Subjective-C
Bringing Context to Mobile Platform Programming

Sebastián González, Nicolás Cardozo, Kim Mens,
Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux

Computing Science Engineering Pole, ICTEAM, UCLouvain
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve, Belgium

Abstract. Thanks to steady advances in hardware, mobile computing platforms are nowadays much more connected to their physical and logical environment than ever before. To ease the construction of adaptable applications that are smarter with respect to their execution environment, the context-oriented programming paradigm has emerged. However, up until now there has been no proof that this emerging paradigm can be implemented and used effectively on mobile devices, probably the kind of platform which is most subject to dynamically changing contexts. In this paper we study how to effectively realise core context-oriented abstractions on top of Objective-C, a mainstream language for mobile device programming. The result is Subjective-C, a language which goes beyond existing context-oriented languages by providing a rich encoding of context interdependencies. Our initial validation cases and efficiency benchmarks make us confident that context-oriented programming can become mainstream in mobile application development.

1 Introduction

New computing platforms are interrelated to their physical execution environment through all kinds of sensors that are able to measure location, orientation, movement, light, sound, temperature, network signal strength, battery charge level, and so forth. At the logical level, even traditional desktop and server platforms are getting exposed to richer environments in which they can find network services of all sorts. Both at the physical and logical levels, the live environment in which applications execute is acquiring a central role. If equipped with higher levels of context-driven adaptability, software systems can become smarter with respect to their environment and to user needs, exhibit emergent properties, be resilient in the face of perturbations, and generally fit better in the technical ecosystem in which they are used.

Unfortunately, most software systems do not meet the high adaptability expectations that stem naturally from their connectedness to their environment. Most applications exhibit fixed functionality and are seldom aware of changing contexts to adapt their behaviour accordingly. Many chances of delivering improved services to users and network peers are thus missed. We hypothesise that a major reason for this lack of adaptability is the unavailability of appropriate context-aware programming languages and related tool sets. Current programming technology does not put programmers in the right state of mind, nor does it provide adequate abstractions, to program context-aware applications.

Starting from this observation, Context-Oriented Programming (COP) has been introduced as a novel programming paradigm which eases the development of adaptable behaviour according to changing contexts [4,9]. COP offers an alternative to hard-coded conditional statements and special design patterns to encode context-dependent behaviour. COP thereby renders code more reusable and maintainable. Unfortunately, current COP languages do not run on mobile platforms —probably the kind of platform for which context is most relevant, thus offering the most promising possibilities for development of context-aware applications. Furthermore, COP languages still lack dedicated facilities to model the knowledge of the situation in which applications execute.

With the aim of having a COP language that runs on a mobile platform, we set out to develop an extension of Objective-C, one of the most widespread programming languages for mobile systems. The result is Subjective-C, a new COP language extension aimed at easing the construction of context-aware mobile applications. In Subjective-C, object behaviour depends on the context in which it executes. Hence, observed behaviour is not absolute or objective, but rather of a more relative or *subjective* nature [17]. A minimum amount of computational reflection available in Objective-C suffices to add the necessary abstractions which allow the straightforward expression of context-specific behaviour.

Subjective-C is not a mere reimplementation of the concepts behind main COP languages like Ambience [8] and ContextL [4]. Subjective-C goes beyond the simple inheritance relationships that are possible between context objects in Ambience, and between layer classes in ContextL, by providing explicit means to encode more advanced interdependencies between contexts. Not only does Subjective-C allow for more kinds of dependencies, but also they can be expressed in a domain-specific language developed especially for this purpose, making the declaration of such dependencies more readable.

To validate Subjective-C, we implemented three proof-of-concept applications that run on actual smartphones. These case studies showed the feasibility of programming context-aware applications using subjective programming abstractions, with a noticeable increase in software understandability, maintainability and extensibility. Furthermore, efficiency benchmarks show that the performance impact of COP abstractions in Subjective-C is negligible, to the point that in some cases it can even improve performance.

The remainder of this paper is organised as follows. Section 2 introduces the basics of context-oriented programming in Subjective-C. Section 3 goes on to explain context relations in detail. Section 4 presents the reflective implementation technique we used to add a subjective layer on top of Objective-C. Section 5 briefly presents three validation cases we conducted to assess the advantages and disadvantages of Subjective-C. Section 6 reports on the efficiency benchmarks we carried out. Section 7 discusses limitations and future work. We present related work in Section 8, and draw the paper to a close in Section 9.

2 Context-Oriented Programming in Subjective-C

Context-Oriented Programming (COP) is an emerging programming paradigm in which contextual information plays a central role in the definition of application behaviour [4,8,18]. The essence of COP is the ability to overlay adapted behaviour on top of existing default behaviour, according to the circumstances in which the software is being used. Such adaptations are meant to gracefully adjust the service level of the application, following detected changes in the execution environment. COP languages provide dedicated programming abstractions to enable this behavioural adaptability to changing contexts. This section presents the COP core on which Subjective-C is based. The fundamental language constructs are introduced progressively, as core mechanisms are explained.

2.1 General System Architecture

Subjective-C has been conceived for a fairly straightforward system architecture, illustrated in Fig. 1. Context information is received mainly from two sources. Firstly, there is a context discovery module which collects sensor data to make sense of the *physical* world in which the system is running, and also monitors network services to make sense of the *logical* environment. Logical context changes can also be signalled internally by the running application, for instance when switching to secure or logging mode. Having all context changes at hand, the context management module analyses the current situation and might chose to prioritise some of the context changes, defer others for later application, drop context changes that have become outdated, and solve possible conflicts stemming from contradictory information and adaptation policies (for instance, *increasing* fan speed due to overheating, versus *reducing* fan speed due to low battery charge). It then commits a coherent set of changes to the active context representation, which directly affects application behaviour.

