Context-Oriented Software Engineering: A Modularity Vision

Tetsuo Kamina
University of Tokyo
kamina@acm.org

Tomoyuki Aotani
Tokyo Institute of Technology
aotani@is.titech.ac.jp

Hidehiko Masuhara
Tokyo Institute of Technology
masuhara@acm.org

Tetsuo Tamai
Hosei University
tamai@acm.org

Abstract

There are a number of constructs to implement context-dependent behavior, such as conditional branches using `if` statements, method dispatching in object-oriented programming (such as the state design pattern), dynamic deployment of aspects in aspect-oriented programming, and layers in context-oriented programming (COP). Uses of those constructs significantly affect the modularity of the obtained implementation. While there are a number of cases where COP improves modularity, it is not clear when we should use COP in general.

This paper presents a preliminary study on our software development methodology, the context-oriented software engineering (COSE), which is a use-case-driven software development methodology that guides us to a specification of context-dependent requirements and design. We provide a way to map the requirements and design formed by COSE to the implementation in our COP language ServalCJ. We applied COSE to two applications in order to assess its feasibility. We also identify key linguistic constructs that make COSE effective by examining existing COP languages. These feasibility studies and examination raise a number of interesting open issues. We finally show our future research roadmap to address those issues.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—Methodologies

General Terms Design, Languages

Keywords Context-oriented programming; Methodology; Use cases

1. Introduction

Context awareness is a major concern in many application areas. It refers to the capability of a system to appropriately behave with respect to its surrounding contexts. A context is identified by observing behavioral changes in the application. An example of a context-aware application is a ubiquitous computing application that differently behaves in relation to situations such as geographical location, indoor or outdoor environment, and weather. In this case, some specific states or situations are contexts. An adaptive user interface is also context aware as it provides different GUI components (behavior) depending on the user's current task (contexts).

There are a number of constructs to implement context-dependent behavior, such as conditional branches using `if` statements, method dispatching in object-oriented programming (such as the state design pattern), and dynamic deployment of aspects in aspect-oriented programming (AOP). Context-oriented programming (COP) [20] also provides another mechanism to implement context-dependent behavior, which is called a *layer*. Uses of those constructs significantly affect the modularity of the obtained implementation, and research in COP shows a number of cases where COP can modularize variations of context-dependent behavior that are difficult to modularize by using other approaches.

However, it is not clear when we should use COP in general because of the lack of a methodology to find context-dependent behavior from the requirements. This problem consists of a stack of subproblems. First, the definition of a context is not clear. A context implies a specific state of a system and/or an environment that affects the system's behavior, but we may find a very large number of such states and environments from the requirements. We need to find the candidate contexts among them and the behavioral variations depending on them that should be implemented by using layers in COP. Second, besides context-dependent behavior, predictable control of change of context-dependent behavior is also important. There are complex relations between contexts (that affect the application's behavior) and variations of behavior, which make the modification of behavioral changes with respect to a change in the specification error prone. Thus, systematic identification of changes in contexts and variations of behavior is required. Third, we need to design modules and dynamic changes of behavior from the identified contexts. For example, selecting modularization mechanisms for context-dependent behavior and context changes is important because these behavior and context changes may be scattered over the whole execution of the application. Finally, we need to map the design to the implementation. A number of COP mechanisms have been proposed thus far [7, 9, 14, 17, 19, 20, 23]; among them, we need to select an appropriate mechanism to implement a design artifact.

This paper presents a preliminary study of our software development methodology, context-oriented software development (COSE) that organizes the specifications of contexts and variations of behavior depending on them. By giving an overview of the devel-

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.
Copyright © 2014 ACM 978-1-4503-2772-5/14/04…$15.00.
http://dx.doi.org/10.1145/2577080.2579816

opment process with COP, even if it is not in depth, it would lead us to further research on each stage of the development process. We hypothesize methods to find context-dependent behavior, and they are validated through two case studies. We provide a mechanized modular mapping from a specification developed by COSE to an implementation in our COP language ServalCJ [26][1]. This preliminary study raises a number of interesting open issues. To address these issues, we finally present a future research plan to further explore the effectiveness of COSE that covers a number of research areas including requirements engineering and programming language implementation.

Methodology. On the basis of the use-case-driven approach [21], COSE represents the requirements for a context-aware application using contexts and context-dependent use cases. A context is represented as a Boolean variable that represents whether the system is in that context[2]. A context-dependent use case is a specialization of another use case applicable only under some specific contexts. From these requirements, COSE further derives a design model that is eventually translated into a modular implementation. This design method classifies variations of context-dependent behavior into those implemented by appropriate mechanisms such as layers in COP and other traditional mechanisms such as class hierarchies and `if` statements. This classification drives mechanized mapping from requirements to implementation. We choose ServalCJ as an implementation language because it provides a generalized layer activation mechanism, which supports all existing COP mechanisms as far as we know. This mapping ensures that each specification in the requirements is not scattered over multiple modules in the implementation, and each module is not entangled with multiple requirements.

Case studies. We demonstrate the effectiveness of this method by conducting two case studies of different context-aware applications. The first one is a conference guide system, which serves as a guide for an academic conference including management of an attendee's personal schedule, navigation help inside the venue and around the conference site, and a social networking service (SNS) function such as a Twitter client. The other one is CJEdit, a program editor providing different functionalities relative to cursor position [8]. In these case studies, we successfully organized context-related specifications by applying COSE and directly mapped these specifications to their implementations in ServalCJ.

To examine the existing language features and discuss what features make the methodology effective, we analyze how COSE addresses the aforementioned problems and the key linguistic constructs that make COSE effective through the case studies. We examine several existing implementation techniques to clarify which constructs will be useful for COSE. A notable finding is that, while most existing COP languages directly specify the execution point when the corresponding context becomes active, in the case studies, the implicit layer activation mechanism where context activation is indirectly specified by using conditional expressions is used intensively. Even though the implicit layer activation mechanism may currently suffer from performance problems, it can be a strong tool to separately implement the dynamic changes of behavior specified in the requirements.

[1] This language was previously known as Javanese.

[2] Keays also proposed COP [28], where a context is a named identifier (e.g., location) that identifies the type of *open terms* (holes in the code skeleton) that are filled at runtime with pieces of code corresponding to a specific value of the context (e.g., location="Tokyo"). This paper is based on Hirschfeld's COP [20] where a context is represented as a *layer* that dynamically takes two states, namely active and inactive, and thus can be represented as a Boolean variable.

Research roadmap. Although the case studies indicate that our approach is promising, we also identify a number of interesting open issues, which comprise our future research roadmap. First, to deal with scattered mentions of context-dependent behavior in descriptions of the system-to-be written in inconsistent syntax, we are planning to further develop a systematized method to identify contexts. Second, our approach is based on use cases; however, it is also desirable to explore how similar approaches can be applied when use cases are not appropriate to analyze requirements. Third, we mention issues in the evaluation of our methodology. Fourth, since there is a performance issue in the implicit layer activation, we are planning to study optimization of implicit activation. It is also interesting to analyze when the event-based activation (i.e., the way in which the execution points where context activation occurs are explicitly represented) is useful and desirable. Finally, since both case studies in this paper are standalone and conducted by using just a single language, it is also desirable to study how the same approach can be applied to more sophisticated environments such as distributed, multi-language environments.

Contributions. The main contributions of this paper are as follows:

- Identification of difficulties in the development of context-aware applications and discussion about the existing approaches (Section 2)
- Systematic organization of context-dependent requirements and classification of them into those implemented by appropriate linguistic mechanisms (Section 3)
- Mechanized mapping from the artifacts obtained by COSE to modular implementation in existing COP mechanisms (Section 4)
- Informal evaluation of COSE through case studies, and identification of key linguistic constructs that make it successful (Sections 5 and 6)
- Provision of the future research roadmap (Section 7)

2. Motivation

We explain the motivation to develop a new context-oriented software development methodology by introducing an example of a context-aware application and explaining the difficulties in the development of context-aware applications and the limitations of the existing approaches.

2.1 Example

We introduce a conference guide system, which serves as a guide for an academic conference, as an example of a context-aware application. This system is implemented on an Android smartphone and provides the conference program, management of the attendee's personal schedule, navigation help inside the venue and around the conference site, and a Twitter client to enable the user to submit their comments on the talks during the conference. This system has a couple of context-related behavioral variations listed as follows:

- The conference program is provided online; the user can view the online program using Internet on the smartphone. The downloaded program is cached on the local database in case the online version becomes unavailable. From the program, the user can select sessions that she/he will attend. The selected sessions are listed on the personal schedule. The listing of the selected sessions is available only when some have been selected.

