An Evaluation of the Adaptation Capabilities in Programming Languages [*]

Carlo Ghezzi, Matteo Pradella, Guido Salvaneschi
DEEPSE Group
DEI, Politecnico di Milano
Piazza L. Da Vinci, 32
Milano, Italy
{ghezzi, pradella, salvaneschi}@elet.polimi.it

ABSTRACT

In recent years the need for software applications to adapt to the environment in which they are situated has become common. Beside architectural approaches, language-level support for the development of adaptable and context-aware software have been investigated. Many existing solutions adopt ad hoc programming paradigms such as aspect oriented programming (AOP) or context oriented programming (COP). In this paper we investigate the use of the abstractions offered by traditional object-oriented and functional paradigms for the development of context-adaptable systems. We carry out our analysis along a set of conceptual directions which consider the requirement of functional adaptation beside non functional requirements such as safety and effective modularization. Our analysis were validated though the development of several prototypes of an adaptable cache server which is chosen as the running example through the paper. We provide an estimation of the performance advantages of the techniques based traditional programming languages compared with context-oriented programming.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Object-oriented Programming*
; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design

Keywords

Context, Self-adaptive software, Context-oriented programming

[*]This research has been funded by the European Community's IDEAS-ERC Programme, Project 227977 (SMSCom).

1. INTRODUCTION

Context-awareness in computing is a general term indicating a link between the system behavior and the environment in which a system operates. Emerging fields such as ubiquitous computing and pervasive computing [6] have made context-awareness a common requirement for software systems. Because of the dynamic nature of context, there is a tight relationships between context-awareness and dynamic adaptability. A software application is adaptable if it has the capability of changing its behavior dynamically (i.e. at run time) in response to a request for the fulfilling of a change in its requirements.

The development and the maintenance of adaptable software is challenging for several reasons. Architectures, middleware or languages must support dynamic behavior changing and such feature should be available to programmers in a usable and effective way. The adaptation code often crosscuts the system functionalities and careful engineering is required to manage the separation of concerns. Among the other solutions two programming techniques specifically address these problems. COP is a programming paradigm specifically thought for dynamic adaptation [21]. AOP has been proved useful in realizing the separation of concerns in adaptable systems at development time [27].

This paper investigates the support that most common programming paradigms, object-oriented and functional, offer for the development of adaptable and context-aware systems. Our intent was to probe how the abstractions offered by programming languages can be directly leveraged without resorting to special framework such as COP or AOP. We also evaluated only the direct employment of such abstraction, without considering the even reasonable option of the encapsulation in in *ad hoc* libraries.

We identified a set of dimensions which are significantly involved in context-adaptable applications and we used them to analyze the abstractions offered by different languages. The confidence in our analysis was reinforced through the development of several prototypes of an adaptable system in different languages. Each prototype uses different abstractions for the adaptability and context-awareness concerns. An adaptable cache server was chosen as a case study for the prototypes, which were developed by the students of a master course in advanced software engineering. The cache server is adopted as a working example for our analysis through the paper.

COP is a solution for context adaptation which supports abstractions specifically aimed to manage program adaptations. However these facilities comes at a cost. In order to estimate the performance overhead introduced by particular purpose paradigms, we implemented a set of benchmarks to compare the techniques discussed in this paper with functionally equivalent implementations using COP.

The remainder of this paper is organized as follows: Section 2 presents the related works, Section 3 describes the analysis framework which is used trough the paper. In Section 4 we discuss the language features that can be used to implement context-adaptable systems. In Section 5 we compare the performance of the native language implementations with COP. In Section 6 there are the conclusions and the future work.

2. RELATED WORK

McKinley et al. [23] review the technologies supporting compositional adaptation. Three key enabling technologies are identified: separation of concerns, component-based design and computational reflection [22]. An evaluation of the use of computational reflection as a programming language support for dynamic adaptability can be found in [16]. The problem of dynamic software adaptation to respond to context changes has been extensively tackled from a software architecture standpoint [14, 24, 30, 18, 29].

AOP has been extensively used as a support for the implementation of adaptable systems. This solution is applied in Sadjadi et al. [28] for the development of TRAP/J, a framework for the generation of adaptable Java programs. In TRAP/J, the need for dynamically activating a certain behavior in an application requires the use of AOP in conjunction with behavioral reflection. Dynamic AOP [25] is based on waving aspects at run time. The introduction of Dynamic AOP for adaptation has been proposed in the filed of autonomic systems by Greenwood [20]. This technique is employed for example by Baresi [12] AOP for a loose composition framework for autonomic systems and by Engel [17] for the development of an adaptable operating system kernel.

COP [21] has been proposed as an ad hoc approach to context adaptation and management. This goal is achieved through abstractions that enable application context-awareness without hard-wired conditional statements spread over the application code. Starting from the pioneering work on Lisp [15] many COP extensions have been developed for different languages such as Python Smalltalk, Ruby, JavaScript and Groovy. With the time this effort have extended to less dynamic languages such as Java [21, 9]. A fairly complete comparison of the existing languages with a performance evaluation of the available solutions can be found in [8]. The benefits of COP in the development of context-adaptable software has been demonstrated in the field of mobile service computing and ubiquitous applications [7] [11] and for desktop applications [10].

Although we are also working on adding COP features to an existing language (Erlang) [19], in this paper we wish to explore how far we can actually go by exploiting fairly standard and general linguistic mechanisms provided by non-specialized languages. The purpose is twofold. On the one side our analysis can provide guidance to designing ad-hoc mechanisms. On the other side, it can help the designers who need to use standard languages in adapting a systematic approach to structuring context-aware behaviors.

