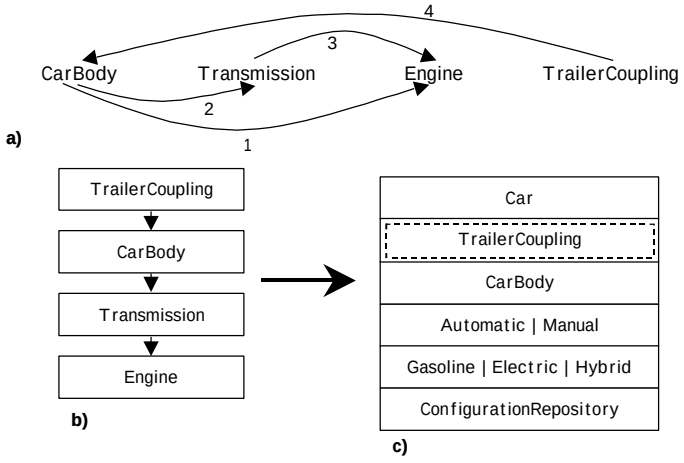

Fig. 4. Derivation of a layered architecture

Sort the categories into a layered architecture. The component categories can be arranged into a hierarchy of layers, where each layer represents a category and the categories that most other categories depend on are moved towards the bottom of the hierarchy (see Fig. 4 b). Finally, we add two more layers: `Car` on the top and

ConfigurationRepository at the bottom (see Fig. 4 c). Car is the top layer identifying all cars. ConfigurationRepository is used to communicate configuration information to all layers. This is possible since a layer may retrieve information from any layer below it. You'll see how this works in Section 5.3. The dashed box around TrailerCoupling indicates that this layer is optional. The Transmission and the Engine layer display the alternative components separated by vertical bars.

Write down the GenVoca grammar. The main idea of the layered architecture in Fig. 4 c is that a component from a given layer takes another component from the layer just below it as a parameter, e.g. CarBody may take AutomaticTrans-mission or ManualTransmission as a parameter, i.e. we may have CarBody[AutomaticTransmission[...]] or CarBody[ManualTransmission[...]]. Using this idea, we can represent this layered architecture as a set of grammar rules (see Fig. 5). Please note that vertical bars separate alternatives and we have abbreviated ConfigurationRepository to Config.

```
Car:                    Car[  CarBodyWithOptTC]
CarBodyWithOptTC:         CarBodyWithTC[ CompleteCarBody] |  CompleteCarBody
CompleteCarBody:          CarBody[ TransmissionWithEngine]
TransmissionWithEngine:   ManualTransmission[Engine] |
                          AutomaticTransmission[Engine]
Engine:                   GasolineEngine[ Config] |  ElectricEngine[ Config |
                          HybridEngine[ Config]
Config:                 speeds, Engine, Transmission,    CarBody, Car
```

Fig. 5. GenVoca grammar for the car product line

At this point the architecture for our car product line is finished. Of course, the steps require an iterative process and prototyping to come up with the final set of components. The latter is also needed to find a stable interface for each of the component categories (each component in a category is required to implement the category interface).

5.3 Implementation Components

Once we have the architecture, we can implement the components. As stated, a component from a given layer takes a component from the layer below it as a parameter, i.e. we need to implement the components as parameterized components. In C++, we can use class templates for this purpose. For example, GasolineEngine can be implemented as follows:

```
template<class  Config_>
struct  GasolineEngine
{ typedef  Config_  Config; //publish  Config as a member type
  GasolineEngine() {  cout << " GasolineEngine "; }
};
```

GasolineEngine takes Config_ as its parameter and publishes it under the new name Config. Any other component that takes GasolineEngine as its parameter can retrieve Config from it using the C++ scope operator ::, i.e. GasolineEngine::Config . ElectricEngine and HybridEngine are implemented in a similar way and are not shown here.

