A Programming Model for Adaptable Java Applications

Marco Autili  Paolo Di Benedetto  Paola Inverardi
Dipartimento di Informatica, Università dell’Aquila, Via Vetoio, L’Aquila, Italy
{marco.autili,paolo.dibenedetto,paola.inverardi}@univaq.it

Abstract
Adaptable applications are becoming the main drivers of the pervasive computing paradigm. The development and the context-aware execution of such applications on mobile devices, often characterized by their heterogeneity and limitedness, is a big challenge and it is far to be solved. The main difficulty is to provide (i) an easy-to-use and powerful programming technique for developers to actually code adaptable applications, and (ii) a context-aware run-time support to properly handle contextual situations. This paper presents a programming model that provides developers with a set of agile and user-friendly extensions to Java for easily specifying generic code in a flexible and declarative way. The generic code specifies both the invariant semantics and the degree of variability of the application. Variability is expressed in terms of adaptable classes that declare adaptable methods, and alternative classes that define them. Then, an ad-hoc preprocessor resolves variability by generating standard Java methods within standard Java classes that, opportunely combined with the core code, make-up different application alternatives, i.e., standard Java applications that represent different ways of implementing an adaptable application specification. The programming model and the generic code preprocessor have been fully implemented in and for Java as Eclipse plugins. They are part of CHAMELEON, a framework that provides both an integrated development environment and a proper context-aware run-time support to adaptable Java applications for limited devices.

Categories and Subject Descriptors D.2.3 [Coding Tools and Techniques]: Object-oriented programming; D.3.3 [Language Constructs and Features]: Classes and objects

General Terms Adaptable Applications, Object-Oriented Programming

Keywords Programming Model, Adaptable Java Applications, Context-awareness

1. Introduction
During the last decade, context-awareness and adaptation have been receiving significant attention in many research areas [5][6][14][25]. The need for adapting software applications becomes obvious for new development paradigms, such as mobile and pervasive computing, when dealing with context variations. The main drivers of the pervasive computing paradigm are then context-aware and adaptable software applications for resource-constrained mobile devices, often characterized by their heterogeneity and limitedness. In our setting, context awareness refers to the capability of the run-time environment of being aware of information that can be used to characterize the situation of the systems it runs (e.g., networks, user needs, resources, device status). Adaptability refers to systems whose behaviour, according to defined behavioural variations, can be adjusted in response to the perception of the situation.

The development and the context-aware execution of adaptable applications is a big challenge for the research community and it is far to be solved. The main difficulty is to provide (i) an easy-to-use and powerful programming technique for developers to actually code adaptable applications, and (ii) a context-aware run-time support to properly handle contextual situations. As discussed in Section 7 in the literature many valuable approaches have been proposed to this purpose [7][11][13][16][21][24] (just to cite some). In particular, the work in [14] points out how the little support that current mainstream programming languages and runtime environments provide for specifying adaptation and for supporting context awareness, respectively, leads to a system design more complex and convoluted than needed. In this work the authors deeply discuss the need for a Context-oriented Programming (COP) approach that support context-dependent behavioural variations. The claim is that COP brings a degree of dynamicity to the notion of behavioural variations that can be compared to the dynamicity that object-oriented programming brought to ad-hoc polymorphism (i.e., overloading). For instance, some object-oriented programming languages, such as Java, brought subtyping polymorphism (sometimes referred to as dynamic polymorphism) using subclasseing (also known as inheritance).

Within this line of research, our goal is to propose a COP approach to the development of context-adaptable applications, which is closer to the common practice of software applications coding. To this aim, in this paper we report our research results by presenting a programming model for adaptable applications. The programming model provides developers with a set of agile and user-friendly extensions to Java for easily specifying generic code in a flexible and declarative way, while being close to the Java programming language. The generic code consists of two parts, the core code and the adaptable code, and allows for specifying both the invariant semantics and the degree of variability of the application, respectively. Concretely, the adaptable code provides developers with extended Java constructs to specify variability in terms of adaptable classes that define adaptable methods, and alternative classes that define them. Then, following the Generative Programming style [8], an ad-hoc preprocessor resolves variability by generating standard Java methods within standard Java classes, opportunely combined, make-up different application alternatives, i.e., standard Java applications that represent different ways of implementing an adaptable application specification.
The programming model and the generic code preprocessor have been fully implemented in and for Java as Eclipse plugins. They are part of CHAMELEON, a framework that provides both an integrated development environment and a proper context-aware run-time support to adaptable Java applications for limited devices \cite{1,4,17}. As it will be clear later, the declarative and generative nature of the programming model spares developers the writing of the adaptation logic by using conditionals to explicitly and a priori treat contextual situations, and all the possible associations with behavioural variations.

The structure of the paper is as follows: Section 2 sets the context by briefly introducing the CHAMELEON framework. The syntactic aspect of our programming model is described in Section 3 by using the extended Java grammar. The policy for specifying behavioural variations in term alternative classes is discussed in \ref{4}. The preprocessing algorithm is described in Section 5 and an adaptable application example is presented in Section 6. Section 7 gives related work, and Section 8 concludes the paper and discusses future directions.

2. Setting the context

CHAMELEON is a framework \cite{1,4,17} for to develop and deploy adaptable Java applications. The CHAMELEON framework architecture is shown in Figure 1.

