Using Mixin Layers for Context-Aware and Self-Adaptable Systems

Brecht Desmet, Jorge Vallejos Vargas, Stijn Mostinckx and Pascal Costanza
Programming Technology Lab – Vrije Universiteit Brussel
Pleinlaan 2 – 1050 Brussels, Belgium
{*bdesmet,jvallejo,smostinc,pascal.costanza*}@*vub.ac.be*

May 29, 2006

Abstract

The use of context information in networks of mobile devices is crucial to respond adequately to the user's expectations. We therefore require that applications dynamically adapt their behaviour according to context changes. Current-day technology typically consists of a series of programming patterns to achieve dynamic behaviour adaptation. However, combining different contexts in such systems has proven to be far from trivial. We propose the use of mixin layers to modularize the context-dependent adaptations separate from the application core logic, together with a composition mechanism that deals with runtime context interactions. Since the classes in mixin layers have no fixed superclasses, they can be combined easily to reflect different combinations of context.

1 Introduction

The Ambient Intelligence vision describes scenarios in which people are pervasively surrounded by interconnected embedded and mobile devices. This pervasiveness introduces new opportunities to make software systems sentient and aware of the context in which they exist. Such context-aware software systems automatically adapt their behaviour according to context changes. They are therefore called context-aware and self-adaptable systems.

The development of such systems exhibits the following main problems. First, low-level sensor data needs to be transformed into *meaningful context information*. Next, context changes frequently imply a *change in behaviour*. Finally, a context description is often composed from different types of information like location, time, temperature etc. This diversity of information may produce a *combinatorial explosion of possible behavioural variants* a system can exhibit.

In this paper, we focus on context-aware systems that exhibit the following specific characteristics. First, we regard the changes in behaviour as variations of the application core logic. Throughout this paper, we will refer to this as *context-dependent adaptations*. This position paper argues for the use of mixin layers to implement these context-dependent adaptations in order to keep the design of context-aware systems manageable. Second, relevant changes in context result in an adaptation of existing behaviour by applying mixin layers to a running system. Finally, different combinations of mixin layers in the inheritance hierarchy reflect the different combinations of context. Since there can exist semantic constraints between context-dependent adaptations, we additionally require a mechanism that constructs valid compositions of mixin layers at runtime.

Throughout this paper, we assume the existence of a mechanism that generates meaningful context information. This mechanism reifies context changes that are relevant to the system as events which announce behavioural adaptations. Such functionality is exhibited by a variety of contemporary tools like ContextToolkit [SDA99] and is outside the scope of this paper. In contrast, we focus on strategies to incorporate context-dependent adaptations of software behaviour.

2 Motivating example

We present the software of a simplified cellular phone as an illustration of a context-aware and self-adaptable system. The phone example consists of the following functionalities. First, the phone harbours a list of contacts, some of them may be marked as VIPs. This information is encapsulated in the `Contacts` class. Second, the `Messages` class provides facilities to read and send messages. Third, the `Journal` class keeps track of all phone and messages traffic. Finally, the main task of the phone is to ring whenever somebody calls and to provide the means to answer calls. This functionality is offered by the `PhoneCall` class. These different functionalities constitute the *application core logic* of the cellular phone.

The behaviour of the application core logic can be adapted at runtime according to context changes. We introduce three *context-dependent adaptations* that each contain two parts: a *context condition* that explains when the adaptation is applicable and the *actual behaviour* of the adaptation with regards to the application core logic.

IgnoreAdaptation If the battery level is low, ignore and log all phone calls except for contacts that are classified as VIPs.

AnswerMachineAdaptation If the time is between 11pm and 8am, activate the answering machine.

RedirectAdaptation If the user is in the meeting room, redirect all calls and messages to the secretary.

Although the three context conditions (battery low, time between 11pm-8am and meeting room location) can all be true at the same time, the behaviour of the adaptations cannot be freely combined. This is because adaptations might contradict each other, like e.g. *IgnoreAdaptation* and *RedirectAdaptation*. In case of a contradiction, the user can make an arbitrary decision about what should happen. For instance, in our phone example, the following set of rules describes the valid combinations of adaptations and how contradictions should be resolved.

Rule I All adaptations can exist individually.

Rule II *IgnoreAdaptation* and *AnswerMachineAdaptation* can coexist. Only VIP contacts will get in touch with the answering machine, all other contacts will be ignored.

