Combining AOSD and CBSD in PacoSuite through Invasive Composition Adapters and JAsCo

Wim Vanderperren[1], Davy Suvée and Viviane Jonckers

Vrije Universiteit Brussel, Pleinlaan 2,
1050 Brussel, Belgium
{wvdperre, dsuvee}@vub.ac.be , viviane@info.vub.ac.be
http://ssel.vub.ac.be

Abstract. In this paper, we build on previous work that combines ideas from visual component-based software development and aspect-oriented software development. In our visual component-based methodology PacoSuite, developed in earlier work, we introduced composition adapters to modularize crosscutting concerns. A composition adapter is visually applied onto a composition pattern and the changes it describes are automatically inserted in that pattern using finite automata theory. The expressive power of a composition adapter is however limited to concerns that alter the exterior behavior of a component, as only the glue code of a component-based application is altered. In this paper, we propose invasive composition adapters, able to express concerns that require more than mere filtering and re-routing. The changes dictated by an invasive composition adapter are automatically inserted into the concerned components and composition patterns. To implement an invasive composition adapter, we employ JAsCo, an aspect-oriented language that is tailored for the component-based context. The approach is validated using a small case-study that successfully modularizes three crosscutting concerns as composition adapters in an e-commerce application.

1 Introduction

Aspect-Oriented Software Development (AOSD) argues that some concerns exist that can not be confined to one single module. Typical examples of such crosscutting concerns are logging and access control. The research to deal with this problem is under constant evolution and in most cases targeted at Object-Oriented Software Development (OOSD). Component-Based Software Development (CBSD) requires looser coupling between components and increased independence of a single component. In addition, components can be flexibly inserted in and removed from a composition, often even at run-time. Current mainstream aspect-oriented approaches, of which AspectJ[1] is undoubtedly the most well-known, do not allow to describe

[1] Supported by a doctoral scholarship form the Fund for Scientific Research (FWO or in Flemish: “Fonds voor Wetenschappelijk Onderzoek”).

aspects which are sufficiently independent and reusable for component-based software development. In addition, removing aspects is often quite difficult or sometimes even impossible. As a consequence, these approaches are not very well suited to be reused in a component-based context. Research that aims at recuperating aspect-orientation in Component-Based Software Development, like for instance Aspectual Components [9] or Caesar [10], mainly focuses on the implementation level by introducing new programming languages. Our approach however strives at introducing aspect-oriented ideas in Component-Based Software Development from design till implementation.

This research is based on the PacoSuite[13,14] component-based methodology that aims at lifting the abstraction level for component composition. PacoSuite improves on standard visual composition tools as it allows components to be wired together based on generic interaction protocols, called "composition patterns", rather than simple event/method pairs. To recuperate aspect-oriented ideas into PacoSuite, a "composition adapter" [13] has been introduced. A composition adapter transforms the original composition patterns to introduce the specified aspects. Technically, a composition adapter is applied by introducing the aspects in the glue-code of a component-based application. As a result, it is impossible to introduce aspects in the components themselves. Several experiments however revealed that it should be possible to adapt the components' interior to express aspects that require more than mere filtering or rerouting.

Therefore, we propose in this paper to implement a composition adapter in an aspect-oriented language. The JAsCo [12] language is a novel AOP language specially tailored for component-based software development and the Java Beans component model in particular. As a consequence, JAsCo is very well suited to serve as an implementation language of an "invasive" composition adapter.

This paper presents a complete overview of our approach. As a result, technical details of algorithms and formal foundations are not discussed. Section 2 introduces the context of this research, and more specifically the PacoSuite component-based methodology and the composition adapter model. Section 3 discusses why "invasive" composition adapters are required and how these are implemented using the JAsCo aspect-oriented component language. Section 4 briefly presents the tools we implemented to support our methodology. Section 5 presents a small case study of an e-commerce application, which is used to validate our approach. Finally, we present some related work and state our conclusions.

2 Research context

2.1 Component-based software development in PacoSuite

The PacoSuite research has been going on for a couple of years at our lab and resulted in the PhD of Bart Wydaeghe [14]. In this paper we do not go into the details of the PacoSuite component-based approach, nor do we discuss how this approach relates to

other approaches. Instead, the approach is shortly sketched because it is used as a basis for the remainder of this paper.

The PacoSuite component-based research mainly focuses on lifting the abstraction level for component-based development. The goal is to enable the plug and play idea of component-based development. Therefore, components are documented with usage scenarios that specify how to employ them. A usage scenario is expressed by a special kind of Message Sequence Chart (MSC) [8]. The main difference with a regular MSC is that the signals are taken from a limited set of predefined semantic primitives. Each of these signals is mapped on the concrete API that performs them. As a result, the documentation of a component is both abstract and concrete at the same time.

