A Declarative Approach for Designing and Developing Adaptive Components

Philippe Boinot, Renaud Marlet*, Jacques Noyé†, Gilles Muller and Charles Consel‡
Compose Group, IRISA / INRIA, University of Rennes I
Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France
E-mail: {pboinot,muller,consel}@irisa.fr, noye@emn.fr

Abstract

An adaptive component is a component that is able to adapt its behavior to different execution contexts. Building an adaptive application is difficult because of component dependencies and the lack of language support. As a result, code that implements adaptation is often tangled, hindering maintenance and evolution.

To overcome this problem, we propose a declarative approach to program adaptation. This approach makes the specific issues of adaptation explicit. The programmer can focus on the basic features of the application, and separately provide clear and concise adaptation information. Concretely, we propose adaptation classes*, which enrich Java classes with adaptive behaviors. A dedicated compiler automatically generates Java code that implements the adaptive features. Moreover, these adaptation declarations can be checked for consistency to provide additional safety guarantees.*

As a working example throughout this paper, we use an adaptive sound encoder in an audio-conferencing application. We show the problems associated with a traditional implementation using design patterns, and how these problems are elegantly solved using adaptation classes.

1. Introduction

A complex system is typically made up from separate components sharing common resources. The behavior of these components and the quality of service that they each provide is interdependent: the system must adapt when the available resources are limited. For this reason, adaptation technologies, such as feedback control mechanisms, are used to implement systems that dynamically react to resource variations. As an example, consider distributed multimedia applications that share network resources. These applications must adapt their behavior according to the network bandwidth to improve performance and to guarantee quality of service [5, 6, 2].

Building adaptive software is difficult, code that implements adaptation is often tangled due to the resource and component dependencies. Furthermore, there is no convenient support in programming languages to easily associate given execution contexts with corresponding behaviors. Current adaptation mechanisms are ad hoc, which impedes maintenance and extensibility.

In this paper, we present a declarative approach for the design and development of adaptive components in an object-oriented language like Java. This approach makes explicit two specific issues: adaptation conditions and adaptation actions. Adaptation conditions express when adaptation should occur, depending on the program state and the execution context. Adaptation actions determine appropriate component behaviors. We declare both conditions and corresponding actions in a concise and precise way using *adaptation classes*, which enrich existing Java classes with adaptive behavior.

These adaptation declarations are processed by a dedicated adaptation class compiler. This compiler generates Java code that implements the adaptive features declared in the adaptation classes. In particular, it automatically instruments the code with introspection mechanisms that capture state changes. These state changes trigger behavior switches according to the adaptation declarations, seamlessly integrating adaptation into programs.

Using the adaptation class approach, the programmer can focus on the basic features of the application, and separately provide adaptation information. The separation of concerns and the conciseness of adaptation classes improve development, maintenance and extensibility. Safety is also improved as the adaptation class compiler performs consistency checking. In addition, as opposed to manual, ad hoc approaches that are inevitably error-prone, the compiler sys-

*Author's current address: Trusted Logic, 5 rue du Bailliage, F-78000 Versailles, France. E-mail: Renaud.Marlet@trusted-logic.fr.

†Author's current address: École des Mines de Nantes, 4, rue Alfred Kastler, BP 20722 F-44307 Nantes Cedex 3, France.

‡Author's current address: LaBRI / ENSERB, 351 cours de la Libération, F-33405 Talence Cedex, France.

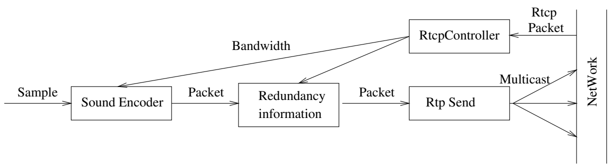

Figure 1. Audio-conference application

tematically instruments the code whenever introspection is needed. Finally, there is no price to pay for performance compared to a manual approach, because the compiler relies on efficient introspection and context switching mechanisms.

The rest of this paper is organized as follows. Sect. 2 presents a working example of an adaptive program in the multimedia domain. Sect. 3 introduces some terms and issues related to adaptation. Sect. 4 then describes two ways of designing an adaptive component: an ad hoc manner and using adaptation classes. In Sect. 5, we present the problems that arise when extending the adaptive component and how we handle them. Sect. 6 then gives an overview of adaptation classes and provides an informal semantics. Sect. 7 surveys related work, and Sect. 8 presents our concluding remarks and future work.

