Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

Exemplar Driven Development of Software Product Lines

Ruben Heradio *[1], David Fernandez-Amoros†[1], Luis de la Torre‡[1], and Ismael Abad§[1]

[1]ETS de Ingenieria Informatica,Universidad Nacional de Educacion a Distancia, Madrid, Spain

Abstract

The benefits of following a product line approach to develop similar software systems are well documented. Nevertheless, some case studies have revealed significant barriers to adopt such 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 made by analogy to an existing product. Furthermore, this paper discusses the capabilities and limitations of different techniques to implement the analogy relation and proposes a new language to overcome such limitations.

1 Introduction

Software Product Line (SPL) engineering has become an important and widely used approach for the efficient development of whole portfolios of software products [1, 2]. The fundamental idea of the approach is to undertake the development of a set of products as a single, coherent development activity. Although the benefits of using a SPL in matter of quality, productivity and time-to-market are well documented [3, 4, 5], some case studies have revealed significant barriers to adopt the SPL approach. For example, in its successful Diesel Engine SPL, Cummins stopped all product deployments for six months [6]. As Krueger argues [7], many organizations can not afford to slow or stop production for six months, even if the potential return of investment is huge. In order to minimize the paradigm shift between conventional software engineering and SPL engineering, Krueger identifies three prominent adoption models:

1. The *proactive approach*, also named big bang approach [8], is like the waterfall approach to conventional software: all product variations on the foreseeable horizon up front are analyzed, designed, and implemented.

2. The *reactive approach* is like the spiral or extreme programming approach to conventional software: one or several product variations on each development spiral are analyzed, designed and implemented.

3. The *extractive approach* reuses one or more existing software products for the product line initial baseline.

*rheradio@issi.uned.es
†david@lsi.uned.es
‡ldelatorre@bec.uned.es
§iabad@issi.uned.es

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

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 SPL engineering.

In this paper, we propose the *Exemplar Driven Development* (EDD) process to develop SPLs, which adopts reactive and extractive approaches. EDD takes advantage of the similarities among domain products to make them by analogy. The starting point of EDD is any domain product built using *conventional software engineering*. Implicitly, this *exemplar* implements 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 *Domain Specific Language* (DSL) compiler [9, 10], which is used during application engineering to get the products automatically. EDD is extractive because reuses existing exemplars as product line initial baseline and is reactive because proposes *domain engineering* [11] as an incremental activity where flexibilization layers, which implement variable SPL requirements, are added to the exemplar in successive development cycles.

Furthermore, this paper shows how to implement the exemplar flexibilization. Code Generation is an increasing popular technique for implementing SPLs that produces code from abstract specifications written in DSLs [12, 13, 14]. The next paradox usually comes up when a DSL compiler is developed: *a DSL is a specialized, problem-oriented language. From the point of view of the DSL user, it is interesting that DSL is as abstract as possible (supporting the domain terminology and removing the low-level implementation details). On the other hand, from the point of view of the compiler developer, the DSL abstraction makes harder to build the compiler. That is, the further DSL specifications are from the final code, the more difficult is to transform them into final code.*

We propose to solve such paradox by taking advantage of a common property to all DSL compilers: the similarities among the final products (i.e., domain product commonalities are the main reason to develop the products jointly as a family, instead of one by one). Instead of synthesizing the final code from scratch or transforming a distant input specification, we suggest to obtain the final products adapting a previously developed domain product to satisfy the input DSL specifications: the exemplar. Figure 1 illustrates this approach, where the generator of a DSL compiler is another compiler which is used to adapt an exemplar according to the DSL source specifications. The figure also represents a possible decomposition of this subcompiler into subgenerators responsible of different sorts of variability.

Template languages, such as XPand of openArchitectureWare[1][15] or XVCL [16], implicitly use this approach, since a text template can be viewed as a piece of an exemplar with slots. The exemplar code that is common to all the domain products is maintained in the template, whereas the variable code is replaced by slots, that are filled with *metacode* which specifies how code must change. Unfortunately, code and metacode are strongly coupled in templates. Indeed, as argued in [17], some domain variability should be implemented as *crosscutting concerns*. When a template engine does not support Aspect Oriented Programming (AOP) [18, 19, 20], templates may suffer *metacode tangling* (multiple variable concerns implemented simultaneously in a template) or *metacode scattering* (a variable concern implemented in multiple templates).

To overcome the templates coupling problem, the metacode should be kept out of the exemplar code. In this case, the exemplar might be processed at:

- lexical level, using *regular expressions* [21]. Nevertheless, though regular expressions can manage text in an agile way, they have serious limitations because they are internally implemented as state machines without memory and cannot manage nested or balanced constructs [22].
- syntactical level, using a *metaparser* such as ANTLR [23] or a *transformation language* such as Strat-

[1] see "openArchitectureWare User Guide Version 4.3.1", available at `http://www.openarchitectureware.org`

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

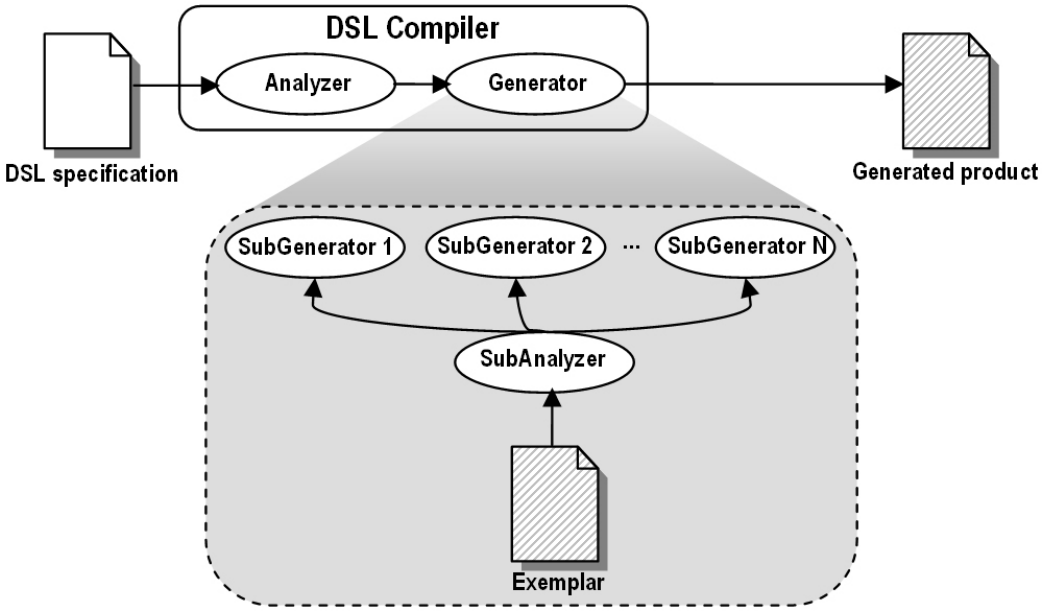

Figure 1: DSL compiler based on the transformation of a domain exemplar

ego [24] or Tom [25]. However, in most cases the simplicity of the exemplar changes does not justify to waste time neither defining the exemplar language grammar nor working with Abstract Syntax Trees (ASTs) [26, 27].

