An introduction to Context-Oriented Programming

Ivo Willemsen
Open Universiteit
Maastricht, The Netherlands
ivo.willemsen@outlook.com

ABSTRACT

The last decade, use-cases have emerged that emphasize the need to cater to different behaviors depending on situation and context changes. Examples are pervasive systems and highly personalized business applications. Conventional programming languages offer constructs to implement context-dependent behavior, like conditional branches such as if statements, but these branches often result in cluttered code and uses of those constructs seriously damage the modularity of the applications. In the early 2000s, a new programming paradigm emerged, called Context-Oriented Programming which intended to mitigate the aforementioned problems, by incorporating context as a part of the programming language, like variables, classes, and functions, etcetera. This paper presents an introduction to Context-Oriented Programming and explains how the concept of behavioral variations based on changing contexts can be exploited using Context-Oriented Programming languages.

Keywords

Context-Oriented Programming, context-aware systems, behavioral variations, layer activations

1. INTRODUCTION

Many applications present behavior that is determined by the context in which it is being used. Examples of different contexts are: Battery level, GPS-location, available connectivity protocols (Wifi/3G/4G), speed of the network, user's preferences, etcetera.

For example, the battery level of a tablet or a smart phone on which an application (Operating System in this case) runs is a context. This context impacts many behavioral aspects of the application and it will not only have affect on the brightness of the screen, but it might also influence the way the Operating System prioritizes running threads and perhaps result in the preventive hibernation of the system.

In order to include context-dependent behavior in applications using most modern programming languages, one option available is to use the Strategy Design Pattern to abstract the context-dependent behavior into separate classes and decide at runtime-level which context-dependent behavior (strategy) to use. Another option would be the usage of conditional statements to find out the context in which a certain program is running. The usage of conditional statements in these cases would result in not adhering to one of the concepts of Object-Oriented Programming, namely to avoid conditional statements to determine polymorphic behavior. Both options are suboptimal as they result in cluttered code which is difficult to reuse and understand and which makes maintenance of the code a very cumbersome activity.

In many situations, behavioral variations are not implemented by a single object, rather it is spread over a group of cooperating objects. Functionality that is dispersed over several cooperating objects is called a crosscutting concern [8]. Plain old Object-Oriented Programming languages don't have *first-class constructs* that allow for modularization and composition of behavioral variations. A first-class construct [7] is a construct which is an element of a language, like a class or a method in Java.

The lack of aforementioned constructs in regular Object-Oriented Programming languages to support the development of context-aware applications, leaves the developer of those applications with the need to implement necessary *boiler-plate code*. What is needed, is a type of language that incorporates those constructs; which allow a developer to focus more on the implementation of business use-cases and less on inventing the wheel of 'context determination, modularization and activation' over and over again.

The next section will zoom into the concept of 'Context', which contains basic Context-Oriented Programming jargon. The concepts that are explained in this section will be referred to in subsequent paragraphs. The concept of behavioral variations is introduced and an explanation is given on how to define behavioral variations in a class by using partial method definitions and layers. Finally, an elaboration is given of several types of behavioral variation activations.

In order to explain the concepts of Context-Oriented Programming in a more vivid way, a sample use-case will be considered where a screen handler dynamically adjusts its behavior depending of that status of the battery level of a smart phone. For example, in case the battery level (context) falls below a predefined level, the screen should be dimmed (behavioral variation) so that operation time of the smart phone can be extended.

2. CONTEXT-ORIENTED PROGRAMMING

In the early 2000s several approaches have been presented to support the development of context-aware applications. These approaches mainly encompass software architectures and component-based design [10]. In 2008 the Context-Oriented Programming paradigm was proposed [5] as a complementary approach for supporting dynamic adaptation to behavioral variations.

This development was triggered by the appearance of a wide range of scenarios where applications had to react differently according to the active context. Conventional Object-Oriented Programming languages did not incorporate language structures that enabled developers to implement behavioral variations to changes in contexts in a modular way.