![Figure 1. CHAMELEON Framework](image)

- **Programming Model.** Referring to Figure 1 the Development Environment (DE) is based on the Programming Model presented in Section 5. The applications’ code is a generic code that consists of two parts: the core and the adaptable code - see in Figure 1 the screen-shoot of the DE implemented as an Eclipse plugin \cite{1}. The generic code is preprocessed by the CHAMELEON Pre-processor (1), also part of the DE, and a set of different standard Java application alternatives is derived and stored into the Application Registry (2).

- **Resource and SLS Models.** The whole framework is built around the Resource Model and SLS Model. The former is a formal model that allows for characterizing the application alternatives with respect to the resources they need to be executed. The latter is a model that permits developers to attach non-functional information at generic code level, and users to specify the required Service Level Specification (SLS), i.e., Qualities of Service (QoS). For a detailed description of the resource and SLS models please refer to \cite{1,4,17}.

  - **Chameleon Server-side.** Still referring to Figure 1 the Abstract Resource Analyzer (ARA - see \cite{1,17} for details) is an interpreter that, abstracting a standard JVM, is able to analyse the application alternatives (3), and derive their Offered SLSs (5.a) and Resource Demands (5.b). ARA is parametric with respect to the characteristics of the execution environment as described through the Resource Consumption Profile sent by the device (4.a). This profile provides a characterization of the target execution environment, in terms of the impact that Java bytecode instructions have on the resources.

  - **Chameleon Client-side.** CHAMELEON-enabled devices (see left-hand side of Figure 1) are devices deploying and running the CHAMELEON client component that is able to retrieve contextual information basing on a defined context-model. A CHAMELEON-enabled device provides a declarative description of the execution context in terms of the resources it supplies (i.e., Resource Supply) and the resource consumption profile.

  - **Customizer.** On the server-side, the resource demands (5.b) – resp., offered SLS (5.a) – of the application alternatives together with the resource supply sent by the device (4.b) – resp., requested SLS (4.c) – are used by the Customizer. The latter, basing on a notion of compatibility and goodness, is able to relatively compare and choose (6.a) the set of “best” suited alternatives, and deliver (6.b) one of them that (via the Over-The-Air (OTA) provisioning technique \cite{20}) can be automatically deployed in the target device for execution. Compatibility is used to decide if an alternative can run safely on a certain device and if its offered SLSs satisfy the requested SLSs. Thus, for instance, a resource supply is compatible to a resource demand if for every resource demanded by the alternative, a sufficient amount is supplied by the execution environment. Roughly speaking, an alternative is considered to be better w.r.t. another one if it is “less resources consuming” and “more SLSs satisfying”.

It is worth to mention that, the development of adaptable applications is carried out as a standard software engineering activity, and is independent from the other “tasks”. The client-side component is a very light-weight component and its tasks can be suitably executed on limited devices. On the contrary, the server-side components have a high computational complexity but they are executed on powerful server machines that (usually) does not suffer resource limitations. At present CHAMELEON and, hence, the programming model are used for context-dependent run-time deployment of adaptable Java applications. This makes the current implementation of the CHAMELEON framework particularly suitable for automatically delivering adaptable applications to limited devices. However, as better argued in Section 8 other object-oriented languages are eligible and other forms of adaptation can be supported. Interested readers can download all the CHAMELEON modules from \cite{1}.

3. Programming Model

This section presents the programming model by focussing on the extensions to the Java language and, hence, on the syntactic aspect of the adaptable code (as part of the generic code).
The behavioural variations of an adaptable application are specified up to the smallest unit of (object-oriented) behavioural description, i.e., up to method definition. Our programming model allows for the specification of how to adapt the application by providing a flexible and declarative way to describe which methods can be adapted, and which are the allowed adaptation alternatives. The approach is that of enriching the standard Java syntax with new constructs that allow the programmer to clearly specify behavioural variations. In the following, we introduce basic concepts underpinning our programming model:

- An **adaptable method** is a method that can be adapted, i.e., its behaviour can vary according to the context condition. It defines the entry-point for a behaviour that can be adapted.
- An **adaptable class** is a class that declares one or more adaptable methods.
- An **alternative class** adapts an adaptable class. It contains the definition of how one or more adaptable methods can be actually adapted. It implements one of the possible actual behaviours that can be chosen when adapting a given adaptable method. An adaptable class can be adapted by one or more alternative classes.
- An **adaptable application** is an application that contains one or more adaptable classes.
- An **adaptation policy** is the set of all the adaptable classes, and their alternative classes, defined within the adaptable code. In other words, the adaptation policy determines the behavioural variations by defining which methods can be adapted and how.
- **Tags** can be used to specify conflicts among adaptable method implementations that cannot coexist in the standard Java classes derived by the preprocessing phase.

In the following, we denote the implementation of a method \( m \) provided by an adaptation alternative \( A \) as \( A.m \), and we refer to a class generated by the preprocessing phase as **derived class**.

### 3.1 Adaptable classes

The declaration of an adaptable class is similar to that of a standard Java class prefixed with the new modifier keyword **adaptable**:

\[
\text{AdaptableClassDeclaration ::= adaptable ClassDeclaration}
\]

**ClassDeclaration** is a standard Java class declaration extended for allowing also the declaration of adaptable methods.