We propose an intermediate solution, the *Exemplar Flexibilization Language (EFL)* [28], that provides new operators to overcome the regular expressions limitations. EFL also supports the integration with parsers to manage marginal complex exemplar modifications. Besides, EFL supports the implementation of *crosscutting generators*, that manage variability scattered over the exemplar, and the decomposition and combination of generators.

The remainder of the paper is structured as follows. Section 2 summarizes EDD. Section 3 describes EFL. Section 4 exemplifies the flexibilization capabilities and limitations of different techniques that are currently used to generalize code, and how EFL overcomes such limitations. Finally, section 5 sums up the conclusions of our work.

2 Exemplar Driven Development

The decision of building a family of systems by using a SPL approach 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.

Figure 2 depicts EDD, which 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 implemented by the exemplar, domain analysis is focused on

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

identifying the variable features, looking for the differences between the exemplar requirements and the requirements of the remaining products.

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

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:

   (a) *Non–invasiveness*. Figure 3 shows how EDD integrates with the Boehm's spiral lifecycle model [29]. 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 in order to introduce the domain variability. When flexibilization modules are not invasive to the exemplar, there is no manual modification of it, facilitating the SPL evolution.

   (b) *Crosscutting flexibilizations*. As argued in [17], some domain variability should be implemented as crosscutting concerns. In the EDD context, the case where one flexibilization module adapts many modules of the exemplar implementation should be supported.

   (c) Applicable to any kind of software artifact. It should be supported the flexibilization of the exemplar documentation, test cases...

   (d) Run–time efficient management of the variability. For example, providing the parametrization of the inter-product variability before the products runtime.

In order to implement the exemplar flexibilization, different techniques commonly used to generalize code may be used. Such techniques can be classified as *internal* and *external*.

1. With internal techniques the flexibilization is implemented using the mechanisms available in the language where the exemplar is written (e.g., using inheritance, genericity, aspects...).

2. With externals techniques the flexibilization is implemented using a different language or tool.

In section 4, we will show how to flexibilize an exemplar using some internal and external techniques typically used to generalize code, identifying their limitations. To overcome such limitations we propose EFL, which is described in the following section.

3 Exemplar Flexibilization Language

A technique for quickly developing a DSL interpreter is embedding it into a *dynamic* general purpose language [9]. This way, all the host language capabilities are implicitly available from the DSL. However, the pay-off is that the DSL concrete syntax has to fit in the host language concrete syntax. EFL is currently implemented applying this technique: it is a library of the Ruby object oriented language. EFL is freely available at `http://rubyforge.org/projects/efl`. As we will see, thanks to Ruby's extensibility, the EFL concrete syntax is reasonably usable.

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

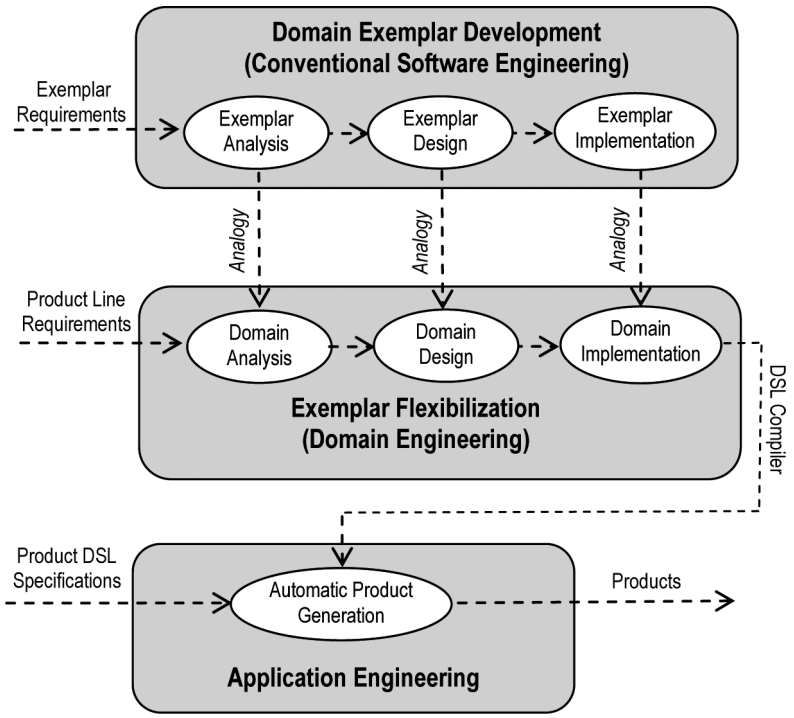

Figure 2: Overview of the EDD process

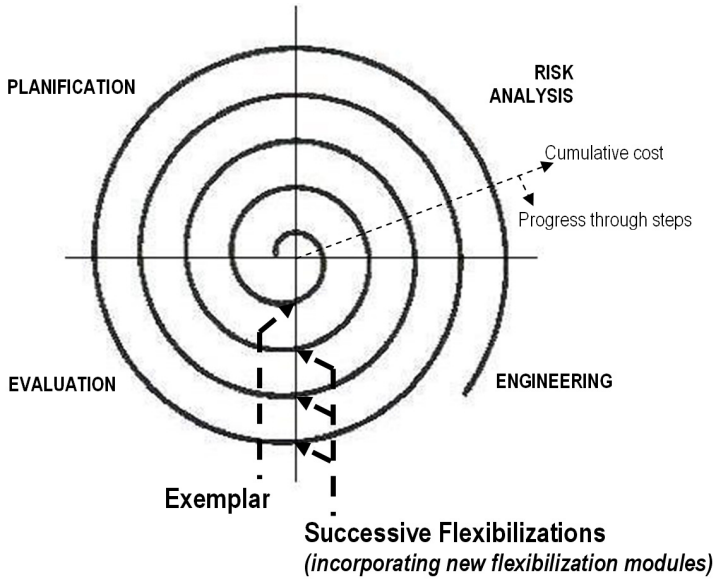

Figure 3: EDD integration with the Boehm's spiral lifecycle model

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

3.1 Defining Generators

Figure 4 shows a simplified EFL metamodel. EFL supports the writing of generators that transform input exemplar files into output final product files according to input DSL specifications. EFL generators are written as Ruby classes that extend from the `Generator` class. This way, generators can be easily reused by mean of the Ruby composition and inheritance capabilities. Alternatively, the following *syntactic sugar* to write generators as objects of the `Generator` class is also available:

```
my_generator = generator {
  << generator definition >>
}
```