Context-Oriented Programming promotes the modularization of context-dependent behavioral variations. It offers abstractions and mechanisms in the form of first-class constructs to define entities that need to change behavior depending on their context. It allows applications to be partitioned into behavioral variations that can be activated at runtime with predefined scopes. These behavioral variations are composed of *partial definitions* for entities, like classes, functions, methods, procedures, etcetera. The essential ingredients of Context-oriented Programming [4] are the following:

- Context
- Behavioral variations
- Layers
- Layer activation
- Scoping

2.1 Context

Context can be defined as any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves [1]. Context is derived from three sources: Environment, system and the actors. Contextual information that originates from actors and the environment reaches the system from outside, but the system itself can also generate context.

A *system* is a set of computational objects, methods, functions that responds with predefined behavior on a per-request basis. An *actor* is a person or another system that is directly involved with the system: It determines the order in which use-cases are executed by communicating directly with the system, by clicking on buttons, sending messages, receiving feedback, etcetera. An example of context generated by an actor is the choices made by a user when accessing certain functionalities that result in differences in behavior. Another example is the support of multiple devices (like an smart phone or a desktop computer) that add to the context-dependent behavior of the system. The *environment* encompasses everything that lies outside the boundaries of the system, which is not directly involved in the relationship between the system and the actors.

Behavioral variations, changes in response, depend on the context that is generated by these three sources. As a result, context-dependent behavior variations can be split up into actor-dependent, environmental-dependent and system-dependent variations.

Context is also commonly described as information that can be computationally accessed in a software system. This definition is kind of vague, as it leaves room for a variety of interpretations:

- The definition does not specify whether the information should originate from outside of the system, the environment, but it also comes from other sources like the actors and the system itself.
- The granularity of abstraction is not enforced by the definition. Context can exist in the system, or can come to the system as fine-grained pieces of information, like indiscrete numerical temperature indications and can subsequently be transformed to more course-grained information classes like 'low', 'medium', 'high'.
- The uniqueness of the context is not imposed. There exist implementations of Context-Oriented Programming languages that enable applications to share only a common context. Other implementations don't follow this model and allow several contexts to exist, which are used in different parts of the application [2]

2.2 Behavioral variations

Context-Oriented Programming languages have first-class constructs that support *behavioral variations* by means of partial definition of modules. Behavioral variations can be *activated*, partly changing the behavior of the application. So, the function of behavior variations is to allow *runtime activation* of a change in behavior and it also allows for the *modularization* of behavior by exploiting the available constructs (by using layers) in the Context-Oriented Programming language.

2.3 Layers

Layers group context-dependent behavioral variations as first-class entities that can be explicitly referred to in a program at runtime. Layers are first-class entities in most Context-Oriented Programming languages, which group partial definitions and can be stored in variables and passed to methods. Several parts of the application can have access to the same layers and determine the exact changes in behavior that have to be done. In case of layers, the first-class entity that implements this functionality is a layered method. This method consists of a base, principle method definition, extended with at least one extra partial definition. This root layer which is always present, representing the base definition, defines the *context-independent* behavior of the application.

The code snippets like the one in figure 1 and in other figures in this paper, are based on the ContextJ implementation (Java-based) [3], but are by no means meant to be compile-ready as they only serve an explanatory goal.

The use-case that is utilized in this paper is a fictitious screen handler and battery handler of a smart phone. This screen handler contains two layers that represent the modularization of behavioral variation with regards to the battery level context. The battery handler checks the level of the battery and sets the layer accordingly.

In figure 1, the base behavior (handle-method) of the Screen-Handler (with no active layers) is to only draw the screen as it is configured at that moment. In case a certain layer has been activated, the corresponding handle-method will be executed. For example, in case the battery handler has detected that the battery is below a certain level, it activates the layer 'LowBattery'. The screen handler will execute the according handle-method (within the LowBattery layer). The proceed-method inside the handle-method will execute the context-independent behavior.