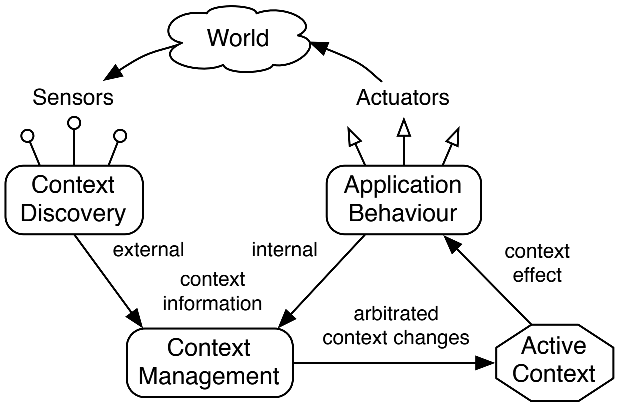


Fig. 1: General architecture for context-aware systems.

```
Context* landscape = [[Context alloc] initWithName: @"Landscape"];
[CONTEXT addContext: landscape];
```

Snippet 1: Subjective-C context definition.

The global architecture proposed here is compatible with more detailed ones such as Rainbow [6], meaning that the more refined subsystems of those architectures can be accommodated within ours. However, the presented level of detail suffices as frame of reference for the explanations that follow.

2.2 Contexts

We define *context* as an abstraction of the particular state of affairs or set of circumstances for which specialised application behaviour can be defined. The context discovery module shown in Fig. 1 is in charge of making sense of perceived data and assigning it a higher-level meaning as contexts. Table 1 shows a few examples. This mapping of data into meaningful contexts is not explored further in this paper.

Sensed data	Contexts
Coordinates = 50°50'N 4°21'E	In Brussels
Battery charge = 220 mAh	Low battery charge
Idle cycles = 11 MHz	High CPU load
Z axis = 0.03	Landscape orientation

Table 1: Environmental data vs. contexts as semantically-rich situations.

The notion of context put forward by Subjective-C is in line with dictionary definitions such as "the situation within which something exists or happens, and that can help explain it"[1] and "the interrelated conditions in which something exists or occurs".[2] This is in contrast to the more general definition of context by Hirschfeld et al. [10] as "any information which is computationally accessible".

In Subjective-C, contexts are reified as first-class objects. A typical context definition is shown in Snippet 1. The landscape context is allocated and given a name. Contexts are declared to the system's context manager by means of the `addContext:` call. As exemplified in Sections 2.3 and 2.4, the `Landscape` context can be used by a smartphone application whose behaviour depends on the spatial orientation of the device.

At any given time, contexts can be either *active* or *inactive*. Active contexts represent the currently perceived circumstances in which the system is running, and only these active contexts have an effect on system behaviour. Snippet 2 shows the way a context can be activated and deactivated. We call such changes from one state to the other *context switches*. Context switches are carried out by the context manager in response to incoming context changes.

[1] `http://dictionary.cambridge.org/dictionary/british/context_1`
[2] `http://merriam-webster.com/dictionary/context`

```
[CONTEXT activateContextWithName: @"Landscape"];
[CONTEXT deactivateContextWithName: @"Landscape"];
```

Snippet 2: Context activation and deactivation.

```
#context Landscape
- (NSString*)getText() {
  return [NSString stringWithString:@"Landscape view"];
}
```

Snippet 3: Context-specific method definition.

2.3 Contextual Behaviour

Subjective-C concentrates on algorithmic adaptation, allowing the definition of behaviour that is specific to certain execution contexts. Programmers can thereby define behaviour that is more appropriate to those particular contexts than the application's default behaviour.

In Subjective-C, defining context-dependent behaviour is straightforward. Adapted behaviour can be defined at a very fine granularity level, namely on a per-method basis. To define methods that are specific to a context, a simple annotation suffices. Snippet 3 illustrates a typical context-specific method definition. The `#context` annotation lets Subjective-C know that the `getText` method definition is specific to the given named context. This version of `getText` should be invoked only when the device is in `Landscape` position. The method is not to be applied under any other circumstances. The general EBNF syntax for context-specific method definitions is as follows:

```
#context ([!]contextName)+
methodDefiniton
```

This is one of the two syntactic extensions Subjective-C lays over Objective-C; the other one is explained in Section 2.4. As can be observed in this general form, it is possible to specialise a method on more than one context. It suffices to provide multiple context names after the `#context` keyword that precedes the method definition. The method is applicable only when all its corresponding contexts are active simultaneously.

As a convenience, Subjective-C introduces method specialisation on the *complement* of a context by means of the negation symbol (`!`). Such complementary method specialisations mean that the method is applicable only when the given context is inactive. Complementary specialisations serve as a shortcut to explicitly defining a complementary context object and associated management policies.

2.4 Behaviour Reuse

For most cases it is of little use to provide a means to define context-specific behaviour but no means to invoke at some point the original default behaviour

```
@implementation UILabel (color)
#context Landscape
- (void)drawTextInRect:(CGRect)rect{
  self.textColor = [UIColor greenColor];
  [superContext drawTextInRect:rect];
}
@end
```

Snippet 4: Sample use of `superContext` construct.

as part of the adaptation. The absence of such a mechanism would lead to the reimplementation of default behaviour in overriding context-specific methods, resulting in code duplication. Subjective-C therefore permits the invocation of overridden behaviour through the `superContext` construct, which has two general forms:

```
[superContext selector];
[superContext keyword: argument ...];
```

Next to the `#context` construct explained in Section 2.3, `superContext` is the second syntactic extension of Subjective-C over Objective-C.

