Delta-Trait Programming of Software Product Lines

Delta-Trait Programming of Software Product Lines*

Ferruccio Damiani[1], Ina Schaefer[2], Sven Schuster[2], and Tim Winkelmann[2]

[1] Università di Torino, Dipartimento di Informatica, 10149 Torino, Italy
ferruccio.damiani@unito.it
[2] Technische Universität Braunschweig, Germany
{i.schaefer | s.schuster | t.winkelmann}@tu-braunschweig.de

Abstract. Delta-oriented programming (DOP) is a flexible approach for implementing software product lines (SPLs). DOP SPLs are implemented by a set of delta modules encapsulating changes to class-based object-oriented programs. A particular product in a DOP SPL is generated by applying to the empty program the modifications contained in the delta modules associated to the selected product features. Traits are pure units of behavior, designed to support flexible fine-grained reuse and to provide an effective means to counter the limitations of class-based inheritance. A trait is a set of methods which is independent from any class hierarchy and can be flexibly used to build other traits or classes by means of a suite of composition operations. In this paper, we present an approach for programming SPLs of trait-based programs where the program modifications expressed by delta modules are formulated by exploiting the trait composition mechanism. This smooth integration of the modularity mechanisms provided by delta modules and traits results in a new approach for programming SPLs, *delta-trait programming (DTP)*, which is particularly well suited for evolving SPLs.

1 Introduction

A *software product line (SPL)* is a set of software systems with well-defined commonality and variability [13, 29]. SPL engineering aims at developing these systems by managed reuse. Products of an SPL are commonly described in terms of *features* [20], where a feature is a unit of product functionality. Feature-based product variability has to be captured in the product line artifacts that are reused to realize the single products. On the implementation level, reuse mechanisms for product implementations have to be flexible enough to express the desired product variability [34].

Today, many product implementations of SPLs are carried out within the object-oriented paradigm. Although class-based inheritance in object-oriented languages provides means for code reuse with static guarantees, the rigid structure of class-based inheritance puts limitations on the effective modeling of product variability and on the reuse of code [27, 17]. *Feature-oriented programming (FOP)* [5, 2, 16, 1] allows to flexibly implement product lines within the object-oriented paradigm by class refinement. In FOP, a product implementation for a particular feature configuration is obtained by composing *feature modules* for the respective features. A feature module contains class

* Work partially supported by MIUR (proj. CINA), Ateneo/CSP (proj. SALT), Deutsche Forschungsgemeinschaft (grant SCHA1635/2-1), and ICT COST Action IC1201 BETTY.

definitions and class refinements. A class refinement can modify an existing class by adding new fields/methods, by wrapping code around existing methods or by changing the superclass. *Delta-oriented programming (DOP)* [31, 33, 32, 9] extends FOP by the possibility to remove code from an existing product (see [33] for a straightforward embedding of FOP into DOP). In DOP, a product implementation is obtained by applying modifications specified in *delta modules* to existing products. Using FOP/DOP for implementing SPLs results in a scenario where class-based inheritance is the mechanism for *intra-product code reuse* (i.e., for reusing code within single products) and FOP/DOP class refinement/modification is the mechanism for *inter-product code reuse* (i.e., for reusing code across different products). Since class-based inheritance does not support low coupling [27, 17], FOP/DOP class refinement/modification do not mix well with class-based inheritance and, as a matter of fact, little or no intra-product code reuse is realized via class-based inheritance in many FOP/DOP SPL implementations [36].

In object-oriented languages with class-based inheritance, classes have two competing roles: (i) generators of objects and (ii) units of reuse. To counter this, traits are introduced as pure units of behavior, designed for flexible, fine-grained reuse [35, 17]. A trait contains a set of methods, which is independent from any class hierarchy. Thus, the common methods of a set of classes can be factored into a trait. The distinctive characteristic of traits is that they can be composed in an arbitrary order and that the resulting composite unit (which can be a class or another trait) has complete control over the conflicts that may arise in the composition and must solve these conflicts explicitly. Since their first formulation [35, 17], various formulations of traits in a JAVA-like setting can be found in the literature (see, e.g., [38, 28, 30, 12, 25, 11, 8, 7]).

In this paper, we present delta-oriented programming for software product lines of trait-based programs. In the proposed approach, delta modules and traits work synergically together for modeling program variability by flexibly supporting both inter- and intra-product code reuse. In particular, the program modifications expressed by delta modules are formulated by exploiting the trait composition mechanism. This smooth integration of delta modules and traits results in a new approach for programming SPLs, that we call *delta-trait programming (DTP)*. As in DOP, intra-product code reuse is rarely achieved by class-based inheritance, but it could be realized by using design patterns in the implementation of the core products [37]. However, when the SPL evolves the patterns used in core products might not be suitable for supporting intra-product code reuse in the new products. So, either we accept to have code duplications in the code of the new products, or we refactor the whole code base which is both undesirable. To mitigate this problem, DTP offers trait-based intra-product code reuse. As *unanticipated SPL evolution* (i.e., evolution involving changes for which developers have not prepared in the original design of the SPL) scenarios become more common in long-living software systems, DTP is a promising approach to alleviate the arising problems.

The paper is organized as follows. Section 2 introduces pure trait-based programming. Section 3 discusses the main design choices made during the development of DTP. Section 4 introduces DTP by a small case study. Section 5 illustrates how DTP supports proactive, reactive and extractive SPL development and is particularly well suited for evolving SPLs. Section 6 discusses related work. Section 7 concludes the paper by outlining some directions for further work.