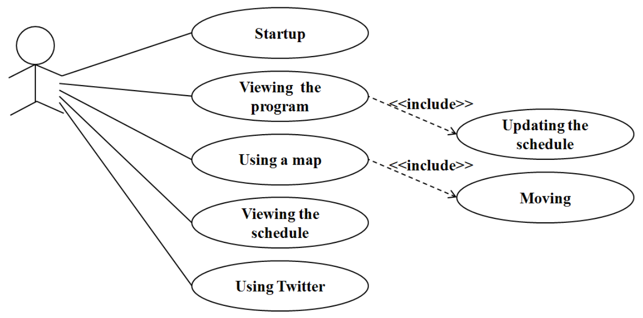

Figure 1. Use case diagram for the conference guide system

- The system provides a map function. When the user is within the conference venue, the map provides a floor plan of that venue. When the user is outside the venue, it provides a city map around the conference site, which is updated when the new position of the user is detected. The positioning is performed on the basis of GPS or the Wi-Fi connection. If the system cannot determine whether it is outdoors or indoors, it provides a static map around the conference site.
- The system provides a Twitter client, which is available only when the Internet is available.

In Figure 1, we summarize the use case diagram for the conference guide system. Besides the "Startup" use case where the user is starting the system, there are four use cases where the user interacts with the system, corresponding to the above itemized listing. Furthermore, the use case "Viewing the program" includes the use case "Updating the schedule" where the user selects sessions to attend, and the use case "Using a map" includes the use case "Moving" where the user is moving and the new position of the user is detected by the positioning system.

2.2 Difficulties

Although this is a simple example, we can observe that there are a number of difficulties in the development of context-aware applications.

Identification of contexts and requirements variability. A context-aware application changes its behavior with respect to current executing context, i.e., there are a number of variations of behavior depending on context. Thus, we need to identify contexts and requirements variability depending on them. For example, in the conference guide system, we may identify contexts such as outdoors, the availability of the list of selected sessions, and the availability of the Internet. However, identification of contexts is not trivial. After the identification of the outdoor context, it is unclear whether we should also identify the indoor context, because it seems that we can represent the indoor context by means of the outdoor context (i.e., indoors=!outdoors).

Different levels of abstraction. Contexts have different abstraction levels, and contexts at the abstract level consist of multiple concrete contexts. For example, the availability of positioning systems depends on the hardware specifications such as the availability of GPS and/or wireless LAN functions. Thus, we need to precisely define contexts in terms of the target machine. This multiple dependency leads to difficulty in precisely defining when the variation of behavior switches at runtime, because there may be a number of state changes in the target machine that trigger a context change, and some states of the executing hardware may barrier or guard the change of abstract contexts.

Multiple dependencies between contexts and behavior. We also need to carefully analyze dependencies between contexts and variations of behavior because some variations depend on multiple contexts. For example, in the conference guide system, if we identify outdoor and indoor situations as different contexts, displaying a static map depends on them, because this behavior is executable only when the system cannot determine whether it is outdoors or indoors. In general, this multiple dependency depends on how we identify contexts, and multiple contexts may barrier or guard the execution of context-dependent behavior. This dependency becomes more complicated when we consider different levels of abstraction of contexts as discussed above.

Requirements volatility in context specification. Technologies for sensing context changes are very complex and evolving continually, indicating that requirements specifications for context sensing are subject to change. For example, at first, it seems appropriate to define the outdoor/indoor contexts on the basis of the status of the GPS receiver. However, this definition may change in the future to use air pressure sensors or other technologies that are not currently implemented in the smartphone (such as an active RFID receiver).

Crosscutting of contexts in multiple use cases. In context-aware applications, a number of contexts are scattered over multiple use cases. For example, in the conference guide system, the conference program is downloaded through the Internet (to let the user access the up-to-date program) only when Internet access is available. Similarly, the availability of the Twitter client depends on the availability of the Internet. Thus, the context "the Internet is available" crosscuts both use cases "Viewing the program" and "Using Twitter." A systematic way to find such a situation and select an appropriate implementation mechanism for this specification is necessary.

Crosscutting of behavior changes. One of the most important properties of context-aware applications is that they change their behavior at runtime. Thus, we need to identify when a variation of behavior switches to another one. However, as discussed above, a variation of behavior may depend on multiple (abstract) contexts, where each context may depend on a number of concrete contexts. Furthermore, changes of such concrete contexts are scattered over the execution of the application. Since their specifications are subject to change, it is desirable to encapsulate them.

Translation to modular implementation. The above difficulties (from the viewpoint of requirements specification) make it difficult to map specifications to modular implementations. We need to carefully trace which requirements are implemented by which modules. It is also desirable that a module in the implementation is not entangled with several requirements but serves only a single requirement. Thus, to support modularity, it is desirable that there is an injective mapping from the specification to the implementation.

2.3 Problems in Existing Approaches

COP languages provide a novel linguistic construct called layers to modularize context-dependent behavior. A number of COP languages have been developed thus far, and some of them share the same abstraction mechanism based on layers and partial methods [7, 9, 14, 19]. On the other hand, little research effort has been devoted for systematizing the design of context-oriented programs. For example, the process of discovering layers from requirements is unclear. Determining when the use of layers is preferable over the use of existing object-oriented mechanisms and `if` statements in order to implement context-dependent behavior also remains unclear. A number of mechanisms have been proposed in COP for dynamic activation of layers. Most of the existing COP languages are based on a dynamically scoped layer activation mechanism using so-called `with`-blocks, which scatters a context activation code over the whole program. Event-based activation of layers with the support of AOP features is proposed to separate the control of layer activation from the base program [9, 23]. A layer activation mechanism that unifies existing COP mechanisms is also proposed [26]. All these mechanisms are useful under the assumption that we have already determined what are contexts and what are behavioral variations depending on them. We require a software development methodology that addresses the aforementioned difficulties.

There have been a number of software development methodologies. Object-oriented methodologies are useful for discovering objects and classes from the requirements and analyzing them. Aspect-oriented software development (AOSD) methodologies [22, 35] are useful for finding crosscutting concerns and modularizing them. Feature-oriented software development (FOSD) [4] is a method that maps feature diagrams [27], which are obtained by analyzing the software to be developed, to implementations. Feature diagrams are useful for analyzing dependencies among features from which software is constructed. Even though these methodologies provide a good starting point to consider how we develop context-aware applications, they do not focus on solutions for the aforementioned difficulties. We need to extend the existing methodologies to systematically identify contexts and behavior depending on them to provide predictable control of change of context-dependent behavior.

Recently, a number of approaches to discover, analyze, and implement contexts and variations of behavior depending on them have been studied. A number of requirements engineering methods [3, 31, 32, 37, 38, 41] mainly focus on discovery and analysis of (abstract) contexts and variations of behavior depending on them. Henrichsen and Indulska propose a software engineering framework for pervasive computing [18]. They do not provide any systematic ways to manage volatile requirements for concrete levels of context, and to modularly implement them. Specifically, they do not identify a set of variations that comprises one single module. Frameworks and libraries for context-aware applications provide context-aware software components and thus enhance reusability, addressing some of the difficulties mentioned above [1, 12, 13, 36]. They are domain specific, and few general solutions for context-aware applications are provided.

The authors previously proposed a metamodel of context-dependent specifications and formalized an injective mapping from specifications to implementations in EventCJ [25]. Although this proposal discusses the entire development process from requirement analysis to the implementation, the mapping from specifications to implementations highly depends on EventCJ [23]. By using this proposal, we can find only context changes that are explicitly triggered by events. Since the metamodel includes detailed specifications of context changes, we need to fix the way of implementation at the earlier stages of development. In contrast, COSE provides a language-independent methodology to elicit contexts and context-dependent behavior[3], enables us to find several types of context changes including implicit ones, and postpones a detailed specification of context changes after designing classes. As a result, the model transformation mentioned in [25] is no longer required but implementations are directly obtained from specifications without transforming the model.

To our knowledge, besides our previous study, this paper is the first attempt to propose a methodology to systematically organize context-dependent requirements and promote modular implementation of them.

[3] Although COSE is language independent, this paper shows mappings from specifications in COSE to implementations in ServalCJ.