Snippet 4 shows an example in which the colour of a label widget changes to green when the orientation of the host device is horizontal (i.e., when the context `Landscape` is active). This example shows in passing that it is possible to modify the behaviour of stock library objects such as UILabel, which have been developed independently, and for which adaptations such as the one in Snippet 4 were not foreseen. It is possible to layer adaptations on top of *any* existing object, without access to its source code. This is made possible by Objective-C's open classes and categories.

3 Context Relations

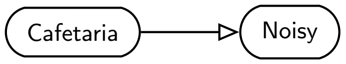
Subjective-C allows the explicit encoding of context relationships. These relationships impose constraints among contexts, which either impede activation or cause cascaded activations and deactivations of related contexts. A failure to respect the natural relationships between contexts could lead to unexpected, undesired, or erroneous application behaviour. All behaviour described in this section is part of the context management subsystem illustrated in Fig. 1.

When a context is switched, the system has to inspect all relations involving the context, checking if the change is consistent with imposed constraints, and performing other switches triggered by the requested one. This section concisely specifies the different relation types and their effect on context switching through four main methods that any context must implement: `canActivate`, `canDeactivate`, `activate` and `deactivate`.

To deal with multiple activations, every context has an *activation counter*, which the `activate` method increases, and `deactivate` decreases (if positive). Only when the counter falls down to zero is the context actually removed from the active context representation.

3.1 Weak Inclusion Relation

Sometimes the activation of a context implies the activation of a related one. For example, if domain analysis yields that cafeterias are usually noisy, then the activation of the `Cafeteria` context should induce the activation of the `Noisy` context. We say that the former context *includes* the latter one. However, the inclusion is a weak one in the sense that the contrapositive does not necessarily hold. Even though there might be no noise, the device might still be located in a cafeteria. The key here is that cafeterias are usually, though not always, noisy. Fig. 2 shows the definition of weak inclusion relations. Activating (resp. deactivating) the source context `Cafeteria` implies activating (resp. deactivating) the target context `Noisy`. The source context can always be activated and deactivated. Conversely, because it is a weak inclusion relation, the target context `Noisy` is not constrained at all by the source context `Cafeteria`. The target context can be activated and deactivated anytime, without consequences on the activation status of the source context.

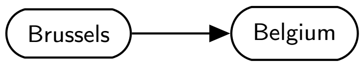


message	source behaviour	target behaviour
canActivate	YES	YES
canDeactivate	YES	YES
activate	target activate	—
deactivate	target deactivate	—

Fig. 2: Weak inclusion relation specification.

3.2 Strong Inclusion Relation

In a strong inclusion relation, the activation of the source context implies the activation of the target context, as in weak inclusions. Additionally, the contrapositive holds: deactivation of the target implies automatically a deactivation of the source. For example, if the current location is `Brussels`, then necessarily the device is also located in `Belgium`. If the current location is not `Belgium`, then it is certainly also not `Brussels`. Fig. 3 shows the definition of such strong inclusion relations. As illustrated by the example, this kind of relation can be used to signal that a specific context is a particular case of a more general one.

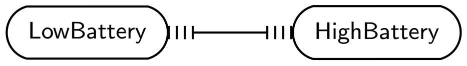


message	source behaviour	target behaviour
canActivate	target canActivate	YES
canDeactivate	YES	source canDeactivate
activate	target activate	—
deactivate	target deactivate	source deactivate

Fig. 3: Strong inclusion relation specification.

3.3 Exclusion Relation

Some contexts are mutually exclusive. For instance, a network connection status cannot be `Online` and `Offline` simultaneously, and the battery charge level cannot be high and low at the same time (note however that it makes sense for two exclusive contexts such as `LowBattery` and `HighBattery` to be simultaneously inactive). This motivates the introduction of *exclusion* relations between contexts, specified in Fig. 4. Note that the exclusion relation is symmetrical.

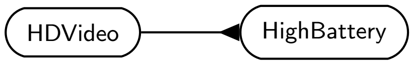


message	source behaviour	target behaviour
canActivate	target isInactive	source isInactive
canDeactivate	YES	YES
activate	—	—
deactivate	—	—

Fig. 4: Exclusion relation specification.

3.4 Requirement Relation

Sometimes certain contexts require other contexts to function properly. For instance, a high-definition video decoding context `HDVideo` might work only when `HighBattery` is active. If `HighBattery` is inactive, then `HDVideo` cannot be activated either. Fig. 5 specifies this *requirement* relation between contexts.



message	source behaviour	target behaviour
canActivate	target isActive	YES
canDeactivate	YES	source canDeactivate
activate	—	—
deactivate	—	source deactivate

Fig. 5: Requirement relation specification.

3.5 Context Declaration Language

For non-trivial scenarios, the programmatic definition of contexts and their relations in Subjective-C can become verbose. As an example, consider the relatively complex code to create just two contexts and an exclusion relation between them, shown in Snippet 5 (left).