The next component is `ManualTransmission`. According to the grammar in Fig. 5, it takes an `Engine` as its parameter:

```
template <class Engine>
struct ManualTransmission
{ typedef typename Engine::Config Config;
  enum { speeds = Config::speeds };
  Engine e;
  ManualTransmission(){ cout << speeds << "- SpeedManualTransmission "; }
};
```

`ManualTransmission` retrieves `Config` from its parameter `Engine` and publishes it under the alias `Config`. This is how `Config`, which is the bottom layer in the hierarchy, is propagated all the way up to the top layer. The line below the `typedef`-declaration shows you how `Config` is used. In this example, `ManualTransmission` retrieves the number of speeds from the `Config` (e.g. 4 or 5). The remaining components `AutomaticTransmission`, `CarBody`, and `CarBodyWithTC` (TC stands for trailer coupling) are implemented in a similar way. Let us just take a look at the last component, namely `Car`:

```
template <class CarBody_>
struct Car
{ typedef typename CarBody_::Config Config; // Config is part of any car!
  CarBody_ cb;
  Car() { cout << "Car " << endl << endl; }
};
```

5.4 Manual Assembly

Now that we have the implementation components, we can build different cars by writing down different configuration repositories, in which the appropriate components are assembled together. For example, the following configuration repository defines a car with a gasoline engine, five-speed manual transmission, and without a trailer coupling:

```
struct Config1
{ enum { speeds = 5 };
  typedef GasolineEngine<Config1> Engine;
  typedef ManualTransmission<Engine> Transmission;
  typedef CarBody<Transmission> CarBody;
  typedef Car<CarBody> Car;
};
```

And a car with an electric engine, four-speed automatic transmission, and without a trailer coupling looks as follows:

```
struct Config2
{ enum { speeds = 4 };
  typedef ElectricEngine<Config2> Engine;
  typedef AutomaticTransmission<Engine> Transmission;
  typedef CarBody<Transmission> CarBody;
  typedef Car<CarBody> Car;
};
```

You can declare an instance of the latter car as follows:

```
Config2::Car c2;
```

Writing configuration repositories is a tedious exercise. The person writing them needs to know what implementation components are available, what are the valid configurations (e.g. an electric or hybrid engine requires an automatic transmission),

and which configurations are more optimal or satisfy some other requirement. Thus, requiring the application programmer to write a configuration repository places quite a burden on her. Even worse, it makes client code too strongly coupled with the architecture and the implementation components since changes to the architecture (e.g. adding a new layer) may require modifying all configuration repositories.

An alternative would be to include all possible configuration repositories in the library. However, this is usually not practicable since there is normally a large number of configurations (e.g. the matrix computation library described in Section 6 would require 1840 configuration repositories) and each of them is usually longer than in the car example. The solution is to generate the configuration repositories out of more abstract descriptions. But before taking a look at how this works, we need to figure out what an "abstract description" means in our context, i.e. how to conveniently order cars.

5.5 Ordering Cars

When you order a car, you don't have to describe to the car dealer from which parts to assemble it. Instead, you specify the class, the line, and the options, which are usually listed in a product information brochure. An example of such a brochure for our car product line is shown in Fig. 6. The brochure specifies features available as standard equipment or as options for each line. Please note that according to the brochure, it is not possible to order an electric or hybrid engine together with a manual transmission.

	BaseLine	CityLine	EcoLine
Engine			
gasoline	●		
electric		●	
hybrid			●
Transmission			
five-speed manual	●		
four-speed automatic	❍	●	●
Options			
trailer coupling	❍	❍	❍