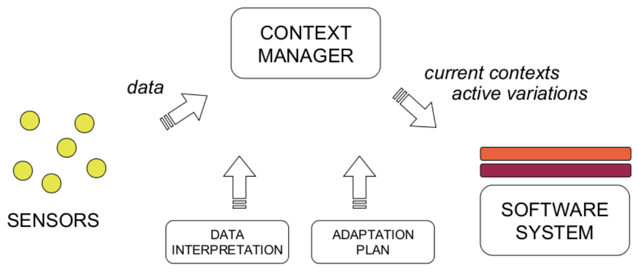

Figure 1: The reference conceptual model of a context-adaptable system.

3. ANALYSIS FRAMEWORK

In this section we introduce the conceptual model to which we refer throughout the paper. We describe the specification of the cache server prototype, we motivate the choice of the languages that were used for the implementation and we introduce the dimensions along with we conduct the analysis of Section 4.

3.1 Conceptual Model

The conceptual model of an adaptable system is shaped in Figure 1. We suppose that the system collects the data of interest through sensors. A *context manager* component receives data from sensors in the form of numerical observable and associate them to *contexts*. We refer as *context* the symbolic observable that captures a condition in their environment or in the system itself that is significant for adaptation.

At each instant of time a set of contexts can be active in the system. The mapping between data and active contexts is determined by the *data interpretation* specification. For example the *context manager* observes from sensor an average bandwidth usage of 100 Mb/sec, it decides on the base of the data interpretation specification that the*low-bandwidth-available* context is active. Each context is associated to one or more *variations* that must be applied to the software system in order to make it adapt to the active contexts. This mapping is expressed by an *adaptation plan*.

We refer as a *variation* each computational unit that can be atomically activated in order to change the behavior of the application. The implementation of the adaptations is strictly related to the system technology and to the language in use.

3.2 Prototypes Implementation

In order to evaluate the support that in practise is given by the abstraction supported by different languages, we developed several prototypes of an adaptable cache server. The specification we adopted was deliberately generic, with the goal of making it simpler to fit the requirements using the idiomatic abstractions of each language.

The server offers caching functionalities, speeding up the retrieval of information generated on other machines. Items stored in the cache are resources like dynamically created web pages, or serialized objects. The server exposes a set of primitives for resource store and retrieval, such as *lookup(key)*, *store(key,value)* and *remove(key)*. The cache server is *context-aware* and *adaptive* in the sense that the behavior of the server can be changed dynamically depending of the active contexts. The following functionalities are the possible server adaptations, i.e. can be activated and deactivated at run time:

Memory Constraint. The server starts performing optimizations aimed to reduce the RAM consumption, which is the critical resource.

Response Time Constraint. The server starts operating with the objective of being as fast as possible optimizing for short response. This requires to minimize the disk access and to make a strong use of the memory.

Low Bandwidth. The server starts minimizing the bandwidth consumption, maybe at the cost of a reduction in the quality of the service, e.g. no transmission of metadata associated to the stored items.

Privacy. The resources starts being ciphered before being stored in the cache and deciphered when requested.

Backup. The server starts saving the resources on a persistent redundant storage in order to be resilient to possible failures.

Security. The server must behave differently depending on the authentication status of the client. Client authentication must make the server adapt and start exposing its full functionalities.

Logging. The server is turned in *debug mode*: each significant operation is logged.

Each group of student was assigned a language and it was asked to realize a proof-of-concept prototype of the specification above. The goal was to demonstrate the support offered by each language in the implementation of the adaptive concerns of the specification. The students were provided with some language-specific features to explore and then left free to use other features that could fit the assigned problem. On the whole, 12 prototypes were developed, with an in-depth investigation of several language features. In Section 4 we report the most interesting results.

3.3 Languages

The languages we chose for our analysis and for the implementation of the prototypes are Erlang [1], Python [3], Ruby [4], F# [2] and Scala [5]. We believe that these languages offer a good test suite for our comparison. They cover all the most widespread programming paradigms, procedural, object oriented, and functional, they also cover both static and dynamic typing. They offer a vast set of different features, which cover all the major features of modern programming languages: classes, objects, interfaces, traits, modules, multiple inheritance, parametric and ad hoc polymorphism, dynamic dispatching, first class functions, closures, algebraic data types, patter matching, eval functions. Erlang is the oldest language in the set, dating back to the '80s, Python and Ruby have been released in the '90s, while F# and Scala have appeared in the last decade. This has an impact on the features that each of them supports and on the general programming style enforced by each language. In the following we briefly point out the characteristic aspects of each of the languages in our test suite.

Erlang. It is an interpreted functional language, with single assignment, and dynamic typing. It has an agent-based concurrency model with native support for fault tolerance and distribution.

Python. It is an interpreted programming language supporting multiple programming paradigms: object oriented, imperative and functional. It is dynamically typed and it supports multiple inheritance and metaprogramming.

Ruby. It is a dynamically-typed object-oriented interpreted programming language. It supports single inheritance and mixins. Metaprogramming is idiomatic and common.

Scala. It is a general purpose programming language integrating features of object-oriented and functional languages. Scala runs on the Java VM and it is compatible with Java libraries or existing application code. It is a statically typed language with type inference, algebraic data types, traits, agent-based concurrency model.