A generator definition is composed of *substitutions*, *productions* and *generations*:

1. A *substitution* describes the interchange of an exemplar code pattern, expressed with a regular expression, to new code. Due to EFL is embedded in Ruby, regular expressions are written in the Ruby notation (delimited with the / symbol). For instance, `my_regexp = /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. The main `Generator` methods to define substitutions are:

   - `sub(reg_exp, text, name = nil)`
     Optionally, substitutions, productions and generations can be named using the `name` string.
   - `gsub(reg_exp, text, name = nil)`

   A *local* substitution (`sub`) expresses the interchange of the first occurrence of the `reg_exp` regular expression to the `text` string. A *global* substitution (`gsub`) expresses the interchange of all the `reg_exp` occurrences. Additionally, the `Generator` class provides the next methods:

   - `del` and `gdel` to delete code from the exemplar.
   - `before` and `gbefore` to insert code before the `reg_exp` occurrences.
   - `after` and `gafter` to insert code after the `reg_exp` occurrences.

   Besides, the EFL substitution capabilities can be easily extended adding the correspondending methods to the `Generator` class.

2. A *production* describes the application of a substitution list to an exemplar file to produce a final product file. `Generator` provides the next method to define productions:

   - `prod(input_file, output_file,`
     `sub_list = nil, name = nil)` The order of the substitutions in `sub_list` is irrelevant. If the `sub_list` is not specified, it will contain implicitly all the substitutions defined before the current production.

   EFL supports the detection of undesirable overlaps among the code patterns of the `sub_list` substitutions.

3. A *generation* executes a list of productions. `Generator` provides the next method for generations:

   - `gen(prod_list = nil)` The order of the productions in `prod_list` is irrelevant. If `prod_list` is not specified, it will contain implicitly all the productions defined before the current generation.

   EFL supports the detection of undesirable collisions among the productions of a generation.

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

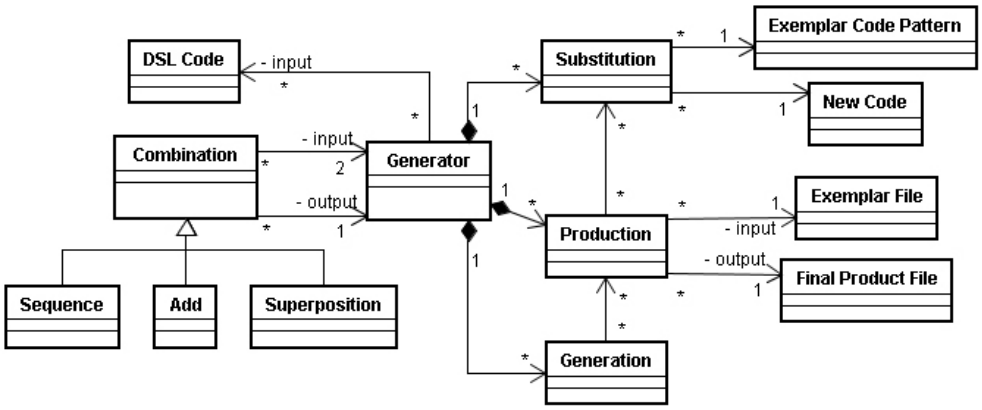

Figure 4: Simplified EFL metamodel

3.2 Combining Generators

For writing complex exemplar transformations, EFL provides the next binary operators to combine two generators `g1` and `g2`:

1. *Sequence*. Executes `g1` first and `g2` later:

   ```
   g1.gen
   g2.gen
   ```

2. *Add*. Returns a new generator which substitutions and productions are the union of the substitutions and productions of `g1` and `g2`:

   ```
   (g1 + g2).gen
   ```

3. *Superposition*. 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:

   ```
   (g1 << g2).gen
   ```

Moreover, these operators can be combined among them. For example, you can write: `((g1 << g2) + g3 + g4).gen`

3.3 EFL Capabilities to Overcome the Regular Expressions Limitations

3.3.1 The Zoom Operator

There are two main types of regular expressions engines: the *Deterministic Finite Automaton* (DFA) and the *Nondeterministic Finite Automaton* (NFA). Being irrelevant for DFA engines how the regular expressions are written, the behaviour of NFA engines, however, depends on the representations of the regular expressions (e.g., a NFA engine follows different ways to match the equivalent regular expressions `regexp1 = /to(ni(ght|te)|knight)/` and `regexp2 = /tonite|toknight|tonight/` against the “tonight" string). According to Friedl [30], most of the programming languages implement NFA engines because give more

control to the programmer, since the representation of a regular expression sets the way the NFA engine backtracks during the matching resolution. Besides, NFA engines provide interesting features, such as capturing parentheses and the associated backreferences (`$1, $2...`), and lazy quantifiers.

Writing a complex and time-efficient regular expression for a NFA engine may be quite hard. To simplify this work, EFL provides the *zoom operator* (`>`) that supports the step-by-step writing of regular expressions. Thanks to this operator, regular expressions can be chained to progressively specify a text pattern; i. e., the expression:

```
regexp1 > regexp2 > regexp3 >
... > regexpN
```

matches the `regexp2` against the text matched by the `regexp1`, the `regexp3` against the text matched by the `regexp2`, etcetera.

3.3.2 Anti-patterns

Sometimes, it is useful to express a pattern in negative terms: instead of specifying the features we are interested in, describing characteristics to exclude some matching candidates. To support the writing of such *anti-patterns*, many regular expression engines provide the next constructs:

- The *negated character class* `[^...]`, which matches any character that is not listed into the character class.
- The negatives *look-ahead* `(?!...)` and *look-behind* `(?<!...)` These look-around constructs do not "consume" any text, but they look forward or backward to "see" if their subexpressions cannot be matched. For example, the evaluation of the `/Ruben (?!Heradio)/` regular expression against the "Ruben Garcia" string only matches the text "Ruben" (i.e. the negative look-ahead queries if anything different of "Heradio" follows "Ruben", but does not consume "Garcia"). Unfortunately, Ruby does not support the negative look-behind construct.

EFL provides two new constructs for writing anti-patterns:

1. *Complement* (`o`) is an unary-operator that inverts the matching of a regular expression. That is, `o(regexp1) > regexp2` matches the `regexp2` out of the text matched by the `regexp1`.
2. *Minus* (`-`) is a binary-operator that excludes candidates for matching. That is, `regexp1 - regexp2` captures the text that is matched by the `regexp1` but not matched by the `regexp2`.