Figure 1 illustrates a usage scenario of the well-known *Juggler* bean. One participant of a usage scenario represents the component itself and the other participants represent the environment the component expects. In this case, only one environment participant is specified, namely the *Toggler* participant. This usage scenario documents that the *Juggler* component expects consecutive start and stops. The START primitive is implemented by *startJuggling* and the STOP primitive is implemented by *stopJuggling*.

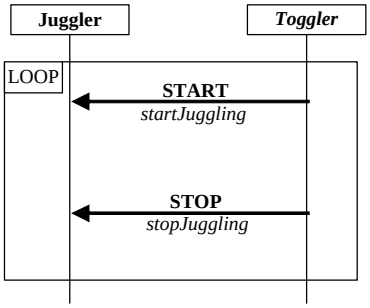


Figure 1: Usage scenario of a Juggler component.

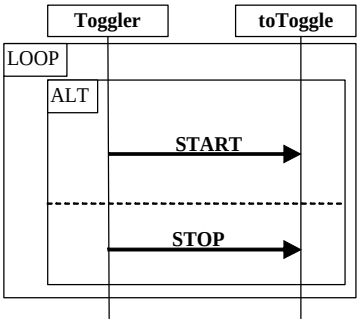


Figure 2: Toggling composition pattern.

Explicit and reusable composition patterns are introduced as higher-level connectors. A composition pattern is an abstract specification of the interaction between a number of roles and is also expressed using an MSC. The signals between the roles originate from the same limited set of semantic primitives. This allows comparing the signals in a usage scenario of a component with these in a composition pattern. Figure 2 illustrates a generic toggling composition pattern. This composition pattern specifies that the *Toggler* participant consecutively sends either a START or a STOP to the *toToggle* participant. A possible application of this composition pattern is a simple visual interface that allows toggling the *Juggler* component from a single *JButton* component. To build this application, the *Juggler* component is mapped on the *toToggle* role and the *JButton* component is mapped on the *Toggler* role. Notice that even this simple collaboration can not be wired by most visual composition environments because the collaboration itself requires state. The *JButton* always fires the same *actionPerformed* event and depending on the state of the interaction *startJuggling* or *stopJuggling* has to be called onto the *Juggler* component.

The documentation of components and composition patterns allows checking the compatibility of a component with a role. The glue-code that constraints the behavior

of the components and that translates syntactical compatibilities is generated automatically. Both algorithms are based on finite automata theory.

2.2 Composition Adapters

Some concerns can not be cleanly modularized using composition patterns and components as they are spread among different entities. As a result, editing, adding and removing such a concern becomes a cumbersome and error-prone task. To solve this problem, a new concept has been proposed, called a composition adapter. The next paragraphs present this solution using the run-time checking of timing constraints as a concrete example. For checking timing constraints dynamically using our current concepts, every composition pattern needs to be adapted in the same way to include an extra role that intercepts all communication between the other roles. Of course, when the application goes into the production phase, the dynamic timing aspect needs to be removed from the application. Consequently, the involved composition patterns need to be altered again.

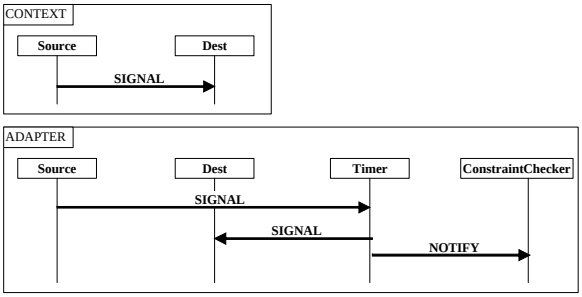


Figure 3: Dynamic timing verification composition adapter.

In order to modularize this timestamping aspect, a composition adapter is used. A composition adapter is able to describe adaptations of the external behavior of a component, independently of a specific API. A composition adapter is again documented using a special kind of MSC and consists of two parts: a context part and an adapter part. Figure 3 depicts the composition adapter that is used to modularize the timing aspect. The context part of a composition adapter describes the behavior that needs to be adapted. This can be a simple signal send as in Figure 3, but can very well be a full protocol. The adapter part specifies the adaptation itself. In the case of the dynamic timing composition adapter, every signal between the *Source* and *Dest* role will be rerouted through a *Timer* role. The *Timer* role is responsible for taking a timestamp and notifying the *ConstraintChecker* role. The *ConstraintChecker* role is responsible to verify whether every signal it is notified of, does not violate a timing constraint. The component that is mapped on the *ConstraintChecker* role could do the verification process offline and/or run on a different CPU to minimize the disruption of the system.