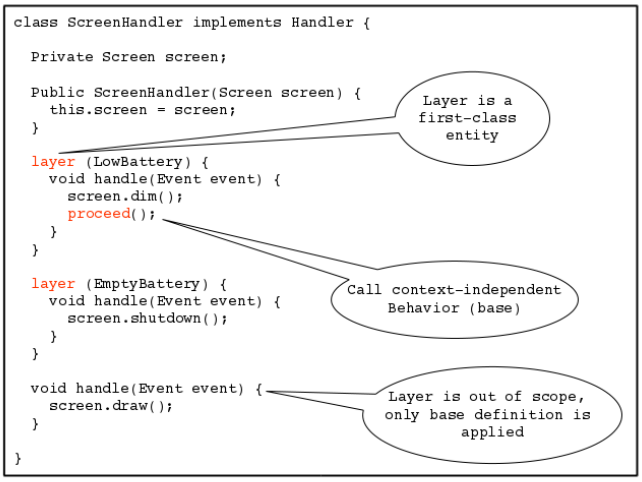

Figure 1: Layers, behavioral variations and base/partial definition

Of course, the above use-case and sample implementation is a very simplified abstraction of what could be a real solution, depending on defined functional and non-functional requirements.

The *layer modularization technique* that has been used to allocate layers in figure 1, is called the *layers-in-class* style allocation. On a per-class basis, layers are defined. This means that layer definitions are *scattered* over all the classes to which a certain behavioral variation should be applied. When there are too many classes to which a certain behavioral variation must be applied, there is another technique to avoid the aforementioned problem, which is called *class-in-layer* allocation of layers. The most optimal choice depends on the number of behavioral variations and classes.

2.4 Layer activation and scoping

The main reason for the existence of Context-Oriented Programming languages is the need to dynamically adapt to different behavioral variations in an elegant and modular way, so this paradigm must provide for regularly changing application behavior. The concept of changing application behavior is implemented by activating and deactivating layers at a certain point in time during the execution of the application. *Layer activation* is achieved by language constructs that ensure that layers are added at runtime, such that certain partial method definitions have an influence on the actual behavior of an application [6].

By default, only the root layer is active at runtime. So only the base definition associated with the root layer will directly influence the behavior of the application. Other layers can be activated and deactivated by using specific keywords that have been added to the programming language or the extension.

The following subparagraphs will shed a light on the different activation mechanisms that have been described in implementation proposals or are available in implementations. It needs to be said that regarding this subject, there is no consensus among either the Context-Oriented Programming language that have been developed or many specifications for Context-Oriented Programming languages that have been published throughout the years. Layer activation and deactivation mechanisms are a complex matter and the most suited approach depends a lot on the architecture and the proposed use-case of the Context-Oriented Programming language.

2.4.1 *Dynamically scoped activation*

Most Context-Oriented Programming languages support the use of *dynamically scoped activation*. A series of layers can be activated by the use of the keyword 'with'. Multiple uses of with-statements in nested calls will add the specified layer to the configuration of layers that are active at that point. The scope of a set of certain active layers is determined by the with-block. At the moment that a with-block terminates, the scope expires and affected layers are removed (*implicit deactivation*) from the configuration of active layers.

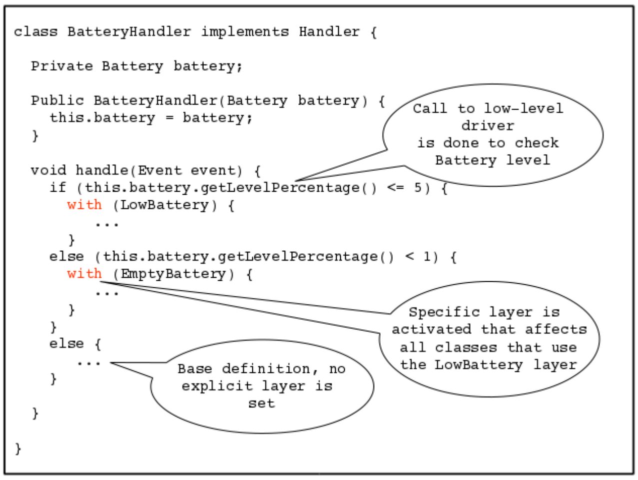

Figure 2: Dynamically scoped activation

The concept of *dynamically scoped activation*, or *direct layer activation*, is depicted in figure 2. In case the call to the low-level driver detects a low battery level, the according layer is set by the 'with' statement. All classes that use this layer will then be affected by this change in context and these classes will activate their behavioral variation dynamically. In this example, the screen handle will dim the screen before proceeding with the execution of the context-independent behavior.