   Sometimes, it is quite hard "to find the precise regular expression", general enough to match all the text of interest and particular enough to ignore the rest. Using the minus operator, this problem can be solved in several steps: first, a more general regular expression is written without worrying about catching some undesirable text and, then, the matching is progressively adjusted by subtracting one or more particular regular expressions.

Figure 5 illustrates several examples of the zoom, complement and minus operators. The top row shows several regular expressions built combining these operators and the bottom row highlights the result of matching the regular expressions against a given text.

3.3.3 Managing Nested Constructs

As it was mentioned in the introduction, regular expressions cannot actually manage nested or balanced constructs because they are internally implemented as state machines without memory. For example, a regular expression for matching any number of balanced parentheses cannot be written, because when the

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

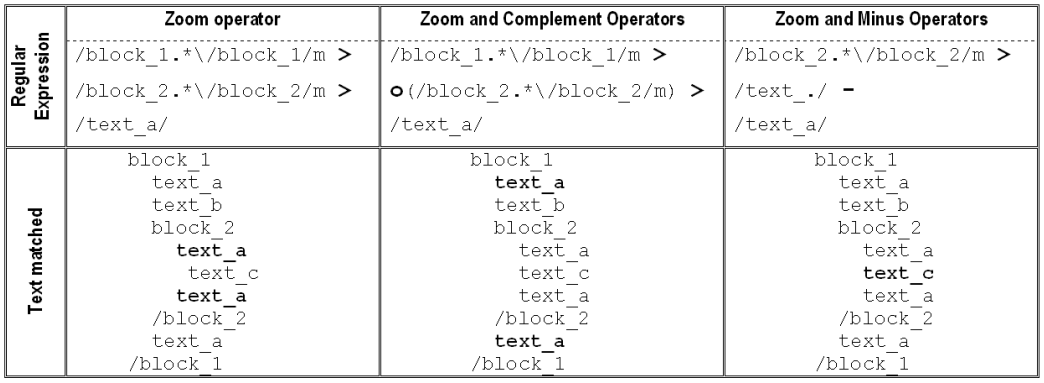

Figure 5: Examples of the zoom, complement and minus operators

state machine finds the first close-parenthesis, is not able to “remember" how many open-parentheses has processed before. However, it is possible to write a regular expression for matching *until a fixed* number of balanced parenthesis. For example, the next three regular expressions match until one, two and three balanced parentheses respectively:

1. `/[(][^()]*[)]/`
2. `/[(][^()] | [(][^()]*[)] )*[)]/`
3. `/[(][^()] | [(][^()] |`
   `[(][^()]*[)])*[)])*[)]/`

Writing a `/[(] ...[)]/` regular expression for each particular case is quite hard and repetitive. Fortunately, this work can be automatized using the Ruby meta-programming capabilities. For example, the `nested_parentheses` function in Figure 6 receives a `levels` number of balanced parentheses and generates the corresponding regular expression. For example, a regular expression for ten balanced parentheses would be obtained with `nested_parentheses(10)`. Internally, this method makes a string that contains the Ruby code for the corresponding regular expression and, then, calls the `eval` method for asking to the Ruby interpreter to evaluate the string (i.e., we are writing Ruby code that: (i) writes more Ruby code and (ii) executes the new code).

```
1  def nested_parentheses(levels)
2     eval('@level0 = "[(][^()' + '])*[)]"')
3     (1..(levels-2)).reject {|i|
4       eval("@level#{i}" + '= "[(][^()' +
5         '] | #{@level' + "#{i-1}" + '} )*[)]"')
6     }
7     if levels > 1 then
8       eval("@level#{levels-1}" +
9         '= /[(][^()' +'] | #{@level' +
10        "#{levels-2}" + '} )*[)]/mx')
11      eval "return @level#{levels-1}"
12    else
13      eval "return /#{@level0}/mx"
14    end
15 end
```

Figure 6: Generating regular expressions to match nested parentheses

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

3.3.4 EFL Integration with Parsers and Text Template Engines

Sometimes, regular expressions are not the best way to write certain exemplar changes. You may want to work at syntactical level (i.e., against a AST) or to use a text template.

Since EFL is embedded in Ruby, EFL generators can integrate parsers (Racc[2] and Rockit[3]) are two currently available metaparsers for Ruby) and text templates (ERB[4] is a valuable text template engine for Ruby). Figure 7 shows how to do this inside substitutions, maintaining the EFL support for detecting undesirable overlaps among the substitutions of a production and the possible collisions among the productions of a generation. The first substitution parameter is a very general regular expression that sets the exemplar scope for the parser or the template. The second parameter calls the parser or the template engine for processing the scoped text and producing the new code. Note that the scope is captured with parentheses and then is passed to the parser or the template through the associate backreference (`$1`).

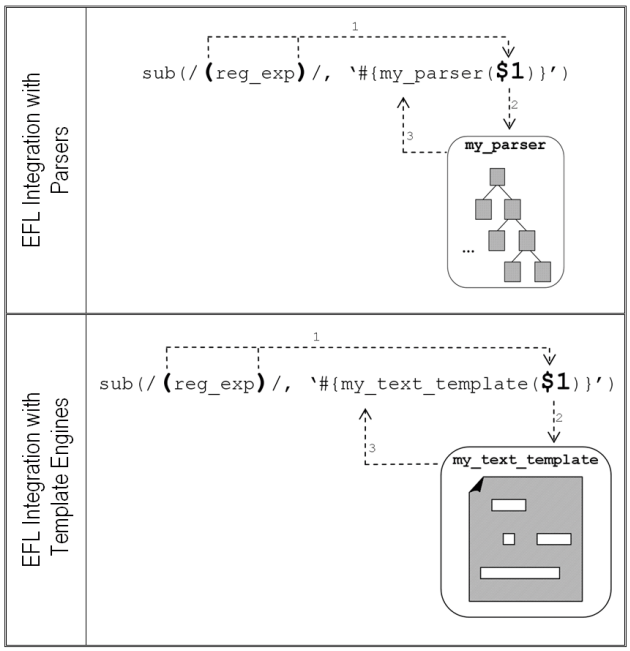

Figure 7: Example of how to integrate a parser or a text template with EFL

4 Example: a SPL for List Containers

This section solves a simplified version of the "list container" example proposed in [31] using different flexibilization techniques. The aim of the example is to develop a portfolio of list containers written in

[2] http://rubyforge.org/projects/racc/
[3] http://sourceforge.net/projects/rockit/
[4] http://raa.ruby-lang.org/project/erb/