F#. It is a multi-paradigm programming language that encompasses the functional and the object-oriented styles. It targets the .NET Framework and it is heavily influenced by OCaml. It is a strongly typed language with type inference, algebraic data types, and single inheritance.

3.4 Dimensions

In the following we delineate the dimensions that we considered in our analysis. They take into account both aspects related to the adaptation process, such as the dynamic variations *activation* and *combination*, and non functional concerns such as variations modularization.

Abstraction. This dimension defines which language abstraction are used to implement variations and the basic behavior. We consider only native abstractions without the emulation through the use of other constructs or libraries.

Generic code. A key point in the development of context-aware systems is to allow programmers to write code whose behavior is context-dependent, without that context adaptation is explicitly triggered in each location of the control flow where it should occur. This dimension captures how the code is written in order to make adaptation transparent to the programmer.

Variation combination. Since multiple contexts can be active in the system at the same time, more than one variation can be active. If the active variations affect the same functionalities of the software, they must be combined together. Static combination is performed when the system is built up, and require to foresee all the possible configurations in advance. Dynamic combination is done when the application is running. This dimension captures how variations are combined together and with the basic behavior in order to determine the final behavior of the system.

Variation activation. This dimension pinpoints the mechanism through which variations are activated and start affecting the behavior of the system. For example a variation can be activated assigning an object to a reference or passing a parameter to a function.

Modularization. This dimension captures how variations and the basic behaviors are organized in the code base. Such dimension can be strictly related to the abstractions with which variation are expressed. For example in object-oriented languages classes are both a language abstraction and the basic code modularization unit.

Safety. This dimension captures how the occurrence of errors associated to context-adaptation is avoided. In many cases the compiler in statically typed languages provides guarantees against the uprise of adaptation-related errors.

4. LANGUAGE FEATURES

4.1 Inheritance and Subtyping

The object-oriented paradigm is by far the most spread in software development. For this reason we firstly address the point of how the common features of the object-oriented languages can be used to develop context-adaptable software. Since inheritance and subtyping have a central role in structuring object-oriented programs it is interesting to discuss how a context adaptable system can be designed leveraging these features.

One of the possible uses of the inheritance in programming languages is to support reuse of an implementation. An entity B in a program *inherits* form an entity A if a set of functionalities of B are taken from A. Inheritance between classes is the fundamental mechanism for code reuse provided by object-oriented languages. Subtyping, instead, refers to interface compatibility: a class B is a subtype of a class A if an instance of A can be replaced by an instance of B. In most object-oriented languages these orthogonal concepts coincide since classes are used for inheritance relationships as well as for (sub)type declarations.

Since classes are the main modularization facility in object-oriented languages, it is reasonable to adopt such abstraction for the implementation of variations. We suppose that the basic behavior is implemented in a class A, while the variation is implemented with a class $A1$ that inherits from A. In object-oriented languages a class can modify the behavior exposed by its superclass overriding some of the superclasses' methods. This does not necessary mean complete method substitution, since from inside the overriding method a call to *super* can execute the overridden (i.e. the superclass') implementation. The use of *super* allows to modify the behavior of a method by adding operations before or after its execution instead of completely replacing it.

Abstraction. Variations are implemented as classes.

Generic code. Generic code is written leveraging reference polymorphism. For example generic code uses references of (static) type A to which can be assigned instances of subtypes of A, such as $A1$. Reference polymorphism guarantees uniform management of both adapted and non adapted objects.

Variation combination. Variation combination is achieved by dynamic method dispatching, which chooses the subclass implementation of a method if available, and through the use of *super* calls to the parent class method.

Variation activation. Variation activation is given by the instantiation of the class implementing the variation, instead of the base class. This solution allows to perform the adaptation only at instantiation time, which limits the overall adaptability of the application.

Modularization. Variations and basic behaviors are implemented with classes, which are usually the main modularization unit in object-oriented languages.

Safety. In languages with static type system the compiler assures that all the functions callable on the basic objects are callable also in the adapted objects and all the *super* calls have a corresponding implementation in the supertype.

The effect of single or multiple inheritance deserves to be carefully discussed since it have strong consequences on code replication. In languages with single inheritance, classes are allowed to reuse the code of only one existing class. This limits the use of this feature for the implementation of adaptable systems.

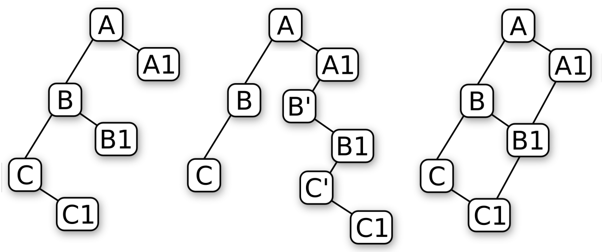

Figure 2: The inheritance hierarchy for the adaptation to a single context. A naive implementation single inheritance (left), the functionally correct one (center) single inheritance, and the case with multiple inheritance (right).

Suppose that the inheritance hierarchy of an application is structured as follows: a class A is subclassed by a class B, which is in turn subclassed by C. The software normally operates in a basic condition with no active context; occasionally an adaptation must be performed to the context *context1*. When *context1* is active, the class $A1$ is instantiated instead of A. Now consider how to design the class hierarchy that can model the system. A first attempt can be to preserve the $A - B - C$ chain and add to each of this classes the associated adapted class (Figure 2 left). However, with this approach, if class $B1$ is instantiated in an object $b1$, $b1$ does not implement the functionalities associated to the adaptation of A to the context *context1*, although $b1$ is of class A and therefore should expose also A's part of adaptation. This issue is correctly addressed by the hierarchy in Figure 2 (center). In this case the class $B1$ indirectly inherits from the class $A1$. Therefore the instantiation of $B1$ results in an object that implements not only the variation associated to B, but also the variation associated to A.