Observing that it is cumbersome to describe context settings programmatically, we developed a small Domain-Specific Language (DSL) for this purpose.

```
Context* on = [[Context alloc] initWithName:@"Online"];
Context* off = [[Context alloc] initWithName:@"Offline"];
[CONTEXT addContext:on];
[CONTEXT addContext:off];
[on addExclusionLinkWith:off];
```

```
Contexts:
Online
Offline
Links:
Online >< Offline
```

Snippet 5: Manual creation of contexts and their relations in Subjective-C (left) versus equivalent code using the context declaration language (right).

In this DSL, contexts are declared simply by naming them, and their relations established by means of the following textual notation:

- Weak Inclusion: `A -> B`
- Strong Inclusion: `A => B`
- Exclusion: `A >< B`
- Requirement: `A =< B`

The right side of Snippet 5 shows how the context set-up on the left side is obtained using the context declaration language. This language permits the edition of contexts and their relations with all the advantages brought by a DSL: it has a more intuitive notation that can be understood even by non-programmers, it results in more succinct code, and eases rapid prototyping.[3]

In its current version, Subjective-C does not check inconsistencies among context relationships at the moment they are created (for example if `A => B` on the one hand but `A >< B` on the other); as mentioned earlier, it checks for inconsistencies when contexts are switched, preventing any contradictory change. Support for earlier checks is part of our future work.

4 Implementation

Most existing COP implementations exploit meta-programming facilities such as syntactic macros and computational reflection provided by the host object model to modify method dispatch semantics, thereby achieving dynamic behaviour selection. It is no surprise that these implementations have been laid on top of dynamic languages that permit such level of flexibility.

For approaches based on more static languages such as ContextJ for Java [1], existing implementations use a dedicated compiler. This is also the case of Subjective-C, in which the compiler is just a small language transformer to plain Objective-C. However, Subjective-C does not intercept method dispatch as other approaches do. Rather, it precomputes the most specific methods that become active right after every context switch. This original implementation technique, explained in this section, is possible thanks to some of the dynamic features offered by Objective-C. Section 6 presents an efficiency comparison of the two approaches (method precomputation versus method lookup modification).

[3] A script integrated in the build process of the IDE parses the context declaration files written in the DSL and translates them into equivalent Objective-C code. The result is compiled together with regular source code files.

```
@implementation UILabel (color)
- (void)Context_Landscape_drawTextInRect:(CGRect)rect {
  self.textColor = [UIColor greenColor];
  SUPERCONTEXT(@selector(drawTextInRect:), [self drawTextInRect:rect]);
}
@end
```

Snippet 6: Context-specific version of `UILabel`'s `drawTextInRect:` method translated to plain Objective-C.

```
[MANAGER
  addMethod:@selector(Context_Landscape_drawTextInRect:)
  forClass:[UILabel class]
  forContextNames: [NSSet setWithObjects: @"Landscape", nil]
  withDefautSelector:@selector(drawTextInRect:)
  withPriority:0];
```

Snippet 7: Registration of a context-specific method.

4.1 Method Translation

Context-specific methods, explained in Section 2.3, have the same signature as the original method containing the default implementation. For instance, the `drawTextInRect:` method from Snippet 4 has the same signature as the standard method furnished by Apple's UIKit framework.[4] The intention of adapted methods is precisely to match the same messages the original method matches, but then exhibit different behaviour in response to the message.

To disambiguate method identifiers that have been overloaded for multiple contexts, and thus distinguish between the different context-dependent implementations sharing a same signature, Subjective-C uses name mangling. Name mangling is a well-known technique in which identifiers are decorated with additional information from the method's signature, class, namespace, and possibly others pieces of information to render the decorated name unique. In Subjective-C, the selector of any context-specific method is mangled by prefixing the `Context` keyword, followed by the name of all contexts on which the method has been specialised. The name of complementary contexts (explained in Section 2.3) is prefixed with `NOT`. The different name parts are separated by underscores. As an example, Snippet 6 shows how the name of the `drawTextInRect:` method from Snippet 4 is mangled. The snippet also shows the translation of the `superContext` construct, discussed further in Section 4.3.

The different method versions are registered to the context manager by automatically generated code, shown in Snippet 7. The priority index lets the context manager order method implementations to avoid ambiguities. This ordering is discussed further in Section 7.

[4] UIKit provides the classes needed to manage an application's user interface in iOS.

```
Method current_method =
 class_getInstanceMethod(affectedClass, defaultMethodName);
Method selected_method =
 class_getInstanceMethod(affectedClass, mangledMethodName);
method_setImplementation
 (current_method, method_getImplementation(selected_method));
```

Snippet 8: Reflective method replacement to achieve predispatching.

4.2 Method Predispatch

Contrary to existing COP implementations, Subjective-C does not modify the method lookup process of its host language. Rather, it determines the method implementations that should be invoked according to the currently active context at context-switching time. The chosen methods become the *active methods* in the system. The set of active methods is recalculated for every change of the active context. We call this process *method predispatch*.

Method predispatch is made possible by the ability to dynamically replace method implementations in Objective-C. As sketched in Snippet 8, the predispatcher uses the reflective layer of Objective-C to exchange method implementations.[5] The currently active implementation is replaced by a version that is most specific for the active context. It can very well be that the old and new versions are the same, in which case the method switching operation has no effect.

This implementation technique would be less easy to achieve in other members of the C language family such as C++, due to the lack of a standard reflective API that enables the manipulation of virtual method tables. A non-reflective implementation would probably involve compiler– and platform-specific pointer manipulations to patch such tables manually.