2 Pure trait-based programming

Pure trait-based programming [12, 11, 7, 10, 7] aims at supporting low coupling and at maximizing the opportunities of reuse. It completely separates the competing roles of object generators and units of code reuse, and the competing roles of types and units of code reuse. Namely, trait names are not types and class-based inheritance is ruled out.

In this section, we summarize TRAITJ, a pure trait-based programming language (based on [10]), which highlights the specific characteristics of pure trait-based programming. The syntax of TRAITJ is given in Figure 1, where we use the overbar notation for sequences (as in [19]) and where the big square brackets '$\left[$' and '$\left]$' denote an optional element of the syntax. A program consists of interface declarations, trait declarations, and class declarations. The syntax of interface declarations ID, method headers H, field declarations F, method declarations M, field initializations FI and class constructors K is similar as in JAVA (without considering, e.g., visibility modifies and checked exception declarations).

ID	::=	**interface** I $\left[\textbf{extends}\ \overline{\text{I}}\right]$ { $\overline{\text{H}}$; }	*interface*
H	::=	S m ($\overline{\text{S}}$ $\bar{\text{x}}$) | **void** m ($\overline{\text{S}}$ $\bar{\text{x}}$)	*method header*
S	::=	I | **boolean** | **int** | ⋯	*type*
TD	::=	**trait** T **is** TE	*trait*
TE	::=	{$\overline{\text{TM}}$} | T | TE + TE | T[$\overline{\text{TA}}$]	*trait expression*
TM	::=	F; | H; | M	*trait member*
TA	::=	**exclude** m | m **aliasAs** m | m **renameTo** m | f **renameTo** f	*trait alteration*
F	::=	S f	*field*
M	::=	H { ⋯ }	*method*
CD	::=	**class** C **implements** $\overline{\text{I}}$ **by** TE { $\overline{\text{FI}}$; $\overline{\text{K}}$ }	*class*
FI	::=	F | F = ⋯	*field initialization*
K	::=	C($\overline{\text{S}}$ $\bar{\text{x}}$){ ⋯ }	*constructor*

Fig. 1. TRAITJ syntax (I ∈ interface names, T ∈ trait names, C ∈ class names, m ∈ method names, f ∈ field names, x ∈ variables names)

A *trait declaration* associates a name to a *trait expression*. We say that a trait declaration is: *basic* (or *flat*) if its body consists of a basic trait expression {$\overline{\text{TM}}$}, *composite* (or *non-flat*) otherwise. A *basic trait expressions* {$\overline{\text{TM}}$} declares a set of provided methods together with their required fields and required methods. *Provided methods* are the methods defined in the trait, which will be included in any class using the trait. *Required fields* and *required methods* are fields and abstract methods which are assumed to be available in any class using the trait. The provided/required methods and the required fields of a trait can be directly accessed in the body of the provided methods of the trait. For instance, the following trait declaration

```
trait T1 is { int y;  int getX();  int getY() { return this.y; }  void setY(int value) { this.y = value; }
              String toString() { return "(" + this.getX() + "," + this.getY() + ")"; } }
```

associates the name T1 to a basic trait expression which provides the methods getY, setY and toString and requires the field y and the method getX.

The type system checks that each field/method requirement declared in a basic trait is *used* by some method m provided by the basic trait (that is, the required method/field is selected on **this** in the body of m).

Traits are building blocks that can be used to compose classes and other traits by means of a suite of trait composition operations. In the following, we illustrate the semantics of trait composition by associating to each composite trait declaration a flat

trait declaration with the same semantics. This way of specifying the semantics of traits, called *flattening* [35, 17, 28], is quite common in the literature on traits.

The *symmetric sum* operation, +, merges two traits to form a new trait. The summed traits must be disjoint (i.e., they must not provide identically named methods) and consistent (i.e., required fields with the same name and required/provided methods with the same name must have the same type). For instance, given the trait declaration

```
trait T2 is { int x; int getX() { return this.x; } void setX(int value) { this.x = value; } }
```

the composite trait declaration **trait** TPoint **is** T1 + T2 is equivalent to the flat trait declaration

```
trait TPoint is { int x; int y; int getX() { return this.x; } int getY() { return this.y; }
                  void setX(int value) { this.x = value; } void setY(int value) { this.y = value; }
                  String toString() { return "(" + this.getX() + "," + this.getY() + ")"; } }
```

The operation **exclude** forms a new trait by removing a method from an existing trait. For instance, the composite trait declaration **trait** T3 **is** T1[exclude toString] is equivalent to the flat trait declaration

```
trait T3 is { int y; int getY() { return this.y; } void setY(int value) { this.y = value; } }
```

(where the method requirement **int** getX() from T1 has been automatically dropped, since getX is not used by the provided methods of T3), and the composite trait declaration **trait** T4 **is** T1[exclude getY, exclude setY] is equivalent to the flat trait declaration

```
trait T4 is { int getX(); int getY(); String toString(){return "("+this.getX()+","+this.getY()+")";} }
```

(where the field requirement **int** y from T1 has been automatically dropped, while the excluded method getY has been changed into a requirement since it is used by the provided methods of toString).

