A Context-Oriented Software Architecture

Kim Mens
Université catholique de Louvain
Belgium
kim.mens@uclouvain.be

Nicolás Cardozo
Universidad de los Andes
Colombia
n-cardoz@uniandes.edu.co

Benoît Duhoux
Université catholique de Louvain
Belgium
benoit.duhoux@student.uclouvain.be

Abstract

Context-aware systems must manage the dynamic selection, activation, and execution of feature variants according to changing contexts, detected from data gathered from their surrounding execution environment. Many context-oriented programming languages focus only on the implementation level by providing appropriate language abstractions for implementing behavioural variations that can adapt dynamically to changing contexts. They often ignore or presuppose the existence of mechanisms to deal with earlier aspects such as the gathering of sensory input and context discovery. In this paper we propose a layered software architecture that reconciles all these aspects in a single implementation framework, which can be customized by application programmers into actual context-aware applications. This framework is currently being implemented in Ruby on top of a reimplementation of the Phenomenal Gem context-oriented language.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Languages, Domain-specific architectures; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures, modules

General Terms Languages, Design

Keywords Context-oriented programming, Software architecture, Implementation framework

1. Introduction

Context-awareness is important, for example, when building pervasive systems for the IoT, where devices (through sensors and actuators) may vary their behaviour to interact with one another, or when creating personalised applications that adapt to users' preferences and context of use. Context-aware systems adapt their execution behaviour dynamically to particular situations based on contextual information discovered from their surrounding execution environment, such as weather conditions, localisation information, remaining battery power, user actions, system state, and so on.

1.1 Context-Oriented Programming

In context-aware systems, behavioural variations are often managed by using a multitude of `if`-statements, scattered all over the code, to dispatch the different application behaviour, as exemplified below for the `receiveCall` method of a phone device.

```
def receiveCall(call)
  ...
  if LowBattery
    phone.forwardCall(call, self.number)
  end
  if QuietEnvironment
    phone.vibrate(5)
  end
end
```

To address the maintainability issues ensuing from handling such scattered `if`-statements, recent developments in programming language research have given rise to the novel paradigm of Context-Oriented Programming (COP) [7], a programming language approach to enable programming behavioural variations to changing contexts of use in a more modular fashion.

For example, the `LowBattery` case of the above call reception feature could be modularised as follows in the Phenomenal Gem [14] COP language:

```
context LowBattery do
 adapt :receiveCall do
  phone.forwardCall(call, self.number)
 end
end
```

Different such variants would be associated to different contexts, thus enhancing modularity by reducing tangling and scattering of contextual feature variants with respect to the default behaviour and other contexts.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author(s). Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

COP'16, July 19 2016, Rome, Italy
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM ACM 978-1-4503-4440-1/16/07?$15.00 . . . $15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2951965.2951971

1.2 Adaptive Software Systems

Nonetheless, such code level handling of feature variants, their interactions and activations, based on contextual information discovered from the surrounding environment, becomes a daunting task for larger systems. To minimise such complexity, Salehie and Tahvildari [15] proposed an architecture for adaptive systems in the form of a Monitoring, Analysis, Planning, Execution, and Knowledge (MAPE-K) control feedback loop. In this domain, many different proposals to define an architecture of adaptive systems have been proposed [1, 6, 13, 16] (cf. Section 3). Most of these proposals, as exemplified in Figure 1, take a relatively high-level view of the system, showing how the different components defining its architecture can be adapted.

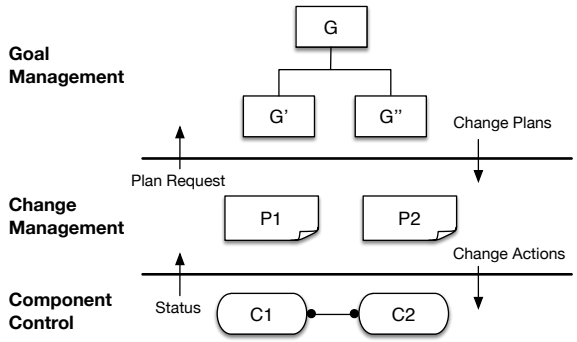

Figure 1. Adaptive Systems' Architecture Model [13]

However, such high-level architectural views of adaptive systems are rather coarse-grained and do not highlight the different variations and components that programmers need to code.