Rule III *IgnoreAdaptation* and *RedirectAdaptation* cannot coexist. *RedirectAdaptation* has priority.

Rule IV *AnswerMachineAdaptation* and *RedirectAdaptation* cannot coexist. *AnswerMachineAdaptation* has priority.

3 Mixin layers

The notion of mixin layers [SB98] was introduced by Smaragdakis et al. as an implementation technique to support refinement of collaboration-based designs. A mixin layer is a modularisation unit that encapsulates different mixin classes each refining a single class of the collaboration. Such mixin classes (or just mixins) are also commonly known as abstract subclasses. The distinguishing feature between ordinary and abstract subclasses is that the latter have parameterized superclasses. This property enables the instantiation of mixins with various superclasses and thus supports reusability.

In practice, refinement by using mixin layers is achieved through the ability to add or specialize methods and classes. Moreover, mixin layers can also refine other mixin layers because they can be composed in an inheritance hierarchy, yielding a layered design. Since mixin layers are both cross-cutting (layers can affect multiple classes) and hierarchical (layers refine existing behaviour instead of invasively modifying it), we can represent them graphically as a grid structure with the layers positioned horizontally and the affected classes vertically. This is illustrated in Figure 1.

Amongst others things, mixin layers are extensively used to encapsulate the various functionalities in a software product line. In this domain, called feature-oriented programming, techniques to isolate features from the application core logic are of considerable importance to allow one to compose features when developing a product variant. Since feature compositions are not influenced by

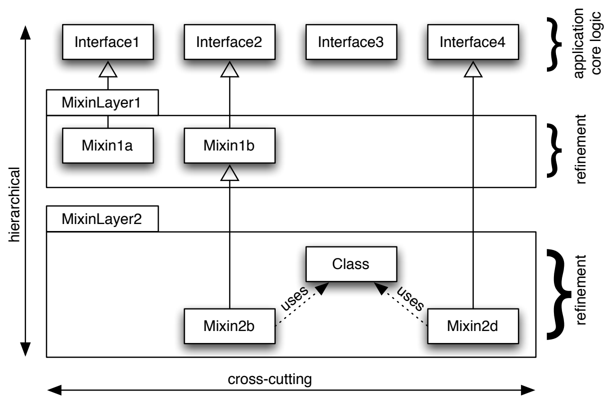

Figure 1: Mixin layers.

runtime factors such as context events, they can be validated at design time using, for example, the Feature-Oriented Domain Analysis (FODA) model [KSJ+90]. The latter represents a hierarchical decomposition of all features and the relationships between them.

It is our belief that the use of mixin layers can offer important contributions to the development of context-aware and self-adaptable software. The basic idea is to separate the context-dependent adaptations from the application core logic and modularize them using mixin layers. The combination of adaptations can be realized by ordering the mixin layers correctly in the inheritance hierarchy.

We apply this idea to our phone example from Section 2 by putting the behavioural part of the *IgnoreAdaptation*, *AnswerMachineAdaptation* and *RedirectAdaptation* in the `IgnoreLayer`, `AnswerMachineLayer` and `RedirectLayer` respectively. Figure 2 illustrates how the design looks like if the battery level is low and no other context condition is satisfied. The `IgnoreMixin` uses the `Contacts` to filter phone calls that are not from VIPs. Furthermore, `RegisterIgnore` registers all calls that are ignored in the `Journal`.

We have several reasons to believe that our approach leads to more manageable software designs. First, context-dependent adaptations can be cross-cutting (e.g. *IgnoreAdaptation* affects different parts of an application) and hierarchical (e.g. rule II combines *IgnoreAdaptation* and *AnswerMachineAdaptation*). The notion of mixin layers is a suitable candidate to modularize these context-

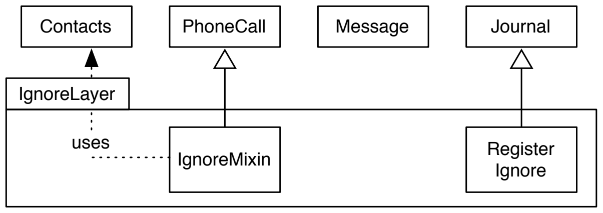

Figure 2: System design if battery level is low.

dependent adaptations because they cover both dimensions. Next, the use of mixin layers to implement the context-dependent adaptations separate from the application core logic naturally enforces separation of concerns. The fact that mixin layers can be instantiated with various superclasses reduces the coupling between the mixin layers and the application core logic. This subsequently increases the reusability of mixin layers within the same application. Finally, because of the clean separation of concerns and low coupling, it becomes more convenient for an application to evolve over time. For instance, extending an application with a new context-dependent adaptation consists of adding a new mixin layer. Furthermore, the extensions can be realized without affecting the application core logic. In order to make this approach scalable, we need more support to model the runtime relationships between the context-dependent adaptations. This issue is discussed in Section 4.2.