The operation **aliasAs** forms a new trait by adding a copy of an existing method with a different name. When a recursive method is aliased, the recursive invocations in the body of the new method are not renamed. For instance, given the trait declaration

```
trait T5 is
{ int x; void resetX(){if (this.x<0){this.x=-x; this.resetX();} else if (this.x>0){this.x--; this.resetX();}} }
```

the composite trait declaration **trait** T6 **is** T5[resetX aliasAs resetXaux] is equivalent to the flat trait declaration

```
trait T6 is {
  int x;
  void resetX() { if (this.x < 0){this.x=-x; this.resetX();} else if (this.x > 0){this.x--; this.resetX();} }
  void resetXaux() { if (this.x < 0){this.x=-x; this.resetX();} else if (this.x > 0){this.x--; this.resetX();} }
}
```

The operation **renameTo** creates a new trait by renaming all the occurrences of a required field name or of a required/provided method name from an existing trait. For instance, the composite trait declaration **trait** T7 **is** T1[y renameTo x, getY renameTo getX, setY renameTo setX] is equivalent to the flat trait declaration

```
trait T7 is { int x; int getX() { return this.x; } void setX(int value) { this.x = value; }
              String toString() { return "(" + this.getX() + "," + this.getX() + ")"; } }
```

Since traits do not introduce any state, a class assembled from traits has to declare and initialize the fields required by its constituent traits (non-explicitly initialized fields ar implicitly initialized, as in JAVA). For instance, the class declaration

```
interface IPoint  {  int getX();  int getX();  void setX(int value);  void setY(int value);  String toString();  }
interface IColor  {  void setColor(String name);  String toString();  }
interface IColoredPoint extends IPoint, IColor  { }
```

Listing 1: Interfaces IPoint, IColor and IColoredPoint

```
class CPoint implements IPoint by TPoint
  {  int x = 0;     int y;     CPoint() { }     CPoint(int x, int y) { this.x = x; this.y = y; }  }
```

defines a generator of objects of type IPoint (the interface IPoint is defined at the top of Listing 1) with two constructors. The class, which is built by using the trait TPoint (defined above in Section "Trait sum"), has the same semantics of the JAVA class obtained by removing the clause "**by** TPoint" from the class header and inserting in the class body the code of the methods provided by TPoint.

The following example shows the flexibility of traits. A trait TColor introduced for building a class CColor:

```
trait TColor
  { String name;    void setColor(String name){this.name = name;}    String toString(){return this.name;} }
class CColor implements IColor by TColor { String name = ""; }
```

(the interface IColor is defined in Listing 1) can be straightforwardly reused for building a class CColoredPoint:

```
trait TColoredPoint is TPoint[toString renameTo pointToString]+TColor[toString renameTo colorToString]+{
  {  String pointToString();    String colorToString();
     String toString() { return this.pointToString() + ":" + this.colorToString(); }  }
class CColoredPoint implements IColoredPoint by TColoredPoint { int x=0;   int y=0;   String name=""; }
```

(the interface IColoredPoint is defined in Listing 1) supporting the same kind of reuse provided by multiple class-based inheritance.

Pure trait-based programming targets a scenario where a trait, which was developed for a particular purpose, may later be adapted and reused in a completely different context. For instance, trait TPoint introduced for defining a point in a plane can be reused to define a counter:

```
interface ICounter {  int getValue();    void setValue(int value);    String toString();  }
trait TCountert is TPoint[exclude setY,  exclude getY,  exclude toString,
                          x renameTo n,  getX renameTo getValue,  setX renameTo setValue]  +
  {  int n;    String toString() { return n; }    void increment() { n++; }  }
class CCounter implements ICounter by TCounter { int n = 0; }
```

3 Design choices for DTP

In this section, we discuss the main design choices made during the development of DTP. We choose a *pure* trait-based programming language as the language for writing the products of the SPL because pure trait-based programming has been developed in order to maximize the opportunities of reuse. In particular, we consider TRAITJ since, in previous work [10], it has been used to directly implement SPLs.

We approached the challenge of designing a suitable notion of delta module for trait-based programs by exploring the possibility to adapt DOP delta modules. A DOP

delta module is a container of modification operations to a JAVA program. The modifications may add, remove or modify interfaces and classes. Modifying an interface means changing the super interfaces, or adding or removing methods. Modifying a class means: ***(i)*** changing the super class; ***(ii)*** adding or removing fields; and/or ***(iii)*** adding, removing or modifying methods. The method-modification operation can either replace the method body by another implementation, or wrap the existing method using the **original** construct (similar to the **Super** construct in AHEAD [5])—the call **original**$(\cdots)$ expresses a call to the method with the same name before the modifications and is bound at the time the product is generated. Since TRAITJ interfaces are literally JAVA interfaces, the DOP delta operations for adding, removing or modifying an interface can be straightforwardly adopted for defining delta modules for trait-based programs. Also the operations of adding or removing classes and traits do not pose design challenges and can be straightforwardly defined.

The main challenge is to define suitable delta operations for expressing modifications to traits. As a first attempt, we have tried to adapt to traits the class-modification operations provided by DOP (see above), thus defining delta operations for modifying the body TE of a trait definition **trait** T **is** TE by: ***(i)*** replacing the used traits (i.e., the trait names occurring in TE) by arbitrary trait expressions; ***(ii)*** adding or removing field requirements and method requirements; and/or ***(iii)*** adding, removing or modifying (possibly using the **original** construct) methods. However, through some experiments, we realized that such delta operations are quite complex to use and that the delta operation on methods (point *(iii)* above) is less flexible than the TRAITJ composition operations, which include also method/field renaming.