Finally, from Snippet 8 it can be observed that the default method and its context-dependent adaptations belong to the same class. Since Objective-C features open classes, methods can be added to any existing class. Open classes make it possible for Subjective-C to add context-specific methods to any user-defined, standard or built-in class, without access to its source code. Adaptability of third-party code is one of the strongest advantages brought by Subjective-C, and is another area in which other members of the C language family would fall short in implementing a similar mechanism (because they lack open classes).

4.3 Super-Context Calls

Snippet 6 shows how the `superContext` construct from Snippet 4 is translated to plain Objective-C. Snippet 9 shows the definition of the `SUPERCONTEXT` preprocessor macro used by the translated code. This macro replaces the current method implementation by the next one in the method ordering corresponding to the given class, default selector and currently active context, invokes the newly set implementation, and reverts the change to leave the system in its original state.

[5] Besides instance methods, it is also possible to manipulate class methods through `class_getClassMethod`, but the details are inessential to the discussion.

```
#define SUPERCONTEXT(_defaultSelector, _message) \
  [MANAGER setSuperContextMethod:_defaultSelector forClass:[self class]]; \
  _message; \
  [MANAGER restoreContextMethod:_defaultSelector forClass:[self class]];
```

Snippet 9: Macro definition used to translate `superContext` constructs.

5 Validation

This section summarises three case studies we developed to assess the qualities of Subjective-C to respectively create a new context-aware application from scratch, extend an existing application so that it becomes context-aware, and refactor an existing application by exploiting its internal modes of operation (i.e. logical contexts, as opposed to making it adaptable to physical changes).

Home Automation System The goal of this case study is to build a home automation system using Subjective-C from the ground up. The system permits the use of a smartphone as remote control to regulate climatic factors such as temperature, ventilation and lighting, and to command household appliances such as televisions. The remote control communicates through the local network with a server system, which simulates these factors and appliances, in a home with a kitchen, bathroom, bedroom and living room. Each room is equipped with a different combination of windows (for ventilation regulation), heating, air conditioning and illumination systems. The remote control application adapts its user interface and behaviour dynamically according to the simulated context changes coming from the server.

This case study heavily uses the context declaration language introduced in Section 3.5. Fig. 6 shows a graphical overview for the home server implementation. The relatively complex relations for this proof-of-concept system show that the definition of a dedicated context declaration language is justified. In a full-fledged home automation system the contexts and their relations could be even more intricate, and defining all entities programmatically would result in complex code.

Device Orientation The goal of this case study is to extend an existing Objective-C application with context-oriented constructs. The original Device Orientation application is a proof-of-concept whose basic functionality is to display a text label which dynamically adjusts its display angle so that it remains parallel to the ground, regardless of the physical device orientation. The application extension consists in changing the text and colour of the label according to orientation changes in the x and z axes. The guideline is to be able to introduce said extensions with minimal intervention of the original source code. Several code snippets from this case study are used throughout this paper.

Regarding efficiency, Device Orientation switches contexts as frequently as every 100 milliseconds to keep the label constantly parallel to the ground. We observed no apparent slowdown with respect to the original application: the label adapts swiftly to orientation changes.

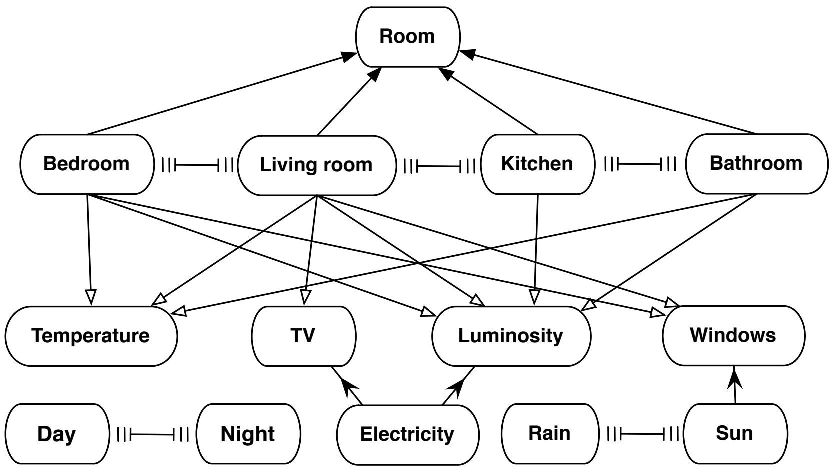


Fig. 6: Context and relations in the Home Automation case study.

Accelerometer Graph The goal of this case study is to perform a behaviour-preserving refactoring of an existing application using contexts, to assess the impact on source code quality. The chosen application is Accelerometer Graph, developed by Apple to illustrate the use of filters to smooth the data obtained from an iPhone's accelerometer. The application presents a graph of read accelerometer data against time. It can work in standard (default) or adaptive mode. Independently from these modes, it can work in low-pass or high-pass mode. Due to these operation modes, the original application presents some cases of conditional statements related to the operation mode, and code duplication. The refactored version avoids the conditionals and the duplication by modelling the different operation modes as contexts.

From the experience gathered in the described case studies, we have observed that extensibility and maintainability are particularly strong points of Subjective-C. These main strengths come from the separation of concerns between the base application and its context-specific adaptations. Subjective-C allows the adaptation of any method of the application, and all such method adaptations that correspond to a given context can be modularised and furnished as a single unit. Further details and in-depth discussion of the case studies are provided by Libbrecht and Goffaux [13].