2. Example: an adaptive sound encoder

Adaptation is crucial for multimedia [6]. Consider a live audio application for audio-conferencing. The quality of the audio depends primarily on the number of lost packets and the delay variations between successive packets. Furthermore, the average end-to-end delay must be small to allow interactions between participants. Therefore, the application must adapt to the available network bandwidth to respect the temporal constraints, and provide performance guarantees regarding loss rate or maximum delay [5, 2]. Bandwidth control mechanisms can adjust to network congestion by adapting the size of the packets to the load of the network [2].

Fig. 1 illustrates the general structure of the sending part of the adaptive audio-conference application *freephone* [3]. Before transmitting sound samples to the subscribers of the audio-conference, specific treatments reduce the amount of information sent over the network. First, the sound is compressed in the `SoundEncoder` component. To increase the tolerance to packet loss redundant information is added which is used to reconstruct lost packets (the `RedundancyInformation` component). The `RtpSend` component broadcasts the packet. The protocol used by audio-conferencing applications such as freephone is RTP (Real-time Transport Protocol) [5, 15], which provides feedback on the transmitted data as RTCP (Real-time Transport Control Protocol) control frames. The `RtcpController` component receives these frames and computes feedback information to approximate the network bandwidth available. This estimate is used by the `SoundEncoder` and `RedundancyInformation` components to adapt the compression rate and the amount of redundancy information.

To vary the compression rate, we can choose among different encoding algorithms [3], as shown in Table 1. Choosing among these algorithms makes it possible to vary an 8 kHz sound speech sample from 5.6 kbits/s to 48 kbits/s. *ADPCM* [12, 17] is a differential encoder which can be parameterized from 16 kbits/s to 48 kbits/s with a precision parameter (between 2 and 6). *LPC* [8] and *GSM* [16] are fixed low bit-rate encoders. Using these encoding algorithms, the application is able to adequately react to bandwidth variations.

Encoding	bit rates (kbits/s)
ADPCM(6)	48
ADPCM(2)	16
GSM	13.3
LPC	5.6

Table 1. Bit rates of the encoding algorithms

3. Adaptation basics

An adaptive component is a component that is able to adapt its behavior to different execution contexts. Adaptation can be *static* or *dynamic*. Static adaptation corresponds to configuration, i.e. performing adaptation before execution depending on fixed parameters. Dynamic adaptations are run-time changes depending on the execution context.

There are two ways to implement dynamic adaptation: *adaptation on change* and *adaptation on action*. Adaptation on change installs new behaviors each time that the execution context changes. When adaptive functionality is used, the execution context does not need to be checked; installed behaviors are blindly performed. On the contrary, adaptation on action inspects the execution context each time that an adaptive functionality is called, to determine which behavior should be performed.

The choice of which strategy is best (on change or on action) depends on both the frequency of change in the execution context and the number of calls of adaptive actions. In the following, we focus on adaptation on change. This strategy is usually the most appropriate for multimedia components because the same set of actions are often repeatedly called, whereas context changes require updating behaviors only when given thresholds are reached. Given that the rate change of the execution contexts is often lower than the frequency of action calls, adaptation on change requires less run-time management compared to adaptation on action.

Adaptation on change can be decomposed into three stages:

Introspection. Collecting information about the execution context. This introspection can be implemented by inserting guards that monitor given program states and notify the adaptation controller (the second stage) in case of changes. In our working example, the introspection stage is implemented by the `RtcpController` component, which sends a notification to the `SoundEncoder` and `RedundancyInformation` components when the bandwidth changes.

Control. Adaptation conditions are expressed over the guarded program states. These predicates, often represented by threshold tests, determine which behavior is the most appropriate. In the example, the `SoundEncoder` component implements this control mechanism. Each time the `RtcpController` signals a bandwidth change, the `SoundEncoder` evaluates adaptation conditions based on the new bandwidth value, to choose the appropriate encoding algorithm.

Installation. Depending on the adaptation conditions, new behaviors are installed, replacing previous ones. In our example, the `SoundEncoder` installs one of the different encoding algorithms, balancing quality and packet sizes.

In this paper, we focus on the `RtcpController` and `SoundEncoder` components, and their relationship. We show that the traditional implementation of a simple adaptive system raises several problems in terms of construction and evolution.

4. Building an adaptive component

In this section, we study the implementation of the `SoundEncoder` component for a wireless network. Typically, a wireless network is characterized by low bandwidth, for which reason we use the two low bit-rate encoders, LPC and GSM.

The choice between these two algorithms depends on the available network bandwidth. For this reason, we need to create a relation between the `RtcpController` component, which estimates the network bandwidth, and the `SoundEncoder` component, which selects the appropriate encoding algorithm.

We build this adaptive component in two ways: first, we use an ad hoc technique based on design patterns and study its advantages and disadvantages. Then, we present the design of the same component using our declarative approach and examine its benefits.