Thus, we realized that a flexible trait-modification operation can be expressed by replacing the body of the trait with a new trait expressions. The new trait expression may contain occurrences of the **TOriginal** keyword, which refers to the trait with the same name before the modification and is bound at the time the product is generated. In this way, a smooth integration of the modularity mechanisms provided by delta modification operations (modeling inter-product code reuse) and by trait composition operations (modeling intra-product code reuse) is achieved.

4 Delta-trait programming

In order to illustrate the main concepts of delta-trait programming, we use a case study of a simple product line of data structures for sequences, that we call the Sequences PL.

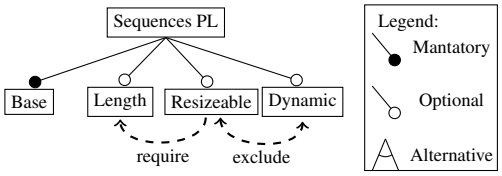

Fig. 2. Feature Model for the Sequences PL

Figure 2 depicts the feature model of the Sequences PL as a feature diagram. The Sequences PL has five products. Each product provides a stack and a queue data structure. The base product (implementing only the Base feature) provides a fixed capacity stack and queue implementing the empty/full tests and the canonical insertion/extraction operations. The other products offer additional functionalities: an operation for getting the number of existing ele-

ments in a stack/queue (feature Length), and in mutual exclusion either operations for changing the capacity of a stack/queue (feature Resizeable) or for the automatic management of the capacity (feature Dynamic). Since it would not be sensible to change the capacity of a stack/queue without knowing the number of contained elements, the feature Resizeable requires the feature Length.

The syntax of DELTATRAITJ is given in Figure 3. Delta modules are containers of modification operations to programs. In the context of trait-based programs, delta operations may add, remove or modify interfaces, traits or classes. Modifying an interface means changing the super interfaces, or adding or removing methods. Modifying a trait means replacing its body or wrapping the existing trait body by means of the **TOriginal** construct. Modifying a class means changing the implemented interfaces, or replacing/wrapping the trait expression providing the methods, or adding/removing/modifying field initializations, or adding/removing/modifying constructors. Modifying a constructor means replacing its body or wrapping the existing constructor body by means of the **COriginal** construct.

DD ::= **delta** D { $\overline{\texttt{DO}}$ } *delta module*
DO ::= IO | TO | CO *delta operation*
IO ::= ID | **remove** I | *interface operation*
modify interface I [**extends** $\overline{\text{I}}$] { $\overline{\texttt{HO}}$; }
HO ::= H | **remove** m *header operation*
TO ::= TD | **remove** T | **modify** TD *trait operaton*
CO ::= CD | **remove** C | *class operations*
modify class C [**implements** $\overline{\text{I}}$] [**by** TE] { $\overline{\texttt{FO}}$; $\overline{\texttt{KO}}$ }
FO ::= FI | **remove** f | **modify** FI *field operation*
KO ::= K | **remove** C($\overline{\text{S}}$ $\bar{\text{x}}$) | **modify** K *constructor op.*

Fig. 3. DELTATRAITJ delta modules syntax (D ∈ delta module names and ID, H, TD, TE, CD, FI, K, m, f, x are defined in Fig. 1). The body TE of the trait definition TD specified by a trait-modify operation **modify** TD may contain occurrences of the **TOriginal** keyword, which denotes the original version of the trait. The body $\{\cdots\}$ of the constructor definition K specified by a constructor-modify operation **modify** K may start with **COriginal**$(\cdots)$, which represents a call to the original version of the constructor.

A delta-trait product line (similar to a delta-oriented product line) consists of a *code base* (containing the delta modules) and a *product line declaration*. The code in Listings 2 and 3 is a code base for the Sequences PL. The product line declaration creates the connection to the product line variability specified in terms of product features. Listing 4 shows a product line declaration for the Sequences PL. The product line declaration: ***(i)*** Lists the product features. ***(ii)*** Describes the set of *valid* feature configurations. In the examples, the valid feature configurations are represented by a propositional formula over the set of features. We refer to [4] for a discussion on other possible representations. ***(iii)*** Describes the possible application orders of the delta modules by defining a total order on the sets of a partition of the delta modules. Delta modules in the same set can be applied in any order, while the order of the sets must be respected. The ordering allows the programmer to enforce semantic requires-relations that are necessary for the applicability of the delta modules. In Listing 4, the ordering is represented by writing an ordered list of the delta module sets after the keyword **deltas** $\{\ldots\}$. ***(iv)*** A delta module name can have an application condition to evaluate for which feature configurations the

delta module has to be included in the code of the corresponding product. In Listing 4, the application condition is represented by a propositional constraint over the set of features, given by **when** clauses. Since only feature configurations that are valid according to the feature model are used for product generation, the application conditions are understood as a conjunction with the formula describing the set of valid feature configurations. In Listing 4, the delta modules DBase, DLength, DResizeable and DDynamic are associated to the features Base, Length, Resizeable and Dynamic, respectively. Moreover, in order to realize the feature Resizeable, both the delta modules DResizeable and DResisazableOrDynamic must be applied. In order to realize the feature Dynamic, both the delta modules DResizeableOrDynamic and DDynamic must be applied.