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

C++. The PL scope is modeled in Figure 8 by a *Feature Diagram* (FD), which is a widely used notation to depict the commonalities and variabilities of the products supported by a PL [32, 33]. A FD is represented as a hierarchically arranged set of features with different relations among those features. Figure 8 includes three kinds of relations: mandatory (pointed by simple edges ending with a filled circle; e.g., all Lists in the portfolio have a Ownership feature), alternative (pointed by edges connected by an arc; e.g., External reference, Owned reference and Copy are the mutually exclusive values for Ownership) and optional (pointed by simple edges ending with an empty circle; e.g., a List may have the Tracing feature). In this particular example, the meanings of the features are:

1. ElementType specifies the type of the elements stored in the list.
2. Ownership specifies how a list stores its elements:
   (a) External reference: the list keeps references to the original elements and is not responsible for element deallocation.
   (b) Owned reference: the list keeps references and is responsible for element deallocation.
   (c) Copy: the list keeps copies of the original elements and is responsible for their allocation and deallocation.
3. LengthCounter specifies if there is available a counter of type LengthType to know the length of the list.
4. Tracing indicates if a list traces its operation by logging function calls to the console.

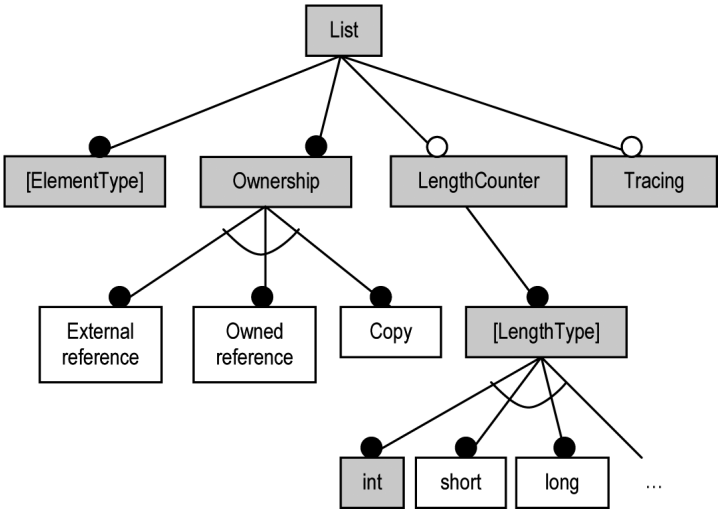

Figure 8: Feature diagram that models a SPL of list containers

Lets suppose that the exemplar of Figure 9, which implements in C++ the shadowed features in Figure 8, is available at the beginning of the SPL development. According to the EDD approach, an analogy relation will be defined to automatically derive the remaining products from the exemplar.

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

```
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 9: An existing exemplar: a list of elements of type `MyClass` that implements features Copy Ownership, int LengthCounter and Tracing

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

4.1 Flexibilization with internal techniques

Internal techniques provide an easy way to express the exemplar adaptations (i.e., they avoid the manipulation of intermediate 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:

- 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 choosing among alternative mechanisms (the difficulties to choose the best variability mechanism for a given problem are illustrated in the chapter 6 of [34], where Coplien tries to systematize such election).

C++ provides two ways of parametrizing *types*: *genericity* and *inheritance*. Nevertheless, both mechanisms are invasive (i.e., the lines 3, 5, 7, 12, 15, 17, 33 and 36 in the Figure 9 have to be changed). Whereas the inheritance flexibilization uses abstract classes and late binding, genericity manages the inter-product variability at compile-time. Figure 10 shows a new version of the exemplar, where feature ElementType is implemented using genericity (i.e., C++ templates).

Sections 4.1.1 and 4.1.2 discuss how to implement the exemplar flexibilization for the features Ownership, LengthCounter and Tracing by using inheritance and aspects respectively.

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 the following 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 should be deleted in Figure 9: attribute `length_` in line 5, expression `length_(computedLength())` in line 8, sentence `length_ = computedLength()` in line 26 and, methods `length` and `computedLength` in 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 in [35]). 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., it is invasive). Figure 10 shows the new exemplar resulting from such manipulation.

Figure 11 shows a flexibilization of the new exemplar based on multiple inheritance, where classes `ExtRefList`, `OwnRefList` and `CopyList` implement feature Ownership; class `LengthCounterList` implements feature LengthCounter; and class `TracingList` class implements feature Tracing. Figure 11 also exemplifies how to get a list container `MyList` with Copy ownership, LengthCounter and Tracing.

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