4.1. Making an adaptive component "by hand"

Because adaptation is difficult to apprehend in a complex system, implementing an adaptive component requires a structured approach. A natural way to organize component implementation in an object-oriented language is to use design patterns [9]. Design patterns capture common structures that arise when designing and implementing programs, providing a good framework for conveying program design expertise. Over the last few years, they have become a standard tool in the design of object-oriented programs.

A selection of well-known design patterns can be used to express the three stages of our adaptive `SoundEncoder`. To build this adaptive component, we rely on:

- a subscription mechanism to the network observer (`RtcpController`) that can notify all the subscribers of guarded state changes (Introspection and Control stages),
- a mechanism to modify a behavior (Installation stage).

Following [4], we use three design patterns to implement these features: *Observer*, *Strategy* and *Facade*. The Observer pattern allows a set of subscribers to be notified of any state change. The Strategy pattern describes the implementation of an object that can dynamically change its behavior. The Facade pattern provides an interface to encapsulate a set of objects. The composition of these design patterns is shown in Fig. 2. Implementation details can be found in [1].

Introspection. The `RtcpController` must notify the `SoundEncoder` when the network bandwidth changes. To model this dependency, we use the Observer pattern. This pattern allows a set of objects, the *observers*, to be automatically notified when a state changes in an observed object, the *subject*. In our example, the Observer pattern describes how to establish the relation between the `RtcpController` (the subject) and the `EncoderStrategy` (an observer).

Control. When the bandwidth value changes, the appropriate conditions must be checked to determine which encoding algorithm is the most appropriate. To model this dependency, we could use a Mediator pattern. In practice, this level of indirection is not needed here. It is easier to just extend the observer so that it notifies the subscribers only when an update is required, specifying explicitly which algorithm must be installed.

Installation. Update requests specify which encoding algorithm is to be used as the current behavior. To model this, we use the Strategy pattern, which allows dynamic changes in object behavior by letting complementary behaviors vary independently from the clients who use it. It is implemented by defining classes that encapsulate the different algorithms. These algorithms are implemented with `Gsm` and `Lpc` classes

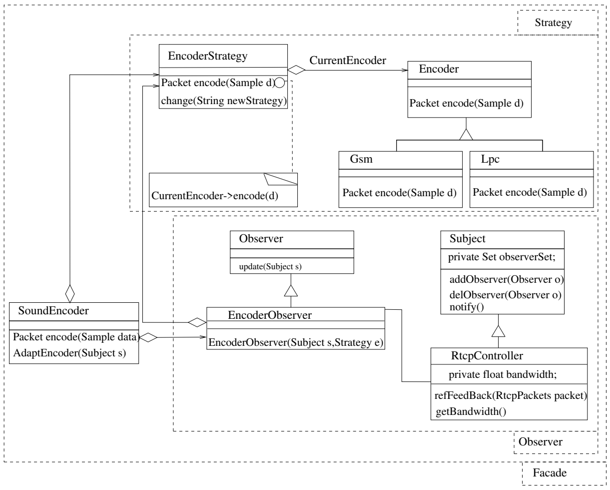

Figure 2. UML diagram of the cooperation between Encoder and RtcpController

> which provide an `encode` method that transforms sampled sound into compressed packets. The `EncoderStrategy` class defines the interface to the adaptive behavior.

Last, the classes implementing the Observer and Strategy functionality are encapsulated within the `SoundEncoder` class using the Facade pattern and providing the proper, unified, interface to the component.

Usage. At instantiation time, the `SoundEncoder` component is connected to the `RtcpController` component in order to be informed of network bandwidth changes. When the bandwidth changes, the `update` method of the `EncoderObserver` object is invoked by the `RtcpController`. The `EncoderObserver` object queries the transmitter for its new state with the `getBandwidth` method. Given the bandwidth, the `EncoderObserver` object then determines the best encoding strategy. It invokes the `change` method of the `EncoderStrategy` object to switch the encoding algorithm.

The `SoundEncoder` must create an instance of each behavior, which can be done either *lazily* (when needed) or *strictly* (at instantiation time). Strict creation initially consumes more resources than lazy creation, but ensures responsiveness after initialization has completed, which makes it preferable for our component. Thus, the system first creates all adaptive behaviors, after which the `SoundEncoder` component is synchronized with the `RtcpController` component by invoking the `update` method, in order to acquire the actual bandwidth.

Assessment. Design patterns describe solutions to specific problems in object-oriented software design. For example, the structure of the Strategy pattern provides different implementations of the same behavior offering a way to support a variety of algorithms and facilitating future extensions. The use of design patterns isolates different functionality (creational, structural and behavioral) and splits up behaviors (Strategy, Mediator, Observer...). However, in the case of adaptation, using design patterns has some drawbacks.