### 3.2 Adaptable methods

An adaptable method is declared in a way similar to a standard Java method having the keyword **adaptable** as prefix:

\[
\text{AdaptableMethodDeclaration ::= adaptable MethodHeader;}
\]

**MethodHeader** represents the method modifiers, the return type and the signature. An adaptable method must be declared within an adaptable class. Similarly to Java abstract methods, adaptable methods are not provided with their implementations when declared. In fact, the actual implementations are provided by alternative classes.

### 3.3 Alternative classes

The new keywords **alternative** and **adapts** (see Figure 3) are introduced to declare an alternative class and to specify the class it adapts, respectively:

\[
\text{AdaptationAlternativeDeclaration ::= alternative class Identifier adapts Identifier ClassBodyDeclaration}
\]

Before diving into details, in Figure 2 we present a glimpse of how the new constructs introduced by the programming model look like. Figure 2 shows an adaptable class **HelloWorld** declaring the adaptable method `main(String[] args)`, and an adaptation policy specifying the alternative classes `Alt1` and `Alt2`, which define different behaviours for the method `main`. After the preprocessing phase, this very simple adaptation policy will produce two different **HelloWorld** standard Java classes: one having the implementation of `main(String[] args)` provided by the alternative class `Alt1`; the other one having the implementation provided by `Alt2`. These two classes represent two different application alternatives of the “HelloWorld” application: the first one simply displays text; the second one displays “Hello World” as an animated gif image. Obviously, the latter provides a better result but requires a graphic display and a supporting graphic library.

```java
adaptable public class HelloWorld {
    adaptable public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
adaptable class Alt1 adapts HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
adaptable class Alt2 adapts HelloWorld {
    public static void main(String[] args) {
        Display.showAnimation("Hello World");
    }
}
```

**Figure 2.** A glimpse of the programming model constructs

Specifically, through the **adapts** keyword, the second **Identifier** specifies either an adaptable class or another alternative class that, in turn, has been declared as adaptable (see below). The main purpose of an alternative class is to provide the (standard Java) implementation of (a subset of) methods declared as adaptable in the class specified through the **adapts** keyword. The modifiers, the return type and the signature of such methods, must be exactly the same as the ones specified by the adaptable method declaration. Thus, **ClassBodyDeclaration** is an extended standard
Java class body declaration, that permits not only to define standard Java methods and variables (see, for instance, the method m2 in Figure 3), but also to (re-)declare adaptable methods.

Still referring to Figure 3, the alternative A4 defines both the adaptable methods m1 and m2 of C. This allows the programmer to establish a sort of dependency relationship between method adaptations. In fact, when two or more adaptable methods are defined in the same alternative class, the choice of that alternative for adapting one of the methods forces the selection of all the other methods. Thus, if the alternative A4 is chosen for adapting the method m1, then the method A4.m2 is implicitly coupled with it. Moreover, the alternative A1 implements the adaptable method m1() and the standard Java method m2(). Again, this implicitly defines a dependency between A1.m2() and A1.m1(). The same holds for variables declared within alternative classes. Starting from the above considerations, we can state the following general principle for our programming model: an adaptation alternative, must be considered as a whole, i.e., the choice of an alternative A for adapting a method, implies the choice of all the other members of A.

### 3.3.1 Adaptable alternative classes

An alternative class can declare additional adaptable methods. In this case, the alternative must be declared as adaptable itself. This implies that an alternative class can adapt another alternative class.

```java
adaptable alternative class B1 adapts C {
    void m1() { ... }
    adaptable void m3();
}
adaptable alternative class B2 adapts B1 {
    void m3() { ... }
}
adaptable alternative class B3 adapts B1 {
    void m3() { ... }
}
```

**Figure 4.** Alternative adapting another adaptation alternative

Figure 4 shows an example of an adaptable alternative B1 that adapts the class C of Figure 3. B1 declares the additional adaptable method m3, and the alternatives B2 and B3 define two different implementations for it.

An alternative can also explicitly redefine as adaptable one or more adaptable methods. Also in this case, the alternative itself must be declared as adaptable. Figure 5 shows an example of another adaptable alternative D1 for the adaptable class C. The alternative D1 redeclare as adaptable the method m2(), whose implementation is then provided by D2 and D3.

```java
adaptable alternative class D1 adapts C {
    adaptable void m2();
    void m1() { ... }
}
adaptable alternative class D2 adapts D1 {
    void m2() { ... }
}
adaptable alternative class D3 adapts D1 {
    void m2() { ... }
}
```

**Figure 5.** Alternative redeclaring adaptable an adaptable method

The possibility of having adaptable alternatives, creates a hierarchy between alternatives that defines further dependency relationships between method adaptations. In particular, the implementations of a method redeclared as adaptable by an alternative A, must be necessarily coupled with the implementations of the other (adaptable or standard) methods provided by A together with its variables. For example, referring to Figure 3, D2.m2() and D3.m2(), must be necessarily coupled with D1.m1(). This implies that the preprocessing phase will never generate a derived class for the adaptable class C containing, for example, D2.m2() and A1.m1().

Note that, the programming model permits an alternative to implement only the adaptable methods that have been declared as adaptable by the class/alternative it directly adapts. For example, in Figure 4, B2 cannot provide an implementation for the adaptable method m2() declared by C. Moreover, as formalized in Section 4.3 (see Constraint 2), an alternative cannot redefine a non adaptable method which has been already defined in the adapted class/alternative.