6 Benchmarks

As mentioned in Section 1, one of the main advantages of COP is that it offers an alternative to hard-coded conditional statements. By helping to avoid such statements, COP renders code more reusable and maintainable. However, this advantage would be nullified if the penalty in performance would be prohibitively high. Therefore, to assess the cost of using COP abstractions, we mea-

```
-(void) test:(int) mode {
  if (mode == 1)
    result = 1;
  else if (mode == 2)
    result = 2;
  ...
  else if (mode == N)
    result = N;
  else
    result = 0;
}
```

```
#context C1
-(void) test {
  result = 1;
}
...
#context CN
-(void) test {
  result = N;
}
-(void) test {
  result = 0;
}
```

Snippet 10: Dummy test methods in Objective-C (left) and Subjective-C (right) with $N + 1$ behavioural variants. The choice of the variant depends on the application's current operating mode and the application's current execution context, respectively.

sured the difference in execution time between an application that uses contexts in Subjective-C and an equivalent application that uses conditional statements in Objective-C. Our benchmark consists of a dummy application that runs in $1+N$ possible operation modes (the default one plus N variants). In Objective-C these modes are encoded as integers stored in a global variable, on which application behaviour depends. In Subjective-C the alternatives are represented as contexts. Snippet 10 illustrates the two approaches. For the sake of the benchmark, the `test` method merely produces a side effect by assigning the `result` global variable. Since the execution cost of such an assignment is negligible, the cost of `test` is dominated by the cost of method invocation. Additionally, the Objective-C solution incurs the cost of testing the branches in the conditional statement. For sufficiently high values of N, this cost becomes considerable. In Subjective-C there is no additional cost associated to the choice of a behavioural variant during method invocation, because such choice has been precomputed at context-switching time.

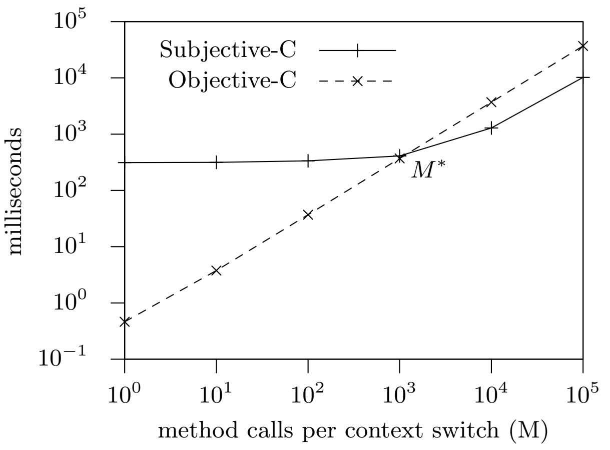
Naturally, the question is how the costs of conditional statement execution in Objective-C and context switching in Subjective-C compare. To measure the difference, we invoke the `test` method M times for every context change, as shown in Snippet 11. In Objective-C, `test` execution time depends on the number of branches K that need to be evaluated in the conditional statement. In Subjective-C, `test` execution time is constant, but at context-switching time it is necessary to iterate over the first K possible methods to find the one that needs to be activated. The results of the comparison between these two approaches are shown graphically in Fig. 7a for $N = 50$ and $K = 50$. The test application was run in debugging mode on an iPhone 3GS with iOS 4.0. In the case illustrated in Fig. 7a, context switching reaches the efficiency of conditional statement execution at about 1150 method calls per context switch. Beyond this point, Subjective-C is more efficient than Objective-C; the execution time in

```
for (int i = 0; i < 1000; i++) {
  if (i % 2)
    [CONTEXT activateContextWithName:@"CK"]; // mode = K;
  else
    [CONTEXT deactivateContextWithName:@"CK"]; // mode = 0;
  for (int j = 0; j < M; j++)
    [self test];
}
```

Snippet 11: Code to measure the relative cost of context changes with respect to context-dependent method invocation in Subjective-C; the Objective-C counterpart is analogous and is therefore just suggested as comments.

both approaches will tend to grow linearly, although Objective-C will have a considerably higher slope due to conditional statement execution,[6] besides the cost of method invocation which is incurred in both approaches. Fig. 7b summarises the intersection points for various values of N and K, including the case of Fig. 7a.

(a)



(b)

N	K	M^*
5	1	8230
10	1	4708
50	1	3405
5	5	4521
10	10	2653
50	50	1148

Fig. 7: Performance comparison of Objective-C and Subjective-C; (a) illustration with logarithmic scale for the case $N = 50$, $K = 50$, and (b) summary of efficiency meeting points M^* for various values of N and K.

The benchmarks just discussed use contexts that are not linked through any of the relations introduced in Section 3. Though not shown here for space limitations, we have carried out a few benchmarks to assess the impact of relations on context activation [13]. The presence of exclusion relations increases slightly the time of activation (i.e. an extra check for every excluded context); in inclusion

[6] This difference is not apparent in Fig. 7a because of the logarithmic scale.

relations, the (de)activation must be not only accepted but also propagated on the chain of included contexts. Processing inclusion relations is about 4 times more costly than processing exclusion relations.

In another benchmark we assessed the cost of activation according to the number of methods associated to the switched context. As can be expected, the activation time increases linearly with the number of methods (e.g. if a context has twice as many methods, it takes twice as much time to switch).