2.4 Hypotheses

To address the aforementioned descriptions of context-dependent behavior and problems in existing approaches, we propose the following hypotheses that are assumed in COSE to identify contexts and context-dependent behavior.

HYPOTHESIS 2.1. *The factors dynamically changing the system behavior are candidates for contexts.*

A context is one of the factors that changes the system behavior. Thus, it is a good starting point for identifying contexts to focus on factors that change the system behavior.

HYPOTHESIS 2.2. *A context can be represented as a Boolean variable.*

In many cases, a factor that changes the system behavior takes only two states. For example, the situation whether the user is outdoor takes just two states, yes or no. The availability of the network also takes two states, available or unavailable. The battery level also takes two states, low or not low. Each of these factors can be represented as a Boolean variable.

In some cases, such factors may take more than two states. For example, a location may take a number of values such as "Tokyo," "Lugano," and so on. In such cases, we can consider that each value as a context. For example, we can consider a context like that "whether the user is in Tokyo." This may results in quite a large number of contexts (e.g., we may list thousands of cities), and it is hard to prepare such listing. In general, COP requires pre-listing of variations of behavior, and contexts with a large number are unlikely modularized by using COP but implemented in other techniques such as abstraction over parameters. Thus, in the following sections, we assume that a context can be represented as a Boolean variable.

HYPOTHESIS 2.3. *If multiple variations of context-dependent behavior share the same context, and if such variations are not the specializations of the same behavior, they should be implemented by using a layer.*

This hypothesis explains the situation where "unrelated" variations of behavior are eventually found to be executable in the same situation. This is the situation where the same context is scattered over a number of behavioral variations in the system. A layer in COP can modularize such crosscutting behavior. On the other hand, if the context affects only one single variation of behavior, or if such variations are specialization of the same behavior, we may also consider other implementation mechanisms such as `if` statements and method dispatching in object-oriented programming.

3. Specifying Context-Dependent Requirements and Design

We propose COSE, a use-case-based methodology for context-oriented software engineering. It represents the requirements for a context-aware application using contexts and context-dependent use cases. A context is represented as a Boolean variable that represents whether the system is in that context. A context-dependent use case is a specialization of another use case applicable only in some specific contexts.

On the basis of this requirements model, COSE further derives a design model that is eventually translated into a modular implementation, as shown in Section 4. COSE is based on the use-case-driven approach. It provides a systematic mapping from context-dependent use cases to modules provided by existing COP languages, namely *layers*, just as Jacobson proposed the AOSD method, where each use case is implemented by using an aspect

Table 1. Listing of contexts: the first stage

name	description
hasSchedule	the user has registered at least one session or not
hasNetwork	the Internet is available or not
outdoors	the situation is outdoors or not
hasPositioning	the positioning systems are available or not
batteryLow	the battery level is low or not

[22]. Our design method classifies variations of context-dependent behavior into those implemented by appropriate implementation mechanisms such as layers in COP and those implemented by other traditional mechanisms such as class hierarchies and `if` statements. The following design constituents are identified:

1. Groups of context-dependent use cases, each of which shares the same contexts. Context-dependent use cases in the same group simultaneously become applicable when the contexts hold. To modularize dynamic behavioral changes, they should be modularized into a layer in COP languages.
2. Classes participating in the use cases by applying the standard use-case-driven approach.
3. Detailed specification of contexts based on the identified classes and frameworks on which the system depends.

In the following sections, we overview each step of COSE using the conference guide system example introduced in Section 2.

3.1 Identifying Contexts and Context-Dependent Use Cases

The first step of COSE is to identify contexts and context-dependent use cases. We extend the original use-case-driven method in [21] with context-dependent use cases that are applicable only in specific contexts. By observing use cases, we can see that there exist a number of variations of behavior with respect to some situations or state of the system. As explained in Hypothesis 2.1, these factors changing the system behavior are candidates for contexts. For example, in the conference guide system, we can identify a use case "Startup" where the user starts the system. We can then identify two specializations of "Startup," namely "Startup scheduler" that prepares the menu for the user's schedule, and "Startup Twitter" that prepares the menu for the Twitter client. All these specializations are applicable only when some situations hold such as the availability of the user's schedule and availability of the Internet. Another example is the use case "Using a map," which is specialized to three use cases "Using a city map," "Using the floor plan," and "Using a static map," which are applicable when the user is outdoors, when the user is indoors, and when the system cannot determine the user's situation, respectively.

More precisely, a context in our model is defined as a Boolean variable that represents whether the system is in that context or not. We list the candidates for contexts in the conference guide system in Table 1. This is the very early stage of listing candidates for contexts that are directly observable from behavior of the system-to-be, and should be refined at later steps.

One important criterion on which we rely to identify contexts is that each context should not depend on other contexts, because such dependencies imply that a context can be represented in terms of others. A context and other contexts should be orthogonal, or if they are not orthogonal, they should be exclusive. In the above listing, we can find such a dependency: the situation where no positioning systems are available is a subcase of the situation where the user is not outdoors, because (assuming that the conference guide system determines the situation using positioning systems) the detection of an outdoor situation relies on the availability of positioning systems. Thus, the context outdoors and hasPositioning are divided into three contexts representing outdoors, indoors, and no positioning is available, and the final one is exactly the case where the system cannot determine whether it is outdoors or indoors. The refined listing of contexts is shown in Table 2.

Table 2. Refined listing of contexts

name	description
hasSchedule	the user has registered at least one session or not
hasNetwork	the Internet is available or not
outdoors	the situation is outdoors or not
indoors	the situation is indoors or not
batteryLow	the battery level is low or not

Table 3. Use cases for the conference guide system

name	context
Startup	
Startup scheduler	hasSchedule
Startup Twitter	hasNetwork
Viewing the program	
Viewing the online program	hasNetwork
Updating the schedule	
Using a map	
Using a city map	outdoors
Using the floor plan	indoors
Using a static map	!outdoors && !indoors
Moving	
Moving when outdoors	outdoors
Viewing the schedule	hasSchedule
Using Twitter	hasNetwork
Updating timeline frequently	!batteryLow
Updating timeline infrequently	batteryLow

Note that, as discussed in Section 2, requirements for context changes are often volatile. Thus, at this stage, it is preferable to keep contexts abstract to be prepared for future changes of requirements.

A context-dependent use case is a use case annotated with a proposition where ground terms are contexts that specifies when this use case is applicable. Context-dependent use cases for the conference guide system are summarized in Table 3. The names of use cases are listed in the left column, and conditions in terms of contexts that represent when the use case is applicable are listed in the right column. A name with an indent represents that this use case is a specialization of the use case listed in the above row with the italic font. A use case with an empty condition is context independent.

3.2 Grouping Context-Dependent Use Cases

A situation where multiple use cases are applicable in the same context implies that the context-dependent behavior is scattered over those use cases. To modularize dynamic behavioral changes, these context-dependent use cases should be grouped into one module that is enabled (activated) when the condition holds, and disabled (deactivated) when the condition does not hold. This is the situation where the Hypothesis 2.3 explains, which is rephrased in terms of the use case driven method as follows: *if multiple context-dependent use cases that are not specializations of the same use case share the same context, their behavior should be implemented by using a layer.*

Table 4. Groups of context-dependent use cases

context	use case
hasSchedule	Startup scheduler Viewing the scheduler
hasNetwork	Startup Twitter Viewing the online program Using Twitter
outdoors	Using a city map Moving when outdoors
indoors	Using the floor plan
!outdoors && !indoors	Using a static map
hasNetwork && !batteryLow	Updating timeline frequently
hasNetwork && batteryLow	Updating timeline infrequently

Table 5. Classes for each layer

layer	classes	position
`HasSchedule`	`MainActivity`, `Schedule`	class-in-layer
`HasNetwork`	`MainActivity`, `Program`, `Twitter`	class-in-layer
`Outdoors`	`Map`	layer-in-class
`Indoors`	`Map`	layer-in-class
`StaticMap`	`Map`	layer-in-class

Table 4 lists the groups of context-dependent use cases. We can see that three contexts, hasSchedule, hasNetwork, and outdoors, are assigned to multiple context-dependent use cases. Thus, these use cases are grouped into a layer; from now on, we rename these contexts by capitalizing the first character like `HasSchedule`, `HasNetwork`, and `Outdoors`, respectively, following the tradition of the naming of layers in COP languages.