When a composition adapter is applied onto an existing instantiated composition pattern, the context roles of the composition adapter need to be mapped onto roles of the composition pattern. For example, suppose that the communication between the *Tog-*

gler and *toToggle* roles of the composition pattern in Figure 2 needs to be timed. The *Source* role of the timing composition adapter of Figure 3 has to be mapped onto the *Toggler* role of the composition pattern. Likewise, the *Dest* role has to be mapped onto the *toToggle* role. As a result, the START and STOP signals are not send directly to the *toToggle/Dest* role but are re-routed through the *Timer* role. After sending the START or STOP signal to the *toToggle/Dest* role, the *ConstraintChecker* role is notified.
To automatically apply a composition adapter onto a given composition we developed an algorithm based on finite automata theory. In this paper, we do not discuss this algorithm. A full explanation can be found in [13].

3 Invasive Composition Adapters

The critical reader might have noticed that the composition adapter approach to enable run-time checking of timing constraints is not very accurate. Currently, the timestamp of the event is taken when it arrives at the component mapped on the *Timer* role. As a result, there is at least an inaccuracy because of the delay of this message send. If the application works distributed, this delay can not be neglected. Certain sophisticated component systems use a scheduler to pass messages to components. This scheduling process imposes yet another delay, making the timestamp even less accurate. As a result, our composition adapter approach to check timing constraints at run-time is not very well suited if a high precision is desired. The only way to achieve a correct timestamp is to alter the mapped components themselves so that the timestamp is taken before a message is sent or received. A composition adapter however, is only able to alter the exterior behavior of a component by ignoring or rerouting messages. Aspects that require other adaptations can not be described using composition adapters, which is a major limitation.

3.1 JAsCo Implementation

One solution to represent aspects that require interior adaptations of components consists of providing a composition adapter with an implementation in an aspect-oriented programming language. The JAsCo [12] AOP language is especially designed for Component-Based Software Development and the Java Beans component model in particular. JAsCo improves on current aspect-oriented approaches as it allows specifying aspects on interactions specific for the employed component model (e.g. events in Java Beans). In addition, JAsCo allows attaching and detaching aspects at run-time, which is a major advantage in enterprise component-based applications. Therefore, we propose to implement a composition adapter using a JAsCo Aspect Bean. Figure 4 illustrates the implementation of the Aspect Bean that represents the timestamping aspect introduced in the previous section.
The *DynamicTimer* Aspect Bean contains two parts: a *TimeStamp* hook (line 15 till 26) and a set of regular Java class members (line 2 till 13), which are used to implement the notifying behavior of the *timestamping*-aspect. The *TimeStamp*-hook is re-

sponsible for describing in an abstract way **when** the normal execution of a method should be cut (i.e. like a pointcut in AspectJ [1]) and **what** extra behavior there should be executed at that precise moment in time. The constructor of the *TimeStamp*-hook (line 17 till 19) specifies that it can be deployed on every method that takes zero or more arguments as input. The constructor-body (line 18) specifies that the behavior of the *TimeStamp*-hook should be performed whenever *method* is executed. The behavior of the *TimeStamp*-hook is described by making use of the behavior methods *before* and *after* (line 20 till 25). The *before*-method specifies that a timestamp should be taken prior to the execution of *method*. The *after*-method specifies the notification of the observers of the *TimeStamp*-hook afterwards. This notification towards the listeners of the *DynamicTimer* aspect is done by firing a *timeStampTaken* event (line 11).
As an Aspect Bean specifies its context in an abstract way to enhance reusability, each aspect needs to be initialized explicitly by means of a connector. An example of a connector that initializes the *TimeStamp*-hook with a concrete context is shown in section 3.3.

```
1  class DynamicTimer {
2    private Vector obs = new Vector();
3    public void removeTimeListener(TimeListener o) {
4      obs.remove(o);
5    }
6    public void addTimeListener(TimeListener o) {
7      obs.add(o);
8    }
9    protected void notifyListeners(Method m, long t) {
10     for (int i = 0;i < obs.size();i++) {
11       ((TimeListener)obs.elementAt(i)).timeStampTaken(m,t);
12     }
13   }
14
15   hook TimeStamp {
16     private long timestamp;
17     TimeStamp(method(..args)) {
18       execute(method);
19     }
20     before() {
21       timestamp=System.currentTimeMillis();
22     }
23     after() {
24       notifyListeners(method,timestamp);
25     }
26   }
27 }
```

When?

What?

Figure 4: The JAsCo-aspect for dynamic timing. The TimeStamp hook contains the crosscutting timestamping behavior.