● standard equipment
❍ options available for a surcharge

Fig. 6. Product information brochure for our car product line

We can easily implement this brochure in C++. Since a client specifies a car by stating the desired line and options, we need to provide the vocabulary representing the lines (BaseLine, CityLine, and EcoLine) and options (transmission and trailer coupling). The enumeration type `Transmission` will be used to specify the transmission and `Options` will be used to state whether a trailer coupling is available or not:

```
enum Transmission {  fiveSpeed,  fourSpeedAutomatic };
enum Options      { none,   trailerCoupling };
```
Now we need the vocabulary representing the three lines. We will model this vocabulary using types rather than enumeration constants since they will contain a member indicating the required transmission:

```
template < int transmission_ =  fiveSpeed>
struct  BaseLine
{ enum { transmission =  transmission_,
         line         = baseline
       };
};
```
BaseLine is available with either manual transmission (standard) or automatic transmission (option). Therefore, we have the template parameter `transmission` with five-speed manual transmission as default. The second member is just a line identifier, so that we can verify at compile time whether a line type is `BaseLine` or not based on the value of `BaseLine::line` . Here is the declaration for `baseline` and the two remaining line identifiers:

```
enum Line { baseline,  cityline,  ecoline }; //for internal use only
```
Finally, we have the two remaining lines CityLine and EcoLine:

```
struct  CityLine
{ enum { transmission =  fourSpeedAutomatic,
         line         =  cityline
       };
};

struct  EcoLine
{ enum { transmission =  fourSpeedAutomatic,
         line         =  ecoline
       };
};
```
Please note that whenever you select a CityLine or an EcoLine, you automatically select a four-speed automatic transmission. Thus, the vocabulary for ordering cars is designed to exclude the possibility of specifying an illegal feature combination (e.g. electric engine and manual transmission). In general, when you design a domain-specific language for “ordering” different systems or components, you may either prevent illegal feature combinations by structuring the language as in the car example or by having an extra “buildability checking” step in the generator. The first option is preferred if the configuration space is highly irregular, i.e. there are many illegal combinations compared to the total number of combinations (i.e. the probability of a mistake is high). The second option is better if there are only few illegal combinations. In this case, the language is simpler and the few possible mistakes are caught by the generator. For example, if the constraint was that a hybrid engine requires an automatic transmission, we could specify a car using three enumeration types `Engine` , `Transmission` , and `Options` . The generator would be responsible for detecting the illegal combination of a hybrid engine and a manual transmission.

Another issue in the design of a domain-specific language is the level of specificity it supports. Ideally, it should support a spectrum from being unspecific (e.g. “give me a car”) to being able to specify details about the implementation components, or even

providing user-defined components in the specification. The trade-offs here include the required level of detail, the stability of the architecture, and the complexity of the configuration space. In any case, supporting different levels of specificity requires defining default settings and default computation rules.

5.6 The Generator

The generator takes a specification of a system or component and returns the finished system or component.

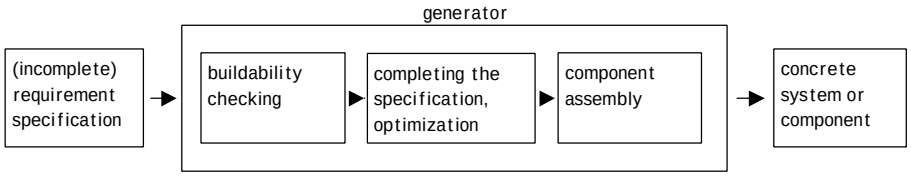

Fig. 7. Stages of a configuration generator

In general, a *configuration generator* [CE99a] performs the following steps (see Fig. 7): it checks if the specified system can be built, completes the specification (by computing defaults), and assembles the implementation components. In our car example, however, there is no buildability checking (since the domain-specific language doesn't give an opportunity to specify illegal feature combinations) and there are no computed defaults (as you'll see in a moment, the few direct default settings are specified in the parameter list of the generator).

Of course, we could implement the generator as a pre-processor generating configuration repositories in C++ source code. A better alternative, however, is to use the built-in metaprogramming capabilities of C++, i.e. *template metaprogramming*. Without explaining all the detail, let us just state that C++ templates constitute a compile-time, Turing-complete sublanguage of C++. In other words, you can use the template instantiation process to perform arbitrary computations at compile time (this was first observed by Erwin Unruh [Unr94]). Meanwhile, there is a whole set of programming idioms and principles based on this idea, which are collectively referred to as "template metaprogramming" [Vel95, CE99b]. For the purpose of this article, you only need to understand that the generator is implemented as a template taking the description of a car as its parameter and returning the finished car type in a specially designated member, which is by convention called `RET` (which stands for `RETURN`). To make this discussion more concrete, let's take a look at how you would create an instance of a BaseLine car with a four-speed automatic transmission and a trailer coupling:

```
CAR_GENERATOR<BaseLine<fourSpeedAutomatic>,trailerCoupling>::RET car1;
```

or an EcoLine car with a four-speed automatic transmission and without a trailer coupling:

```
CAR_GENERATOR<EcoLine>::RET car2;
```