Now, the question is how to treat the remaining context-dependent use cases. Even though they do not share the condition with other use cases, some of them still have a relationship with other layers in that a subterm of their condition is the condition that activates the layer. For example, the condition for "Using a static map" includes a subterm outdoors, which is the condition that activates the layer `Outdoors`. To uniformly control dynamic changes of behavior, activation of "Using a static map" should be managed in the same way as that of `Outdoors`. Thus, we also identify the context-dependent use case "Using a static map" as a layer, namely `StaticMap`. Similarly, we identify the context-dependent use case "Using the floor plan" as a layer, namely `Indoors`.

Other context-dependent use cases are not implemented by using layers. They are conceptually the same as alternative use cases, and the behavioral variations represented by such use cases should be implemented by traditional OO mechanisms such as inheritance and `if` statements.

3.3 Designing Classes

Each layer in COP consists of (partial) definitions of classes. By straightforwardly extending the original use-case-driven approach, we can identify classes and methods participating in each layer.

First, from use case scenarios, we identify the names of classes. Due to the limited space, we do not describe the details, but briefly illustrate the result. Since the conference guide system is an Android application, each view of the application should be implemented as a subclass of the `android.app.Activity` class from the Android SDK framework[4]. The use case "*Startup*" identifies the `MainActivity` class, which will implement the main view of the application. Similarly, in the use cases "*Viewing the program*," "*Using a map*," "*Viewing the schedule*," and "*Using Twitter*," we identify an `Activity` class for each of them, namely `Program`, `Map`, `Schedule`, and `Twitter`. There are some other helper classes; however, only the `Activity` classes participate in the context-dependent behavior.

[4] `http://developer.android.com/sdk/`

Table 5 summarizes this assignment of classes for each layer. While layers `HasSchedule` and `HasNetwork` consist of multiple classes, other layers consist of just one class `Map`. This table also shows the preferred ways to allocate layers. There are two alternative ways to allocate layers: the class-in-layer style allocates the (partial) classes that implement the context-dependent behavior in the layer, while the layer-in-class style allocates the layer within the class. When a layer is scattered over several classes, the class-in-layer style is preferable, while when a class is scattered over several layers, the layer-in-class style is better. Note that some COP languages support only one style [6]. In this case, we need to conform to the style provided by the implementing language.

3.4 Designing Detailed Specification of Contexts

The contexts identified above are abstract. Since we have identified a number of classes in use case scenarios, we can now provide more concrete definitions for them. In the following, we define when the context becomes active in terms of classes identified above and classes from the framework. As explained later, specifications for some contexts are complex; thus, we need to identify more fine-grained contexts that comprise the specified context.

Section 3.1 defines that the context hasSchedule holds when the user has registered at least one session to attend from the conference program. In terms of the Android SDK framework, this is represented as "a query on the `SQLite` instance returns at least one result." Thus, we define when the layer `HasSchedule` becomes active as follows, which is read as "the `getCount` method on the result of a `query` on an `SQLite` instance (namely `db`) returns an integer value that is greater than 0":

```
HasSchedule(SQLite db) ::
  db.query(..).getCount() > 0
```

Similarly, by inspecting the specification of the Android SDK framework, we define when the layer `HasNetwork` becomes active as "the result of the `getDetailedState` method on the result of `getActiveNetworkInfo` on a `ConnectivityManager` instance (namely `cm`) is equal to `NetworkInfo.DetailedState.CONNECTED`":

```
HasNetwork(ConnectivityManager cm) ::
  cm.getActiveNetworkInfo().getDetailedState() ==
    NetworkInfo.DetailedState.CONNECTED
```

The cases for outdoors and indoors contexts are more complex. They are affected by multiple states of the running machine. First, to determine whether the user is outdoors, the GPS device should be available. Second, the conference guide system determines whether the user is in the conference venue by using the SSID of the connecting wireless LAN, which means that the wireless LAN connection should be available. Thus, the activation of layers `Outdoors` and `Indoors` is determined in terms of more fine-grained contexts:

```
Outdoors :: !WifiAvailable && GPSAvailable
Indoors :: WifiAvailable
```

In other words, `Outdoors` and `Indoors` are composite layers [24].

The context `WifiAvailable` is defined as follows, assuming that `isWifiConnected` is an application method that returns `true` when the wireless LAN is connected and its SSID is some predefined value:

```
WifiAvailable :: Config.isWifiConnected()==true
```

The context GPSAvailable is defined as follows using the isProviderEnabled method provided by the framework:

```
GPSAvailable ::
  LocationManager.isProviderEnabled(
    LocationManager.GPS_PROVIDER) == true
```

4. Mapping to Implementation

This section demonstrates how the facts discovered by COSE are systematically translated into a program with existing COP mechanisms. We choose ServalCJ [26], which is a successor of EventCJ [23], as an implementation language because it provides a generalized layer activation mechanism that supports most existing COP mechanisms. A context in ServalCJ is defined as a term of temporal logic with a call stack, which can represent most existing layer activation mechanisms. For example, it can specify two events, of which one activates the corresponding context and the other deactivates that context (as in EventCJ's event-based layer transition). ServalCJ can also specify a control flow under which the corresponding context is active (as in JCop [9]). ServalCJ can select the target where such context specifications are applied, and that target can be a set of objects (*per-instance* activation) or the whole application (*global* activation). Furthermore, ServalCJ supports *implicit activation*, where activation of a context is indirectly specified by using a conditional expression. As shown in the following sections, our methodology clarifies that this mechanism is notably useful for modular implementation.

A ServalCJ program comprises a set of classes, layers, and *context groups* where dynamic layer activation and the target for this activation are specified. Layers and classes identified in Sections 3.2 and 3.3 are directly implemented in layers and classes in ServalCJ. Context specifications in Section 3.4 are directly implemented in context groups in ServalCJ. We explain the details in the following sections.

4.1 Implementing Layers

As in other COP languages, layers and partial methods comprise the mechanism for modularization of context-dependent behavior in ServalCJ.

Figure 2 shows an example of layers and partial methods in ServalCJ for the main view of the conference guide system. The class MainActivity extends the Activity class provided by the Android SDK framework, and overrides the onResume method, which is called from the framework when this view resumes the execution. This method displays the main menu of the conference guide system as buttons for viewing the conference program and using the map. MainActivity also declares two layers HasSchedule and HasNetwork. These layers define the context-dependent behavior of MainActivity[5]. HasSchedule defines the behavior when there is at least one session that the user would like to attend, and HasNetwork defines the behavior when the Internet is available. These layers extend the original behavior of onResume by declaring *after* partial methods, which are executed just after the execution of the original method when the respective layer is active[6]. For example, when HasSchedule is active, onResume also displays the menu button to check the user's schedule.

[5] Although Table 5 shows that it is preferable to implement these layers in the class-in-layer style, in Figure 2, they are implemented in the layer-in-class style because currently ServalCJ only supports the layer-in-class style.

[6] There are also *before* and *around* partial methods that execute before the execution of the original method and instead of the original method, respectively, when the respective layer is active.

```
class MainActivity extends Activity
    implements View.OnClickListener {
  private GridLayout layout;

  @Override
  protected void onResume() {
    super.onResume();
    layout = new GridLayout(this);
    layout.addView(makeMenu("program", "Program"));
    layout.addView(makeMenu("map", "Map"));
  }

  private Button makeMenu(String tag,
                          String label) {
    ..
  }

  layer HasSchedule {
    after protected void onResume() {
      layout.addView(makeMenu("schedule",
                              "Schedule"));
    }
  }
  layer HasNetwork {
    after protected void onResume() {
      layout.addView(makeMenu("twitter",
                              "Twitter"));
    }
  }
}
```

Figure 2. Layers and partial methods in ServalCJ

```
contextgroup Network(ConnectivetyManager cm)
    perthis(this(ConnectivityManager)) {
  context HasNetworkContext if(
    cm.getActiveNetworkInfo().getDetailedState()
      ==NetworkInfo.DetailedState.CONNECTED);
  activate HasNetwork when HasNetworkContext;
}
```

Figure 3. Context group responsible for activation of HasNetwork

4.2 Implementing Layer Activation

In COP languages, we can dynamically activate and deactivate layers, and ServalCJ provides declarative ways to perform such layer activation. These declarations are directly obtained from the design of detailed contexts discussed in Section 3.4.