3.2 Documentation

The *DynamicTimer* Aspect Bean can now be used as an implementation for a composition adapter. When this composition adapter is applied onto a given component composition, the *DynamicTimer* Aspect Bean has to be initialized with the corresponding component communication.

However, the composition adapter model needs to be altered to cope with this implementation. Therefore, we propose an enhanced version of a composition adapter, called an "invasive" composition adapter. Figure 5 illustrates the invasive composition adapter that documents the *DynamicTimer* Aspect Bean of Figure 4. Messages in the context part of an invasive composition adapter can be mapped on a hook. In the case of Figure 5, the SIGNAL message is mapped on the *TimeStamp* hook. As a result, every message between the component that is mapped on the *Source* role and the component that is mapped on the *Dest* role will be given as input to the *TimeStamp* hook constructor. As a consequence, those messages are affected by the logic implemented by the *DynamicTimer* Aspect Bean. As a result, a timestamp is taken *before* the original method is executed and an event is fired *afterwards.*

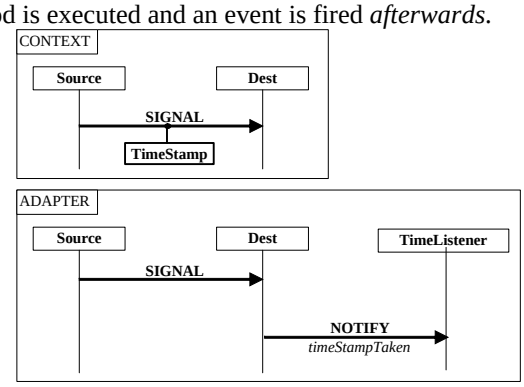


Figure 5: Invasive Composition Adapter model for the DynamicTimer Aspect Bean.

The adapter part documents what the effect of the application of the *DynamicTimer* Aspect Bean will be. In the example of Figure 5, every signal between a certain source and destination component is still sent in the same way. However, the *DynamicTimer* Aspect Bean specifies that a timestamp has to be taken before an adapted method is executed (see Figure 4, line 20 till 22). This behavior is not documented in the composition adapter as it is internal to the Aspect Bean and no communication with other components is involved. As a consequence, this behavior is not relevant for verifying compatibility and to generate glue-code.

The *DynamicTimer* Aspect Bean specifies an after method (see Figure 4, line 23 till 25) that causes a *timeStampTaken* event to be fired (see Figure 4, line 11). This behavior has to be documented in the composition adapter as it requires communication with other components. Therefore, the adapter part specifies that the *Dest* role notifies a *TimeListener* participant after the original method is executed. For instance, a component that checks timing constraints could be mapped on the *TimeListener* participant.

Messages that are sent or received as a result of the changes described by a JAsCo Aspect Bean require an implementation mapping. In Figure 5, the NOTIFY message is implemented by firing the *timeStampTaken* event. The implementation mapping of the NOTIFY to *timeStampTaken* is required to be able to generate glue-code that will call the correct method of the component that is mapped on the *TimeListener* role. Notice that the component that is mapped onto the *TimeListener* role does not have to understand the *timeStampTaken* event. Glue-code that translates the *timeStampTaken*

event into one or more methods of the mapped component can be automatically generated using the documentation of Figure 5.

3.3 Applying an invasive composition adapter

Similar to a regular composition adapter, applying an invasive composition adapter implies to first map the context roles of the invasive composition adapter onto roles of the composition pattern. Imagine that the *DynamicTimer* invasive composition adapter of Figure 5 needs to be applied onto the Toggling composition pattern of Figure 2. To enable this, the *Source* role of the invasive composition adapter has to be mapped onto the *Toggler* role of the composition pattern. Likewise, the *Dest* role has to be mapped onto the *toToggle* role of the composition pattern. As a consequence, the composition pattern is changed to incorporate the extra behavior, namely firing an event after each START and STOP the *toToggle* role receives.
Afterwards, components can be mapped on the enhanced composition pattern. For example, we could again map the *Juggler* component onto the *toToggle* role. However, the *Juggler* component is now incompatible with the enhanced *toToggle* role, because it does not send a NOTIFY after each START and STOP. Due to the application of the JAsCo *DynamicTimer* Aspect Bean, the *Juggler* component will behave in that way. It is just the documentation of the *Juggler* that has become inconsistent. Therefore, the documentation of a component is transformed to cope with the adaptations described by the invasive composition adapter. In this way, the documentation of the *Juggler* component specifies that it sends a NOTIFY after each START and STOP.
It is possible to automatically transform a composition pattern and a component usage scenario as specified by an invasive composition adapter using algorithms based on finite automata theory. The details of these algorithms are however outside the scope of this paper.

```
1  connector TimeConnector {
2                                          Where?
3    DynamicTimer.TimeStamp timer =
4      new DynamicTimer.TimeStamp ( {
5        void Juggler.startJuggling(),
6        void Juggler.stopJuggling() } );
7
8    timer.before();
9    timer.after();
10 }
```

Figure 6: The JAsCo connector for dynamic timing of the JButton and the Juggler.

