Software Product Line Development by Analogy

Rubén Heradio and José A. Cerrada

Departamento de Ingeniería de Software y Sistemas Informáticos
Universidad Nacional de Educación a Distancia, Juan del Rosal, 16, E-28040, Madrid
{rheradio, jcerrada}@issi.uned.es

Abstract. Although the benefits of taking a product line approach to develop similar software systems are well documented, early case studies have revealed significant barriers to adopt this approach. In order to minimize the paradigm shift between conventional software engineering and software product line engineering, this paper presents a new development process where the products of a domain are generated by analogy to an existing product. This paper also discusses the capabilities and limitations of different techniques, currently applied to generalize code, to implement the analogy relation and proposes a new language to overcome the limitations.

1 Introduction

The benefits of taking a SPL (Software Product Line) approach to develop similar software systems, in matter of quality, productivity and time-to-market, are well documented [1, 2]. However, early case studies have revealed significant barriers to adopt a SPL approach. For example, in its successful Diesel Engine SPL, Cummins stopped all product deployments for six months [1]. As C. Krueger argues [3], many organizations can’t afford to slow or stop production for six months, even if the potential ROI (Return Of Investment) is huge.

In order to minimize the paradigm shift between conventional software engineering and SPL engineering, C. Krueger [3] identifies three prominent adoption models:

1. The proactive approach to SPLs, also named big bang approach [4], is like the waterfall approach to conventional software (you analyze, architect, design, and implement all product variations on the foreseeable horizon up front).
2. The reactive approach is like the spiral or extreme programming approach to conventional software (you analyze, architect, design, and implement one or several product variations on each development spiral).
3. The extractive approach reuses one or more existing software products for the product line’s initial baseline.

The reactive approach is appropriated when it is difficult to predict the requirements for product variations or if organizations must maintain aggressive production schedules with few additional resources during the transition to a product line approach. The extractive approach is very effective for an organization that wants to quickly transition from conventional to software product line engineering.
In this paper, we present the EDD (Exemplar Driven Development) process to develop SPLs, which adopts reactive and extractive approaches. EDD takes advantage of the similarities among domain products to make them by analogy. Figure 1 illustrates EDD. The EDD starting point is any domain product built using conventional software engineering. This exemplar implements implicitly the intersection of all the domain product requirements. Next, the exemplar is flexibilized to satisfy the domain variable requirements that are out of the intersection. That is, an analogy relation is defined in a formal way to derive products automatically from the exemplar. The result of the exemplar flexibilization is a DSL (Domain Specific Language) compiler, which is used during application engineering to get the products automatically. EDD is extractive because reuses existing exemplars as product line's initial baseline and is reactive because proposes domain engineering as an incremental activity where flexibilization layers, which implement variable SPL requirements, are added to the exemplar in successive development cycles.

This paper is focused on how to implement the exemplar flexibilization and is structured as follows. Section 2 summarizes EDD. Section 3 presents the new language EFL (Exemplar Flexibilization Language) that supports the flexibilization of any software artifact. Section 4 exemplifies the flexibilization capabilities and limitations of different techniques that are currently used to generalize code, and how EFL overcomes these limitations. Section 5 lists successful applications of EDD and EFL to solve several examples taken from the generative programming literature and to develop real SPLs. Finally, the section 6 summarizes the presented work.

2 Exemplar Driven Development

The decision to make a SPL is usually taken when repetitive work is detected in a domain or when business opportunities are identified in the extension of a successful product. Therefore, when the SPL development starts there is often available an exemplar of the domain. EDD tries to maximize the reuse of this exemplar applying intensively the idea of analogy in all the domain engineering activities.

1. Domain analysis. EDD domain analysis is based on the exemplar analysis. Since mandatory features, common to all domain products, are implicitly in the exemplar requirements, the domain analysis is focused on identifying the variable
features, looking for the differences between the exemplar requirements and the requirements of the remaining products.

2. **Domain design.** EDD derives the SPL architecture from the exemplar architecture. The domain design specifies what adaptations of the exemplar design are necessary to transform it into the design of any other product.