4 Dynamic mixin layers

This section describes the consequences of implementing the behavioural part of context-dependent adaptations using mixin layers. As is illustrated in the previous section, mixin layers hold some promise since they allow modularisation of a single adaptation, which my cross-cut the application, in a single abstraction. Unfortunately, mixin layers are applied once prior to the construction of the software product and they have no identity at runtime. To allow context-aware software to be written using mixin layers it is required that they can be pluggable at runtime. This pluggability should be supported by a mechanism that constructs compositions of mixin layers that adhere to the semantic constraints between them. In addition, the program state can further constrain the composition mechanism. All these issues are discussed in the following sections.

4.1 Dynamic activation

The behavioural adaptations are accomplished by activating and deactivating mixin layers at runtime according to context changes. This pluggability can be achieved by redefining classes at runtime. Existing instances of redefined classes should be updated accordingly. Furthermore, the activation of several mixins that are part of the same layer should be an atomic operation. On the one hand, this might look like a harsh requirement to implement in a static language like Java. On the other hand, by using the reflective capabilities of dynamic languages such as CLOS or Smalltalk, it is much more straightforward to perform class redefinitions at runtime.

In other words, behavioural adaptations in our approach take place at the meta level in the sense that we redefine classes and update existing instances accordingly. This contrasts existing aproaches like in Context-Toolkit [SDA99] and WildCAT [DL05] that basically employ event-handler systems. These systems apply event-handlers in response to context changes and are therefore completely situated at the base level.

4.2 Dynamic composition mechanism

Next to the dynamic activation of mixin layers, we also require a composition mechanism that is able to dynamically reconfigure mixin layer compositions according to context changes. It is important to mention that our notion of a dynamic composition mechanism contrasts the static approaches of feature-oriented programming (FOP) at two levels. First, the selection of mixin layers that are part of the composition are computed automatically based on context information. In FOP, this selection is done manually at design time. Second, the composition of mixin layers evolves over time as the context changes. On the contrary, the compositions in FOP are not supposed to change at runtime and are therefore fixed at design time using program synthesis techniques. Hence, we conclude that there exists a huge gap between existing static composition mechanisms where the composition does not change at runtime and the kind of dynamic composition

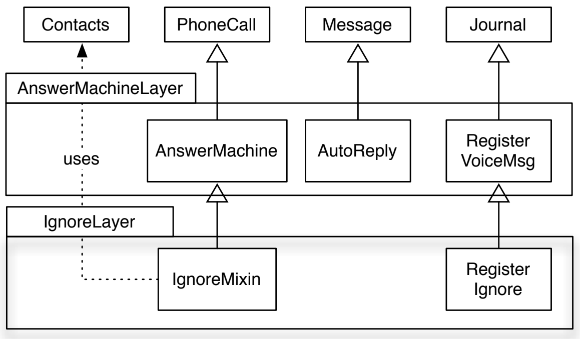

Figure 3: System design if both battery level is low and time is between 11pm and 8am.

mechanism that we require to reconfigure compositions at runtime according to context changes.

We illustrate the idea with the phone example from Section 2. Consider that the user's phone enters a low battery level. This means that the `IgnoreLayer` is to be applied to the application core logic as presented in Figure 2. The functionality provided by this layer is to ignore and log all calls from contacts which are not classified as VIP to conserve battery power. Now suppose the clock strikes 11pm, triggering a context change to activate the `AnswerMachineLayer`. It is the responsibility of the dynamic composition mechanism to fulfill this request by adding the `AnswerMachineLayer` to the current composition of mixin layers at runtime. The result of this context change is shown in Figure 3.