A product is *valid* if it corresponds to a valid feature configuration. The generation of a product for a given feature configuration consists of two steps (that can be performed automatically): ***(i)*** find the selected delta modules (that is, the delta modules with a satisfied application condition); and ***(ii)*** apply them to the empty program in any linear ordering that respects the total order on the partition of the delta modules. A delta module is applicable to a program if: ***(i)*** each interface/trait/class to be added does not exist; ***(ii)*** each interface/trait/class to be removed or modified exists; ***(iii)*** each interface-modify operation is such that each method to be removed exists and each method to be added does not exist; and ***(iv)*** each class-modify operation is such that each field to be removed exists and each field to be added does not exist. During the product generation, the selected delta modules must be applicable in the given order (otherwise the product generation fails). In particular, the first delta module (which is applied to the empty product) must contain only additions. I.e., its body must be a TRAITJ program.

Listing 2 illustrates the DBase delta module that, when applied to the empty product, generates the product with the feature Base. Applying the delta module DBase is mandatory for all feature configurations. It creates the classes for the basic data structures Stack and Queue. The functionality of the data structures are described in the interfaces IStack and IQueue with the methods pop, push and enqueue, dequeue. Both interfaces are extending the interface ISequence which describes the functionality to check the status of a sequence. The trait TSequence implements the functionality of the interface ISequence with an array of objects (for storing the elements of the sequence) and an integer field (for storing the number of elements currently in the sequence). The traits TStack and TQueue implement the interfaces IStack and IQueue and extend the trait TSequence. The data structures are instantiated in the two classes CStack and CQueue which use the implementation of the according traits and the description of the interfaces.

The delta module DLength for the Length feature (in Listing 3) modifies the interface ISequence to add the method getLength and adds the implementation of the method to the trait TSequence. The depending traits and classes will be automatically updated by these modifications. The delta module DResizeableOrDynamic (Listing 3) contains the commonality of the related features Resizeable and Dynamic. It implements the method resize which can be used to increase the capacity of the data structures. The delta module DResizeable for the Resizeable feature (Listing 3) extends the interface ISequence with the two methods resize and getCapacity. In this module only the getCapacity method is implemented which returns the capacity of the data structure. The delta module DDynamic implements the Dynamic feature (Listing 3) which automatically coordinates the

```
delta DBase {

interface ISequence {
  boolean isEmpty();
  boolean isFull();
}

interface IStack extends ISequence {
  void push(Object e);
  Object pop();
}

interface IQueue extends ISequence {
  void enqueue(Object e);
  Object dequeue();
}

trait TSequence is {
  int length;
  Object[] elements;
  boolean isEmpty() { return (this.length == 0); }
  boolean isFull()
    { return (this.length == this.elements.length); }
}

trait TStack is TSequence + {
  int length;
  Object[] elements;
  boolean isFull(); //required Methods
  boolean isEmpty(); //required Methods
  void push(Object o) {
    if (this.isFull()) throw new IllegalStateException();
    this.elements[this.length] = o;
    this.length++;
  }
  Object pop() {
    if (this.isEmpty()) throw new IllegalStateException();
    this.length--;
    Object o = this.elements[this.length];
    this.elements[this.length] = null;
    return o;
  }
}

trait TQueue is TSequence + {
  int length;
  Object[] elements;
  int first;
  boolean isFull(); //required Methods
  boolean isEmpty(); //required Methods
  void enqueue(Object o) {
    if (this.isFull()) throw new IllegalStateException();
    this.elements[(this.first + this.length)
                  % this.elements.length] = o;
    this.length++;
  }
  Object dequeue() {
    if (this.length == 0)
      throw new IllegalStateException();
    this.length--;
    Object o = this.elements[this.first];
    this.first=(this.first + 1) % this.elements.length;
    return o;
  }
}


class CStack implements IStack by TStack {
  Object[] elements;
  int length = 0;
  CStack(int capacity)
    { this.elements  = new Object[capacity]; }
}

class CQueue implements IQueue by TQueue {
   Object[] elements;
   int length = 0;
   int first = 0;
   CQueue(int capacity)
     { this.elements  = new Object[capacity]; }
}


} // end of DBase
```