3. **Domain implementation.** The exemplar is flexibilized to provide its automatic adaptation to satisfy input DSL specifications. The implementation techniques to flexibilize the exemplar should support the next desirable capabilities:

   - Non-invasiveness. Figure 2 shows how EDD integrates with the Boehm's spiral lifecycle model [5]. In the first development cycle an exemplar is built using conventional software engineering. In the next successive cycles, flexibilization modules are added to the exemplar to introduce the domain variability. When the flexibilization modules are not invasive to the exemplar, the manual modification of the exemplar in every development cycle is avoided, facilitating the SPL evolution.

   - Crosscutting flexibilizations. As argued in [6], some domain variability should be implemented as crosscutting concerns. In the EDD context, a flexibilization module should be able to adapt different modules of the exemplar implementation.

   - Applicable to any kind of software artifact. It should be supported the flexibilization of the exemplar documentation, test cases…

   - Efficient management of the variability. For example, providing the parametrization of the inter-product variability before the products runtime.

A considerable variety of techniques, commonly used to generalize code, have been evaluated in [7] to flexibilize exemplars. These techniques can be classified into internals, if the flexibilization is implemented using the mechanisms available in the language where the exemplar is written (for example, using inheritance, genericity, aspects…), and externals, if a different language or tool is used. Section 4 exemplifies the use of some available internal and external techniques, and characterizes important limitations of these techniques. We propose the EFL language to overcome these limitations.

![Figure 2. EDD integration with the Boehm's spiral lifecycle model](image-url)
3 Exemplar Flexibilization Language

EFL is an external flexibilization technique that supports the desirable capabilities cited in the previous section (non-invasiveness, crosscutting flexibilizations, applicable to any kind of software artifact and with efficient variability management). Figure 3 illustrates a typical DSL compiler written in EFL, where the generator is in addition another compiler which is used to adapt the exemplar to satisfy the DSL source specifications. In this subcompiler, the flexibilization is usually decomposed into subgenerators responsible for different variability aspects.

For the sake of clarity, figure 4 shows a simplified EFL metamodel, where the details about the DSL analysis and the exemplar analysis have been suppressed. EFL supports the writing of generators that transform input exemplar files into output final product files according to input DSL specifications. An EFL generator is composed of substitutions, productions and generations.

1. A substitution describes the interchange of an exemplar code pattern to new code. Crosscutting generators often apply the same substitutions over different exemplar files. To avoid the repetitive writing of substitutions and support their reuse, substitutions are independent from the exemplar files and the final product files.

2. A production describes the simultaneous application of a substitution set to an exemplar file to produce a final product file. EFL provides the automatic detection of undesirable overlaps among the code patterns of the substitutions of a production.

3. A generation executes simultaneously a set of productions. EFL provides the automatic detection of undesirable collisions among the productions of a generation.

Two generators G1 and G2 can be combined with the operators sequence, add and superposition.

1. Sequence(G1, G2) executes G1 first and G2 later.
2. **Add(G1, G2)** returns a new generator which substitutions and productions are the union of the substitutions and productions of G1 and G2.

3. **Superposition(G1, G2)** updates the substitutions and productions of G1 with the substitutions and productions of G2. Those with the same name are overwritten and the remaining ones are added.

The *add* and *superposition* operators check automatically the collisions between generators.

To build complex exemplar flexibilizations, the operators can be combined. For example, **Sequence(Add(Superposition(G1,G2),G3),G4)**.

EFL is currently implemented as a library of the Ruby object oriented language. Section 4.3 exemplifies the use of this implementation, freely available at [http://www.issi.uned.es/miembros/pagpersonales/ruben_heradio/rheradio_english.html](http://www.issi.uned.es/miembros/pagpersonales/ruben_heradio/rheradio_english.html)

![Figure 4. Simplified EFL metamodel](image)

### 4 Example: the “List Container”

This section solves a simplified version of the “list container” example proposed in [8] using different flexibilization techniques\(^1\). The goal of this problem is to build a generative model that supports the automatic development of list containers written in C++. Figure 5 is the feature diagram that models the SPL variability according to the notation proposed in [9]. Summarizing the features: **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 squared brackets; **Ownership** specifies 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., owner reference), or keeps copies of the original elements and is responsible for their allocation and deallocation (i.e., copy); **LengthCounter** specifies if there is available a counter of type **LengthType** to know the length of the

---

\(^1\) The code of all the flexibilizations discussed in this section is available at: [http://www.issi.uned.es/miembros/pagpersonales/ruben_heradio/rheradio_english.html](http://www.issi.uned.es/miembros/pagpersonales/ruben_heradio/rheradio_english.html)
list; and finally, *Tracing* specifies if the list traces its operation, by logging operation calls to the console.