It is common that there exist relationships between mixin layers that must be taken into account in order to construct valid compositions. The relationships between the mixin layers of our phone example are conceptually explained in Section 2 by means of four rules. For instance, rule I and II stipulate that `AnswerMachineLayer` and `IgnoreLayer` can exist either individually or together. If both mixin layers coexist, we actually have a new behavioural variant: Only VIPs get in touch with the answering machine. This variation is the result of organizing the two mixin layers correctly in the inheritance hierarchy. In this case, we require an ordering constraint between `AnswerMachineLayer` and `IgnoreLayer` to end up with the desired behaviour. We conclude that the composition mechanism should be aware of all relationship information between mixin layers in order to construct valid compositions at runtime.

We can reuse some concepts of the FODA model to a certain degree to express how mixin layers can be composed in a dynamic environment. Unfortunately, FODA lacks the expressiveness to model the runtime constraints between mixin layers. For instance, the phone example requires the addition of temporal constraints like the "comes before" and "gives priority to" relationships in order to capture the semantics of rules I-IV. The result is presented in Figure 4.

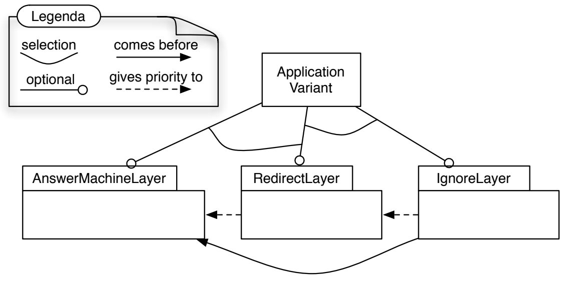

Figure 4: Feature diagram of cellular phone example.

The following relationship information can be derived from the feature diagram. The `AnswerMachineLayer` and `RedirectLayer` cannot coexist. If both mixin layers are applicable according to the associated context conditions, that is if the user is in the meeting room and the time is between 11pm and 8am, the priority goes to `AnswerMachineLayer`. An equivalent relationship holds for `RedirectLayer` and `IgnoreLayer`. In contrast, `AnswerMachineLayer` and `IgnoreLayer` can coexist if they appear in the correct order.

It becomes clear that the dynamic composition mechanism is actually a kind of *logic reasoning system* that derives valid compositions of mixin layers at runtime. The model that describes all valid combinations of mixin layers (like Figure 4) determines the decisions of the reasoner. Every time a change in context arises, the dynamic composition mechanism computes a new composition based on the current composition and a request to activate or deactivate some mixin layers. In other words, the responsibility of the dynamic composition mechanism is to maintain the semantic constraints between the context-dependent adaptations.

4.3 Consistency

The information encoding the relationships between mixin layers is not necessarily sufficient to compute valid compositions. The program state can sometimes constrain the activation or deactivation of certain mixin layers. For instance, let us consider a system that transforms an internal representation to both HTML and PDF. We implement this behaviour using two mixin layers, one for each output format. Furthermore, both layers refine the methods `begin()`, `transform()` and `end()` that logically depend on each other. There is a potential consistency problem if the layers that contain the formatting behaviour are switched between the `begin()` and `transform()` methods calls or between the `transform()` and `end()` method calls.

A possible solution is to introduce a locking system with regard to mixin layers. The concrete idea is as follows. Once the `begin()` method is called, the mixin layer that contains the transformation behaviour should be locked. This means that it cannot be deactivated by the composition mechanism. As soon as the `end()` method has finalized the transformation task, the mixin layer may be unlocked. From that moment on, the composition mechanism is allowed to deactivate the mixin layer. This mechanism ensures consistent program behaviour by introducing runtime constraints in the composition mechanism.

5 Position statement

We advocate to implement context-aware and self-adaptable systems using a layered design approach because this leads to more manageable software designs. The basic idea is to separate context-dependent adaptations from the application core logic and put them into separate modularisation units. This paper gives several reasons why the mixin layers approach is a good candidate to implement the context-dependent adaptations.

Next, we propose to describe relationships between mixin layers in a declarative way. This enables a dynamic composition mechanism to construct and apply valid compositions of mixin layers according to context changes. The combination of using mixin layers and a declarative language to describe relationships between mixin layers is a powerful mechanism to deal with the continuously varying behaviour of context-aware systems.

Finally, it is important that the dynamic composition mechanism can be constrained at runtime. A concrete example is the possibility to lock activated mixin layers for a certain time to enforce consistency.

References

[DL05] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[KSJ$^+$90] K. Kang, S.Cohen, J.Hess, W.Novak, and S.Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, U.S.A., nov 1990.

[SB98] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, New York, NY, USA, 1999. ACM Press.