1.3 Contribution

In this paper we propose an architecture for context-oriented software systems that *reconciles the practicality of encoding variations at the code level with COP languages, with the clarity provided by high-level architectures of adaptive systems.* The purpose of our architecture is to cover all levels required to enable context-orientation in a software system, while making explicit which parts are provided as part of the adaptation framework (Section 2.2), and which are to be coded by the application programmer (Section 2.3). We are currently implementing this architecture on top of our reimplementation of Phenomenal Gem [14], a COP language extension for Ruby.

2. A Context-Oriented Architecture

2.1 A Layered Architecture

Our proposed context-oriented software architecture, depicted in Figure 2, is a layered architecture consisting of 4 layers. The *interaction layer* is in charge of the interaction between the context-oriented system and its external environment. The *discovery layer*'s purpose is to determine the current context of use based on the information received from the interaction layer. The *handling layer* is in charge of activating the appropriate contexts determined by the discovery layer, as well as selecting, activating, and executing the feature variants corresponding to those contexts. These three layers constitute the *adaptation framework* (Section 2.2), which can be regarded as an implementation framework containing the machinery upon which context-oriented program(mer)s can rely. The fourth *application layer* (Section 2.3) contains the different components to be provided by an application programmer. The framework calls these application components as needed.

2.2 The Adaptation Framework

Interaction Layer The interaction layer is in charge of gathering information from the system's surrounding environment. That is, information coming from the physical environment (*e.g.,* geographical localisation), the user (*e.g.,* user preferences), or the system's computing platform (*i.e.,* hardware and software conditions, such as the battery status). Environmental information is gathered by two components: *User Input* and *Sensors*. The *User Input* component gathers information coming from the user (*e.g.,* user preferences, particular user actions or user input). *Sensors* gather information through physical devices available in the environment, as well as information monitored from the system's executing platform (*e.g.,* hardware conditions).

Discovery Layer This layer interprets and reasons over the information gathered from the external environment, with the objective of extracting those contexts that are semantically relevant for the application. It identifies which contexts may become (in)active, according to the information received from the interaction layer via a set of listeners provided by the application programmer, combined with the context declarations specified by that programmer. The *Interpretation* component first filters the received information with a set of filters provided by the programmer. The *Reasoning* component decides which of the declared contexts are relevant to the application.

Handling Layer The handling layer analyses the different kinds of dependencies that exists in and between contexts and their features, to manage the (de)activation and selection of contexts and their features, in order to adapt the system's behaviour in a consistent way. It is partitioned in two sublayers.

Context handling decides which contexts get activated. The *Context Activation* component verifies if a context can be (de)activated based on its declared