The implementation of CAR_GENERATOR is given in Fig. 8. It uses the templates IF and SWITCH, which correspond to the familiar selection statements if and switch (their implementation is described in [CE99b, CE98]).

```
template <class Line =  BaseLine<>, int options_ = none>
struct CAR_GENERATOR
{ typedef CAR_GENERATOR<Line,options_> Generator;

  //parse the car spec
  enum { line_ =  Line::line,
         transmission_ =  Line::transmission
       };

  // assembly components
  enum { speeds_ = (transmission_ ==  fiveSpeed) ? 5 : 4 };

  typedef SWITCH<line_,
            CASE<baseline,GasolineEngine<Generator>, //see footnote  [8]
            CASE<cityline,ElectricEngine<Generator>,
            CASE<ecoline,HybridEngine<Generator>
          > > > >::RET  Engine_;
  typedef IF<(transmission_ ==  fiveSpeed),
             ManualTransmission< Engine_>,
             AutomaticTransmission< Engine_>
          >::RET  Transmission_;
  typedef IF<(options_ ==  trailerCoupling),
             CarBodyWithTC< CarBody< Transmission_> >,
             CarBody< Transmission_>
          >::RET  CarBody_;

  typedef  Car<CarBody_> RET; //return the finished car!

  //provide the  Config[9]
  struct  Config
  { typedef Engine_ Engine;
    typedef  CarBody_  CarBody;
    enum { speeds       =  speeds_,
           line         =  line_,
           transmission =  transmission_,
           options      =  options_,
         };
    typedef RET Car;
  };
};
```

Fig. 8. C++ implementation of the car generator

The advantage of the implementation using template metaprogramming is that the generator can be used simply as any other template and we don't need any extra pre-processors. The interesting aspect of this kind of metaprogramming is that the metacode performing the configuration at compile time is part of the library of

[8] Since we pass Generator to each engine instead of Config, the engine implementation components GasolineEngine , ElectricEngine , and HybridEngine need to be slightly modified to retrieve the Config from their Generator parameter.

[9] Please note that Config becomes part of any generated car type and can be later accessed by other generators, e.g. generators for algorithms displaying cars.

domain-specific concepts as any other code implementing the concepts (e.g. the implementation components).

6 Applications

We've selected the car example for this paper for pedagogical reasons. You'll find some more computer-oriented examples in [CE99a] (a list container) and in [EC99] (a bank account). We have used the techniques presented here in real-scale systems:

Generative Matrix Computation Library (GMCL) [GMCL, Cza98, Neu98] is a generative C++ library for matrix computations. It contains a configuration generator for generating different kinds of matrix types (different element types, density, storage formats, memory allocation, and error checking) and another kind of generator for generating optimized implementations of matrix expressions, e.g. `(A+B)*(C+D)`. GMCL comprises 7500 lines of C++ code and is capable of generating 1840 different, highly-efficient matrix types.

Generative Matrix Factorization Library [Kna98] contains a configuration 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.

Generative Library for Statistics in Postal Automation [OSVA99] contains configuration generators for different kinds of counters, timers, and statistic algorithms. It is being used by Siemens Electrocom, a world leader in postal automation.