```
template <class ElementType>
class List {
protected:
  ElementType* head_;
  List<ElementType>* tail_;
public:
  List(ElementType&h, List<ElementType> *t=0)
  {
    setHead(h);
    setTail(t);
  }
  virtual void setHead(ElementType& h) = 0 {};
  virtual ElementType& head()
  {
    return *head_;
  }
  virtual void setTail(List<ElementType> *t)
  {
    tail_ = t;
  }
  List<ElementType> *tail() const
  { return tail_; }
};
```

Figure 10: Exemplar manipulated to make possible the flexibilizations based on inheritance and AOP (using AspectC++)

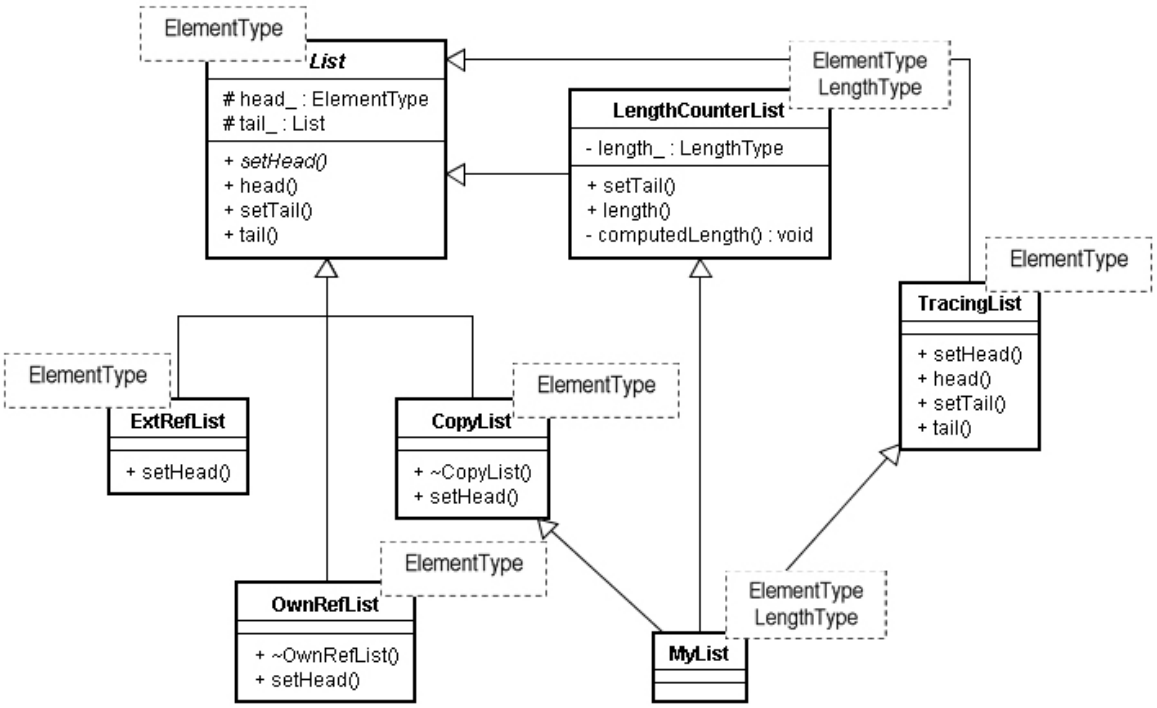

Figure 11: Exemplar flexibilization based on multiple inheritance

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

However, this solution has a drawback: multiple inheritance introduces ambiguity that the compiler can not solve. For example, when method `setTail` of the class `MyList` is called, the compiler is unable to decide between the execution of the method `setTail` of `LengthCounterList` or the execution of `setTail` of `TracingList`. This ambiguity can be solved using single inheritance instead of multiple inheritance. But, as Figure 12 shows, such solution introduces excessive redundancy (i.e., a combinatory explosion of classes). Finally, Figure 13 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).

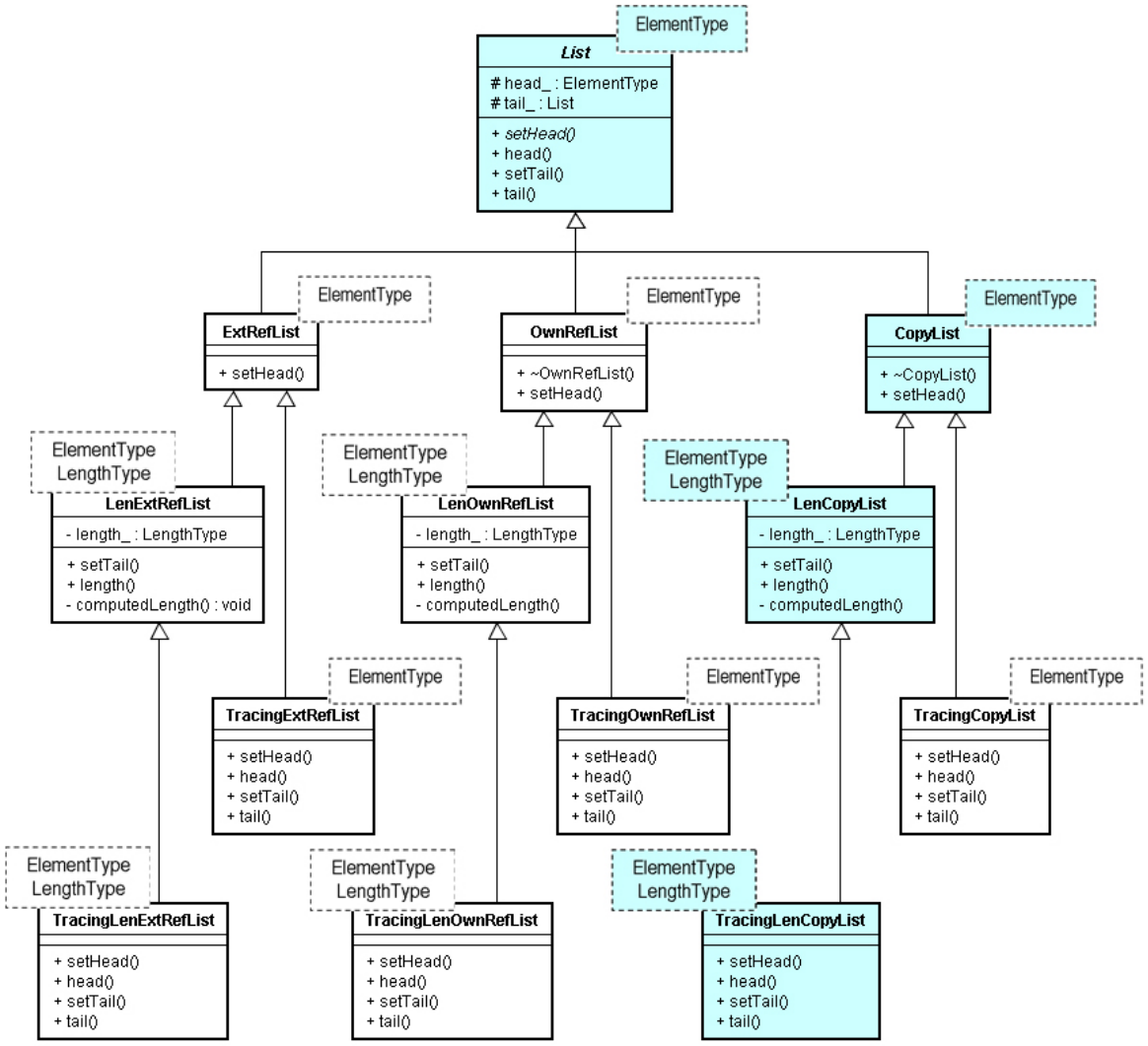

Figure 12: Exemplar flexibilization based on single inheritance

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

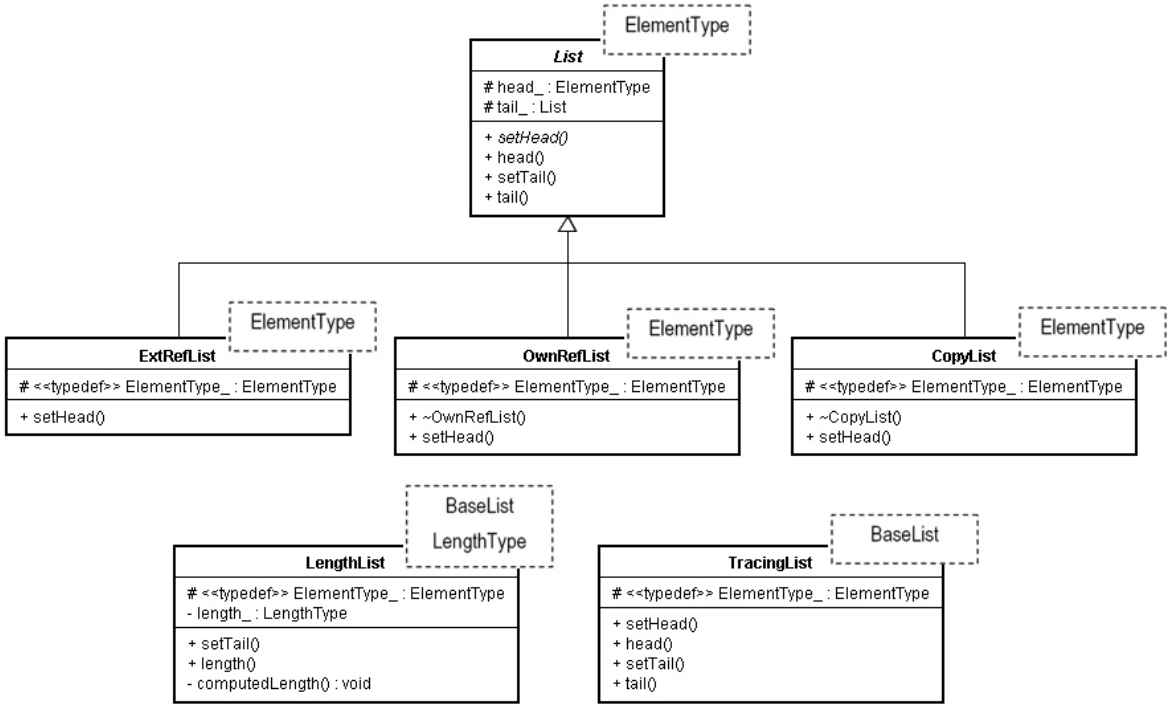

Figure 13: Exemplar flexibilization based on parametrized inheritance

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

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 Aspect Oriented Programming (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[5]. So, an ApectC++ flexibilization would imply to manipulate the exemplar to transform it into Figure 10. Besides, AspecC++ weaving is restricted to non-templated code[6]. Therefore, the implementation of features ElementType and LengthType using genericity should be substituted by an inefficient implementation based on inheritance.

4.2 Flexibilization with external techniques

4.2.1 Text templates

A template can be viewed as a piece of an exemplar with slots. The exemplar code that is common to all the domain products is maintained in the template, whereas the variable code is replaced by slots, that are filled with metacode which specifies how code must change. Figure 14 shows a piece of the exemplar flexibilization applying the ERB Ruby library for text templates, where metacode (i.e., the Ruby code that implements the variable domain features) is embraced within symbols <% and %>.

The main drawback of this template–based solution is that code and metacode are strongly coupled. Indeed, if the template engine does not support AOP, 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 ElementType variability, whereas the metacode in lines 5, 6, 7, 13 and 15 manages the LengthCounter variability.