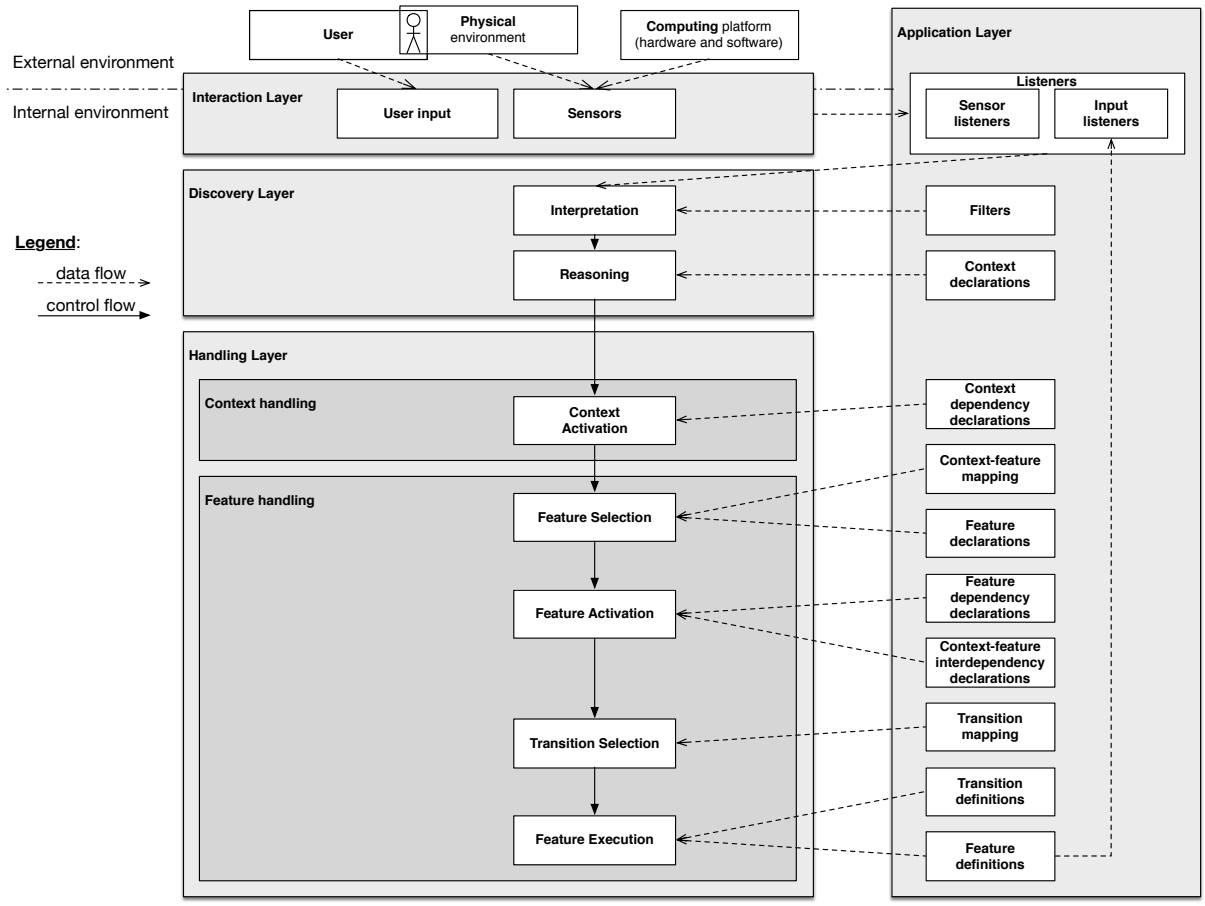

Figure 2. Our context-oriented software architecture

dependencies. It can also keep track of the activation state of contexts such as how many times a context has been activated, and what is its activation scope.

Feature handling manages the selection, activation, and execution of features whenever a context is active (*i.e.,* the system's behaviour). Each of this sublayer's responsibilities are realised by separate components. *Feature Selection* is in charge of acquiring all features associated to the currently active contexts. *Feature Activation* verifies whether the selected features can be (de)activated consistently according to their declared dependencies with other features and contexts. Similar to the *Context Activation* component, this component also manages the scoping of features. *Transition Selection* determines what transitions need to be performed upon the (de)activation of certain features. These transitions, declared by programmers, may vary with respect to the currently active contexts in the system. Finally, the *Feature Execution* component manages the execution of the aforementioned transitions and features to adapt the system's behaviour at run time.

2.3 The Application Layer

The application layer specifies the different implementation components to be provided by an application developer to customise the adaptation framework (Section 2.2) into an actual context-oriented application.

Listeners The first components to be provided by a programmer to customise the framework are the listeners. They intercept data coming from the user (*Input listeners*) or generated by sensors (*Sensor listeners*) and are implemented according to the Observer pattern: each listener is an Observable. Whenever a listener intercepts some data, it converts it to JSON to preserve this data and its types, and notifies the *Interpretation* component of the framework. The code at the top of the next page simulates a listener retrieving data from a GPS sensor.