When both the roles of the composition pattern and invasive composition adapter are filled with components, a connector in the JAsCo language is generated to initialize the abstract Aspect Bean with its correct context. Figure 6 illustrates the connector that would be generated from the example described above. This connector specifies the initialization of the *TimeStamp*-hook with both the *startJuggling* and *stopJuggling* methods of the *Juggler* bean (line 3 till 6). Afterwards, the execution of both the *before* and *after* behavior methods is specified (line 8 till 9). As a consequence, both

the *startJuggling* and *stopJuggling* methods are adapted to take a timestamp before their execution and to fire an event afterwards.

4 Tool Support: PacoSuite

The ideas introduced in this paper are implemented in a visual component composition environment called PacoSuite. PacoSuite consists of two visual tools, called PacoDoc and PacoWire, and the command-line tools required by the JAsCo language. PacoDoc is a visual editor for documenting components, composition patterns and composition adapters. PacoWire is our actual component composition environment that allows visually applying a component onto a role of a composition pattern. The drag and drop action is refused when the component is detected to be incompatible with the composition pattern. Composition adapters can also be visually applied on a given composition of components. The changes dictated by a composition adapter are automatically applied using the algorithms mentioned in this paper. In case of an invasive composition adapter, the JAsCo tools are executed transparently to the user. When all the component roles are filled, the composition is checked as a whole and glue-code is generated automatically. Figure 7 illustrates some screenshots of this tool suite.

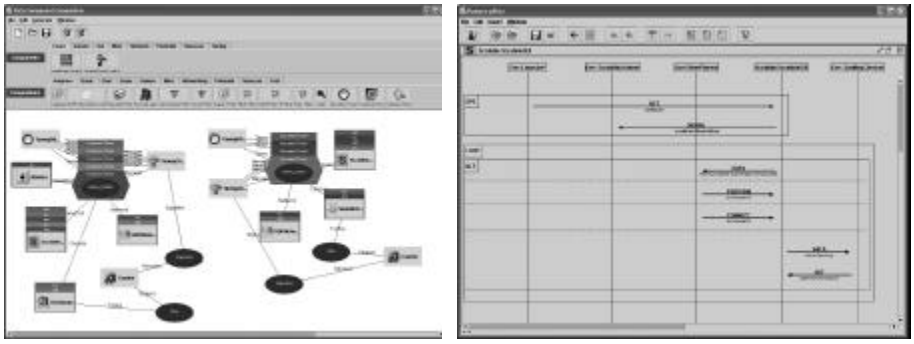


Figure 7: Screenshots of PacoSuite. The left screenshot illustrate the visual component composition environment PacoWire. The rectangles represent components, the ovals stand for composition patterns and the hexagonal shapes symbolize invasive composition adapters. The right screenshot shows the documentation of a Scrabble component in the PacoDoc tool.

5 Wiring the PhotoLab Application in PacoSuite

It can be argued that using an invasive composition adapter for specifying timing constraints validation is not really necessary. Indeed, a regular composition adapter is also able to describe this concern, only the accuracy of the timestamps differs. Therefore, we present a small case-study that introduces crosscutting concerns that really require an invasive composition adapter.

The case study at hand is a distributed e-commerce application for printing digital pictures. The e-commerce application consists of two sub-applications: a client and a server. The client application allows users to submit digital pictures together with the requested number of prints and size to a server. The server calculates the price for this order, bills it onto the credit card of the customer and returns a receipt.
Figure 8 illustrates the composition pattern that specifies the interaction required for the client application. This composition pattern consists of four roles: the *Graphical User Interface* role, the *Network* role, the *Launcher* role and the *Checker* role. The client application communicates with the server application through the *Network* role. The *Checker* role is responsible for validating the data submitted by the user. In the PhotoLab example, this means checking whether the selected file is a valid JPEG picture. The composition pattern specifies the following interaction: First, a *Launcher* role is responsible for starting the application. Afterwards, the user interface sends the login information of the user to the *Network* role. The server then checks the login and password of the user and returns the result. In case of false login information, the application stops. Otherwise, the GUI interface for selecting the pictures to print is started. Afterwards, the user interface requires the *Checker* role to check the validity of the supplied user data. When the user accepts the order, the order is sent over the network and the network returns the receipt. Notice that this composition pattern is not only suited for this digital photo printing application, as it can be reused in other e-commerce client applications as well.