The design of the adaptive `SoundEncoder` component is complex (see Fig. 2 and [1]). The use of the Strategy and Observer design patterns for the `SoundEncoder` component introduces communication overhead and increases the number of classes. In our example, it is necessary to declare 9 classes bound together by 5 references. In addition, the object interaction protocol is computationally expensive. It requires 2 indirections for each call of the `encode` method and 3 calls for each modification of the `bandwidth` field (4 if the current encoder has to be changed). Furthermore, the decomposition into different subcomponents makes code modification difficult. The dependencies between objects require propagation of all modifications to all the classes in order to keep the code coherent, as will be apparent in Sect. 5.

4.2. Declaration of an adaptive encoder

To simplify the design and development of adaptive components, we propose to declare adaptive behaviors and automatically generate code from these declarations. The principle of our approach is to separate the adaptation declaration from the basic implementation of the main program. We use *adaptation classes* to declare which encoding algorithm to use depending on the bandwidth computed by the `RtcpController` component. The adaptation class compiler evaluates the declarations, creates the `SoundEncoder` component, and modifies the `RtcpController` to plug in the introspection mechanism.

We reuse the `Encoder` class hierarchy and the `RtcpController` class of Sect. 4.1. The `Encoder` class defines an abstraction of a sound encoder and an interface to the encoding method. This abstract class is refined by concrete subclasses (`Gsm` and `Lpc`) which each implement an encoding algorithm. The `RtcpController` class is the network interface for receiving control packets. These are our starting point for designing the adaptive component.

We define an adaptation class `SoundEncoder` (shown in Fig. 3) to adapt the abstract class `Encoder`. The fields and the interface of the `Encoder` class are used as information to define when and how to substitute behaviors. The adaptation class introduces a reference `rtpControl` to an `RtcpController` object to access the value of the current bandwidth. It defines a constructor to initialize this

```
adaptclass SoundEncoder adapts Encoder {
   RtcpController rtpControl;
   SoundEncoder (RtcpController _rtpControl) {
     rtpControl=_rtpControl; }
   when (rtpControl.bandwidth <  13.3) Lpc();
   when (rtpControl.bandwidth >= 13.3) Gsm();
}
```

Figure 3. Declaration of adaptive behaviors on Encoder class

reference, linking the adaptive component with the RtcpController object. Two adaptation conditions (when statements) are introduced, which express conditions over the value of the bandwidth field of the rtpControl object. If this value is less than 13.3 kbits/s, then the current behavior is defined by the class Lpc, otherwise it is defined by the class Gsm. To access the value of the private bandwidth field, the adaptation class compiler introduces an accessor method into the RtcpController class.

The adaptation class compiler can use the information defined in adaptation classes to generate the implementation of the adaptation mechanisms. We know at compile time which observers are needed and we have a precise description of the information required to switch behaviors. Thus, we can generate an implementation specific to the adaptation needs:

- We can simplify the connection mechanism by replacing the management of the set of observers by a reference to each observer.
- It is possible to make observer-specific update protocols, since we know exactly what information is needed about each change. The subject can send observers detailed information as an argument to the update method. This optimization reduces the cost of each state change.
- One of the indirections otherwise needed for calling the encode method can be eliminated. The compiler automatically produces a complete component with all necessary mechanisms. It is not necessary to use a design pattern like Facade to provide a concise interface.

The compilation of this adaptation class modifies the RtcpController class and creates the SoundEncoder class. The RtcpController class is modified by adding a subscription mechanism and analyzing the code to insert guards to detect modifications of the bandwidth field. These modifications do not affect the behavior of the RtcpController; they are transparent for any other objects that may use RtcpController. The resulting SoundEncoder class is composed of references to adaptive behaviors and methods for implementing adaptive mechanisms.

Usage. We insert the generated SoundEncoder component into the program. The use of this class is similar to the use of the SoundEncoder component defined by hand in the Sect. 4.1. First, we create an rtcpSession object to receive and handle RTCP control frames. Second, we create an instance of the SoundEncoder and pass it a reference to the rtcpSession object. Then, the soundEncoder object connects to the rtcpSession object in order to be notified of each modification to its state, and it initializes all possible behaviors defined in the adaptation class (strict creation). To synchronize with the initial rtcpSession bandwidth, the soundEncoder object calls the acUpdate method and determines an initial behavior.

When the value of the bandwidth field is modified, the rtcpSession object notifies the soundEncoder object by calling the update method with the bandwidth value. The soundEncoder object re-computes its adaptation state and applies the appropriate encoding algorithm, as specified in the adaptation class (see Fig. 3).