However another problem arises. Since the activation of $B1$ implies the activation of $A1$, $B1$ must inherit form $A1$. But $B1$ already inherits from B. Due to the single inheritance, the only solution is to replicate the code of B in a class B'. Now $B1$ inherits from B' (a replication of B) which in turns inherits from $A1$.

Now consider the case with multiple inheritance as in Figure 2 (right): variations inherit from their "natural" parent and from the variation of the parent. We assume a linearization algorithm that gives precedence to left parents before the right ones and to children before parents, so that the linearization starting from $C1$ is $C - B1 - B - A1 - A$. It is evident that in this case there is no need for replication.

The problem of code replication is exacerbated in the presence of multiple contexts. Consider the case of single inheritance and two possible contexts, *context*1 and *context*2. All the combinations must be considered, such as the activation of both the $A1$ and the $A2$ variation, and the activation of $A1$ or $A2$ only. This leads to a combinatorial explosion, and to heavy code replication. For example the class B is replicated four times. (Figure 3) Also in this case multiple inheritance solves the problem leading to the hierarchy in Figure 4. Dots represent classes that do not need to define any code, since they inherit all the functionalities from other classes.

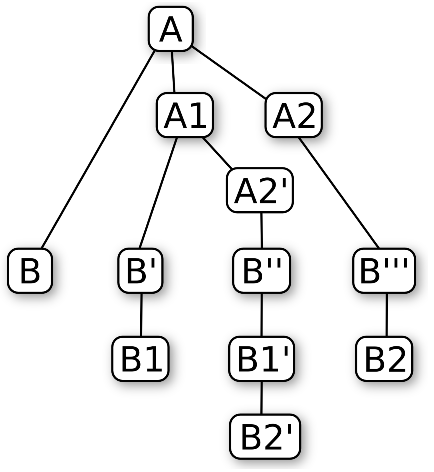

Figure 3: The inheritance hierarchy for the adaptation to two different contexts in the case of single inheritance.

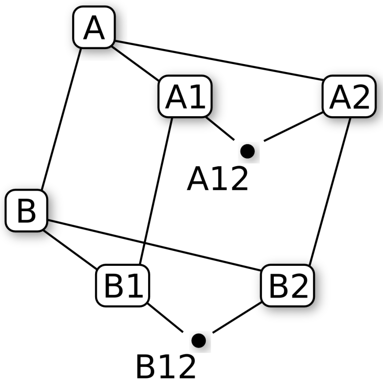

Figure 4: The inheritance hierarchy for the adaptation to two different contexts in the case of multiple inheritance.

In Figure 5 and 6 we show an example of how multiple inheritance can be used for the software adaptation and as a solution to the aforementioned issues. Scala allows to define *traits*, which are an implementation of mixins [13]. Scala traits are added in an object when it is instantiated, augmenting its functionalities. Since traits are a mechanism orthogonal to the Scala single class inheritance and an object can be augmented an arbitrary number of traits, they introduce a way to obtain an analogous of multiple inheritance in the language.

The basic behavior of the cache server is implemented in the `CacheServer` class, which exposes the `lookup` and the `store` methods. The `CryptoVar` trait implements a variation which adds cryptographic functionalities to the `CacheServer` class, overriding the `lookup` and `store` methods. Now suppose that a version of the server is developed which is specific for text files. This specialized version allows for searching the matches of a regular expression in the cached files. The `TxtCacheServer` trait specializes the `CacheServer` class implementing the searching algorithm in the `searchCache` method.

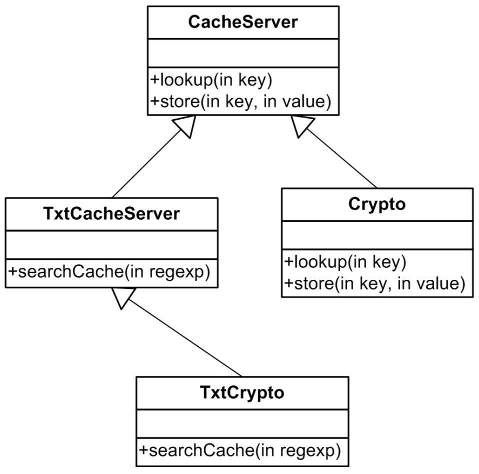

Figure 5: UML diagram of the cache server example implemented in Scala.

In case the cryptographic protection is activated on the content of the cache, it is no longer possible to search for a regular expression matching inside cached files, but files must be decrypted on-the-fly to perform the searching. This feature is a variation of the textual cache server basic behavior, and is implemented in the `CryptoTxt` which overrides the `search` method in `TxtCacheServer` doing the on-the-fly decryption.

The `performOperation` method in the application contains the generic code which performs the method calls on the cache server object. In `normal` context, a `CacheServer` is instantiated with the `TxtCacheServer` trait. When the `Crypto` context is active, a `CacheServer` is instantiated with the `TxtCacheServer` trait and the variations associated to the `Crypto` context, i.e. the `CryptoTxt` and the `CryptoVar` are added. With respect to Figure 2 (right), `CacheServer` is the class implementing the original behavior and maps to the A class, `TxtCacheServer` is an extension of the basic behavior and maps to the class B, `CryptoTxt` is a variation of `CacheServer` and maps to A1, `CryptoVar` is a variation of `CacheServer` and maps to B1.