7 Active Libraries

As you saw, the implementation of the generator required metaprogramming. We have used template metaprogramming for this purpose. This was OK, but not perfect due to debugging problems (there is no debugger for the C++ compilation process!) and long compilation times (C++ compilers are not optimized for that kind of strange template programming!). Nevertheless, we were able to demonstrate the idea of putting metacode into domain-specific libraries. In a sense, you can think of the metacode performing compile-time configuration and optimization as extending the C++ compiler. In general, you would also like to have other kinds of metacode extending just about any aspect of a programming environment. This brings us to the idea of *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." An example of a system supporting this idea is Intentional Programming (IP), which is being developed at Microsoft Research [Sim96]. IP is an extendible programming environment which lets you contribute

metacode to extend any aspect of the system including the debugger, the compilation system, and the source editing and display system. It is important to note that the program source IP operates on is not text but an active object structure. This way, all kinds of domain-specific notations are possible (both textual and graphical) and the programmer can actually *interact* with his program code while coding. The wide availability of such extendible programming environments will bring a wholly new dimension to Generative Programming.

8 Conclusions

A component is always a part of a well-defined production process. For example, a brick is a component in the process of building houses not cars. Often cited criteria such as binary format, interoperability, language independence, etc., are always relative to the production process. For example, if you need containers in C++, STL components are just fine. If you need to build GUI windows, you need visual components (e.g. a JavaBean). If you need language-independent, distributed components, you may use CORBA.

Just as it took several decades for the idea of interchangeable parts to be widely used in manufacturing, the transition to interchangeable software components will not happen instantly. In particular, there is a cultural change required on the part of customers, consultants, and vendors to accept solutions based on standard componentry rather than "artistic" individual solutions. The introduction of interchangeable software components requires product-line architectures to be in place. Only that way it is possible to easily and quickly say whether a component offers what a given system expects or not. Thus, we'll need more architectural standardization in different industries before the idea of software components truly takes off.

If you can assemble your components manually, you can also automate the assembly process using a generator. Automation is a logical step once you have a plug-and-play architecture in place. However, just as there is an additional cost to developing reusable software rather than just single systems, there is an additional cost to automation. The availability of standard architectures and components and industrial-quality metaprogramming environments based on the idea of active libraries will help reaching the break-even point more quickly.

One final note: Generation can be performed both statically and dynamically, so your metaprogramming environment should allow you to execute metacode at different times!

Note: The complete source code for the car example is available at http://nero.prakinf.tu-ilmenau.de/~czarn/esec99

References

[Arr94] G. Arrango. Domain Analysis Methods. In *Software Reusability*, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, 1994, pp. 17-49

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. Wiley, UK, 1996

[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

[CE99a] K. Czarnecki and U. Eisenecker. Synthesizing Objects. In *Proceedings of ECOOP'99 - Object-Oriented Programming*, LNCS, Springer-Verlag, 1999, see http://nero.prakinf.tu-ilmenau.de/~czarn/ecoop99/

[CE99b] 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

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

[Cox90] B. J. Cox. Planning the Software Industrial Revolution. In *IEEE Software*, November 1990, see http://www.virtualschool.edu

[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

[EC99] U. Eisenecker and K. Czarnecki. Generative Programmierung: Teil 1 & 2. To appear in the German edition of the *Microsoft System Journal*, no. 4 and 5, 1999

[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

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

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

[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

[KLL+97] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open Implementation Design Guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE)*, 1997, pp. 481-490

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP '97)*, M. Aksit

and S. Matsuoka, (Eds.), Springer-Verlag 1997, pp. 220-242, see http://www.parc.xerox.com/aop

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

[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

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

[OSVA99] Objektorientierte Softwarewiederverwendung in verteilten Architekturen. Final report of the OSVA project. Sponsored by the German Federal Ministry of Education, Science, Research and Technology (BMBF), grant no. 01IS605A4, Mai 1999

[Par76] D. Parnas. On the design and development of program families. In *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, 1976, pp. 1-9

[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 *Proceedings of the 12th European Conference Object-Oriented Programming (ECOOP'98)*, LNCS 1445, Springer-Verlag, 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

[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/

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

[Vel95] T. Veldhuizen. Using C++ template metaprograms. In *C++ Report*, vol. 7, no. 4, May 1995, pp. 36-43, see http://monet.uwaterloo.ca/blitz/

[Wil97] A. Willey. Technology Transition: An Historical Perspective. June 6, 1997, online article available at http://www.virtualschool.edu