Yet in one more benchmark we evaluated the activation time according to number of contexts that specialise a given method. In our implementation, the execution time grows linearly with the number of contexts that implement the same method, because finding the most specific method involves a linear search in a list of available methods.[7]

The efficiency of Subjective-C depends highly on its usage. Most benefits are obtained for contexts that are switched infrequently with respect to the rate of usage of affected methods. Fortunately, this is the case for most common scenarios, because context changes are usually linked to physical phenomena such as orientation changes, temperature changes, battery charge changes, network connections, and so forth. In particular, in each of the three case studies described in Section 5, the Subjective-C implementation did not give rise to apparent performance penalties. This being said, we can think of a few kinds of contexts which could be switched very rapidly, for instance software memory transactions implemented as contexts [7], used in tight loops. For these cases the penalty incurred by Subjective-C could become detrimental to overall performance. However, for most practical cases we can conclude that COP abstractions do not incur a performance penalty that would bar them from mobile platform programming.

7 Limitations and Future Work

Even though Subjective-C is usable for application development on mobile devices as suggested in Section 5, it still has rough edges we need to iron out. This section describes the most salient ones, starting with the more technical and going over to the more conceptual.

Super-Context Translation A caveat of the implementation presented in Section 4.3 is the impossibility to retrieve the return value from a `superContext` message. Our current solution consists in having a different syntax when the return value is needed, which complements the definition of the `superContext` construct given in Section 2.4:

```
[superContext selector] => variable;
[superContext keyword: argument ...] => variable;
```

This syntax is translated to a variant of the `SUPERCONTEXT` macro which expands the message as `variable = _message` instead of just `_message`.

[7] We have invested no effort yet in improving this straightforward implementation.

The additional syntax shown here is due to a particularity of our current implementation, but we see no fundamental reason why it could not be avoided in a more sophisticated version of the compiler.

Context Scope and Concurrency Context activation in Subjective-C is global. All threads share the same active context and see the effects of context switching performed by other threads. This can give rise to race conditions and behavioural inconsistencies if concurrent context switches occur at inappropriate execution points [9]. However, note that this issue does not stem from a conceptual error. As discussed in Section 2.1, any computing device is embedded in a physical and logical execution environment, which all applications running on the device share. For instance, it is natural that all applications become aware of a `LowBattery` condition, or a reorientation of the device to `Landscape` position. Rather than avoiding the problem of shared contexts altogether by limiting the scope of context effects to individual threads, our open research challenge and line of future work consists in detecting the execution points at which shared context changes are safe to perform, whether automatically or with some sort of assistance such as source code annotations.

Nevertheless, having made a case for global contexts, we do believe that adding support for local contexts, representing the running conditions of particular threads, would be a useful addition to Subjective-C. For instance, one thread could run in `Debug` or `Tracing` mode simultaneously with other threads running in default mode.

Behaviour Disambiguation Whenever multiple methods are applicable for a given message and active context configuration, the context manager should be able to deterministically define which of the methods is to be invoked. For example, suppose there are two versions of `UILabel`'s `drawTextInRect:` method, respectively specialised on the `Landscape` and `LowBattery` contexts. If both contexts are active at any given time, it is unclear which of the two versions of `drawTextInRect:` should be applied first.

Currently, the choice is based on a priority assigned to every method. Default methods have always less priority than context-dependent methods. For two context-dependent methods, the priority is given by the order in which the compiler comes across the method definitions.[8] Hence, the later a method is found, the higher its priority. Clearly this ad hoc mechanism to automatically determine priorities is insufficient. A better solution is to define a version of the ambiguous method specialised on the set of conflicting contexts (in the example, `Landscape` and `LowBattery`), and resolve the ambiguity manually in that specific method implementation. This solution cannot be used for every possible combination as this would result in a combinatorial explosion. A line of research is to help predicting which ambiguities arise in practice by analysing context relations, and provide declarative rules to resolve remaining ambiguities based on domain-specific criteria.

[8] Note that Objective-C open classes can be defined across multiple files.

8 Related Work

COP-like ideas for object-oriented systems can be traced back as far as 1996. The Us language, an extension of the Self language with subjective object behaviour [17], inspired our work since the early stages. In Us, subjectivity is obtained by allowing multiple perspectives from which different object behaviour might be observed. These perspectives are reified as *layer* objects, and hence, Us layers are akin to Subjective-C contexts.

The contemporary notion of COP has been realised through a few implementations, in particular ContextL [4] which extends CLOS [2], Ambience [9] which is based on AmOS [8], and further extensions of Smalltalk [10], Python [14] and Ruby,[9] among others. Most existing approaches, with the exception of Ambience, seem to be conceptual descendants of ContextL, and therefore share similar characteristics. None of these COP languages is similar to Subjective-C in that they affect method dispatch semantics to achieve dynamic behaviour selection, whereas Subjective-C uses method predispatching, introduced in Section 4.2.

Subjective-C is inspired on our previous work with Ambience. In particular, both languages use the notion of contexts as objects representing particular run-time situations, described in Section 2.2. Further, contexts in Ambience are also global and shared by all running threads, an issue discussed in Section 7. Whereas in Ambience it would not be difficult to adapt the underlying AmOS object model to support thread-local contexts, in Subjective-C we do not control the underlying object system inherited from Objective-C.

An issue barely tackled by existing COP approaches is the high-level modelling of contexts and their conditions of activation. Subjective-C makes a step forward in this direction by introducing different types of relations between contexts, explained in Section 3. This system of relations bears a strong resemblance to some of the models found in Software Product Line Engineering (SPLE). Unfortunately, thus far SPLE has focused mostly on systems with variability in static contexts [15]. Variability models such as Feature Diagrams (FDs) [11] and their extensions have not been geared towards capturing the dynamism of context-dependent behavior. More recent work on variability models acknowledges the concept of dynamic variability in SPLE [3,5,12].