A comparison between Figure 2 (right) and Figure 5 shows that the inheritance relationship between the `CryptoTxt` trait and the `Crypto` trait is not statically enforced. However the effect of this lacking link is obtained by adding traits when the server is instantiated. In fact the dispatching order obtained by trait addition is the same that would come out from applying a linearization algorithm to the tree with `CryptoTxt` inheriting from both `TxtCacheServer` and `Crypto`: `CryptoTxt-TxtCacheServer-Crypto-CacheServer`. Note that in Scala when traits are added to an object through the `with` keyword, traits on the right come first.

4.2 First Class Functions

Functions are an abstraction available in practically any programming language. They are cornerstone constructs in the case of functional languages. Therefore it is natural to explore their use as a solution to the issue of the development of context-aware software.

A programming language is said to support first class functions if functions are values in the computation. In these languages functions can be created during the execution (via

```
object Main extends Application {
  def performOperations(ctx :String) = {
    val cs = ctx match {
      case "Normal" =>
        new CacheServer() with TxtCacheServer
      case "Crypto" =>
        new CacheServer() with CryptoTxt
                  with TxtCacheServer with  CryptoVar
    }
    cs.store("key1","val1")
    cs.lookup("key1")
    cs.searchCache("regexp")
  }
  // In real world ask the context to a context manger
  performOperations("Normal")
  performOperations("Crypto")
}

class CacheServer {
  def lookup(key :String) :String = {
    // search for the value
  }
  def store(key :String, value :String) =
     // store the value
}

trait CryptoVar extends CacheServer {
  override def lookup(key :String) :String = {
    val value = super.lookup(key)
    // Value decryption ...
  }
  override def store(key :String, value :String) = {
    val encrypted = // encrypt value
    super.store(key, encrypted)
}}

trait TxtCacheServer extends CacheServer {
  def searchCache(regexp :String) = {
    // Search ...
}}

trait CryptoTxt extends TxtCacheServer {
  override def searchCache(regexp :String) = {
    // Search with decryption...
}}
```

Figure 6: The cache server Scala implementation leveraging traits for context adaptability.

lambda expressions), passed as arguments to other functions, used as return values and stored in data structures. Some programming languages also offer specific operators for function combination. For example the composition operator » in F#, given two functions g and h, returns a function f which performs the subsequent application of g and h.

Functions can be used to implement the different basic behaviors of the application and to modularize the associated variations. If the language does not allow to pass functions as parameters, choosing at run time the function to execute according to the active context must be done explicitly with an if chain, which results in poor engineering. First class functions are needed for the dynamicity that is required in adaptable software. In fact code can be made generic through references to functions which are assigned only at runtime. Moreover functions representing basic functionalities and functions representing variations can be dynamically combined in order to obtain the needed behavior.

Abstractions. Functions are used to represent variations and basic behaviors in the program.

Generic code. Program code manipulates function references passing them around and calling them when needed. The specific implementations are bounded to the references only at run time.

Variation combination. Variations can be combined through the combination of the functions that actually implement them. For example if two variations are implemented in the functions `var1(par)` and `var2(par)` and the basic behavior is the function `base(par)`, the combination of the two variations with the basic behavior is given by `base(var2(var1(par)))`.

Variation activation. Variations activation is done by binding a function body to the function reference used in the generic code.

Modularization. Functions usually are not the fundamental modularization entity in a program, since usually they are included in a bigger modularization unit such as classes or modules. In the development of context-aware software these coarser-grain entities can be used to group all the functions associated to the same context.

Safety. Strongly typed languages guarantee type consistency in the use of function references (assignment and function call) and in function combination.

Figure 7 shows an example use of first class functions to support software adaptability. The cache server can activate logging and cryptography with the backup of the stored items. The `operateOnCahce` function contains the generic code of the application. The `store` function reference used by `operateOnCache` is bound to the implementation returned by `variationCombinator`.

The function `variationCombination` combines the variations trough the » operator on the bases of the active context. The available variations are the functions `loggingVar`, `cryptoVar` and `backupVar`, and the basic behavior is the function `basicStore`. When the `SimpleContext` context is active, only the `basicStore` function is used, which implements the basic store functionalities. When the `DebugContext` is active, a logging facility is added to the store operation combining the `loggingVar` function with the `basicStore` function. Finally, when the `SafeContext` is active, the `backupVar`, `cryptoVar` and `basicStore` are combined to obtain the `store` function to be executed in the generic code.

4.3 Modules as Values

Modules are language entities that help improving the separation of concerns and the maintainability. Modules interact with the rest of the program through declared interfaces which express *provide* and *require* relationships with other modules.

Some languages, such as Erlang, support modules as values: modules can be directly involved in the computation, being assigned to variables or passed to a function. More importantly, fully qualified function calls (i.e. function calls which explicitly specify the module from which the function is taken) can be performed using a variable as a the module. For example in Erlang the expression `Mod:fun1()` denotes a call to the function `fun1` in the module bound to the `Mod` variable.