### 3.4 Method tags

Within (adaptable) alternative classes, the definitions of adaptable methods can be enriched by tag specifications. Tags are simple (comma separated) labels prefixed to a method definition that express conflicts among method implementations that cannot coexist in the standard Java classes derived by the preprocessing phase:

```
tag (comma separated tag identifiers list) method definition
```

The tag mechanism is simple and intuitive. The developer defines a list of tags and specifies that some tags conflict with other ones. During the preprocessing phase, the choice of a method m tagged by T prevents the CHAMELEON preprocessor to combine m with other methods whose tags conflict with T. Obviously, a method cannot be tagged with conflicting tags.

Hence, while multiple method definitions within an adaptation alternative permits developers to establish a sort of “vertical dependency” between methods, tags can be used to force some kind of “horizontal dependency” crosscutting different alternative classes.

```xml
<conflicts>
    <tag id="T1"/>
    <tagconflict id="T2"/>
    <tagconflict id="T4"/>
</tag>
<conflicts>
```

**Figure 6.** A tag conflicts specification file

```java
alternative class E adapts C {
    tag(T1) void m1() { ... }
}
altspace alternative class F adapts C {
    tag(T2, T5) void m2() { ... }
}
```

**Figure 7.** Tag specification for an adaptable method definition

Conflicts are specified by editing a simple XML file supported by our DE Eclipse plugin (see Figure 1). For instance, in Figure 6, the XML element `<tag id="T1">` defines a tag identified by T1, and the element `<tagconflict id="T2">` specifies the conflicting tag T2. Such a specification establish a relation among tags that is symmetric, i.e., T1 \(\perp\) T2 \(\Rightarrow\) T2 \(\perp\) T1, and not transitive, i.e., T1 \(\perp\) T2 and T2 \(\perp\) T3 \(\Rightarrow\) T1 \(\perp\) T3. Figure 7 shows an example of two alternatives E and F for the adaptable class...
C in Figure 3. Considering the tag specification in Figure 6, we associate tags to methods in order to prevent the preprocessor from generating a derived class containing both \( E.m1() \) and \( F.m2() \).

Figure 8 formally specifies how our programming model extends the Java language grammar. The extension uses the same nomenclature as the one used in [12], and implicitly reuses all the definitions there provided (reported in Italic).

4. Adaptation policy properties

This section introduces a formal structure that, for each adaptable class, defines its adaptation policy. To this end, we introduce the notion of alternatives tree that allows us to model an adaptable application as a forest of alternatives trees.

4.1 Alternatives tree

In our programming model, each alternative adapts exactly one adaptable class or alternative class. In turn, each adaptable class or alternative class can be adapted by more than one alternative. This permits us to represent the relationships between an adaptable class and its alternatives using a tree structure. The following definition introduces the notion of alternatives tree, which gives a formal representation to the adaptation policy of a class.

**Definition 1** (Alternatives tree). An alternatives tree is a finite, labelled, directed tree where the root represents an adaptable class, internal nodes represent adaptable alternatives, and leaves represent non adaptable alternatives. The children of a node represent all the possible alternatives for adapting it. Each node is labelled with a 5-tuple \( N = (l, V, M, aM, IaM) \), where:

- \( l \) is the name of the class/alternative.
- \( V \) is the set of class/instance variables defined in \( N \).
- \( M \) is the set of standard Java methods defined in \( N \).
- \( aM \) is the set of adaptable methods declared in \( N \).
- \( IaM \) is the set of (possibly tagged) adaptable methods implemented in \( N \).

Figure 9 shows the alternatives tree for the adaptable class \( C \) of Figures 3 considering the alternatives of Figures 4, 5, and 7. A node is represented as a rectangle.

Note that, an adaptation policy can produce inconsistent derived standard classes. This is, for instance, the case of two alternatives of the same adaptable class that declares two variables having the same name, or the case of an adaptable method that is not implemented by any alternative. In this section, we define some properties that adaptation policies should have in order to avoid the generation of inconsistent classes. Accordingly, we define constraints over the programming model to help enforcing these properties. To express these properties and constraints, we rely on the structure of alternatives tree.

4.2 Completeness of the adaptation policy

Since the declaration of an adaptable method does not provide an implementation for it, the next property follows:
**Property 1 (Completeness of the adaptation policy).** For each adaptable method there must exist at least one alternative class that defines an implementation for it.

This property states that an adaptation policy must cover all the adaptable methods that constitute the behavioural variations of an adaptable application. This ensures that all the variations, specified at adaptable code level, can be resolved by the preprocessor. To guarantee the completeness of the adaptation policy the following constraint is defined:

**Constraint 1.** For each adaptable method adaptM declared in an adaptable class/alternative C, there must exist at least an alternative that adapts C, which provides an implementation for adaptM or redeclares adaptM as adaptable.

This constraint guarantees that for each adaptable method adaptM declared in a node N of the alternatives tree, at least an implementation for adaptM is provided in the subtree of N.