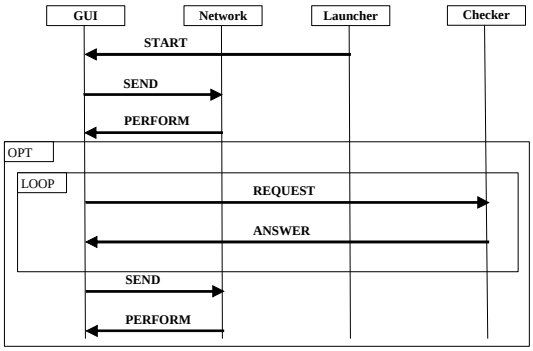


Figure 8: ECommerceClient composition pattern.

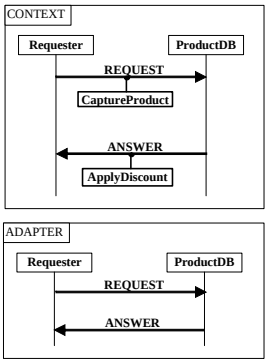


Figure 9: OldProductDiscount invasive composition adapter.

Figure 10 illustrates a screenshot of the *ECommerceClient* composition pattern in the PacoSuite tool with filled-in components. The oval with squares connected to it via lines is the representation of a composition pattern in the PacoSuite tool. Each square represents a role of the composition pattern and the name of the role is specified on both the square and the line that connects it to the composition pattern (see *Launcher* role in Figure 10). Components are also represented by squares containing an icon and the name of the component. In the screenshot of Figure 10, four components are present on the canvas. The *PhotoLabGUI*, *Network* and *PhotoRenderer* components are already mapped onto roles of the *ECommerceClient* composition pattern. The *JButton* component still has to be dragged onto the remaining *Launcher* role.

The interaction required for the server side of the application is specified by another composition pattern with six roles, called *ECommerceServer*. The interaction is somewhat similar to the *ECommerceClient* composition pattern because it has to respond to the messages sent by the client. Therefore, the *ECommerceServer* composition pattern is not further discussed.

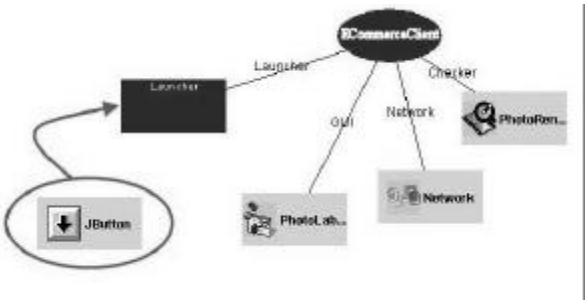


Figure 10: PhotoLabClient application in the PacoSuite environment. The component composer is about to map the JButton component on the Launcher role of the ECommerceClient composition pattern.

The PhotoLab application can be visually wired in the PacoSuite tool without programming a single line of code. However, several concerns can be identified that are not the main functionality of the digital photo printing application and that crosscut the functional decomposition of the system. Without AOP, these concerns are spread and duplicated among different components and composition patterns. The next paragraphs present three crosscutting concerns and capture these concerns successfully as composition adapters. The first two crosscutting concerns are modularized using an invasive composition adapter. The JAsCo implementation of both these invasive composition adapters is however not discussed.

5.1. Obsolete Product Discount aspect

The first concern consists of a business rule that specifies a discount for obsolete products. In this case, the obsolete product is a photo paper format that is no longer in use. Business rules have already been identified as crosscutting concerns in literature. Ossher et al [11] modularize business rules as hyperslices and Cibrán et al [4] implement a business rule linking layer in AspectJ.
To introduce the obsolete product discount in the PhotLab application, extra behavior has to be inserted in the product database to be able to store and use the old product information. Remember, that this component is not designed to cope with obsolete products, so the aspect has to insert that behavior. As a result, the product database returns a discounted price for older products.
Figure 9 illustrates the *OldProductDiscount* invasive composition adapter that is able to capture this concern. *OldProductDiscount* is implemented by a JAsCo Aspect Bean which declares two hooks: *CaptureProduct* and *ApplyDiscount.* The context part declares that this invasive composition adapter is applicable on a consecutive REQUEST and ANSWER. Notice that a different hook is mapped on both the primi-

tives of the context part. The *CaptureProduct* hook is responsible for capturing all relevant information of the price request of a certain product. The *ApplyDiscount* acts on the answer of the request and changes the result if the product is considered obsolete. The adapter part of the *OldProductDiscount* invasive composition adapter declares that the request and answer are sent in the same way as before. Indeed, this invasive composition adapter only changes the interior behavior of the components that are mapped onto the *ProductDB* and *Requester* roles.