```
1  class <%=@list_specificacion['Element Type']%>List {
2  private:
3    <%=@list_specificacion['Element Type']%>* head_;
4    <%=@list_specificacion['Element Type']%>List* tail_;
5    <% if @list_specificacion['Length Counter Type']%>
6    <%=@list_specificacion['Length Counter Type']%> length_;
7    <% end %>
8  public:
9    <%=@list_specificacion['Element Type']%>List
10     (<%=@list_specificacion['Element Type']%>&h,
11      <%=@list_specificacion['Element Type']%>List *t=0):
12     head_(0), tail_(t)
13     <% if @list_specificacion['Length Counter Type']%>
14     , length_(computedLength())
15   <% end %>
16     { setHead(h); }
17 ...
```

Figure 14: Exemplar flexibilization using text templates

4.2.2 EFL

Figure 15 shows an exemplar flexibilization using the available EFL implementation in Ruby, where the ElementType, Ownership, LengthCounter and Tracing are managed by the generators ElementType, Ownership,

[5] see page 28 of "AspectC++ Language Reference. Version 1.6", available at http://www.aspectc.org/fileadmin/documentation/ac-language

[6] see page 19 of "AspectC++ Compiler Manual. Version 1.1", available at http://www.aspectc.org/fileadmin/documentation/ac-compilerma

Draft version of the paper published in Expert Systems with Applications, doi:10.1016/j.eswa.2012.05.004

LengthCounter and Tracing. Such flexibilization has good modularity, is concise, non-invasive and manages the inter-product variability before runtime. Generators in Figure 15 include *substitutions* (e.g., line 3 defines the substitution of all the occurrences of the MyClass code pattern for the value of the element_type string) and *productions* (e.g., line 18 defines a production that applies the substitutions defined in lines 11, 12 and 14 to the exemplar file to produce the out file).