**4.3 “Scope globality” of the class adaptation policy**

An alternative class is semantically linked to the adaptable class C it adapts, i.e., all the adaptable methods implemented by the alternative class should be considered as if defined within C. In fact, as it will be clear in Section 5, the preprocessing phase suitably welds the alternative classes into the set of standard Java classes derived for C. These Java classes have a definition for each adaptable method provided by some alternative class. Moreover, an alternative can define its own additional standard methods and instance-class variables. These additional members will then be integrated into the derived classes associated to the top-level adapted class, as if they were declared primarily there. Similarly, alternatives that adapt other adaptable alternatives (descendants of C) are all welded into standard Java classes derived for C. These concepts are generalized by the following property:

**Property 2 (“Scope globality” of the class adaptation policy).**

The scope of an adaptation alternative A in the alternatives tree is that of the adaptable class labelling the root node, enriched with all the members declared in the descendant alternative classes along the path from the root to A.

This property implies that the standard Java access modifiers do not affect alternatives and that an alternative has access to all the private elements of all its ancestors in the alternatives tree. In particular, an alternative can access the following members declared in its ancestor class/alternatives: instance and class variables, standard Java methods and methods declared as adaptable. Note that, it is not mandatory to provide an implementation for all adaptable methods in the ancestor alternatives. In fact, the completeness of the adaptation policy guarantees that, after preprocessing, an implementation for these methods will be surely provided (e.g., by sibling nodes).

In order to avoid the generation of a class having multiple overlapping member declarations, we impose the following constraint:

**Constraint 2.** For each alternative class A, the names used to declare additional members must be unique within the alternatives tree A belongs to. Formally, for each pair of nodes N1 = (l1, V1, M1, aM1, IaM1) and N2 = (l2, V2, M2, aM2, IaM2) in the alternatives tree, the following relations must hold: V1 ∩ V2 = ∅, M1 ∩ M2 = ∅, aM1 ∩ aM2 = ∅, M1 ∩ aM2 = ∅, M2 ∩ aM1 = ∅ and, obviously, M1 ∩ IaM2 = ∅, M2 ∩ IaM1 = ∅.

**4.4 Fitness of the class adaptation policy**

The presence of an alternative A within an adaptation policy is justified only if A contributes to the definition of at least a derived standard class:

**Property 3 (Fitness of the class adaptation policy).** For an alternative class A to make sense within the adaptation policy, it must contribute to generate at least a derived class. Specifically, according to the tag conflicts specification, for each alternative A, there must be a derived class that contains all the members defined in A, i.e., implementations of adaptable methods, variables and standard Java methods.

An alternative class A is always associated to an adaptable class or another adaptable alternative class. To guarantee that a derived standard class contains at least a member defined by A, a strong constraint that ties adaptable method declarations and implementations is needed:

**Constraint 3.** For an alternative class A that adapts a class/alternative E at least one of the following conditions must hold:

- A implements a subset of the adaptable methods that are declared in E, or
- A redeclares as adaptable a subset of the adaptable methods that are declared in E and defines additional standard Java methods or variables that must be used by at least a descendent of A.

Besides, the tag conflicts specification must not prevent the alternative A (i.e., some of the members A defines) to contribute to the definition of at least a derived standard class.

Note that constraint 3 also requires that a tag conflicts specification does not invalidate the property. For example, if two methods of an alternative are tagged with conflicting tags, the alternative will never be chosen as the choice of an alternative implies the choice of all its members (but this is not possible because they are conflicting).

**5. Preprocessing phase**

The preprocessor analyses the generic code and links each adaptable method to the different implementations provided by the alternative classes according to the programming model rules. It further checks the validity of the adaptation policy properties. In this way, we recall that, for each adaptable class C into the adaptable code, we obtain a set of standard Java classes (i.e., derived classes) that represent different versions of C. It is worth to highlight that, especially for non-trivial generic code specifications, the developers is not (and he/she is not required to be) a priori aware of the number and the structure of all the derived classes that will be generated automatically by the CHAMELEON preprocessor. As better argued later, the declarative nature of the programming model and its tool support do not force developers to be a priori aware of all the possible contextual situations, and all the possible behavioural variations for handling context changes.

In the remainder of this section, we briefly describe how the preprocessor resolves the variability of an adaptable class. To this end, it performs a post-order visit of the alternatives tree, and derives a set of standard Java classes that will be then used for generating different application alternatives.

The generation of the derived classes, for the adaptable class, starts from the leaves. Each leaf is a non-adaptable alternative and hence, it does not contain any adaptable method to be resolved. Basically, at each node N, all the children are recursively resolved obtaining different modes for adapting them. Then, for each node N, all the permutations of the different modes are generated, and the valid (non-conflicting) ones, which cover all the adaptable methods of N, are selected. For instance, the resolution of the adaptable alternative B1 in Figure 9 will lead to the modes...
shows all the possible valid modes for the adaptable class C. AdaptApp = {B1.m1(), B2.m3()}, B1 = {B1.m1(), B2.m3()}, B2 = {B1.m1(), B3.m3()}

Figure 10. Class alternatives of the C adaptable class of Figure 9

{B1.m1(), B2.m3()} and {B1.m1(), B3.m3()} (see B1 in Figure 10), which combine the two ways of adapting the method m3() with the method m1(). The same holds for D1. Figure 10 also shows all the possible valid modes for the adaptable class C that will be used for generating all the derived classes for C. The latter, suitably combined with the core code, make-up different versions of the adaptable application, i.e., different application alternatives, suitably combined with the method m2().

6. An adaptable application example

In this section we describe how the CHAMELEON programming model has been used to implement an adaptable MIDlet application that is able to display a Mandelbrot fractal filling the screen of (possibly limited) devices.