Let's suppose that the exemplar of the figure 6 is available at the beginning of the SPL development. This exemplar implements the shadowed features in the feature diagram. According to the EDD approach, an analogy relation will be defined to automatically derive the remaining products from the exemplar. That is, the bold code in the figure 6 will be flexibilized to adapt the exemplar according to the *ElementType*, *Ownership*, *Length-Counter*, *LengthType* and *Tracing* features.

![Figure 5. Feature diagram of a simple list container](image)

**4.1 Flexibilization with internal techniques**

Internal techniques provide an easy way to express the exemplar adaptations (i.e., they avoid the manipulation of internal representations of the exemplar, such as abstract syntax trees) and take full advantage of the built-in facilities of the exemplar implementation language (for example, the host language type system can help us to detect flexibilization errors). However, this approach has several disadvantages:

- As argued in section 2, the flexibilization of any kind of software artifact should be supported. Unfortunately, some artifacts are written in languages with reduced capability to manage the variability (for example, think about the flexibilization of HTML documentation using HTML).

- Most of the variability mechanisms available in the current programming languages are able to perform only a certain kind of flexibilizations. On the other hand, one flexibilization can be made using different mechanisms. As a consequence, many flexibilizations get complicated because require the combined use of several mechanisms or to choose among alternative mechanisms (the difficulties to choose the best variability mechanism for a given problem are illustrated in the chapter 6 of [10], where J. O. Coplien tries to systematize this election).

C++ provides two ways to parametrize types: *genericity* and *inheritance* (through abstract classes and late binding). Unfortunately, the both mechanisms are invasive (i.e., the lines 3, 5, 7, 12, 15, 17, 33 and 36 in the figure 6 have to manually changed). In our case, genericity provides a better flexibilization since manages the inter-product variability at compile-time. Therefore, the *ElementType* and *LengthType* flexibilizations will be made using C++ templates (see the code within the *<* and *>* symbols in the figure 7).
Sections 4.1.1 and 4.1.2 discuss the Ownership, LengthCounter and Tracing flexibilizations using inheritance and aspects respectively.

```
class List {
    private:
    MyClass* head_;  
    List* tail_; 
    int length_; 
    public:
    List(MyClass& h, List *t=0):
        head_(0), tail_(t), length_(computedLength())
    { setHead(h); }
    ~List() {
        delete head_; 
    }
    void setHead(MyClass& h) {
        cout << "setHead(" " h "")" << endl;
        head_ = new MyClass(h);
    }
    MyClass& head() {
        cout << "head()" << endl;
        return *head_; 
    }
    void setTail(List *t) {
        cout << "setTail(t)" << endl;
        tail_ = t;
        length_ = computedLength();
    }
    List *tail() const {
        cout << "setTail(t)" << endl;
        return tail_; 
    }
    const int& length() const 
    { return length_; }
    private:
    int computedLength() const 
    { return tail_?tail()->length()+1:1; }
};
```

Figure 6. An existing exemplar: a list of MyClass elements with Copy Ownership, int LengthCounter and Tracing

4.1.1 Inheritance

The flexibilizations related to Ownership, LengthCounter and Tracing can be implemented in classes that inherit from the exemplar. These classes will non-invasively adapt the exemplar adding and overwriting methods and attributes. However, the flexibilization requires other adaptations, not supported by the inheritance mechanism:
1. Removing attributes, expressions, sentences and methods from the base class. For example, to generate a list container without *length counter* the next elements have to be deleted in figure 6: the *length_* attribute (line 5), the *length_* (computedLength()) expression (line 8), the *length_* = computedLength() sentence (line 26) and the *length* and the computedLength methods (lines 33-37).

2. Changing the *private* attributes and methods on the base class to *protected*, to make them accessible from the derived classes.

3. Modifying the base class destructor. The C++ compiler ensures that all destructors are always called (see chapter 14 of [11]). The base class destructor ~List should be modified to prevent the element deallocation in list containers with *external reference*.