Figure 11 illustrates the application of the *OldProductDiscount* invasive composition adapter onto the *ECommerceServer* composition pattern. The *Requester* and *ProductDB* roles are mapped respectively onto the *Logix* and *productDb* roles of the *ECommerceServer* composition pattern. Afterwards the *PhotoLab* and *SimpleDatabase* components have been mapped onto these combined roles. As a consequence, the behavior of these two components is changed invasively by the JAsCo Aspect Bean implementation of the *OldProductDiscount* invasive composition adapter.

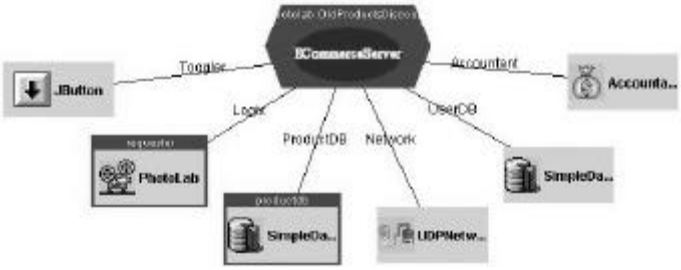


Figure 11: Applying the OldProductDiscount invasive composition adapter onto the ECommerceServer composition pattern.

5.2. SecureLogin aspect

The following two crosscutting concerns we identify are security concerns. The regular digital photo printing application does not consider any security measurements to protect the login of customers. It is for example possible to attack the server in a brute-force way by just iterating over all possible passwords. A good security measurement would consist of blocking user accounts after *x* number of false attempts for a certain period of time. This logic is however not the core functionality of the application and the overhead of the protection might not always be desirable. Therefore, we model the login protection concern as an invasive composition adapter. This allows us to reuse this concern in different applications without hard-coding it and scattering it in all components.

Figure 12 illustrates the *SecureLogin* invasive composition adapter. *SecureLogin* is implemented by a JAsCo Aspect Bean which declares two hooks: *CaptureUser* and *BlockUser*. Similar to the *OldProductDiscount* invasive composition adapter, *SecureLogin* specifies these two hooks in the context part. The *CaptureUser* hook captures the user where the request handles about and *BlockUser* hook changes the answer of the *Requester* role to a blocking answer if necessary. The application of this composition adapter causes the external behavior of the component that is mapped on

the *Requester* role to change. Indeed, after the ANSWER is sent to the *Network* role, the *Requester* role possibly sends a *userBlocked* event to interested observers. This event can be captured by a security user interface so that a human security manager might decide to unblock this account in case of a mistake or to take further actions in case a possible hacker is detected. It is however also possible to just map a statistics component or an artificial intelligence security agent component on this role as long as they adhere to the specified abstract protocol.

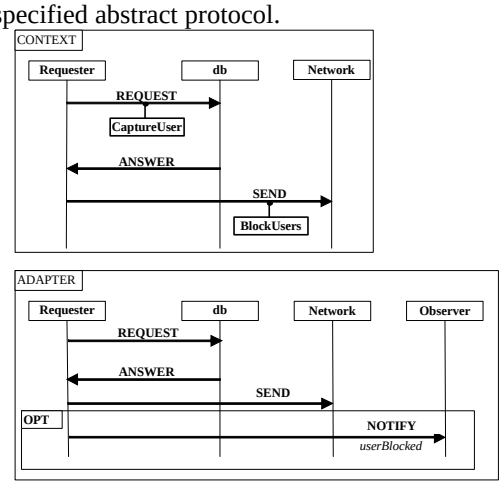


Figure 12: SecureLogin invasive composition adapter.

5.3. Encryption aspect

Another security problem is the transmission of data over the network. Currently, all information (including login information) is sent over the network in plain ASCII. While this might be sensible in some cases, we want to have the possibility to encrypt data that is sent over the network. In order to do this, we can employ a regular composition adapter that reroutes all the messages sent to the network component through an encryption component. Likewise, messages received from the network component can be rerouted through a decryption component.

In this way, the encryption concern is captured by a composition adapter and can be easily removed from the application. In addition, reusing this aspect becomes straightforward. The same encryption aspect is for example also applied in an internet chat application.

Notice that the encryption aspect has to be applied on both the client and server application in order to be useful. Indeed, only applying the encryption aspect onto the server application causes encrypted data to be sent to the client application that does not know how to cope with this.