Assessment. Our approach separates adaptation aspects from the encoding functionality. First, it reduces the complexity of the adaptive SoundEncoder component design. The implementation only needs the Encoder and RtcpController classes and one adaptation class; our approach separates adaptation mechanisms from the encoding functions. Second, performance is improved automatically (instead of improving the pattern-based code by hand). The generated code requires one indirection for each call of the encode method, and one call for each modification of the bandwidth field (or two if it is necessary to change the current behavior). Last, evolution is easier. The evolution of an adaptive component is done by redefining the adaptation classes. The compiler automatically propagates all modifications into the code. Such an evolution is explored in the next section.

5. Evolution of an adaptive component

In the previous section, we defined an adaptive component with the ability to change behaviors depending on network bandwidth variations. Suppose that this component needs to be used in another environment, namely for audio-conferencing on the web. The available bandwidth is higher than for a wireless network, so we can raise the audio quality by increasing the bandwidth adaptation domain of our audio-conferencing application. This section presents the impact of this extension of the SoundEncoder component, namely adding a new encoding algorithm, Adpcm, which extends the adaptive encoder up to 48 kbits/s (see Table 1).

The addition of a new encoding algorithm in the Encoder hierarchy must be propagated to all components that are using these classes. Conceptually, the modifications are minor. We just have to extend the adaptation domain of the `Gsm` behavior and add new adaptation conditions to the object `Adpcm`:

- when the bandwidth is between 16 and 48 kbits/s, use `Adpcm(2)`,
- when the bandwidth is greater than 48 kbits/s, use `Adpcm(6)`.

In the rest of this section, we show how this extension can be implemented, first using the ad hoc method, and then our declarative approach.

Ad hoc extension. To add these behaviors into the adaptive component defined in Sect. 4.1, we need to modify the `EncoderStrategy` and `EncoderObserver` classes in order to propagate state modifications to all the components as illustrated in Fig. 4. First, we must add references to the new behaviors (1) and modify the `EncoderStrategy` constructor to create and initialize them (2). Second, we have to add the assignments to the `CurrentEncoder` variable in the `change` method (3). Last, we extend the control mechanism in the `update` method (4). This example illustrates that a simple change like the addition of new behaviors in an adaptive component is a non-trivial task. It implies modifications throughout the entire program.

Extension by declaration. With adaptation classes, the generation of an adaptive component is automatic. The addition of a new algorithm in the `SoundEncoder` components requires only the modification of the `SoundEncoder` adaptation class (see Fig. 5), and a recompilation.

Alternatively, adaptation class inheritance (explained in detail in the next section) could be used. Fig. 6 shows the `ExtendedSoundEncoder` adaptation class which is a subclass of `SoundEncoder`, that inherits all of the `SoundEncoder` definitions. It defines two levels of predicate evaluation:

- if the `bandwidth` is greater than 16, then the `ExtendedSoundEncoder` behaviors are applied,
- otherwise, it relies on the conditions defined in `SoundEncoder` to find an appropriate behavior.

This use of adaptation class inheritance necessitates the modification of the program using the encoder: the newly generated `ExtendedSoundEncoder` class must be substituted for the `SoundEncoder` class.

6. Adaptation classes

The evolution of an adaptive component is problematic since its dependencies are scattered throughout the program

```
...
class EncoderStrategy {
  Encoder currentEncoder,
          gsmEncoder,
          lpcEncoder,                                   1
          adpcm2Encoder,
          adpcm6Encoder;
  public void change(String newStrategy) {                                   3
    if (newStrategy=="Gsm") { currentEncoder=gsmEncoder; }
    else if (newStrategy=="Lpc") { currentEncoder=lpcEncoder; }
    else if (newStrategy=="Adpcm2") { currentEncoder=adpcm2Encoder; }
    else currentEncoder=adpcm6Encoder;
  }
  ...
  public EncoderStrategy() {
    gsmEncoder=new Gsm();                                   2
    lpcEncoder=new Lpc();
    adpcm2Encoder= new Adpcm(2);
    adpcm6Encoder= new Adpcm(6);
  }
}
...
class EncoderObserver extends Observer {
  EncoderStrategy strategy;
  void update(Subject s) {
    float bandwidth=((RtpSession) s).getBandwidth();
    if (bandwidth<13.3) strategy.change("Lpc");                       4
    else if (bandwidth<16) strategy.change("Gsm");
    else if (bandwidth<48) strategy.change("Adpcm2");
    else strategy.change("Adpcm6");
  }
  ...
}
```