Listing 2: Delta module DBase

size of the data structures. When the feature Dynamic is selected, the operation for testing whether a stack/queue is full (that is present in all the other products, including the base product) is not present. Therefore, the delta module DDynamic removes the method isFull from the interface ISequence. Then it creates a new trait TDynamic which implements new methods for the insertion and extraction of objects in the data structure. The traits for the stack and the queue are then combined with the trait TDynamic in which the original insertion and extraction methods are renamed to fit the required methods of the trait TDynamic. The methods of trait TDynamic are renamed to fit the description from the interfaces IStack or IQueue. Additionally, the classes for the stack and queue data structure are extended with a new field for the minimal capacity of the data structures. In order to realize the feature Resizeable both delta modules DResisazableOrDynamic and DResizeable must be applied, and in order to realize the feature Dynamic both the delta modules DDynamic and DResizeableOrDynamic must be applied.

```
delta DLength {
 modify interface ISequence { int getLength(); }
 modify trait TSequence is TOriginal + {
   int length;
   int getLength() { return this.length; }
 }
}

delta DResizeableOrDynamic {
 trait TResize is {
   Object[] elements;
   int length;
   void resizeAndCopy(int newCapacity, int from) {
     if (this.length > newCapacity)
       throw new IllegalStateException();
     Object[] newElements = new Object[newCapacity];
     for (int i = 0; i <= this.length - 1; i++) {
       newElements[i] = this.elements[(from + i)
                        % this.elements.length];
     }
     this.elements = newElements;
   }
 }
 modify trait TStack is TOriginal + TResize + {
   void resizeAndCopy(int newCapacity, int from);
   void resize(int newCapacity)
     { this.resizeAndCopy(newCapacity,0); }
 }
 modify trait TQueue is TOriginal + TResize + {
   int first;
   void resizeAndCopy(int newCapacity, int from);
   void resize(int newCapacity) {
     this.resizeAndCopy(newCapacity,this.first);
     this.first = 0;
   }
 }
}

delta DResizeable {
 modify interface ISequence {
   void resize(int newCapacity);
   int getCapacity();
 }
 modify trait TSequence is TOriginal + {
   int getCapacity() { return this.elements.length; }
 }
}

delta DDynamic {
 modify interface ISequence{remove boolean isFull();}
 trait TDynamic is {
   Object[] elements;
   int length;
   int minCapacity;
   boolean isFull();
   void resize(int cap);
   void originalInsert(Object o);
   Object originalExtract();
   void insert(Object o) {
     if (this.isFull()) {
       resize(this.elements.length * 2);
     }
     this.originalInsert(o);
   }
   Object extract() {
     if ((this.length <= this.elements.length / 2)
         && (this.elements.length!=this.minCapacity)){
       resize(this.elements.length / 2);
     }
     return this.originalExtract();
   }
 }
 modify trait TStack is
   TOriginal[push renameTo originalInsert,
             pop renameTo originalExtract] +
   TDynamic[insert renameTo push,
            extract renameTo pop]
 modify trait TQueue is
   TOriginal[enqueue renameTo originalInsert,
             dequeue renameTo originalExtract] +
   TDynamic[insert renameTo enqueue,
            extract renameTo dequeue]
 modify class CStack {
   int minCapacity;
   CStack(int capacity) {
     COriginal(capacity);
     this.minCapacity = capacity;
   }
 }
 modify class CQueue {
   int minCapacity;
   CQueue(int capacity) {
     COriginal(capacity);
     this.minCapacity = capacity;
   }
 }
}
```

Listing 3: Delta modules DLength, DResizeableOrDynamic, DResizeable and DDynamic

```
features Base, Length, Resizeable, Dynamic configurations Base
& (Resizeable -> Length) & (!(Resizeable & Dynamic)) deltas
  { DBase }
  { DLength when Length }
  { DResizeableOrDynamic when (Resizeable | Dynamic)}
  { DResizeable when Resizeable }
  { DDynamic when Dynamic }
```

Listing 4: Declaration of the Sequences PL

5 Development and evolution of delta-trait product lines

As an example of unanticipated SPL evolution, consider the case of evolving the Sequences PL by adding three products that additionally contain the feature Peekable. This feature creates a product that, in addition to the classes CStack and CQueue, contains the classes CPeekableStack and CPeekableQueue. These new classes provide a method Object peek(int i) for returning the value of the i-th element of a sequence. Figure 4 depicts the feature model of the evolved Sequence PL. In order to be able to safely peek the elements of a sequence, it is useful to know the length of the sequence. Therefore, the feature Peekable requires feature Length.

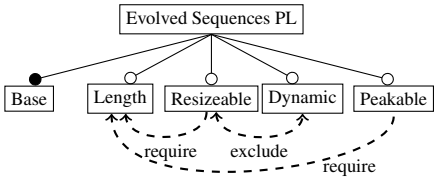

Fig. 4. Feature model for the evolved Sequences PL

The declaration of the evolved Sequences PL is shown in Listing 5. The delta module DPeekable for implementing the feature Peekable is shown in Listing 6. In the application order, the delta module DPeekable is included in the first set of the partition (together with the DBase delta module). Indeed, DPeekable can be safely moved to any other set of the partition, since it does not modify or remove existing interfaces/traits/classes and, thus, it does not interfere with the other delta modules.

The delta module DPeekable adds three new interfaces (IPeekableSequence, IPeekableStack and IPeekableQueue), three new traits (TPeekable, TPeekableStack and TPeekableQueue), and two new classes (CPeekableStack and CPeekableQueue). The last two interfaces IPeekableStack and IPeekableQueue extend the first interface by adding the methods to add and remove an element of the respective data structure. The methods for removing an element (pop and dequeue) no longer return that element. The trait TPeekable provides the implementation of the method peek of the IPeekableSequence interface. The new traits for the peekable data structures are based on the trait TPeekable and the corresponding trait from Listing 2 (TStack and TQueue, respectively)—this straightforward intra-product code reuse would not be possible in DOP, which relies on class-based inheritance for intra-product code reuse. Note that, in the products with feature Peekable, the classes CPeekableStack and CPeekableQueue (which use the trait TPeekableStack and TPeekableQueue, respectively) coexists with the classes CStack and CQueue (which use the trait TStack and TQueue, respectively).

```
features Base, Length, Resizeable, Dynamic, Peekable
configurations Base & (Resizeable -> Length) & (Peekable -> Length) & (!(Resizeable & Dynamic))
deltas
  { DBase}
  { DLength when Length, DPeekable when Peekable }
  { DResizeableOrDynamic when (Resizeable | Dynamic)}
  { DResizeable when Resizeable }
  { DDynamic when Dynamic }
```