The drawing of a fractal can be a simple or a complex task. In the most cases, the more complex is the construction method, the more beautiful is the produced fractal, and the more resources are required to the device (CPU, screen characteristics, memory, ...). In order to display the finest fractal possible while accounting for the characteristics of the device, a context-aware adaptable application is auspicated. Typically, the construction of a Mandelbrot fractal is done iterating the fractal formula to determine if a canvas pixel is in the Mandelbrot set. Then, each pixel is coloured according to some coloring algorithm. Hereafter in this section, we show how the CHAMELEON programming model has been used to implement an adaptable application that provides different versions of a Mandelbrot fractal using different building and colouring methods. If the device is equipped with a very slow CPU, which is not able to compute a fractal locally, we provide an alternative that (if a network connection is available) connects to a remote server, downloads a fractal adapted to the device screen characteristics (sent by the device) and displays it. For the sake of space, we describe the adaptable fractal application by depicting the forest of its alternatives tree in Figure 11 and by reporting only an excerpt of the generic code in Figure 12. Interested readers can find a deeper discussion in [9] and download the complete adaptable code from [1].

Referring to Figure 11 the Fractal adaptable application has two adaptable classes MandelFractalMIDlet and MandelCanvas, and a standard Java class SocketConnection. The latter is a thread that provides a set of methods to connect to a remote server and retrieve a fractal image customized with respect to the screen characteristics sent by the device.

adaptable class MandelFractalMIDlet extends MIDlet {
  adaptable protected void startApp () {
    ...
    public MandelFractalMIDlet () {
      ...
    }
  }
  ...
}

// *************** ALTERNATIVES ***************

alternative class LocalConstruction adapts ...
  ...
  alternative class RemoteConstruction adapts ...

...
on a remote server, and then calls a MandelCanvas object to retrieve and display the fractal. Clearly, the tags remote and local attached to the methods are declared to be conflicting tags.

MandelCanvas is an adaptable class that extends the J2ME library class Canvas and is used to display the fractal. It defines two adaptable methods, namely, generateFractal and MandelCanvas (the constructor). The RemoteGetter alternative provides implementations for these adaptable methods. It is worth noting that, the method MandelCanvas is tagged with remote and this makes the RemoteGetter alternative implicitly associated with the RemoteConstruction alternative. Thus, the generateFractal implementation, provided by RemoteGetter, uses SocketConnection to retrieve the fractal from a remote server, and display it. The LocalBuilder alternative provides an implementation for the adaptable constructor MandelCanvas, while redeclaring as adaptable the method generateFractal. It also declares two new adaptable methods, initColors and pixelColor, and defines a standard Java method drawFractal-Pixel that implements the algorithm to determine if a canvas point is in the Mandelbrot set (according to the Mandelbrot fractal formula [18]). It calls the adaptable method pixelColor to define the color of the pixel. Note that, the MandelCanvas implementation is tagged with local and, hence, the adaptable alternative LocalBuilder and all the alternatives that adapt it are implicitly associated to the LocalConstruction alternative. The LocalBuilder class has three alternatives that provide different implementations of the adaptable method generateFractal. This method iterates over the canvas pixels and, invoking the method drawFractal, displays the fractal on the canvas. The MatrixCanvas alternative, stores the fractal in a matrix and displays it all at a time when the process ends. ArrayCanvas displays a fractal column at a time storing it into an array during the building phase. DirectCanvas draws and displays a fractal pixel at a time. MatrixCanvas, ArrayCanvas and DirectCanvas implement different ways of building the fractal, thus requiring different resources and providing different QoS. In particular, they require, in the given order, a decreasing amount of memory to store the fractal and an increasing time to complete the task, mainly due to the screen refresh frequency [1]. As a consequence, the user perceives different QoSs due to the different “promptness” of the building methods. In the worst case (i.e., MatrixCanvas), the user has to contemplate a “black screen” until the whole drawing process ends.

Figure 13. Fractal alternative coloring algorithms

The three other alternatives of the LocalBuilder class, namely, EscapeTime, EscapeDistance, NormalizedIterations, provide different adaptations for the methods initColors and pixelColor. These alternatives produce, in the given order, images of growing beauty (see Figure 13), but require increasing resource demands (CPU power, number of screen colors, ...).

In Figure 12 it can be noted that additional information can be specified by using CHAMELEON-specific annotations. Supported by the CHAMELEON DE (see Figure 1), developers can easily specify annotations at the generic code level by calling the “do nothing” static methods of the Annotation class. Annotations permit to specify resource demand (Resource Annotation), non-functional Service Level Specification (SLS Annotation), and upper bounds on the number of loop iterations (Loop Annotation) and recursive method calls (Call Annotation). For instance, the method call Annotation.resourceAnnotation(’Network(true)’) specifies that the RemoteConstruction alternative class demands for a network connection on the device.

We recall that the whole framework is based on the resource and SLS models (see Figure 1), that, in particular, allow for specifying conforming resource and SLS annotations, respectively. Annotations are organized in set of standard Java patterns that are injected into the structure of the post-processed code to be then recognized and used by ARA to determine the resource demands and the offered SLAs of the derived application alternatives. Since Annotations are not strictly part of the programming model, we entirely refer to [3] (and references therein) for a detailed description. It is worth to mention that we do not use standard Java annotation mechanism since we need to place annotations also on loops, blocks, and simple statements (by contrast, Java SE 6 permits annotations only on declarations). However, the JSR 308 proposal [10] extends the Java annotation mechanism to permit annotations on any occurrence of a type. There is also interest in extending this proposal so that annotations can be placed not only on type references, but on loops, blocks, and simple statements, as well. Type annotations are planned to be part of the Java 7 and in the future we will consider this extension.