Figure 4. Evolution of the SoundEncoder component

```
adaptclass SoundEncoder adapts Encoder {
  RtcpController rtpControl;
  SoundEncoder (RtcpController _rtpControl) { rtpControl=_rtpControl; }
  when (rtpControl.bandwidth < 13.3)                                      Lpc();
  when (13.3 <= rtpControl.bandwidth && rtpControl.bandwidth < 16) Gsm;()      1
  when (16   <= rtpControl.bandwidth && rtpControl.bandwidth < 48) Adpcm(2);
  when (48   <= rtpControl.bandwidth )                                 Adpcm(6);
}
```

Figure 5. Adding a new adaptive behavior

code. Our declarative approach exposes all adaptation capabilities, and new code can easily be generated by recompiling the modified declarations. This section defines the syntax of adaptation classes and explains the semantics of the language.

The syntax of the adaptation class language is given in Fig. 7. The definition of non-terminals *Identifier*, *FieldDecl*, *ArgumentList* and *ConstructorDecl* are given in the Java syntax [10]. The definition of non-terminal *LinearExpression* is similar to a Java expression, but is slightly constrained as explained later. Adaptation classes are similar to Java classes in their use and syntax. An adaptation class declares adaptation over an abstract Java class. It has a name (both used to build the resulting Java class and to define the inheritance relation), the name of the adapted abstract Java class, and a set of members. These members include fields, constructor declarations, and adaptation conditions which

```
adaptclass ExtendedSoundEncoder extends SoundEncoder {
   when (16  <= rtpControl.bandwidth
        && rtpControl.bandwidth < 48) Adpcm(2);
   when (48  <= rtpControl.bandwidth)  Adpcm(6);
}
```

Figure 6. Inheritance extension of the adaptation class SoundEncoder

```
AcDecl      ::=  adaptclass Identifier AcLink Identifier AcBody
AcLink      ::=  adapts | extends
AcBody      ::=  { (FieldDecl)* (ConstructorDecl)* (AcCond)* }
AcCond      ::=  (when LinearExpression)+ AcBehavior;
AcBehavior  ::=  Identifier ((ArgumentList)? )
```

Figure 7. Adaptation classes syntax

describe the conditions under which adaptive behaviors are applied.

6.1. Adaptation class members

All adaptation declarations are described by the *AcCond* rule. This rule can be divided into two parts: the condition `when` *LinearExpression*, and the action, *AcBehavior*.

Introspection and condition stages. Adaptation conditions express when adaptation should occur, by expressing a condition over the state of the adapted object or its execution context. Conditions are declared with the statement `when` *LinearExpression*, where the predicate *LinearExpression* may reference class fields. An adaptation is applicable when the predicate evaluates to true. In our example (see Fig. 3), the adaptation condition consists of testing the bandwidth value of an `RtcpController` object. For this object to be known by `SoundEncoder`, we declared the reference `rtpControl` in the adaptation class.

The compiler gathers the variables used in the predicates and systematically inserts guards to watch these variables and to notify the adapted component in case of changes (introspection stage). To ensure that all changes to these variables are captured, it is necessary to protect them from being directly written by external objects. The variables must be written through an accessor method which is automatically inserted by the compiler. Consequently, the adaptation context may only contain private fields, such as the `bandwidth` field in the `RtcpController` class.

An incorrect declaration of predicates can cause errors or inconsistencies: if two predicates overlap, the choice of a behavior becomes nondeterministic. An adaptation class must respect *completeness* and *uniqueness*. Completeness guarantees that for every possible situation, there exists an adaptation action. Uniqueness guarantees that for each situation, there is only one adaptation action which is applicable. To enforce these two properties we use the logic language $CLP(\Re)$ [11]. With this language, we can solve linear arithmetic constraints and perform computations over real numbers (non linear constraints are not allowed). As a consequence, when a condition is expressed as a set of linear constraints, it is possible to verify completeness and uniqueness.

Declaration of actions. An action is the substitution of one behavior (i.e. object) by another one. Each adaptive behavior is represented by an object. To enable substitution of objects, they must provide the same interface and must inherit from the same abstract Java class. In our example, all encoding algorithms inherit from the `Encoder` class, and they all implement a version of the `encode` method.

For the program to run, it is necessary to instantiate all classes which represent an adaptive behavior. The creation and the initialization of these objects are declared by their constructors. For example, in Fig. 5, the instantiation of the Adpcm encoder was declared with a prediction value of 6 (for an available bandwidth superior to 48 kbits).

6.2. Adaptation Classes

Adaptation actions are encapsulated in an adaptation class, which represents the unit of adaptation of our approach. An adaptation class is applied to a Java class or extends some other adaptation class by inheritance.