First, detailed context definitions are further grouped on the basis of the variables and contexts that these definitions refer to. For example, HasNetwork refers to an instance of ConnectivityManager (and this is the only context that refers to that instance); thus, it makes up one group, which we call a *context group*.

Figure 3 shows a context group that is responsible for activation of HasNetwork. The first line specifies the name of the context group, which is Network, followed by a specification of how this context group is instantiated. The perthis clause specifies that the instance of Network is associated with an instance of ConnectivityManager (as specified using the this pointcut), which can be referenced through the variable cm.

```
contextgroup Schedule(MainActivity main)
    perthis(this(MainActivity)) {
  context HasScheduleContext if(
    main.scheduleCounter > 0);
  activate HasSchedule when HasScheduleContext;
}
```

Figure 4. Context group responsible for activation of `HasSchedule`

```
contextgroup Situation {
  context WifiAvailable if(
    Config.isWifiConnected()==true);
  context GPSAvailable if(
    LocationManager.isProviderEnabled(
      LocationManager.GPS_PROVIDER)==true);
  activate Outdoors
    when !WifiAvailable && GPSAvailable;
  activate Indoors when WifiAvaileble;
  activate StaticMap
    when !Outdoors && !Indoors;
```

Figure 5. Context group responsible for activation of `Outdoors`, `Indoors`, and `StaticMap`

Line 3 defines a context `HasNetworkContext`, which is used to specify when `HasNetwork` is active. The syntax of context declaration is as follows:

`context` *ContextName Term* ;

It starts with the keyword `context` followed by the name of the context and the specification of when that context is active. There are several ways to specify context activation, e.g., to specify the join points where that context becomes active and inactive, to specify the control flow under which that context is active, and to specify the condition when that context is active. In Figure 3, we specify the condition, which is declared by using the `if` expression. In the `if` expression, we can use any Boolean-type Java expressions, and in this case, we just copy the expression from the definition in Section 3.4.

Line 6 declares when the layer `HasNetwork` is active using an *activate declaration*. The `when` clause specifies the condition when the layer is active in terms of contexts; i.e., if `HasNetworkContext` is active, `HasNetwork` is active.

We can also declare a context group for `HasSchedule` in a similar way. One subtle issue is that the definition of `HasSchedule` contains an expression that requires local database accesses. If the developer has performance concerns, this definition is not preferred, because in ServalCJ, this condition is tested at every call of the layered method (i.e., a method that consists of a set of partial methods). In our case, the definition of `HasSchedule` is refined to access the counter variable that is introduced to `MainActivity` and updated when the local database is updated:

```
HasSchedule(MainActivity main) ::
  main.scheduleCounter > 0
```

The definition of the context group for `HasSchedule` is shown in Figure 4.

The remaining layers are `Outdoors`, `Indoors`, and `StaticMap`. Since they share the same set of context references, we group them into one context group, which is shown in Figure 5. Since this context group does not refer to any instance variables, it specifies no `perthis` and `pertarget` clauses. This context group is a *singleton*, i.e., it is created at the beginning of execution of the application and remains until it terminates.

Context declarations for `WifiAvailable` and `GPSAvailable` are directly obtained from the definitions in Section 3.4. Furthermore, activate declarations for `Outdoors`, `Indoors`, and `StaticMap` are also directly obtained from the definitions in Section 3.4. Note that we can use the logical operators `||`, `&&`, and `!` to compose propositions in the `when` clauses.

Finally, we need to decide the sets of instances where these context groups are applied. ServalCJ supports per-instance activation, where a context group is applied to a specified set of instances, and global activation, where a context group is applied to the whole application. In the conference guide system, we decide that all context groups are global because per-instance activation is not important in this system. All the types of instances that should be under the control of some context groups are subtypes of `Activity`, and their instantiation is totally controlled by the Android SDK framework. There should not be cases where multiple instances of the same `Activity` class coexist simultaneously. A context group is global at the initial setting. Thus, the context groups shown in Figures 3, 4, and 5 are global.

Table 6. Listing of contexts for CJEdit

name	description
cursorOnCode	the cursor is on code
RTF	the text renderer renders comments

5. CJEdit: Another Case Study

This section demonstrates another case study using COSE. CJEdit [8], which was first implemented by Appeltauer, is a program editor that enhances the readability of programs by providing different text-formatting techniques for code and comments. The code part is rendered in a typewriter format with syntax highlighting, and the comment part is rendered in a rich text format (RTF) that supports multiple fonts, text sizes, decorations, and alignments. Furthermore, CJEdit provides different GUI components depending on whether the programmer writes code or comments. For example, when the user is editing code, CJEdit displays the "execute" menu to quickly test the code currently being edited. This application is implemented using the QtJambi framework[7]. We use this example to investigate how COSE fits the development of existing context-aware applications.

Since the original implementation of CJEdit already exists, we do not perform this case study from scratch. We use the original implementation as a prototype of this case study, and by observing the system's behavior, we first derive the contexts listed in Table 6. The context cursorOnCode holds when the cursor is on code. There is also a context for text-rendering regions: RTF holds when the text renderer renders comment regions.

Table 7 lists context-dependent use cases for CJEdit. In CJEdit, we identify the use case "*Editing a program*," which includes another use case "*Displaying the source code*." We derive context-dependent use cases from these use cases. "*Editing a program*" is specialized to different use cases with respect to the cursor's position; "Writing code" is applicable only when the context cursorOnCode holds, and "Writing comments" is applicable only when the context cursorOnCode does not hold. "Displaying the source code" is specialized to three different use cases depending on the text region and the cursor's position; "With syntax highlighting" is applicable only when the context cursorOnCode && !RTF holds; "Without syntax highlighting" is applicable only when the context !cur-

[7] `http://qt-jambi.org`

Table 7. Use cases for CJEdit

name	**context**
Editing a program	
Writing code	cursorOnCode
Writing comments	!cursorOnCode
Displaying the source code	
With syntax highlighting	cursorOnCode && !RTF
Without syntax highlighting	!cursorOnCode && !RTF
RTF format	RTF
Execute	cursorOnCode

Table 8. Layers for CJEdit

layer	**use case**
`CodeEditing`	Writing code *Execute*
`CommentEditing`	Writing comments
`RenderWithHighlighting`	With syntax highlighting
`RenderWithoutHighlighting`	Without syntax highlighting

Table 9. Classes for each layer of CJEdit

layer	**classes**

`CodeEditing`	`TextBlock,TextEditor`
	`FileExecutor`

`CommentEditing`	`TextEditor`
`RenderWithHighlighting`	`SyntaxHighlighter`
`RenderWithoutHighlighting`	`SyntaxHighlighter`

sorOnCode && !RTF holds; "RTF format" is applicable only when the context RTF holds. "*Execute*" is applicable only when the context cursorOnCode holds.

The next step is to group all context-dependent use cases with the same context into one single layer. On the basis of Hypothesis 2.3, we group "*Editing a program*" and "*Execute*" into the layer `CodeEditing`. The remaining context-dependent use cases do not share their context with other use cases; however, we still need to further group each of them as a distinct layer because a subterm of their context is a (subterm of a) context that activates a layer. Thus, we finally obtain the layers listed in Table 8.

Next, we identify the names of classes from the use case scenarios. Table 9 lists important classes for implementing context-dependent behavior. While the `CodeEditing` layer consists of multiple classes, other layers consist of just one class.

Now, we can define the detailed specifications of contexts in terms of classes. As specified in Table 6, the context cursorOnCode holds when the cursor is on code. This condition is represented by using the application method `isCursorOnCode` that returns `true` when the cursor is on code:

```
CursorOnCode :: TextEditor.isCursorOnCode()
```

The context `RTF` is also defined in terms of an application method that returns a text block by inspecting the type of that text block:

```
RTF :: SyntaxHighlighter.getCurrentBlock()
  instanceof RTFBlock
```

We directly implement the above systematized specification using ServalCJ. Layers listed in Table 8 are translated to layer declarations in ServalCJ. Figure 6 illustrates the layers affecting the behavior of `TextEditor`. They change the arrangement of GUI components by introducing partial methods, which are executable only when the corresponding layer is active.

```
class TextEditor {
  void showWidgets() { .. }
  void showToolbars() { .. }
  void showMenu() { .. }

  layer CodeEditing {
    after void showWidgets() { .. }
    after void showToolbars() { .. }
  }
  layer CommentEditing {
    after void showMenu() { .. }
    after void showToolbars() { .. }
  }
}
```

Figure 6. Layers and partial methods for CJEdit

```
contextgroup CJEdit(TextEditor editor)
    perthis(this(TextEditor)) {
  context CursorOnCode if(
    editor.isCursorOnCode());
  context RTF if(
    editor.getHighlighter().getCurrentBlock()
      instanceof RTFBlock);
  activate CodeEditing when CursorOnCode;
  activate CommentEditing
    when !CursorOnCode;
  activate RenderWithHighlighting
    when CursorOnCode && !RTF;
  activate RenderWighoutHighlighting
    when !CursorOnCode && !RTF;
}
```

Figure 7. Context group for CJEdit

The detailed specification of contexts is also directly implemented in the context group shown in Figure 7. Since the methods used in the specification of contexts are instance methods, we bind the instance of `TextEditor` with the local variable `editor` of the context group `CJEdit`, and the condition that specifies which context is active is specified by using the corresponding variable. Activate declarations that specify when the corresponding layers are active are directly obtained from Tables 7 and 8.