2.4.2 Indefinite activation

Dynamically scoped activation is the preferred mechanism of activation of layers in case of functional behavioral adaption. Situations exists where certain non-functional crosscutting aspects need to be weaved into the application as part of an initial configuration setup. Initially they are applied when probably the applications starts up, and changes to the configuration of active layers are not made during the rest of the execution flow. An example is the incorporation of a logging facility, where a system can be started in debug mode in case issues arise in a production environment. After the production outage has been solved, the system's logging option can be reset and the system can be brought up again in normal mode. To facilitate this, *static layer activation* or *indefinite activation* can be applied [10]. Note that using indefinite activation implies that there are no scope restriction: All classes that have behavioral variations are affected by the layers that are being configured.

A variation to indefinite activation, is implicit activation. Like, indefinite activation, implicit activation influences all the classes without scope restriction, but in the latter case, an explicit configuration of the active layers is done during the execution of the flow.

2.4.3 Per-object activation

This type of activation allows configuration of active layers to be performed on single instances of classes. It can be used as an 'add-on' and usually it is applied in combination with dynamically scoped activation. In cases where, on top of applying a globally scoped activation of behavior, it is also necessary to apply a certain behavioral variation on a small subset of instances of classes, explicit setting the appropriate layer on the instance-level can be done [10]. Figure 3 illustrates the concept of per-object activation.

```
class SmartPhone {
  private Layer layer;
  private ClientRepository clientRepository;
  public SmartPhone(ClientRepository clientRepository, Layer layer) {
    clientRepository = clientRepository;
    layer = layer;
  }

  private Client getClient(int id) {
    clientRepository.setWithLayer(layer);
    return clientRepository.get(id);
  }

}
```

Layer is only activated in the ClientRepository, does not affect other instances

Figure 3: Per-object activation

3. FURTHER READING

This paper gives an introduction to Context-Oriented Programming. A next step could be to get more information regarding Context-Oriented Software Architecture [9]. Also sources are available that discuss Context-Oriented Programming in a more detailed fashion, explaining more about activation mechanisms, design patters and modularization [10]. This paper does not discuss an implementation, there are papers available that elaborate more on real implementations, like the ContextJ implementation for Java [3].

4. CONCLUSION

The purpose of this paper was to introduce novices to Context-Oriented Programming and explain how applications developed with Context-Oriented Programming languages implement behavioral variations based on changing contexts.

A simple hypothetical use-case has been presented and a simplified Context-Oriented Programming language, based on ConttextJ, with a reduced subset of functionalities was used to show how layers, that encapsulate behavioral variations, can be defined. Several types of activation mechanisms were presented that can be used in order to link behavioral variations based on context changes.

5. REFERENCES

[1] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307, London, UK, UK, 1999. Springer-Verlag.

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

[3] 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.

[4] P. Costanza. Context-oriented programming in contextl: State of the art. In *Celebrating the 50th Anniversary of Lisp*, LISP50, pages 4:1–4:5, New York, NY, USA, 2008. ACM.

[5] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Systems and Software*, 7(3):125 – 151, 2008.

[6] T. Kamina, T. Aotani, H. Masuhara, and T. Tamai. Context-oriented software engineering: A modularity vision. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 85–98, New York, NY, USA, 2014. ACM.

[7] R. Keays and A. Rakotonirainy. Context-oriented programming. In *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDe '03, pages 9–16, New York, NY, USA, 2003. ACM.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[9] K. Mens, N. Cardozo, and B. Duhoux. A context-oriented software architecture. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, COP'16, pages 7–12, New York, NY, USA, 2016. ACM.

[10] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801 – 1817, 2012.