Class structure changes. An adaptation class declares adaptive aspects of an existing Java class. The adaptation class compiler takes as input a Java program and a set of adaptation classes which controls how adaptation features are added to the program code. An adaptation class is applied to an abstract Java class; this abstract class corresponds to a set of behaviors in the form of subclasses, each subclass implements a concrete behavior. The adaptation class compiler creates a subclass of the abstract superclass which can switch between concrete behaviors depending on an execution context. Furthermore, the compiler inserts guards throughout the program code to notify the adaptive object of any context modification.

Fig. 8 shows a diagram of the classes used in Sect. 4.2 to build the adaptive `SoundEncoder` component. The `Encoder` class is the abstract class to be adapted; the `Gsm` and `Lpc` classes define concrete encoding algorithms; and the `RtcpController` class contains the execution context to be monitored. Fig. 9 shows the result of compilation. The compiler creates the `SoundEncoder` class as a subclass of the `Encoder` class, and references are created to access the concrete behaviors and to notify of any state changes in the `RtcpController`.

Method mapping. Each abstract method defined in the abstract superclass (`Encoder`) is implemented in the adaptive class (`SoundEncoder`) as a wrapper to a concrete

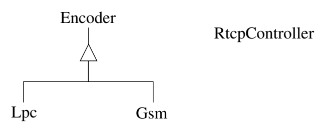

Figure 8. Before compilation

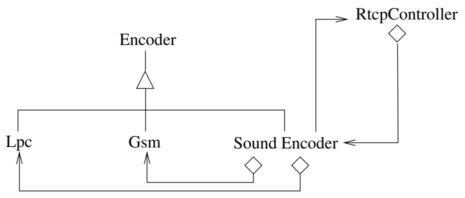

Figure 9. After compilation

method. This mapping is trivial since the methods have the same parameters.

State mapping. It may be necessary to share a variable between all adaptive behaviors. In our framework, we share a variable when it is defined in the abstract Java class or any of its superclasses. The sharing of these variables is implemented by copying shared state when a behavior is switched to, using the `acCopyEnv` method which is added by the adaptation class compiler (see [1] for details). As an example of shared variable, assume that one wants to count each call of the `encode` method of each encoder instance, one could add a variable `counter` to the class `Encoder`. This variable would be incremented each time an `encode` method is called. But when the encoding algorithm changes to adapt to the network bandwidth, if the compiler did not handle shared variables, one would have to copy the value of the variable from the current behavior to the new one.

Inheritance relation. Rather than directly adapting an existing Java class, an adaptation class can extend another adaptation class by inheritance. The inheritance relation is similar to that of an object-oriented language, enabling the definition of a hierarchy of increasingly refined adaptation classes. The use of inheritance in adaptation classes allows us to define a partial order between adaptive behaviors, which again makes it possible to define a decision tree on the adaptation conditions for selecting which behavior to use. Thus, we can add new behaviors in an incremental way, as was illustrated in Fig. 6.

When an adaptation class is extended, our compiler generates a subclass which extends the adaptive Java class. This new subclass overloads the condition adaptation. As an example, recall how the `SoundEncoder` adaptation class was extended with the declaration of an adaptation subclass `ExtendedSoundEncoder` in Sect. 5. This declaration caused the compiler to create a subclass `ExtendedSoundEncoder` of the class `SoundEncoder`.

7. Related work

Predicate classes. Predicate classes [7] are a syntactic and semantic extension of the Cecil language. They allow the applicability of a method to be declared via a *predicate expression*, which is a logical formula over class tests (i.e. testing that an object is a particular class or one of its subclasses) and boolean-valued expressions. Compared with our approach, predicate classes are intrusive. Declaration must be included directly in the code, and each method call requires the evaluation of the associated predicate, which is an on-change adaptation and thus not appropriate in all cases (e.g. multimedia).

Specialization classes. Specialization classes [18] are an extension of the Java language which, as our proposal, makes it possible to dynamically change the behavior of a component, depending on the execution context. These behaviors are however limited to specialized versions of a generic behavior, automatically generated by a program specializer, and corresponding to conditions on the context expressed by predicates parameterized by object fields. Adaptation classes do not automatically produce a specialized behavior, but adaptation classes could be used with specialization classes to provide adaptive components with specialized actions.

Aspect-Oriented Programming. Kiczales et al [13] describe many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some important features, called *aspects*, which cross-cut the basic functionalities of the program. This forces the implementation of these aspects to be scattered throughout the code, resulting in "tangled" code that is difficult to develop and maintain. Aspect-Oriented Programming (AOP) makes it possible to clearly express programs involving such aspects, including appropriate isolation, composition and re-use of the aspect code. A specific compiler, a weaver, injects the declared aspect in the program code. Our approach is a special case of AOP which expresses adaptation concerns. Using adaptation classes, we enrich the capacity of a component to dynamically change its behavior depending on an execution context.