6. Discussing Modularity

The case studies demonstrate our hypotheses on when we should use COP. In this section, we summarize the result of case studies and validate our hypotheses. By comparing ServalCJ with other languages and implementation techniques, we also explore what are the key functionalities of the implementing language to make our approach effective. Finally, the case studies lead us to further research on each stage of the development process from the requirement analysis to the implementation.

6.1 Summary of Case Studies

In Section 2.2, we identified several difficulties in development of context-aware applications. Our approach COSE addresses them as follows.

Identification of contexts and requirements variability. As illustrated in Section 3.1, COSE systematizes identification of contexts by observing behavior of the system-to-be, such as use cases and prototypes. Furthermore, we clarify a criterion that should hold for each context, which is that a context should not be a subcase of other contexts. Requirements variability based on contexts is also represented by context-dependent use cases.

Different levels of abstraction. As discussed in Sections 3.1 and 3.4, COSE provides a concretization process for contexts. A context may be composed of other contexts that are less abstract than the composed context. Each level of abstraction of contexts in the specification is also directly represented by the implementation language using composite layers.

Multiple dependencies between contexts and behavior. As discussed above, because of composite layers, a layer can be composed of a number of contexts.

Requirements volatility in context specification. Each context-dependent use case is represented in terms of abstract contexts, and thus it is rigorous for future changes of detailed specifications of concrete contexts. For example, in the conference guide system, the specification of the outdoor context may change according to future evolution of sensor technologies that detect outdoor and indoor situations. Context-dependent use cases depending on the outdoor context will not be affected by such changes because the detailed specification of the outdoor context is abstracted from the context-dependent use cases. We may also separately perform such changes because definitions of contexts are encapsulated in context groups in ServalCJ.

Crosscutting of contexts in multiple use cases. COSE groups a number of variations of behavior that are executable under the same contexts and scattered across multiple use cases into one single layer. As discussed in Section 3.2, it also provides a guideline for when to use COP.

Crosscutting of behavior changes. Dynamic changes of contexts and behavior depending on them, which are scattered across the whole execution of the program, are separated as specifications of contexts and directly implemented by using context groups. Specifically, definitions of such changes are declaratively specified and totally separated from the base program.

Modular translation to the implementation. Layers and classes identified in Sections 3.2 and 3.3 are directly implemented in layers and classes in ServalCJ. Context specifications in Section 3.4 are directly implemented in context groups in ServalCJ. Each requirement in the specification is not scattered across multiple modules in the implementation, and each module is not entangled with multiple requirements.

6.2 Validating the Hypotheses

The results of case studies discussed above confirm the validity of Hypotheses 2.1 and 2.2. The case studies reveal that the factors changing the system behavior are actually "candidates" for contexts, and each context can be represented as a Boolean variable. This representation of contexts further derives a criterion to identify contexts, which is that each context at the abstract level should not depend on other contexts. A context and other contexts should be orthogonal, or if they are not orthogonal, they should be exclusive. This criterion enhances the exhaustiveness of contexts and makes it easy to discuss the equivalence between contexts.

For the Hypothesis 2.3, however, we need to further discuss the validity of our decision to implement the variations of context-dependent behavior using layers, because there are other alternatives to implement such variations, and above case studies do not discuss the cases where we do not use COP even when COSE indicates that we should use that.

We can validate it by using Tables 4 and 5. First, the layers `HasSchedule` and `HasNetwork` crosscut across multiple classes, and thus the same concern may scatter over those classes if we naively implement them using `if` statements. Applying design patterns may also produce this scattering problem. Extracting such scattered code as a common superclass requires an additional class hierarchy, which may be orthogonal to the existing hierarchies. Applying multiple inheritance, mixins [11], and traits [40] makes it difficult to take a look at the all classes that are composed with the same context-dependent behavior. In contrast, layers in COP provide a good solution to separate such concerns. More importantly, using the techniques other than COP makes it hard to separate behavior changes from the base program, which is possible in (some variants of) COP languages.

On the other hand, the layers `Outdoors`, `Indoors` and `Static-Map` in Table 5 exist only one single class `Map` and thus they do not seem to contribute to separation of crosscutting concerns. From Table 4, however, we can observe that `Outdoors` consists of two use cases, which are implemented by different methods, and using `if` statements would results in scattering of the same conditions over those methods. We may also avoid this scattering by, for example, to allow the `Map` object to have a state of the current situation, and to define behavioral variations for each state by using the state design pattern. The problem in applying design patterns is the scattering and tangling of behavioral changes. The state changes of the `Map` object are triggered by external environment changes, which are observed by the framework. We need to embed state changes of the `Map` object by implementing appropriate event handlers of possibly multiple modules (such as Wifi and GPS related classes). Thus, it becomes hard to localize the overall state changes in the `Map` object. By applying COSE with appropriate COP languages, we can separate such context changes into one single module.

Similar discussion holds in the case study of CJEdit. Thus, all decisions in this paper to implement variations of context-dependent behavior using layers are valid.

6.3 Comparison with Other Activation Mechanisms

The implementation in ServalCJ discussed in Section 4 implies that, in our approach, it is not necessary to *transform* the model of the requirements into that of the implementation. Instead, the implementation is *directly obtained* from the requirements. There are injective mappings from layers and contexts discovered in the requirements to those in the implementing language. Thus, this mapping promotes separation of concerns in that requirements are not scattered across several modules in the implementation, and each module is not entangled with a number of requirements.

The implementations in the case studies rely on the specific linguistic constructs provided by ServalCJ. In this section, we identify what are the properties that the implementing languages should have to make COSE effective, and compare ServalCJ with other languages and implementation techniques, such as ContextJ [7], EventCJ [23, 24], and a pseudo AOP language with the dynamic layer activation mechanism, with respect to those properties. Table 10 summarizes the result of the comparison. The leftmost column shows the numbers and titles of the following sections.

We do not argue that programming languages that do not support features listed below are not useful in COSE. In such languages, we may still apply useful workarounds to implement specifications organized by COSE, which would not be a bad choice in some circumstances such as availability of libraries and a development environment, and programmer's preference. Nevertheless, Table 10 indicates that recent progress in COP languages effectively supports COSE, which will be a good input for future language design.

Table 10. Comparison with other activation mechanisms

	ContextJ	AOP+COP	EventCJ	ServalCJ
6.3.1 Separation of context-dependent behavior[8]	a	a	a	a
6.3.2 Separation of context changes	n/a	a	a	a
6.3.3 Expressing relations between layers and contexts	n/a	n/a	a	a
6.3.4 Implicit activation	n/a	n/a	n/a	a

6.3.1 Separation of context-dependent behavior

First, in COSE, the implementing language should separate context-dependent behavior that is dynamically enabled and disabled from the base program. Layers of COP languages provide an effective way to achieve this purpose. Each partial method implements the context-specific behavior of the base method, and a layer packs all partial methods executable under the same context into one single module. Besides COP, other programming paradigms such as AOP and feature-oriented programming (FOP) [34] also provide such a modularization mechanism; however, for these paradigms, we also require an additional mechanism for dynamic composition of modules. For example, dynamic aspect deployment [10] may be applied for this purpose.

6.3.2 Separation of context changes

We can also see that, in COSE, specifications and implementations of dynamic changes of contexts and behavior depending on them are also separated from other specifications and modules, respectively. From the implementation viewpoint, such dynamic changes can easily be scattered over the whole application execution. Such scattering behavior can be avoided by using the pointcut-advice mechanism provided by AspectJ [29] (provided that it is also equipped with some imperial layer activation mechanism), or other COP languages with AOP features such as EventCJ and JCop [9].