Filters Whenever the discovery layer receives user and sensor data, the first thing that its *Interpretation* component does is filtering out some of that data. Data flows continuously to sensors while the application is running. Analysing this flow constantly would cause significant performance overheads, especially for contexts

```
class GpsListener
  include Observable
  attr_reader :latitude, :longitude
  def initialize(latitude, longitude)
    @latitude = latitude
    @longitude = longitude
    notify
  end
end
```

that do not change often. For example, if the application needs to adapt only to coarse-grained temperature changes (*e.g.,* 'Hot' or 'Cold'), there is no need to check the temperature sensor every second to decide whether a context change should be triggered. Depending on the application domain, coarser time intervals could be used. To achieve this, the programmer can write a time filter for the temperature listener to ensure that data gets analysed only once every 5 minutes, as shown in the code below. Other types of filters exist as well.

```
class TempFilter < TimeFilter
  def initialize
    super
    @new_wait_to_filter = 5*60
  end
end
```

Context declarations After having filtered and interpreted the data coming from the listeners, and having turned them into contextual objects, the reasoning will verify which of those contexts can be activated, depending on the context conditions expressed by the application programmer in the *Context Declarations*. As shown below on the example of a TEMPERATURE context, a context declaration specifies the (unique) name of the context, whether it is abstract or not, what sensor it is coupled to, what subcontexts it has, and what are its admissibility conditions. For example, the HOT subcontext makes sense only when a temperature of 25°C or higher is retrieved from the temperature sensor.

```
{ "contexts": {
   "isAbstract": true,
   "subcontexts": {
     "Temperature": {
       "isAbstract": true,
       "sensors": ["TempSensor"],
       "subcontexts": {
         "Hot": {
           "condition": "(TempSensor.
               degree >= 25)"
         },
         "Cold": {
           "condition": "(TempSensor.
               degree < 25)"
} } } } } }
```

All context declarations are contained in a single JSON object, from which the *Reasoning* component creates a context graph. The code fragment below shows how this component navigates the graph of all declared contexts to find those that are admissible. To verify if a context is admissible, all its conditions are checked; if satisfied, the context is added to the list of admissible contexts. This process is repeated for all of the context's children, until no contexts remain to be verified in the graph. Once this procedure is terminated, the list of admissible contexts is passed to the *Handling* component.

```
def getAdmissibleContexts(contextGraph, o)
  admissibleContexts = []
  queue = [contextGraph]
  while !(queue.empty?)
    contextGraph = queue.shift
    isOk = contextGraph.applyConditions(o)
    admissibleContexts << contextGraph if
        isOk
    queue = contextGraph.children + queue
  end
    admissibleContexts
end
```

Context dependency declarations Dependencies between contexts (such as exclusion and requirement [5]) are also declared as a JSON object:

```
{ "dependencies": {
    "Temperature": {
      "xor": [] },
    "GPSActivity": {
      "requirement": [ "HighBattery" ] },
    "EmergencyMode": {
      "requirement": [ "LowBattery" ] }
} }
```

For example, the above code excerpt declares a *xor* dependency[1] between the different subcontexts of the TEMPERATURE context (a temperature can be either cold or hot, but not both), a requirement dependency of GPSACTIVITY on HIGHBATTERY (since the GPS consumes too much battery it should not be allowed when the battery is running low), and another *requirement* dependency from the EMERGENCYMODE to the LOWBATTERY context. The semantics of a *requirement* dependency is that the required context must be active before the requiring context can be activated. In this particular case, EMERGENCYMODE requires LOWBATTERY to be active since this mode is only supposed to run in that particular situation, by shutting down essential services to maintain enough power to make emergency calls when needed.
Based on the activation semantics of the context dependency declarations [4] provided by the application

[1] An exclusive-or can be regarded as syntactic sugar for multiple exclusion dependencies. It is a useful notation when multiple subcontexts of a same context all need to be mutually exclusive.

programmer, the *Context Activation* component decides which contexts need to be activated. As in Ambience [9], Subjective-C [10] and Phenomenal Gem [14], contexts can be activated more than once (*e.g.,* a context `UserPresence` could be activated as many times as there are users present). The *Context Activation* component takes care of these activation counters. The activation counter is increased upon each activation and decreased upon each deactivation. A context is considered inactive if its activation counter reaches zero, and is then removed from the list of currently active contexts.