Therefore, inheritance has a limited flexibilization power that involves changing the exemplar by hand (i.e., is not really non-invasive). Figure 7 shows the new exemplar resulting from this manipulation.

```cpp
template <class ElementType>
class List {
  protected:
  ElementType* head_;  // New member to store the head of the list
  List<ElementType>* tail_;  // New member to store the tail of the list
  public:
  List(ElementType& h, List<ElementType> *t=0) {
    setHead(h);
    setTail(t);
  }
  virtual void setHead(ElementType& h) = 0 {};
  virtual ElementType& head() {
    return *head_;  // Needs to be protected
  }
  virtual void setTail(List<ElementType> *t) {
    tail_ = t;
  }
  List<ElementType> *tail() const {
    return tail_;  // Needs to be protected
  }
};
```

Figure 7. Exemplar manipulated to make possible the flexibilizations based on inheritance and AOP (with AspectC++)

Figure 8 shows the flexibilization of the new exemplar based on multiple inheritance, where the ExtRefList, OwnRefList and CopyList classes implement the *Ownership* feature; the LengthCounterList class implements the *LengthCounter* feature; and the TracingList class implements the *Tracing* feature. Figure 8 also exemplifies how to get a list container *MyList* with *copy ownership*, *length counter* and *tracing*.
Unfortunately, multiple inheritance introduces some ambiguity that the compiler can not solve. For example, when the setTail method of the MyList class is called, the compiler is unable to decide between the execution of the setTail method of the LengthCounterList class or the execution of the setTail method of the TracingList class. This ambiguity can be manually solved using single inheritance instead of multiple inheritance. But, as figure 9 shows, this solution introduces excessive redundancy (i.e., a combinatorial explosion of classes). Finally, the figure 10 shows how the multiple inheritance and the single inheritance redundancy can be avoided applying parametrized inheritance (i.e., using genericity to parametrize the base classes of LengthCounterList and TracingList).

4.1.2 Aspect Oriented Programming

The flexibilizations related to Ownership, LengthCounter and Tracing can be implemented as aspects that crosscut the exemplar. However, the available AOP extensions to programming languages often have limitations that hinder totally non-invasive flexibilizations. For example, the only structural changes that AspectC++ supports are 1) the addition of attributes and methods to a class, and 2) the change of the base class of a given class (see page 28 in [12]). So, an AspectC++ flexibilization would imply to manipulate the exemplar to transform it into the figure 7. Besides, the AspectC++ weaving is restricted to non-templated code (see page 19 in [13]). Therefore, the implementation of the ElementType and LengthType features using genericity would be substituted by an inefficient implementation based on inheritance.
Rubén Heradio and José A. Cerrada

Figure 9. Exemplar flexibilization based on simple inheritance

Figure 10. Exemplar flexibilization based on parametrized inheritance
4.2 Flexibilization with external techniques

4.2.1 Text templates

Many DSL tools (Microsoft DSL Tools [14], ANTLR [15], openArchitectureWare [16]…) use text templates to generate code. A template can be viewed as a piece of an exemplar with holes. The exemplar code that is common to all the domain products is maintained in the template, whereas the variable code is replaced by holes, that are filled with metacode which specifies how code must change. Figure 11 shows a piece of the exemplar flexibilization applying the ERB [17] Ruby library for text templates, where metacode (i.e., the Ruby code that implements the variable domain features) is embraced within the `<%` and `%>` symbols.

Unfortunately, code and metacode are strongly coupled in templates. Indeed if the template engine does not support aspect oriented programming, templates may suffer metacode tangling (multiple variable concerns implemented simultaneously in a template) or metacode scattering (a variable concern implemented in multiple templates). For example, the flexibilization showed in figure 11 suffers metacode tangling because metacode in lines 1, 3, 4, 9, 10 and 11 manages the `Element Type` variability, whereas the metacode in lines 5, 6, 7, 13 and 15 manages the `Length Counter` variability.

```ruby
class <%=@list_specificacion['Element Type']%>List {
  private:
  <%=@list_specificacion['Element Type']%>* head_;
  <%=@list_specificacion['Element Type']%>* tail_;
  <% if @list_specificacion['Length Counter Type']%>
  <%=@list_specificacion['Length Counter Type']%> length_;
  <% end %>
  public:
  <%=@list_specificacion['Element Type']%> List
  ( <%=@list_specificacion['Element Type']%>&h, 
    <%=@list_specificacion['Element Type']%>*t=0): 
    head_(0), tail_(t)
    <% if @list_specificacion['Length Counter Type']%>
    , length_(computedLength())
    <% end %>
  { setHead(h); }
  ...
```