In some COP languages, layer activation is controlled in a *per-thread* manner, whereby the generation of the event activating the layer and layer activation occur synchronously. In such languages, it is difficult to separate dynamic changes of behavior. For example, in ContextJ, layer activation is expressed by using the `with`-blocks, which ensures that layers are active only within the explicitly specified dynamic scope:

```
with (activeLayers) { onResume(); }
```

However, context changes are triggered by external events that asynchronously occur with the dynamic change of behavior. For example, in this case, we need to remember the active layer within the body of the event handler that handles the change of contexts to activate context-dependent behavior that does not appear in the scope of the event handler:

```
void someEventHandler(Event e) {
  activeLayers.add(Outdoors);
}
```

In this case, the scattering problem is readily encountered, and the base program is entangled with the concerns about dynamic changes of behavior.

6.3.3 Expressing relations between layers and contexts

From COSE, we can also see that a variation of behavior may depend on multiple contexts. For example, from Table 4, we can see that the use case "Using a static map," which is implemented in the layer `StaticMap`, depends on both contexts outdoors and indoors, one of which, namely outdoors, is further decomposed into two contexts `WifiAvailable` and `GPSAvailable`. To separate context-dependent behavior from the detailed specification of contexts, such an abstraction mechanism is necessary. From the implementation viewpoint, composite layers [24], which are supported by EventCJ and ServalCJ, are useful for this purpose.

6.3.4 Implicit activation

In most existing COP languages, we need to explicitly specify the join point where the context change occurs. In COP languages with AOP features, we perform such specification using the pointcut sublanguage. In COP languages with `with`-blocks, we explicitly inject the layer activation block into the base program. However, from the case studies, we have learned that a more declarative way to specify the condition when the corresponding context is active is heavily used in the context specification, which is directly implemented by using the implicit layer activation mechanism provided by ServalCJ (i.e., the `if` condition that specifies the condition when the corresponding context is active). This fact indicates that, even though it currently suffers from performance problems, the implicit layer activation mechanism can be a strong tool to modularly implement dynamic changes of behavior from the specification.

It is also possible to manually translate implicit layer activation into the explicit activation by identifying the join points where the condition is changed. However, when there are such multiple join points, we need to list all of them, which is an error-prone task. Furthermore, explicitly specifying the join points using pointcut often encounters the fragile pointcut problem [30].

6.4 Open Issues

Our preliminary case studies on COSE raise the following open issues that should be further explored.

First, both case studies in this paper are simple. Although these case studies demonstrate the effectiveness of COSE, they do not promise success in more complex cases. In large systems, we may have a large number of dynamic changes in behavior, some of which are context dependent. Eliciting contexts from such systems may be time consuming. Furthermore, in both case studies, the target system is standalone and implemented by using one single programming language. We should not consider that the results of the case studies immediately imply that we can easily apply COSE to distributed systems implemented by using multiple programming languages.

Second, COSE represents variations of context-dependent behavior using use cases. There should be some cases where we may prefer to use methods other than use cases, such as feature diagrams and goal models. The results in this paper do not ensure that we can also apply similar context-oriented extensions to those methods.

Third, the case studies do not convey compelling results regarding costs and benefits of COSE. The results ensure modularity of the products. However, they do not reveal how such modularity affects the real software production process and the quality of its products. We believe that COSE would have a significant impact on software development, in particular on software maintenance, because it provides comprehensive abstractions, clarifies complex

[8] ServalCJ (and EventCJ) only supports the layer-in-class style. Thus, the same layer may be scattered across multiple classes. In fact, such layers exist in both case studies. This scattering can be addressed by supporting the class-in-layer style in the syntax.

relations between contexts and behavior, and provides good modularity in its products. However, this hypothesis should be validated through a lot of control experiments. Furthermore, the hypotheses explained in Section 2.4 should also be validated through a number of demonstration experiments and industrial software development.

Finally, as mentioned above, there are open issues in the performance of implicit activation, which is heavily used in the case studies. The performance problem of implicit activation is not significant in the case studies. However, this assumption will not always hold in applications of larger sizes. In some cases, we may optimize implicit activation, but there may be other cases where such optimization is not feasible. The case studies do not clarify when to use implicit activation and when to use other mechanisms such as event-based activation.

7. Future Research Roadmap

In this paper, we presented COSE and proposed that it can be employed for the effective development of context-aware applications. Specifications systematized by COSE effectively represent different levels of abstraction of contexts, which makes the system rigorous with respect to the change of detailed definitions of contexts. Context-dependent use cases are used to discover a layer, a modularization unit in COP, from the specification. The injective mapping from specifications to implementations ensures that each specification in the requirements is not scattered across multiple modules in the implementation, and each module is not entangled with multiple requirements. The comparison among several implementation techniques shown in Section 6.3 reveals the key linguistic constructs that make COSE effective and indicates important research directions for context-oriented software development.

This paper presents preliminary studies on COSE. Although these studies reveal that our approach is promising, there are also a number of open issues. In this section, we show our future research roadmap.

7.1 Systematizing Context Identification

The applications mentioned in the case studies are simple, and the number of identified contexts is not large. In large systems, the number of "candidates for contexts" will be very large. Furthermore, the system-to-be will be described by using natural languages including diagrams in inconsistent syntax. In some cases, such descriptions will be scattered over various documents, spreadsheets, and emails. This unstructured piling up of descriptions easily results in a situation where conceptually the same contexts are described in different words and notations.

In Section 3, we list the factors changing the system behavior as candidates for contexts. This is the most fundamental property of contexts. To systematize identification of contexts and deal with a large number of candidates for contexts, more precise criteria to find candidates for contexts will be necessary. For example, for a factor changing the system behavior to be identified as a context in COP, it should affect the behavior of a number of objects in the system. Moreover, all the contexts in the case studies are external with respect to the affected entities.

From this perspective, we are planning to develop a systematic context elicitation process that is applicable in the early stages of a requirements elicitation process.

7.2 Requirements based on Other Methods

Using use cases is a fantastic way to figure out functional requirements of the system-to-be. Use cases do not require any special languages to describe them; thus, people from various backgrounds can easily understand them. Nevertheless, they effectively describe the system behavior. Furthermore, they prevent hasty design; design methods based on use cases are well studied.

However, use cases are not all around. They are not suitable for figuring out non-functional requirements or for describing requirements specifications of platforms such as operating systems and frameworks. There are also a number of methods for analyzing requirements that are not based on use cases. It is natural to raise the question whether it is possible to apply methods similar to that described in this paper to other requirements analysis methods.

Goal-oriented methods for requirements engineering [15, 33] are complementary approaches suitable for eliciting requirements variability and constraints. Non-functional requirements are derived from their *soft goals*. Their variability and constraints may depend on executing contexts. Although a goal-based approach for contextualization is proposed in [3], further research should be conducted to explore, for example, approaches to align goal-based approaches and use-case-driven approaches.

Feature modeling presents a compact representation of all products of a software product line (SPL). Feature models are represented by means of feature diagrams [27]. Features provide requirements for architectures (including non-functional ones) and reusable functions. At the programming language level, layers in COP resemble features in FOP [5, 42]. This similarity indicates that we may develop a context-oriented extension of FOSD [4].

Application of context-oriented software development described in this paper to these major requirements engineering methods will be our new challenge.

7.3 Evaluation

To ensure that our methodology is effective, it will be necessary to perform further evaluation. For example, we need to evaluate the costs and benefits of our methodology, and the validity of the decision to use layers to implement context-dependent behavior instead of other mechanisms, through control experiments that compare our methodology with other software development methods. It is difficult to conduct control experiments, and it will take a long time to derive quantitative evaluations. Meanwhile, we think that it is also important to conduct a number of demonstration experiments and collect experiences of the application of our approach. In particular, we believe that application of our methodology to industrial software development is notably important.

Since one purpose of our study is to enhance modularity, the evaluation will be performed from the viewpoint of modularity. For example, an experimental study of how our approach makes it easy to deal with volatile requirements regarding contexts, and analysis of effects of requirement changes should be performed.