```
1  class ElementType < Generator
2    def initialize(exemplar, out, element_type)
3      gsub(/MyClass/, element_type)
4      prod(exemplar, out)
5    end
6  end
7  class Ownership < Generator
8    def initialize(exemplar, out, ownership_type)
9      case ownership_type
10       when 'External reference'
11         sub /delete head_;/, ''
12         sub /new MyClass\(h\)/ , '&h'
13       when 'Owned reference'
14         sub /new MyClass\(h\)/ , '&h'
15       when 'Copy'
16         # no change
17     end
18     prod(exemplar, out)
19   end
20 end
21 class LengthCounter < Generator
22   def initialize(exemplar, out, length_counter_type)
23     if length_counter_type
24       gsub(/int/, length_counter_type)
25     else
26       gsub(/^.*length.*$/i, '')
27       gsub(/\)\:/, '\0head_(0), tail_(t)')
28     end
29     prod(exemplar, out)
30   end
31 end
32 class Tracing < Generator
33   def initialize(exemplar, out, tracing)
34     gsub(/cout.+$/, '') if !tracing
35     prod(exemplar, out)
36   end
37 end
```

Figure 15: Exemplar flexibilization using EFL

Finally, Figure 16 depicts how the generators are combined to adapt the exemplar cooperatively. Because there are overlaps between the substitutions of generators ElementType and Ownership, they are sequentially combined. On the other hand, generators Ownership, LengthCounter and Tracing are combined with the add operator.

5 Conclusions

We have introduced the EDD process to develop SPLs, which minimizes the product line 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.

We have described how to implement analogy relations to automatically derive all the domain products from existing exemplars by using techniques widespread applied to generalize code. The limitations of

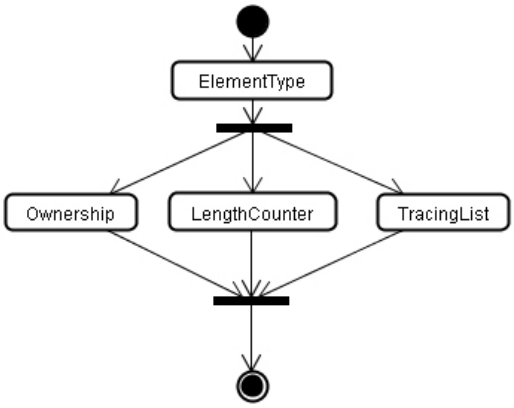

Figure 16: Combination of the EFL generators

such techniques have been exposed and the new language EFL has been proposed to overcome those limitations. At the moment, EDD and EFL have been successfully used to develop: (i) a Data Acquisition SPL for the Astrophysics Institute of the Canary Islands [36] and (ii) a generative model that produces, from abstract specifications, change notifications written in PL/SQL for Oracle databases [37].

References

[1] K. Pohl, G. Bockle, F. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer, 2005.

[2] X. Peng, S.-W. Lee, W.-Y. Zhao, Feature-oriented nonfunctional requirement analysis for software product line, Journal of Computer Science and Technology 24 (2) (2009) 319–338.

[3] C. Dinnus, K. Pohl, Software Product Line Engineering, Springer Berlin Heidelberg, 2005, Ch. Experiences with Software Product Line Engineering, pp. 413–434.

[4] J. Liu, S. Basu, R. R. Lutz, Compositional model checking of software product lines using variation point obligations, Automated Software Engineering 18 (2011) 39–76.

[5] R. Heradio, D. Fernandez-Amoros, L. Torre-Cubillo, A. P. Garcia-Plaza, Improving the accuracy of coplimo to estimate the payoff of a software product line, Expert Systems with Applications 39 (9) (2012) 7919–7928.

[6] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.

[7] C. Krueger, Eliminating the adoption barrier, IEEE Software 19 (4) (2002) 29–31.

[8] K. Schmid, M. Verlage, The economic impact of product line adoption and evolution, IEEE Software 19 (2002) 50–57.

[9] M. Fowler, Domain–Specific Languages, Addison–Wesley, 2010.

[10] M. E. Edge, P. R. F. Sampaio, The design of ffml: A rule-based policy modelling language for proactive fraud management in financial data streams, Expert Systems with Applications 39 (11) (2012) 9966–9985.

[11] I. Reinhartz-Berger, Towards automatization of domain modeling, Data and Knowledge Engineering 69 (2010) 491–515.

[12] J. Herrington, Code Generation in Action, Manning Publications, 2003.

[13] C. Pohl, A. Rummler, V. Gasiunas, N. Loughran, H. Arboleda, F. de Alexandria Fernandes, J. Noye, A. Nunez, R. Passama, M. S. Jean-Claude Royer and, Survey of existing implementation techniques with respect to their support for the requirements identified in m3.2, Tech. rep., AMPLE, Aspect ŰOriented, Model-Driven, Product Line Engineering, Specific Targeted Research Project: IST- 33710 (2007).

[14] J. Lee, J. Park, G. Yoo, E. Lee, Goal-based automated code generation in self-adaptive system, Journal of Computer Science and Technology 25 (6) (2010) 1118–1129.

[15] J. Gallardo, A. I. Molina, C. Bravo, M. A. Redondo, C. A. Collazos, An ontological conceptualization approach for awareness in domain-independent collaborative modeling systems: Application to a model-driven development method, Expert Systems with Applications 38 (2) (2011) 1099–1118.

[16] H. Zhang, S. Jarzabek, A hybrid approach to feature-oriented programming in xvcl, in: 14th International Software Product Line Conference, Jeju Island, South Korea, 2010, pp. 440–445.

[17] M. Voelter, I. Groher, Product line implementation using aspect-oriented and model-driven software development, in: 11th International Software Product Line Conference, Washington, DC, USA, 2007, pp. 233–242.

[18] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, H. Ossher, Discussing aspects of aop, Communications of the ACM 44 (2001) 33–38.

[19] S. A. Vidal, C. A. Marcos, Building an expert system to assist system refactorization, Expert Systems with Applications 39 (3) (2012) 3810–3816.

[20] F. Berzal, F. J. Cortijo, A. Jimenez, Tminer aspects: Crosscutting concerns in the tminer component-based data mining framework, Expert Systems with Applications 37 (9) (2010) 6675 – 6681.

[21] D. F. Barrero, M. D. R-Moreno, D. Camacho, Adapting searchy to extract data using evolved wrappers, Expert Systems with Applications 39 (3) (2012) 3061–3070.

[22] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Prentice Hall, 2006.

[23] T. Parr, The Definitive Antlr Reference: Building Domain-Specific Languages, Pragmatic Bookshelf, 2007.

[24] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser, Stratego/xt 0.17. a language and toolset for program transformation, Science of Computer Programming 72 (1-2) (2008) 52–70.

[25] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, A. Reilles, Tom: Piggybacking rewriting on java, in: 18th International Conference on Term rewriting and applications, Paris, France, 2007, pp. 36–47.

[26] C. Catal, U. Sevim, B. Diri, Practical development of an eclipse-based software fault prediction tool using naive bayes algorithm, Expert Systems with Applications 38 (3) (2011) 2347 – 2353.

[27] V. F. Lopez, R. Aguilar, L. Alonso, M. N. Moreno, Data mining for grammatical inference with bioinformatics criteria, Expert Systems with Applications 39 (3) (2012) 2330–2334.

[28] R. Heradio, J. A. Cerrada, J. C. Lopez, J. R. Coz, Code generation with the exemplar flexibilization language, Electronic Notes in Theoretical Computer Science 238 (2) (2009) 25–34.

[29] B. Boehm, A spiral model of software development and enhancement, ACM SIGSOFT Software Engineering Notes 11 (1986) 14–24.

[30] J. E. Friedl, Mastering Regular Expressions, O'Reilly Media, 2006.

[31] K. Czarnecki, U. Eisenecker, Generative Programming: Methods Tools and Applications, Addison-Wesley, 2000.

[32] K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature-oriented domain analysis (foda) feasibility study, Tech. rep., CMU/SEI-90-TR-21, Software Engineering Institute (1990).

[33] J. Guo, Y. Wang, P. Trinidad, D. Benavides, Consistency maintenance for evolving feature models, Expert Systems with Applications 39 (5) (2012) 4987–4998.

[34] J. O. Coplien, Multi-Paradigm Design for C++, Addison-Wesley, 1998.

[35] B. Eckel, Thinking in C++: Introduction to Standard C++, Volume One, Prentice Hall.

[36] J. C. Lopez-Ruiz, R. Heradio, J. Cerrada, J. Coz, P. L. Ramos, A first-generation software product line for data acquisition systems in astronomy, in: Advanced Software and Control for Astronomy. Marseille, France, 2008.

[37] J. Coz, R. Heradio, J. Cerrada, J. Lopez-Ruiz, A generative approach to improve the abstraction level to build applications based on the notification of changes in databases, in: International Conference on Enterprise Information Systems. Barcelona, Spain, 2008.