Pluggable Composite Adapters. The pluggable composite adapter [14] is a new language construct which allows dynamic adaptation as component customization or component gluing in a non-invasive, encapsulated manner. The adaptation is dynamic in that a component can be replaced as a whole. There is no specific support for expressing behavior adaptation depending on the execution context.

8. Conclusion

In this paper, we have shown that, for designing adaptive components, ad hoc techniques can be complex and introduce communication overhead. Decomposition into different subcomponents makes code modification difficult because implementation of the adaptation is tangled due to resource and component dependencies.

To overcome this software engineering challenge, we have developed a declarative approach for the design and implementation of adaptive components. This approach separates adaptation declarations from the main program code. A compiler generates Java code that fully implements adaptation in the program and automatically instruments the code with an introspection mechanism that captures state changes. With adaptive declaration, the compiler generates optimized mechanisms for introspection and context switching mechanisms. Adaptation classes are integrated with the object oriented paradigm in the sense that they extend existing classes with adaptive behavior, and declare adaptation without disturbing the source program.

We are currently studying how to extend the expressiveness of adaptation classes to use them for static adaptation, i.e. adaptation before execution. By providing constraints on the values guarded by an adaptation class, the compiler can evaluate adaptation conditions and generate specific code which only includes the needed behaviors. As a result, static and dynamic adaptation classes offer an abstraction to define the adaptation time with respect to the context usage of adaptive components. For example, in order to use the sound encoder component with a network protocol which can guarantee a specific bandwidth (e.g. ATM), we would not need dynamic adaptation. The adaptation class compiler could generate a specific sound encoder component without any dynamic adaptation mechanisms. Static adaptation can eliminate dead code and further simplify the adaptation mechanisms, improving overall performance. Taking this idea even further, adaptation classes can be mixed with specialization classes, allowing each behavior to be specialized to given static constraints.

As for applications, we are studying the Java SWING library [19] to re-design the pluggable look-and-feel mechanism to allow the user to choose between a dynamically pluggable look-and-feel or a statically fixed look-and-feel. We believe that statically configuring this feature can improve the performance of the GUI.

Acknowledgments. We would like to thank the members of the Compose group (with special thanks to Ulrik Pagh Schultz) and the anonymous referees for their feedback.

References

[1] P. Boinot, R. Marlet, J. Noyé, G. Muller, and C. Consel. A declarative approach for designing and developing adaptive components. Internal Report 1337, Irisa, Rennes, France, June 2000.

[2] J.-C. Bolot, T. Turletti, and I. Wakeman. Scalable feedback control for multicast video distribution in the internet. In *ACM/SIGCOMM'94*, volume 24, pages 58–67, Oct. 1994.

[3] J.-C. Bolot and A. Vega-Garca. Control mechanisms for packet audio in the internet. In *IEEE Infocom'96*, Apr. 1996.

[4] M. Braux and J. Noyé. Changement dynamique de comportement par composition de schémas de conception. In J. Malenfant and R. Rousseau, editors, *LMO'99 - Langages et modèles à objets*, pages 147–162. Hermès, Jan. 1999.

[5] I. Busse, B. Deffner, and H. Schulzrinne. Dynamic QoS control of multimedia applications based on RTP. *Computer Communications*, Jan. 1996.

[6] C. Diot, C. Huitema, and T. Turletti. Multimedia application should be adaptive. In *IEEE HPCS'95*, pages 23–25, Aug. 1995.

[7] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP'98*, volume 1445 of *LNCS*. Springer-Verlag, 1998.

[8] J. A. Fingerhut. *LPC-10 speech coder software*. Washington University, http://www.arl.wustl.edu/∼jaf/lpc/.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[11] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[12] J. Jansen. *ADPCM Implementation*. ftp://ftp.cwi.nl/pub/audio/adpcm.shar.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, volume 1241 of *LNCS*. Springer-Verlag, June 1997.

[14] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers, 2000.

[15] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. *RFC 1889*, Jan. 1996.

[16] J. Scourias. *Overview of the Global System for Mobile Communications*. http://ccnga.uwaterloo.ca/∼jscouria/GSM/gsmreport.html.

[17] M. H. Sherif, D. O. Bowker, B. G., B. A. Orford, and G. A. Mariano. Overview and performance of CCITT/ANSI embedded adpcm algorithms. *IEEE Transaction on Communications*, 41, Feb. 1993.

[18] E.-N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *ACM OOPSLA'97*. ACM Press, Oct. 1997.

[19] K. Walrath and M. Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley, July 1999.