After the preprocessing phase described in Section 5, the generic code specifying the fractal adaptable application generates the ten application alternatives listed in Table 1. For each alternative, the table reports which alternative classes provide the implementations of which adaptable methods. Clearly, each application alternative is characterized by a different resource demand to be correctly executed, and offer a (possibly) different QoS.

Continuing the considerations initiated in Section 5, the programmer did not know that the simple generic code of the adaptable MIDlet fractal would have generated ten application alternatives (they would have been much more without the specification of conflict tags). Moreover, the programmer does not know (and does not have to) how these applications alternatives are composed, and how many and which global contextual situations will be handled by them. The CHAMELEON context-aware run-time support will do this job.

7. Related work

Our programming model is related to several valuable approaches in the literature. We have analysed these approaches in order to understand what is required to make a programming model adopted by practitioners, while keeping in mind that easy use and simplicity are mandatory requirements. We then decided to propose a new language that (while retaining some features of other formalisms) was as light and intuitive as possible avoiding powerful and complex features not needed for our own purposes.

The strategy design pattern [11] allows for defining a family of algorithms and, by encapsulation, make them interchangeable. Then, the main purpose is to let the algorithms vary independently from clients that use them. Specialization classes [7,24] provide an
extension to the Java language for integrating adaptive behaviour in existing applications. In particular, a set of adaptive behaviour is attached to a given class by defining two kinds of information: the methods that have to be adapted (and their implementation as well) and the conditions under which the adaptation must be done. These conditions must be explicitly expressed through predicates on the internal state (i.e., the instance variables). When the predicate attached to a given adaptation evaluates to true, then the methods of the current class are automatically changed to the matching adaptation. Specialization classes are an approach that is aimed at runtime execution: the application code is pre-processed and instrumented with all the mechanics needed for handling the adaptation. Basically, the instrumentation consists in the insertion of guarded methods that check the conditions expressed through the predicates in all the points in which a variable appearing in a predicate is accessed for being modified. The implementations of the method adaptation are wrapped in separate classes following the strategy design pattern. Similarly to our approach, specialization classes offer a way to describe how to adapt a particular class and a structural model to describe this adaptation. The overhead involved by the instrumentation of the application code for handling the specialization might be not acceptable in our context because of our main target, i.e., limited devices. Moreover, specialization classes force developers to use instance variables for explicitly defining the conditions under which the adaptation must be performed. The declarative nature of our approach, on the contrary, is based on an automatic interpretation of the application variability that combines the independently specified variation points, and calculates the resources required for the execution in the given context.

Objective-C [16] provides categories, that are a way to collect method implementations in separate files. Categories can be added at runtime to a given class obtaining an extension of the class itself or the overriding of some of its methods. Thus, categories can both add new methods and replace existing ones in a given class. Methods defined in a category have full access to all the instance variables within the class, including private variables. This is, in some way, the same behaviour that we have modelled with our alternative classes.

More recently, in [13] the authors propose two programming models that use context models mainly based on situation and preference abstractions. The former is a way to define conditions on the context in term of “fact abstraction” that represents high level views of the context. The latter enables to manage the user preferences in order to set the user context-aware requirements. At each situation changing occurs an event is triggered with some lifetime. Whenever a situation changing occurs an event is triggered with some lifetime condition. The second model presented within the same work is the Branching Model: it offers a novel and flexible means to insert context and preference decision points into the application logic.

Trough the Branching Model a set of preferences are evaluated at run-time to select which program branch visit to better fulfil the user requirement in a certain context situation. A bunch of API are provided to the programmer in order to dynamically choose the right adaptation based on the context evaluation and user preferences.

The context-oriented programming model proposed in [15] is developed as an extension of Python. The context dependent behaviours are kept in a stub repository separated from the running code. The procedure of "context filling" enables the selection of the appropriate stub from the repository in order to fill the gap of the source code. The filling procedure depends on the goal to achieve and the context. A set of adaptation stubs are defined in order to reach the goal for any possible arising context. Hence, the context defines the pertinence scope for the behaviour to apply. This approach requires an a priori global vision of the context. Moreover, it lacks suitable models to represent the context situations and the goals to achieve within the context. Context is modelled by simple xml tags checked during the adaptation phase. Context modelling is provided without the possibility to define relations upon the context elements.

The work in [21] proposes an Aspect Oriented Programming (AOP) approach where the context drives the use of aspects. The aspects are invoked based on the actual context information which is modelled as first class entities. Dynamic AOP extends the original notion of AOP by allowing weaving at load or run time. Dynamic AOP has been shown to be a very suitable mechanism for run time adaptation of applications and services. In [22] the authors propose a system offering dynamic AOP in Java based on AspectU. It supports a wide range of standard AspectU constructs for dynamic cross-cutting, is portable, provides complete method coverage and is compatible with standard JVMs. Again, the dynamic weaving introduces an overhead that might not be suitable for resource constrained devices like the ones we target.