Listing 5: Declaration of the evolved Sequences PL

```
delta DPeekable {
 interface IPeekableSequence extends ISequence {
   Object peek(int i);
 }
 interface IPeekableStack extends IPeekableSequence {
   void push(Object e);    void pop();
 }
 interface IPeekableQueue extends IPeekableSequence {
   void enqueue(Object e);    void dequeue();
 }
 trait TPeekable is {
   int length;    Object[] elements;    int first;
   Object peek(int i) {
     if (i >= this.length)
        throw new IllegalArgumentException();
     return this.elements[(this.first + i)
                          % this.elements.length];
   }
 }
 trait TPeekableStack is
   TStack[pop renameTo topPop] +
   TPeekable + { void pop() { topPop(); }
 }

 trait TPeekableQueue is
       TQueue[dequeue renameTo frontDequeue] +
       TPeekable + {
   void dequeue() { frontDequeue(); }
 }
 class CPeekableStack implements IPeekableStack
                    by TPeekableStack {
   Object[] elements;
   int length = 0;
   int first = 0;
   CPeekableStack(int capacity)
     { this.elements  = new Object[capacity]; }
 }
 class CPeekableQueue implements IPeekableQueue
                             by TPeekableQueue {
    Object[] elements;
    int length = 0;
    int first = 0;
    CPeekableQueue(int capacity)
      { this.elements  = new Object[capacity]; }
 }
} // end of DPeakable
```

Listing 6: Delta module `DPeekable`

As we can see in this example, DTP seems well suited for evolving SPLs, since the developer is allowed to flexibly reuse already existing code both within single products and across different products. Proactive SPL development [23] prescribes to analyze beforehand the set of products to be supported and to plan and develop in advance all reusable artifacts. The Sequence PL case study presented in Section 4 can be seen as an example of proactive product line development, since the feature model defining the scope of the product line is first introduced and then the delta modules and the product line declaration for implementing the products are developed. When applying proactive development, a high upfront investment is required to define the scope of the of the product line and to develop reusable artifacts. Therefore, in order to reduce the adoption barrier Krueger [23] proposed reactive and extractive SPL development. In reactive SPL development, an initial product line that comprises only a basic set of products is created. Then, the initial SPL is evolved in order to deal with changing requirements. The evolved Sequence PL case study presented above can be seen as an example of reactive product line development.

In extractive SPL development, the engineering process starts with a set of existing legacy application that are turned into a product line. For instance, a product line described by the feature diagram in Fig. 4 could be developed from 5 legacy products corresponding to the feature configurations {Base, Length, Resizeable}, {Base, Dynamic}, {Base, Length, Peekable}, {Base, Length, Resizeable, Peekable}, and {Base, Length, Dynamic, Peekable}. Traits have been designed for factoring common methods of a set classes. In [6], a tool is presented for identifying the methods in a JAVA class hierarchy that could be good candidates to be refactored in traits. The tool is an adaptation of the SMALLTALK analysis tool of [24] to a JAVA setting. Since DTP smoothly

integrates delta modules and traits mechanisms, it represents a promising approach for extractive SPL development starting from a set of legacy JAVA applications.

6 Related Work

Schaefer et. al. [34] mention three approaches to support variability and code reuse on the implementation level. The first is the annotative approach which marks the source code in relation to the features of the product line. A prominent instances are conditional compilation, frames [3] and CIDE [21]. The second is the compositional approach where product implementations are built by composing code fragments. Prominent examples of the compositional approach are traits [35, 17], FOP [5, 2, 16, 1] and aspect-oriented programming (AOP) [22]—see also the evaluation presented in [26]. Transformational implementation techniques constitute the third approach which can be considered as an extension of the second and offer more flexible, modular implementation possibilities. Delta-oriented programming [31, 33, 32, 9] is an instance of transformational programming. In this paper, we presented DTP, a novel approach to implement SPL variability by smoothly integrating the modularization mechanisms provided by delta modules and traits, which overcomes some of the limitations of DOP w.r.t. intra-product code reuse.

A comparison of DOP and FOP can be found in [33], and a comparison of DOP and AOP can be found in [9]. Some related work on traits has been quoted in Sect.s 1-2. Recently [10], we have investigated the use of pure trait-based programming to directly implement SPLs. The main difference between the trait-only approach and DTP/DOP/-FOP is that, in the former: ***(i)*** the artifact base consists of a well-formed program consisting of the interfaces, traits and classes of all the products; and ***(ii)*** in order to generate a product is enough to select a subset of these artifacts. In [10], it is shown that the trait composition operations provided by TRAITJ are not enough in order to flexibly modeling inter-product variability alone. To overcome this limitation, a trait parameterization mechanism is proposed. A parametric trait is a trait parameterized by interface names and class names. It can be applied to interface names and class names to generate traits that can be composed to build other (possibly parametric) traits or classes. The trait-only approach looks appealing because the code base has a simpler structure and product generation is straightforward (cf. points (i) and (ii) above). But, the modeling of inter-product variability (which relies both on trait parameterization and on trait composition operations) might be less evident. In the trait-only approach the sole mechanism for reusing interface definition code is interface extension, which is less flexible than the interface-modify operation supported by DTP and DOP. In DTP, inter-product variability is modeled by delta-modules. The trait parameterization mechanism in the underlying trait language would provide additional flexibility.