Modules can support the development of context-aware systems in that the functions implementing the system functionalities, in the flavor suitable for a certain context, can be grouped in the same module. The right module is that

```
type Context =
    DebugContext | SafeContext | SimpleContext


let variationCombinator activeContext =

    let loggingVar(key, value) =
        // logging...
        (key, value)

    let cryptoVar(key, value) =
        // do encription ...
        (key, "crypted" + value)

    let backupVar(key, value) =
        // send to backup server ...
        (key, value)

    let basicStore(key, value) =
        // save in cache ...

    match activeContext with
        | DebugContext -> loggingVar >> basicStore
        | SafeContext ->
backupVar >> cryptoVar >> basicStore
        | SimpleContext -> basicStore


// The generic code using the store function
let operateOnCache store =
    // do stuff ...
    store ("key1", "val1")
    // do stuff ...


// Execute the generic code in a given context
// In real world context is asked to a ContextManager
operateOnCache (variationCombinator DebugContext)
operateOnCache (variationCombinator SafeContext)
operateOnCache (variationCombinator SimpleContext)
```

Figure 7: Context adaptation using first class functions in F#.

chosen accordingly to the active context. Modules as values allow to write generic code in which calls are performed on a module variable which is late-bound to a module chosen on the bases of the context.

Abstraction. Modules are used to implements either a variation or a basic behavior. All the functions inside a certain module are associated to that variation or to the basic behavior.

Generic code. Generic code is written parametrically with respect to the modules on which functions are called. In particular, context-aware functions are called on a module variable, dynamically assigning the actual value of the variable (i.e. the module on which the call is to be done).

Variation combination. Functions inside variation modules can call the functions in other variation modules or in a basic behavior module adding computation before and after the call. This is a form of static combination. Dynamic combination is obtained by using modules as values also inside the variation modules in the same way of top-level context-aware calls. The calls to other variation modules are parametrized with respect to the module on which they are performed and the module must be chosen depending on the current context. This of course requires to propagate the context information into the variation modules.

Variation activation. Variations are activated by binding the module that implements them to the variable on which the context-aware calls are performed. This binding can be changed at any time in the control flow of the program.

Modularization. Modules are the fundamental coarse-grain code unit of many programming languages. For this reason using modules to implement variations appears to be an effective solution which increase concerns separation and maintainability.

Safety. Dynamically typed languages such as Erlang do not give any assurance that the module variable on which a function is called is bounded to a module that really contains that function. In statically typed languages, this kind of safety can be enforced by the compiler.

In Figure 8 we show an example of the use of module values for the implementation of adaptability in Erlang. The `client` module, which calls the API of the cache server, shows how the generic code is written. The calls to the server API are performed on the `Mod` variable which is late-bound to a module according to the current context. The cache server can operate in `save_memory`, `normal` or `debug` context. In the first lines of the `client` module, the proper variation is activated assigning to the `Mod` variable the variation or basic module associated to the active context. In `normal` context the cache server keeps the cached items in memory and uses the basic behavior `high_memory`. When the memory is scarce (i.e. the `save_memory` context is active), the server starts using the disk and the `low_memory` variation is activated. The server also supports a `debug` context which activates the `logging` variation. Note that the logging variation adds the logging facility to the basic behavior and therefore it calls in turn the `high_memory` module to perform the `store` and `lookup` operations.

4.4 Dynamic Object Adaptation

In languages such as Java or C++ the attributes of an object are determined by the class of which that object is an instance. In such languages the structure of the objects (methods and fields) is fixed by its class and cannot be modified at run time. More dynamic languages like Python and Ruby do not have such constraint, allowing to change the structure of an object during the execution.

For example in Python, objects are dictionaries of key-value pairs, where the key is a method signature and the value is the function implementing the method body. When a method is called on the object, the dictionary is searched for an entry whose key correspond to the method signature. When the entry is found, the associated implementation is executed.

The possibility of dynamically changing the methods implemented by an object can be leveraged in the perspective of context adaptation. The modification of the objects' dictionaries allows to change the behavior of the application with an extremely fine granularity. Although this is a powerful feature, it is hardly exploitable *per se* in building up a context-aware application. Because of the low level granularity of this adaptation mechanism, single method changes would be scattered around the application code, which results in poor engineering. An obvious solution is to wrap the use of such functionality inside a library which atomically performs all the dictionary modifications required by a

```
-module(client).                  -module(logging).                 -module(high_memory).     -module(low_memory).

operate(ActiveContext) ->         lookup(Key) ->                    lookup(Key) ->            lookup(Key) ->
  Mod = case ActiveContext of       % Logging ...                     % Perform the lookup.     % Lookup value for Key
    save_memory -> low_memory;      high_memory:lookup(Key).
    normal -> high_memory;                                          store(Key,Value) ->       store(Key,Value) ->
    debug -> logging              store(Key,Value) ->                 % Store the value         % Store (Key,Value)
    end,                            % Logging ...
  Mod:store("key1","val1"),         high_memory:store(Key,Value).   ...                       ...
  % do stuff ...
  Mod:lookup("key1").               ...
  % do stuff ...
```

Figure 8: Adaptation through function calls on parametric modules in Erlang.

context change and exposes a higher level API to the user.

In the spirit of investigating the native constructs that can be *directly* exploited for context adaptation, we chose to organize the structural change of the objects accordingly to templates, i.e. classes. In Python it is possible to dynamically change the class of an object. This allows to use classes as a modularization unit, grouping methods that implement the object behavior in a certain context and which together must be dynamically injected into the object.

Abstraction. Variations are represented as classes that collect methods associated to a certain context condition.

Generic code. References to dynamically adaptable objects are passed around like normal reference are, and method invocation is performed as usual.

Variation combination. Calls to parent classes allow subclasses to add behaviors before and after the call to the superclass. This mechanism allows to arrange the class hierarchy in order to obtain the desired method combinations. This combination occurs when the application is developed. Since Python allows to define classes at run time, the parents of a class and therefore of the adaptable object, can be chosen dynamically. This gives the capability of carrying out run time variation combination.