6 Related Work

The Composition Filter [2] approach makes use of filters attached orthogonally to components, this way extending them with crosscutting behavior. A composition filter is actually very similar to a regular composition adapter as it is only able to specify *interaction* concerns. The interior of a component is normally not affected by a composition filter.
Duclos et al [6] focus on separating crosscutting concerns in legacy systems built using CCM [3]. Similar to PacoSuite, they specify crosscutting concerns at the architectural level. Aspects are applied by generating individually tailored CCM containers that include the aspect's logic. In that sense, their approach is similar to wrapping because they do not allow interior changes to the components.
Filman [7] proposes dynamic injectors to introduce aspects into a given component configuration. He incorporates dynamic injectors into OIF (Object Infrastructure Framework), a CORBA centered aspect-oriented system for distributed applications. The dynamic injector approach is very similar to our non-invasive composition adapter idea because both approaches employ a wrapping and filtering technique to insert crosscutting concerns into a composition of components.
Another more recent approach to recuperate aspect-oriented ideas in component-based software development is event based aspect-oriented programming (EAOP). EAOP [5] allows specifying crosscuts on events and event patterns using a formal language. Similar to the composition adapter approach, EAOP allows specifying aspects on a full protocol of events instead of a set of methods. Our approach extends EAOP by lifting the abstraction level for aspect application from the implementation level to a visual composition environment.

7. Conclusions

In this paper, we introduce an invasive composition adapter in order to specify crosscutting concerns that require interior adaptations of a component on a component-based design level. An invasive composition adapter is an extended version of a regular composition adapter and has an implementation in the JAsCo aspect-oriented language. A component composer is able to visually apply an invasive composition adapter on a given component composition. The invasive composition adapter is verified to be compatible with the composition and is automatically deployed using algorithms based on finite automata theory. Additionally, an invasive composition adapter can be easily removed from a collaboration when the concern is not desired any longer. As a restricted validation of this approach, we successfully modularize three crosscutting concerns in an e-commerce application.
A limitation of this approach is that it is domain dependent. It is possible to agree on a set of semantic primitives to document component interactions for a limited application domain. However, it is unfeasible to come up and agree on a general set of semantic primitives. To avoid the domain dependency, one could just forget about the semantic primitives and only document components with their API. This is of course

at the expense of reusability of components, composition patterns and composition adapters. Another limitation is that this approach is resource intensive. Our current algorithms are of exponential nature and in worst case scenarios this could lead to state explosions. In addition, the glue-code to translate syntactic incompatibilities between components adds an extra level of indirection. This is however the price that has to be paid for the increase of abstraction level and flexibility of component composition in PacoSuite.

9 References

[1] AspectJ Website. http://www.aspectJ.org.

[2] Bergmans, L. and Aksit, M. *Composing Crosscutting Concerns Using Composition Filters.* Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.

[3] Corba Component Model: see http://www.omg.org.

[4] Cibrán, M., D'Hondt, M. and Jonckers, V. *Aspect-Oriented Programming for Connecting Business Rules*. In Proc of BIS 2003. Colorado Springs, USA, June 2003.

[5] Douence, R., Motelet, O. and Südholt, M. *A formal definition of crosscuts.* In Proceedings of the 3rd International Conference on Reflection. Kyoto, Japan, September 2001.

[6] Duclos, F., Estublier, J. and Morat, P. *Describing and Using Non Functional Aspects in Component-based Applications.* In Proc. of AOSD 2002, Enschede, The Netherlands, April 2002.

[7] Filman, R.E. *Applying Aspect-Oriented Programming to Intelligent Synthesis.* Workshop on Aspects and Dimensions of Concerns at ECOOP2000. Cannes, France, June 2000.

[8] ITU-TS. ITU-TS *Recommendation Z.120: Message Sequence Chart (MSC).* ITU-TS, Geneva, September 1993.

[9] Lieberherr, K., Lorenz, D. and Mezini, M. *Programming with Aspectual Components.* Technical Report, NU-CCS-99-01, March 1999. Available at: http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html.

[10] Mezini, M., and Ostermann, K. *Conquering Aspects with Caesar*. In Proc of AOSD 2003, Boston, USA, March 2003.

[11] Ossher, H. and Tarr, P. *Using multidimensional separation of concerns to (re)shape evolving software.* Communications of the ACM, 44(10):43-50, Oct. 2001.

[12] Suvée, D., Vanderperren, W., and Jonckers, V. *JAsCo: an Aspect-Oriented approach tailored for Component-based Software Development.* In Proc of AOSD 2003, Boston, USA, March 2003.

[13] Vanderperren, W. *Localizing crosscutting concerns in visual component-based development.* In Proceedings of Software Engineering Research and Practice (SERP) International Conference. Las Vegas, USA, June 2002.

[14] Wydaeghe, B. PhD dissertation: PACOSUITE: Component Composition Based on Composition Patterns and Usage Scenarios. VUB, November 2001. Available at: http://ssel.vub.ac.be/pacosuite.