Mixin is a programming style where units of functionality are created in a class and then mixed in with other classes. For example, the language Scala and the Fractal component model[2] use mixins. Even though mixins do not specifically target context-aware adaptable applications, they can be related to our programming model since they can be used to construct new classes by combining functionalities defined in other classes. However, the mixin approach requires programmers to explicitly and a priori specify what are the classes to be combined.

Differently from the mixin programming style and the work in [21][22], our approach allows for declaratively and independently specifying behavioural variations that are then automatically combined by the preprocessor and selected by the customizer, according to the resource characteristics of the target execution environment and the SLS specifications.

---

**Table 1.** Fractal application alternatives

<table>
<thead>
<tr>
<th>Applications</th>
<th>Methods</th>
<th>MandelFractalMIDlet</th>
<th>MandelCanvas</th>
<th>generateFractal</th>
<th>initColors</th>
<th>pixelColor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fractal01</td>
<td>startApp</td>
<td>MatrixCanvas</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fractal02</td>
<td>LocalConstruction</td>
<td>LocalBuilder</td>
<td>MatrixCanvas</td>
<td>EscapeDistance</td>
<td>EscapeTime</td>
<td>EscapeTime</td>
</tr>
<tr>
<td>Fractal03</td>
<td>LocalConstruction</td>
<td>LocalBuilder</td>
<td>MatrixCanvas</td>
<td>NormalizedIterations</td>
<td>NormalizedIterations</td>
<td></td>
</tr>
<tr>
<td>Fractal04</td>
<td>LocalConstruction</td>
<td>LocalBuilder</td>
<td>ArrayCanvas</td>
<td>EscapeTime</td>
<td>EscapeTime</td>
<td></td>
</tr>
<tr>
<td>Fractal05</td>
<td>LocalConstruction</td>
<td>LocalBuilder</td>
<td>ArrayCanvas</td>
<td>EscapeDistance</td>
<td>EscapeTime</td>
<td>EscapeTime</td>
</tr>
<tr>
<td>Fractal06</td>
<td>LocalConstruction</td>
<td>LocalBuilder</td>
<td>ArrayCanvas</td>
<td>NormalizedIterations</td>
<td>NormalizedIterations</td>
<td></td>
</tr>
<tr>
<td>Fractal07</td>
<td>LocalConstruction</td>
<td>LocalBuilder</td>
<td>DirectCanvas</td>
<td>EscapeTime</td>
<td>EscapeTime</td>
<td>EscapeTime</td>
</tr>
<tr>
<td>Fractal08</td>
<td>LocalConstruction</td>
<td>LocalBuilder</td>
<td>DirectCanvas</td>
<td>EscapeDistance</td>
<td>EscapeTime</td>
<td>EscapeTime</td>
</tr>
<tr>
<td>Fractal09</td>
<td>LocalConstruction</td>
<td>LocalBuilder</td>
<td>DirectCanvas</td>
<td>NormalizedIterations</td>
<td>NormalizedIterations</td>
<td></td>
</tr>
<tr>
<td>Fractal10</td>
<td>RemoteConstruction</td>
<td>RemoteGetter</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8. Concluding remarks and future directions

In this paper we have presented a programming model for adaptable applications. The programming model provides developers with a set of extensions to Java for agilely specifying behavioural variations in a flexible and declarative way, while being close to the Java programming language. An ad-hoc preprocessor resolves then variability by generating standard Java methods within standard Java classes, which are then automatically combined for deriving different application alternatives.

The declarative and generative nature of the programming model spares developers the writing of the adaptation logic by using conditionals to explicitly and a priori treat all the possible contextual situations, and all the possible associations with behavioural variations. On the contrary, developers separately define the smallest units of behavioural variations up to adaptable methods, and locally specify (through annotations) the resources demanded, upper bounds to iterations and recursive method calls, and the offered quality of service (QoS). Then, the context-aware run-time support offered by CHAMELEON will automatically resolve the overall variability by calculating the global resource demand and offered QoS. Then, the adaptable application is properly customized to the current context condition. In this way, developers are not required to a priori have a global vision of all the possible context situations, rather, unforeseen contexts will be (possibly) handled at run-time by dynamically combining behavioural variations. Note also that, the possibility of separately defining units of behavioural variations, without explicitly writing the adaptation logic, clearly facilitates code maintenance and evolution.

In previous work by the authors [2,3], the CHAMELEON framework has been used to realize a form of service adaptation in the IST PLASTIC project [19], whose goal is the rapid and easy development, deployment and execution of adaptable services for the heterogeneous next-generation networks. In particular, we have used the programming model for the development of two adaptable applications (in the e-Health and e-Learning domains) for consuming and providing services targeted to mobile resource-constrained devices. In particular, in [2] we propose a mechanism that, exploiting ad-hoc methods for saving and restoring the (current) application state, enables services’ evolution again context changes by dynamically un-deploying the no longer apt alternative and subsequently (re-)deploying a new alternative.

As already anticipated, even though at present CHAMELEON and, hence, the programming model are used for context-dependent run-time deployment of adaptable Java applications, other forms of adaptation can be supported. Towards this aim, we are currently formalizing and implementing an object-oriented context-aware computation engine that (supported by a context monitor and a context-switch mechanism, acting with message-passing granularity) allows for also handling context-dependent run-time behaviour. Behavioural variations will be specified by means of our programming model in terms of generic code, and a context-dependent message dispatch will then allow the delivery of messages to the right adaptable object (and hence to the right adaptable method) according to the monitored context.

References