Feature Handling Due to space limitations and since the *Feature Handling* sublayer of the architecture is currently being reimplemented, we explain only the principle of the code that needs to be provided by the application programmer, and how this code is used by the adaptation framework.

Context-feature mapping This mapping links features (or feature variants) to their corresponding contexts. This is similar to the example of Section 1.1 where a call forwarding feature (which specializes the default call reception feature) was linked to the `LowBattery` context.

Feature declarations Similar to context declarations, features form a graph consisting of many features and subfeatures. Features are declared as a JSON object, from which the *Feature Selection* creates a feature graph.

Feature dependency declarations Similar to context dependencies, dependencies can be declared between features. The type of dependencies that can be declared between features are the same as those that can be defined between contexts (*e.g.,* exclusion, requirement).

Context-feature interdependency declarations In addition to having intra-dependencies between contexts or between features, inter-dependencies can exist between features and contexts. Again, we allow for the same types of dependencies as before. Dependencies declared between features, and between contexts and features, are used by the *Feature Activation* component to activate the right features taking into account the constraints imposed by such dependencies.

The duality between features and contexts is novel in our current architecture and approach, and is strongly inspired by the work of Hartmann and Trew [11]. In their work, they use a two-branched feature diagram to model both the different features and their intra-dependencies (one branch), as well as the different contexts and their intra-dependencies (second branch), plus the inter-dependencies between nodes (*i.e.,* features and contexts) in these different branches, to model multiple product lines for software product line chains. Capilla et al. [3] observe that the same kind of modelling approach could be used to model context variability in context-aware systems.

Transition definitions Another novelty in our architecture is that, inspired by work on user interface adaptivity [8], we observed the need for having gentle transitions when (de)activating features when switching from one context to another. Although we still need to explore the use and relevance of such transitions for behavioural adaptations in practice, in the case of user interface adaptations transitions can be useful to give visual clues to the end-user that something has changed, such as fading in or out a new user interface element. The transition definitions, provided by the programmer, contain the code to be executed for these transitions.

Transition mapping The transitions to be applied may depend on the source context (old situation), the target context (current situation), and the feature that needs to be executed. The transition mapping is defined so that the appropriate transitions, if any, get selected.

Feature definitions contain the actual code of the features that need to be executed in certain contexts. They constitute the code of the application to be executed. These feature definitions are similar to second code fragment of Section 1.1, *i.e.,* the call forwarding behaviour, except that the feature definition would contain only the definition of the behaviour, whereas the linking of this feature to the `LowBattery` context is part of the *Context-feature mapping*.

3. Related Work

The development of COP languages focuses on introducing language abstractions to enable dynamic adaptations, loosely following a MAPE-K architecture [2]. However, their design often remains a bit ad hoc, focusing mostly on the application layer of the architecture —that is, the language-level abstractions to modularise and activate adaptations with respect to the context.

Adaptive systems introduce a more structured approach to context-orientation by defining an architectural design to manage the autonomous adaptation of software components [12, 13]. Architectures for adaptive systems are anchored in the MAPE-K loop, describing 3 layers to manage the adaptations of components to context. In their architecture-based framework for adaptation, Cheng et al. [6] introduce Probes to monitor the system's states and Gauges to aggregate monitored information. Information coming from the system (target environment layer) is then used by a model manager component from which architecture adaptations are evaluated, managed, and executed (abstraction / model layer). Once adaptations are introduced, they can be used in the new system model (consumer

layer). FORMS [16] extends the general MAPE-K loop architecture by introducing notions of reflection and distribution into the architectural model. FORMS introduces the idea of working models, which constitute the temporary representation of the system adapting the base model. Working models are deployed according to the information gathered from the monitoring component of the architecture. DuSE [1] presents a general-purpose control loop architecture, providing developers with a framework to assess the trade-off in designing their adaptive software.