7 Conclusion

We presented DTP, a novel approach to implement SPL variability by smoothly integrating the modularization mechanisms provided by delta modules and traits, and realized it in the programming language DELTATRAITJ (which is currently under development)

and compared it with DOP by case studies. DTP overcomes some of the limitations of DOP w.r.t. intra-product code reuse and represents a flexible approach for implementing evolving SPLs (cf. Sect. 5). As unanticipated SPL evolution scenarios become more common in long-living software systems, DTP is a promising approach for decreasing maintenance and development effort in the life cycle of these software systems on the implementation level supporting evolution at coarser levels of abstraction, e.g., the architecture level. In previous work, we have developed type systems for trait-based languages [12, 10, 8] and for DOP [9]. We are currently developing a type system for DTP. We have also developed compositional proof systems for the verification of pure traits [14] and for the verification of DOP SPLs of JAVA programs [18, 15]. In future work, we would like to investigate compositional proof systems for DTP.

References

1. S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.
2. S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proc. of GPCE 2008*, pages 101–112. ACM, 2008.
3. P. G. Bassett. *Framing software reuse: lessons from the real world*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
4. D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. of SPLC 2005*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
5. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proc. of ICSE 2003*, pages 187–197. IEEE, 2003.
6. L. Bettini, V. Bono, and M. Naddeo. A trait based re-engineering technique for Java hierarchies. In *Proc. of PPPJ*, pages 149–158. ACM, 2008.
7. L. Bettini and F. Damiani. Pure trait-based programming on the java platform. In *Proc. of PPPJ 2013*, pages 67–78, New York, NY, USA, 2013. ACM.
8. L. Bettini, F. Damiani, K. Geilmann, and J. Schäfer. Combining traits with boxes and ownership types in a Java-like setting. *Science of Computer Programming*, 78(2):218–247, 2013.
9. L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.
10. L. Bettini, F. Damiani, and I. Schaefer. Implementing type-safe software product lines using parametric traits. *Science of Computer Programming*, 2013. http://dx.doi.org/10.1016/j.scico.2013.07.016.
11. L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. TraitRecordJ: A programming language with traits and records. *Science of Computer Programming*, 78(5):521–541, 2013.
12. V. Bono, F. Damiani, and E. Giachino. On Traits and Types in a Java-like setting. In *Proc. of TCS (Track B)*, volume 273 of *IFIP*, pages 367–382. Springer, 2008.
13. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.
14. F. Damiani, J. Dovland, E. B. Johnsen, and I. Schaefer. Verifying traits: an incremental proof system for fine-grained reuse. *Formal Aspects of Computing*, 2013. http://dx.doi.org/10.1007/s00165-013-0278-3.
15. F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, and I. C. Yu. A transformational proof system for delta-oriented programming. In *Proc. of SPLC - Volume 2*, pages 53–60. ACM, 2012.

16. B. Delaware, W. R. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *Proc. of FOAL*, pages 31–35. ACM, 2009.
17. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.
18. R. Hähnle and I. Schaefer. A Liskov Principle for Delta-Oriented Programming. In *Proc. of ISoLA (1)*, volume 7609 of *LNCS*, pages 32–46. Springer, 2012.
19. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
20. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon Software Engineering Institute, 1990.
21. C. Kastner and S. Apel. Type-checking software product lines - a formal approach. In *Proc. of ASE 2008*, pages 258–267. IEEE, 2008.
22. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *Proc. of ECOOP 2001*, volume 2072 of *LNCS*, pages 327–354. Springer, 2001.
23. C. Krueger. Eliminating the Adoption Barrier. *IEEE Software*, 19(4):29–31, 2002.
24. A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *Proc. ASE 2005*, pages 66–75. IEEE Computer Society, 2005.
25. L. Liquori and A. Spiwack. Extending feathertrait java with interfaces. *Theor. Comput. Sci.*, 398(1-3):243–260, 2008.
26. R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. In *Proc of. ECOOP 2005*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.
27. L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *Proc. of ECOOP '98*, number 1445 in LNCS, pages 355–383. Springer, 1998.
28. O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *JOT (www.jot.fm)*, 5(4):129–148, 2006.
29. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.
30. J. Reppy and A. Turon. Metaprogramming with traits. In *Proc. of ECOOP 2007*, volume 4609 of *LNCS*, pages 373–398. Springer, 2007.
31. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *Proc. of SPLC 2010*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010.
32. I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking of delta-oriented programming. In *Proc. of AOSD 2011*, pages 43–56. ACM, 2011.
33. I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In *Proc. of FOSD 2010*, pages 49–56. ACM, 2010.
34. I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.
35. N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proc. of ECOOP 2003*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.
36. S. Schuster. Design Patterns in Feature-Oriented Programming. Bachelor's thesis, TU Braunschweig, 2012.
37. S. Schuster and S. Schulze. Object-oriented design in feature-oriented programming. In *Proc. of FOSD 2012*, pages 25–28. ACM, 2012.
38. C. Smith and S. Drossopoulou. *Chai*: Traits for Java-like languages. In *Proc. of ECOOP 2005*, volume 3586 of *LNCS*, pages 453–478. Springer, 2005.