Figure 11. Exemplar flexibilization with text templates

4.2.2. EFL

Figure 12 shows the exemplar flexibilization using the available EFL implementation in Ruby, where the `ElementType`, `Ownership`, `LengthCounter` and `Tracing` features are managed by the generators `ElementType`, `Ownership`, `LengthCounter` and `Tracing`. This flexibilization has good modularity, is concise, non-invasive and manages the inter-product variability before runtime.

In the Ruby EFL implementation, the `generators` are classes that inherit from the base class `Generator`.

There is a wide variety of methods (`sub`, `gsub`, `delete`, `before...`) to define substitutions. For example, the line 3 in figure 12 defines the substitution of all
the occurrences of the `MyClass` code pattern for the value of the `element_type` string. The code patterns are expressed by regular expressions. To specify complex code patterns, EFL provides the `>` operator that chains regular expressions to define successive code matchings (i.e., `rel1>rel2` means match the regular expression `rel2` with the result of matching the regular expression `rel1`). Indeed, as EFL generators are common Ruby code, the available Ruby meta-parsers can be used to write complex substitutions (Section 6.5.2 of [7] exemplifies how to write substitutions using the `Racc` meta-parser [18]).

Productions are defined by the `prod` method. For example, the line 18 of the figure 12 defines a production that applies the substitutions defined in lines 11, 12 and 14 to the `exemplar` file to produce the `out` file.

```ruby
12
Rubén Heradio and José A. Cerrada

Figure 12. Exemplar flexibilization with EFL
```
Finally, the figure 13 shows how the generators are combined to adapt the exemplar cooperatively. Because there are overlaps between the substitutions of the `ElementType` and `Ownership` generators, they are sequentially combined. The `Ownership`, `LengthCounter` and `Tracing` generators are simultaneously combined with the `add` operator.

**Figure 13. Combination of the EFL generators**

5 **Practical experience and results**

At the moment, EDD and EFL have been successfully applied to the development of:

1. Several examples taken from the generative programming literature (see chapters 4, 5 and 6 of [7]), including:
   a. A tool that interprets documentation embedded in SQL and turns it into external documentation in HTML (example proposed in the chapter 6 of [19]).
   b. A generator that receives abstract definitions for file formats and produces Java libraries to read the files (example proposed in the chapter 9 of [19]).
   c. The “trash recycler” problem proposed in [20].
   d. The “dictionary” example proposed in the chapter 1 of [21].

2. A generative model that produces stored procedures in Transact SQL to load a Data Warehouse (see section 6.3 in [7]).

3. The `m2unit` tool that generates Modula-2 test cases from embedded code (see section 6.4 in [7]).

4. A Data Acquisition SPL for the Astrophysics Institute of the Canary Islands [22].

5. A generative model that produces, from abstract specifications, change notifications written in PL/SQL for Oracle databases [23].

6 **Conclusions**

In this paper we have introduced the EDD process to develop SPLs, that minimizes the SPL adoption barrier by means of a reactive and extractive approach. The EDD starting point is any domain product built using conventional software engineering.
EDD pursues the reuse of this exemplar applying intensively the idea of analogy to all the domain engineering activities.

This paper is focused on how to implement an analogy relation to automatically derive all the domain products from an existing exemplar. We have solved an SPL example to illustrate how to implement this relation with some techniques that are currently used to generalize code. The limitations of these techniques have been characterized and the new language EFL has been proposed to overcome the described limitations.

Finally, we have summarized successful applications of EDD and EFL to solve several examples taken from the generative programming literature and to develop real SPLs.

References


22. López Ruiz, J. C. Análisis de la metodología "Exemplar Driven Development" y de la herramienta de transformaciones "Exemplar Flexibilization Language". Construcción de una línea de productos para sistemas de adquisición de datos en astronomia. Oct. 2007. (send an email to jlopezvilanova@gmail.com)