The aforementioned architectures relate to our proposal as they describe the components that interact with the environment and adapt the executing system based on such interactions. Our proposal goes beyond this related work in making explicit the close link between the system architecture and the mechanisms to realise adaptations at the programming language level.

4. Conclusion

We presented our 4 layer architecture (interaction, discovery, handling, and application layer) depicting all necessary components for building context-oriented systems. The architecture can be regarded as an implementation framework, the code of which partly is provided by the adaptation framework (the interaction, discovery and handling layers) and partly needs to be provided by the application developer (the application layer). The first 3 layers contain the machinery responsible for interacting with the external environment, for discovering contexts, selecting and activating contexts and features, and for executing the application. The fourth layer, provided mostly by the application developer, contains the declaration of listeners, filters, contexts, features, their mapping, and the intra- and inter-dependencies between them. This architecture emerged as part of an ongoing reimplementation effort of the Phenomenal Gem COP language. Whereas the current implementation of the architecture is still a proof-of-concept, as future work we will improve upon this initial implementation to make it into a full-fletched implementation architecture. To demonstrate its usefulness we will use it to build case studies of realistic context-oriented software systems with Phenomenal Gem.

References

[1] S. S. Andrade and R. J. d. A. Macêdo. Architectural design spaces for feedback control concerns in self-adaptive systems. In *Software Engineering and Knowledge Engineering*, SEKE'13, Jun 2013. ISBN 2325-9086.

[2] A. Cádiz, S. González, and K. Mens. Orchestrating context-aware systems: A design perspective. In *Workshop on Context-Aware Software Technology and Applications*, pages 5–8. ACM Press, 2009. ISBN 978-1-60558-707-3.

[3] R. Capilla, O. Ortiz, and M. Hinchey. Context variability for context-aware systems. *Computer*, 47(2):85–87, 2014.

[4] N. Cardozo, S. González, R. Van Der Straeten, K. Mens, J. Vallejos, and T. D'Hondt. Semantics for consistent activation in context-oriented systems. *Jour. of Information and Software Technology*, 58(0):71–94, February 2015. ISSN 0950-5849.

[5] N. Cardozo Álvarez. *Identification and management of inconsistencies in dynamically adaptive software systems*. PhD thesis, UCL, 2013.

[6] S.-W. Cheng and D. Garlan. Stitch: A language for architecture-based self-adaptation. *Jour. of Systems and Software, Special Issue on State of the Art in Self-Adaptive Systems*, 85(12), December 2012.

[7] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Dynamic Languages Symposium*, pages 1–10, Oct. 2005.

[8] C.-E. Dessart, V. Genaro Motti, and J. Vanderdonckt. Showing user interface adaptivity by animated transitions. In *Symposium on Engineering Interactive Computing Systems*, pages 95–104. ACM, 2011.

[9] S. González, K. Mens, and A. Cádiz. Context-Oriented Programming with the Ambient Object System. *Jour. of Universal Computer Science*, 14(20): 3307–3332, 2008. ISSN 0948-6968.

[10] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux. Subjective-C: Bringing context to mobile platform programming. In *Proceedings of the Int. Conf. on Software Language Engineering*, 2011.

[11] H. Hartmann and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *Software Product Line Conference*, SPLC '08, pages 12–21, Sept 2008.

[12] IBM. An architectural blueprint for autonomic computing. Technical Report 3, IBM Research, June 2005.

[13] J. Kramer and J. Magee. Self-adaptive systems: An architectural challenge. In *Int. Conf. on Software Engineering*. ACM, 2007.

[14] T. Poncelet and L. Vigneron. The Phenomenal Gem: Putting features as a service on Rails. Master's thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, June 2012.

[15] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, May 2009. ISSN 1556-4665.

[16] D. Weyns, S. Malek, and J. Andersson. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *Transactions on Autonomous and Adaptive Systems*, 7(1), 2012.