Variation activation. A variation is activated by assigning to an object the class that implements the variation. Variation deactivation is implemented by restoring the original object class. Variation (de)activation can occur in any time along the control flow, and it is not limited to instantiation time, with the variation that remains active for the whole object's lifetime (see Section 4.1).

Modularization. A class contains all the methods associated to a certain variation.

Safety. Since this mechanism dynamically changes the type of the object on which the adaptation is performed, no guarantee can be given on the safety of methods calls that are performed. Therefore, for example, a call to a non existing method on the adapted object fails at runtime. Classes that implement variations must be designed carefully in order to avoid this type of errors.

In Figure 9 we show the dynamic activation of the backup functionality on the cache server. The class `CacheServer` defines a `store` and a `lookup` methods. The `adapt` method simply wraps the class change of the `CacheServer` object. The class change is carried out by assigning the `__class__` attribute of the object. The class `BackupServer` implements a `store` method which performs the backup before delegating the real store to the parent `CacheServer` class. When the `adapt` function is called on the `srv` object, the type of `srv` is changed from `CacheServer` to `BackupServer`, and `srv` starts exposing the backup functionalities.

```
class CacheServer(object):
    def store(self, key, value):
        # Store value

    def lookup(self, key):
        # Lookup value for key
        return value

    def adapt(self, newClass):
        self.__class__ = newClass


class BackupServer(CacheServer):
    def store(self, key, value):
        # Backup (key,value)
        CacheServer.store(self, key, value)

# Generic code
srv = CacheServer()
def run():
    srv.store("key1","val1")
    srv.adapt(BackupServer)
    srv.store("key1","val1")
```

Figure 9: Adaptation of a single object changing the methods it implements via dynamically switching the object class.

5. COMPARISON WITH COP

Over the years, several languages have been extended with contextual features. Each language supports the fundamental COP concept of language-level context adaptation in a way which is specific of its programming model. For this reason, COP languages constitute a family in which each implementation comes with its own constructs and peculiarities.

In this paper we consider the most common COP model, which allows to define variations as partial method definitions such as *before*, *after* and *around* methods. Such definitions are declared inside first class entities called *layers*. The `with(`*activeLayers*`){`*codeBlock* `}` statement activates in the execution of *codeBlock* the variations declared inside *activeLayers*. The `proceed` function is similar to the `super` call in Java and executes the implementation of the method in the next active layer or the original method if there are no more active layers. In the following we analyze the COP mecha-

```
backupLayer = layer("backupLayer")

class CacheServer(object):
    @base
    def store(self, key, value):
        fact(F_CONST)
    @base
    def lookup(self, key):
        fact(F_CONST)
    @around(backupLayer)
    def store(self, key, value):
        fact(F_CONST)
        proceed(key, value)

a = CacheServer()

def run():
    a.store("key1","val1")
    with activelayer(backupLayer):
        a.store("key1","val1")
```

Figure 10: The cache server implemented in ContextPy.

nism using the dimensions identified in our framework.

Abstraction. Variations are implemented as layers constituted of partial method implementations. Layers are first class entities specifically introduced for context adaptation.

Generic code. Generic code is placed inside the scope of the `with` statement. Method calls are dispatched according to the active variations .

Variation combination. If several layers are active at the same time, the partial method definitions inside each of them are combined with the original methods.

Variation activation. Variations are activated using the `with` statement. The activation is dynamically scoped to the code block.

Modularization. COP provides different solutions: the most noticeable are the modularization along layers (*class-in-layer*) or along classes (*layer-in-class*). For example ContextPy [31] obey the layer-in-class discipline, because layers are declared in the syntactic scope of classes (Figure 10). Other cop languages such as ContextL [15] allows to define layers apart from the classes they augment.

Safety. Statically typed COP languages guarantee that the method combinations triggered by layer activations are type safe.

5.1 Performance

The examples presented in this paper were all reimplemented in Python and in ContextPy, taking advantage of the context-related abstraction offered by this language. As an example, Figure 10 shows the COP version of the prototype in Figure 9, implemented in ContextPy using layers. The reason why we chose Python is twofold. On the one hand, Python is a multiparadigm language that supports all the features presented in this paper. On the other hand, its COP counterpart ContextPy is a relatively fast implementation [8] among mainstream languages supporting all the required features.

The experimental results are shown in Figure 11. Each line correspond to a prototype. The data refer to 10^5 executions of a combination of variations activations and methods calls, like `run` in Figure 10. Since COP features mainly introduce performance penalties on method dispatching, we repeated the experiments with different amounts of computations carried out inside each method. The t_{pass} column corresponds to empty method bodies. The t_n columns show the results when a method body computes the factorial of n. To keep the examples of the previous sections unchanged, the comparison only makes sense between each example and its layer-based implementation, since the examples have a different amount of functions/methods involved. The *fn* column gives the number of method calls for each benchmark execution. Since each benchmark is executed 10^5 times, in the case of the benchmark in Figure 10 (line 7 of Figure 11) we have a total amount of $3 \cdot 10^5$ calls to the `store` method, including the associated around method.

	t_{pass}	t_1	t_{10}	t_{100}	*fn*
Inheritance	0.53	1.33	1.90	21.4	9
Layers	5.01	5.64	6.76	26.8	
First class functions	0.44	0.75	1.18	14.45	6
Layers	4.24	4.82	5.26	18.54	
Modules as values	0.35	0.82	1.42	18.35	8
Layers	5.44	6.04	6.75	24.08	
Dynamic obj. adapt.	0.25	0.47	0.75	7.10	3
Layers	1.95	2.22	2.44	8.87	

Figure 11: Performance comparison between COP and native language implementations. All values are expressed in seconds.