Also related to COP in general, and Subjective-C in particular, is the family of dynamic Aspect-Oriented Programming approaches. PROSE [16] for instance is a Java-based system using dynamic Aspect-Oriented Programming (AOP) for run-time adaptability. Since dynamic aspects can be woven and unwoven according to context, dynamic AOP can be a suitable option to obtain dynamic behaviour adaptation to context. Dynamic AOP buys flexibility (for instance, the ability to express join points that capture only certain invocations of a given method, instead of every invocation) at the expense of more conceptual and technical complexity (e.g. additional join point language and abstractions for aspect definition).

[9] `http://contextr.rubyforge.org`

9 Conclusions

The field of Context-Oriented Programming (COP) was born in response to a lack of adequate programming abstractions to develop adaptable applications that are sensible to their changing execution conditions. Observing that no existing COP language allowed us to experiment with context-oriented mobile application programming, we set out to develop an extension of one of the most widely used languages for mobile devices, namely Objective-C. The result is the Subjective-C language,which furnishes dedicated language abstractions to deal with context-specific method definitions and thus permits run-time behavioural adaptation to context. Subjective-C objects are less "objective" than those of Objective-C in that their expressed behaviour does not depend entirely on the messages they receive, but also on the current execution context.

Subjective-C goes beyond existing COP approaches by providing an explicit means to express complex context interrelations. The different relations have a corresponding graphical depiction and textual representation, to ease their description and communication among developers and domain experts. Although we cannot guarantee that the set of supported relation types is complete enough to express all relevant context settings, we have found it to be sufficient in practice for now.

Subjective-C introduces an original implementation technique that trades context switching efficiency for method execution efficiency. Our experience with the validation cases shows that this technique results in no noticeable efficiency penalties, despite the relatively resource-constrained host platforms on which these applications run. Even more, under some circumstances efficiency is improved as compared to using plain Objective-C. From the same experience we have observed that Subjective-C seems to achieve its ultimate software engineering goals, which are improved modularisation, increased readability, reusability, maintainability and extensibility of context-aware software.

Acknowledgements

This work has been supported by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels, and by the Interuniversity Attraction Poles Programme, Belgian State, Belgian Science Policy. We thank the anonymous reviewers for their comments on an earlier version of this paper.

References

1. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ: Context-oriented programming for Java. Computer Software of The Japan Society for Software Science and Technology (6 2010)
2. Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., Moon, D.: Common Lisp Object System specification. Lisp and Symbolic Computation 1(3/4), 245–394 (1989)

3. Cetina, C., Haugen, O., Zhang, X., Fleurey, F., Pelechano, V.: Strategies for variability transformation at run time. In: Proceedings of the International Software Product Line Conference. pp. 61–70. Carnegie Mellon University, Pittsburgh, PA, USA (2009)
4. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of ContextL. In: Proceedings of the Dynamic Languages Symposium. pp. 1–10. ACM Press (Oct 2005), co-located with OOPSLA'05
5. Desmet, B., Vallejos, J., Costanza, P., De Meuter, W., D'Hondt, T.: Context-oriented domain analysis. In: Modeling and Using Context. pp. 178–191. Lecture Notes in Computer Science, Springer-Verlag (2007)
6. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
7. González, S., Denker, M., Mens, K.: Transactional contexts: Harnessing the power of context-oriented reflection. In: International Workshop on Context-Oriented Programming. pp. 1–6. ACM Press (2009), 7 July 2009. Co-located with ECOOP.
8. González, S., Mens, K., Cádiz, A.: Context-Oriented Programming with the Ambient Object System. Journal of Universal Computer Science 14(20), 3307–3332 (2008)
9. González, S., Mens, K., Heymans, P.: Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In: Proceedings of the Dynamic Languages Symposium. pp. 77–88. ACM Press (Oct 2007), co-located with OOPSLA'07
10. Hirschfeld, R., Costanza, P., Haupt, M.: An introduction to context-oriented programming with ContextS. In: Generative and Transformational Techniques in Software Engineering II. Lecture Notes in Computer Science, vol. 5235, pp. 396–407. Springer-Verlag, Berlin, Heidelberg (2008)
11. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute (Nov 1990)
12. Lee, J., Kang, K.C.: A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In: Proceedings of the International Software Product Line Conference. pp. 131–140. IEEE Computer Society Press, Los Alamitos, CA, USA (2006)
13. Libbrecht, J.C., Goffaux, J.: Subjective-C: Enabling Context-Aware Programming on iPhones. Master's thesis, Ecole Polytechnique de Louvain, UCLouvain (Jun 2010)
14. von Löwis, M., Denker, M., Nierstrasz, O.: Context-oriented programming: Beyond layers. In: Proceedings of the 2007 International Conference on Dynamic languages. pp. 143–156. ACM Press, New York, NY, USA (2007)
15. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, Berlin, Heidelberg (2005)
16. Popovici, A., Gross, T., Alonso, G.: Dynamic weaving for aspect-oriented programming. In: Proceedings of the International Conference on Aspect-Oriented Software Development. pp. 141–147. ACM Press (2002)
17. Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. Theory and Practice of Object Systems 2(3), 161–178 (1996)
18. Vallejos, J., González, S., Costanza, P., Meuter, W.D., D'Hondt, T., Mens, K.: Predicated generic functions: Enabling context-dependent method dispatch. In: Baudry, B., Wohlstadter, E. (eds.) Software Composition. Lecture Notes in Computer Science, vol. 6144, pp. 66–81. Springer-Verlag (2010)