7.4 Implicit Activation

In both case studies in this paper, contexts are implemented by means of context conditions. As mentioned above, this fact implies the importance of implicit layer activation. However, there is a performance problem in implicit layer activation. A naive implementation strategy is to evaluate the condition that specifies when the corresponding context is active at every call of the layered methods, and when that condition holds and the corresponding context is not active, then that context is activated. This strategy will not produce a serious problem if the number of layered method calls is not so high. However, in the case where calls of layered methods frequently and repeatedly occur (e.g., in the case where calls of layered methods are included within a loop statement), this strategy may result in a serious performance problem.

Thus, to develop an optimization mechanism for implicit layer activation so that the evaluation of the context condition occurs only when necessary is an important research topic. There are several approaches for this purpose.

One approach is to develop an ad hoc method that optimizes parts of the program where calls of layered methods may frequently occur, such as loop statements. For example, if we can determine that the context condition will never change during the execution of the loop, we may rewrite the loop so that the context condition is evaluated just once at the entrance of that loop.

For a more effective approach, we may research a method to statically analyze when the value of the context condition changes. For example, assuming that c is a condition for the context `C`, if we can derive a pair of predicates (p, q) for which it can easily be checked that $p \implies c$ and $q \implies \neg c$, we can insert evaluations of c where the values for p or q change. We are currently considering an application of predicate abstraction for model checking for this purpose.

In both cases, the optimization requires whole program analysis because the change of context condition may occur anywhere in the program execution. To make the whole program analysis lightweight and feasible in the case when the whole code is not available for analysis, it is also necessary to study the application of whole program analysis without the whole program [2] to COP programs.

The emphasis on implicit activation does not mean that event-based activation of contexts is not necessary. First, in the case where layered methods are frequently called and optimization of implicit layer activation is difficult for some reason, event-based activation should be used. There are also some cases where the specification of context is defined in terms of events (even though this did not happen in our case studies). For example, there may be a specification of stateless objects whose contexts are changed by clicking buttons. In this case, it is better to implement context activation in an event-based manner than to introduce a state for each object to manage context activation using the implicit activation mechanism. There are also some cases where context changes can be observed from both conditions and events.

The problem is that there are no clear guidelines about when to use implicit activation and when to use the event-based mechanism. To create such guidelines, we need to study this problem both from the programming language perspective and the programming practice one. From the programming language perspective, as mentioned above, it is necessary to figure out the feasibility of efficient implementation of implicit activation. Meanwhile, formalization of implicit activation is also desirable to precisely study the semantics of implicit activation. We think that implicit activation is a special case of functional reactive programming (FRP) [16] in that the change of condition (value) reactively changes the result of activation (computation). Understanding implicit activation in terms of FRP may further clarify the semantics of implicit activation.

From the programming practice perspective, through a number of other case studies, we are planning to discover common *patterns* in context activation, which will serve as guidelines.

7.5 Distributed, Multi-Language Environment

Both case studies in this paper are standalone applications written in one single programming language. In real products, however, systems are implemented by using multiple programming languages and sometimes comprise a number of components and services over networks. To apply our methodology to such systems, there are two problems.

First, to our knowledge, ServalCJ is the only language that has all the desirable properties shown in Section 6.3. We need to explore how to realize the mechanism supported by ServalCJ in a wide range of programming languages including those suitable for high performance computing such as C and C++ and scripting languages such as JavaScript.

Second, little research effort has been devoted in COP for sharing the same context among multiple application processes. Sharing context among processes over the network is possible in programming languages supporting network-transparent communications between processes such as ContextErlang [39]. Further research is necessary to support network-transparent context in other programming models, and develop a mechanism to share contexts among multiple programming languages, which possibly communicate with each other over the network.

On the basis of these technical elements, we will further study the applicability of COSE to more realistic and sophisticated software development situations.

References

[1] Gregory D. Abowd, Christopher G. Atkeson, Jason Hong, Sue Long, Rob Kooper, and Mike Pinkerton. Cyberguide: A mobile context-aware tour guide. *Wireless Networks*, 3(5):421–433, 1997.

[2] Karim Ali and Ondřej Lhoták. Whole-program analysis without the whole program. In *ECOOP'13*, volume 7920 of *LNCS*, pages 378–400, 2013.

[3] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. Goal-based self-contextualization. In *CAiSE 2009*, pages 37–43, 2009.

[4] Sven Apel and Christian Kästner. On overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[5] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE'05*, pages 125–140, 2005.

[6] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *COP'09*, pages 1–6, 2009.

[7] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 28(1):272–292, 2011.

[8] Malte Appeltauer, Robert Hirschfeld, and Hidehiko Masuhara. Improving the development of context-dependent Java application with ContextJ. In *COP'09*, 2009.

[9] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Proceedings of the International Conference on Software Composition 2010 (SC'10)*, volume 6144 of *LNCS*, pages 50–65, 2010.

[10] Ivia Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 135–173, 2006.

[11] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA 1990*, pages 303–311, 1990.

[12] Cinzia Cappiello, Marco Comuzzi, Enrico Mussi, and Barbara Pernici. Context-management for adaptive information systems. *Electronic Notes in Theoretical Computer Science*, 146:69–84, 2006.

[13] Stefano Ceri, Florian Daniel, Federico M. Facca, and Maristella Matera. Model-driven engineering of active contet-awareness. *World Wide Web*, 10:387–413, 2007.

[14] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *Dynamic Language Symposium (DLS) '05*, pages 1–10, 2005.

[15] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[16] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP'97*, pages 263–273, 1997.

[17] Carlo Ghezzi, Matteo Praella, and Guido Salvaneschi. Programming language support to context-aware adaptation–a case-study with Erlang. In *SEAMS'10*, pages 59–68, 2010.

[18] Karen Henrichsen and Jadwiga Indulska. A software engineering framework for context-aware pervasive computing. In *PERCOM'04*, 2004.

[19] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *GTTSE 2007*, volume 5235 of *LNCS*, pages 396–407, 2008.

[20] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[21] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Pearson Education, 1992.

[22] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Pearson Education, 2005.

[23] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *AOSD '11*, pages 253–264, 2011.

[24] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Introducing composite layers in EventCJ. *IPSJ Transactions on Programming*, 6(1):1–8, 2013.

[25] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Mapping context-dependent requirements to event-based context-oriented programs for modularity. In *Workshop on Reactivity, Events and Modularity (REM 2013)*, 2013.

[26] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. A unified context activation mechanism. In *COP'13*, 2013.

[27] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[28] Roger Keays and Andry Rakotonirainy. Context-oriented programming. In *MobiDE'03*, pages 9–16, 2003.

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Grisword. An overview of AspectJ. In *ECOOP'01*, pages 327–353, 2001.

[30] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software*, 2004.

[31] Alexiei Lapouchnian and John Mylopoulous. Modeling domain variability in requirements engineering with contexts. In *ER 2009*, volume 5829 of *LNCS*, pages 115–130, 2009.

[32] Sotirious Liaskos, Alexei Lapouchnian, Yijun Yu, Eric Yu, and John Mylopoulos. On goal-based variability acquisition and analysis. In *RE'06*, pages 79–88, 2006.

[33] Lin Liu and Eric Yu. Designing information systems in social context: a goal and scenario modelling approach. *Information Systems*, 29(2):187–203, 2004.

[34] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97*, volume 1241 of *LNCS*, pages 419–443, 1997.

[35] Awais Rashid, Peter Sawyer, Ana Moreira, and João Araújo. Early aspects: a model for aspect-oriented requirements engineering. In *RE'02*, pages 199–202, 2002.

[36] Daniel Saliber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *CHI'99*, pages 434–441, 1999.

[37] Mohammed Salifu, Bashar Nuseibeh, Lucia Rapanotti, and Thein Than Tun. Using problem descriptions to represent variability for context-aware applications. In *VaMoS 2007*, 2007.

[38] Mohammed Salifu, Yujun Yu, and Bashar Nuseibeh. Specifying monitoring and switching problems in context. In *RE'07*, pages 211–220, 2007.

[39] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: Introducing context-oriented programming in the actor model. In *AOSD'12*, 2012.

[40] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 248–274, 2003.

[41] Alistair Sutcliffe, Stephen Fickas, and McKay Moore Sohlberg. PC-RE: a method for personal and contextual requirements engineering with some experience. *Requirements Engineering*, 11(3):157–173, 2006.

[42] Fuminobu Takeyama and Shigeru Chiba. Implementing feature interactions with generic feature modules. In *Software Composition 2013*, volume 8088 of *LNCS*, pages 81–96, 2013.