Our findings highlights the significant overhead of COP implementations, that at worst can be one order of magnitude slower than the native languages counterparts. Such poor performances are probably due to the fact that COP languages are still immature and were developed without optimization in mind. However this result makes even more interesting the systematic investigation of the use of traditional language constructs in the development of context adaptable systems.

6. CONCLUSION AND FUTURE WORK

In this work we analyzed the support given by the abstractions offered by different programming languages for the development of context-aware adaptable software. To carry on our investigation we proposed a set of dimensions which we believe can characterize the fundamental concerns of language-level adaptation. Our purposes for the future point to two directions. On the one hand we plan to develop new dimensions to take into account elements that in this paper have been neglected. On the other hand we intend to analyze with the framework delineated in this paper also the use of programming paradigms such as AOP, COP, or implicit invocation (II) [26] for the development of adaptable software. We expect that this work will require a generalization of the dimensions identified so far and the specification of new dimensions.

7. REFERENCES

[1] *http://erlang.org.* Reference website for open-source Erlang.

[2] *http://research.microsoft.com/fsharp.*

[3] *http://www.python.org/.*

[4] *http://www.ruby-lang.org/.*

[5] *http://www.scala-lang.org/.*

[6] G. D. Abowd and E. D. Mynatt. Charting past, present, and future research in ubiquitous computing. *ACM Trans. Comput.-Hum. Interact.*, 7:29–58, March 2000.

[7] M. Appeltauer and R. Hirschfeld. Explicit language and infrastructure support for context-aware services. In *GI Jahrestagung (1)*, pages 164–170, 2008.

[8] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A comparison of context-oriented programming languages. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM.

[9] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara. Contextj: Context-oriented programming with java. In *Proceedings of the JSSST Annual Conference 2009*, Shimane University, Matsue, Shimane, Japan, September 16, 2009.

[10] M. Appeltauer, R. Hirschfeld, and H. Masuhara. Improving the development of context-dependent java applications with contextj. In *International Workshop on Context-Oriented Programming*, COP '09, pages 5:1–5:5, New York, NY, USA, 2009. ACM.

[11] M. Appeltauer, R. Hirschfeld, and T. Rho. Dedicated programming support for context-aware ubiquitous applications. In *Proceedings of the 2008 The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 38–43, Washington, DC, USA, 2008. IEEE Computer Society.

[12] L. Baresi and G. Tamburrelli. Loose compositions for autonomic systems. In C. Pautasso and E. Tanter, editors, *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 165–172. Springer Berlin / Heidelberg, 2008.

[13] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA'90*, pages 303–311. ACM Press, 1990.

[14] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee. Software engineering for self-adaptive systems: A research road map. In *Dagstuhl Seminar Proceedings*, volume 8031. Springer, 2008.

[15] P. Costanza. Language constructs for context-oriented programming. In *In Proceedings of the Dynamic Languages Symposium*, pages 1–10. ACM Press, 2005.

[16] J. Dowling, T. Schäfer, V. Cahill, P. Haraszti, and B. Redmond. Using reflection to support dynamic adaptation of system software: A case study driven evaluation. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering, Papers from OORaSE 1999*, pages 169–188, London, UK, 2000. Springer-Verlag.

[17] M. Engel and B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 51–62, New York, NY, USA, 2005. ACM.

[18] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[19] C. Ghezzi, M. Pradella, and G. Salvaneschi. Programming language support to context-aware adaptation - a case-study with Erlang. *Software Engineering for Adaptive and Self-Managing Systems, International Workshop, ICSE 2010.*

[20] P. Greenwood and L. Blair. L.: Using dynamic aspect-oriented programming to implement an autonomic system. Technical report, Proceedings of the 2003 Dynamic Aspect Workshop (DAW04 2003), RIACS.

[21] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), Mar. 2008.

[22] P. Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 147–155, New York, NY, USA, 1987. ACM.

[23] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37:56–64, July 2004.

[24] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[25] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, AOSD '02, pages 141–147, New York, NY, USA, 2002. ACM.

[26] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In J. Vitek, editor, *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference, Paphos, Cyprus*, volume 5142 of *Lecture Notes in Computer Science*, pages 155–179, Berlin, July 2008. Springer-Verlag.

[27] S. M. Sadjadi, P. K. McKinley, and B. H. C. Cheng. Transparent shaping of existing software to support pervasive and autonomic computing. In *Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, DEAS '05, pages 1–7, New York, NY, USA, 2005. ACM.

[28] S. M. Sadjadi, P. K. Mckinley, B. H. C. Cheng, and R. E. K. Stirewalt. TRAP/J: Transparent generation of adaptable java programs. In *In Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04), Agia*, 2004.

[29] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: a combined approach to self-management. In *SEAMS '08: Softw. eng. for adaptive and self-managing systems*, pages 1–8, New York, NY, USA, 2008. ACM.

[30] R. N. Taylor, N. Medvidovic, and P. Oreizy. Architectural styles for runtime software adaptation. In *WICSA/ECSA 09*, 2009.

[31] M. von Löwis, M. Denker, and O. Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 143–156. ACM Digital Library, 2007.