Towards Traceability between AO Architecture and AO Design

Andrew Jackson[1], Pablo Sánchez[2], Lidia Fuentes[2], Siobhán Clarke[1]

[1]Distributed Systems Group,
Dept. of Computer Science,
Trinity College Dublin, Ireland,
{Andrew.Jackson, Siobhan.Clarke} @cs.tcd.ie

[2]Dpto. de Lenguajes y Ciencias de la Computación,
ETSI Informática,
University of Málaga, Spain
{pablo,lff}@lcc.uma.es

Abstract
A significant challenge of software engineering is dealing with change. Traceability is a key issue when designing and maintaining large systems that are prone to change. Although AOSD is gaining maturity, variances in how AOSD concepts are realized still exist. This means that traceability for AOSD is problematic. In this paper we focus on providing traceability support from AO Architecture Design (AOAD) to AO design (AOD). To concretely illustrate traceability, we focus on particular scenarios and centre on two specific approaches – DAOP-ADL for AOAD and Theme/UML for AOD.

1 INTRODUCTION
A significant goal of software engineering is dealing with change. Traceability is a key issue when designing and maintaining large systems that are prone to change. The main purpose of traceability is to be able to identify how and where change affects software artifacts different stages in the software development lifecycle.

Aspect oriented software development (AOSD) is a paradigm that underlies the entire software development lifecycle. Approaches to AOSD that have been proposed, typically focus on one stage of the lifecycle. In addition, although AOSD is gaining maturity, variances in how AOSD concepts are realized still exist. This means that traceability for AOSD is problematic. In this paper we focus on providing traceability support from AO Architecture Design (AOAD) to AO design (AOD).

A prerequisite to providing traceability support between architecture and design is mapping the realization of AOSD concepts. Such traceability support may be provided by recording instances where design elements have been derived from architectural elements in accordance with a well defined mapping. It should be noticed that there may be several alternatives when creating design elements from architectural elements. The alternatives and justifications for choosing one alternative should also be recorded.

Recording traceability information between architecture and design in this manner has the following benefits:

1. The relationship between architectural and design elements is explicitly described.
2. The effect of architectural change on design can be identified through examining the recorded relationships.
3. The justifications for choosing alternatives can be reviewed to ensure they are the most appropriate in the context of the change.

To concretely illustrate traceability support, we focus on particular scenarios and centre on two specific approaches – DAOP-ADL [11] for architecture and Theme/UML [4] for design.

We have selected these proposals due to the strong variance in their underlying models of AO. Theme/UML is based on a symmetric concern separation model and in contrast DAOP-ADL is a asymmetric approach that models components and aspects. Using disparate approaches such as these provide a better illustration of the benefits of traceability from architecture to design.

The motivating scenario is based on an Auction System case study. In this scenario we assume two development iterations. In the first iteration we present a means for recording the alternatives available to the architect when specifying architecture and the justification for the selection of one specific alternative. Means for recording the alternatives available to the designer based on the selected architecture are also presented. In the second iteration we introduce architectural changes and demonstrate the benefits of recording the relationship between architecture and design.

We continue this paper by providing background (Section 2) with a description of the case study on which we base this investigation. We also describe AOAD and AOD focusing on DAOP-ADL and Theme/UML. We then describe a mapping between the constructs of both approaches (Section 3). We follow this by investigating traceability between architecture and design (Section 4). Before concluding the paper (Section 6) and discussing future work we describe related work (Section 5).

2 BACKGROUND

2.1 The Auction System

In this paper we take motivating scenarios from an Auction Case Study[1] to illustrate the traceability problem and our proposed solution.

Our motivating problem is based on two iterations in the development of the Auction Case Study. Iterative development is a cumulative process. New use cases may be addressed and previously addressed use cases are refined, in different iterations.

In this scenario, the first iteration is used to illustrate the mapping between architecture and design. One set of use cases is addressed and information to support traceability is gathered. The second iteration is used to show how traceability can help deal with change, by refining decisions made based on the use cases addressed in the first iteration. Figure 1 depicts the use cases that are addressed in the first iteration.

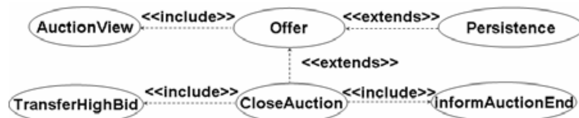


Figure 1 Use Cases description of Auction System

2.2 AO Architecture

The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of these elements, and the relationships among them. [2]

A component can be defined as "a unit of composition with contractually specified interfaces and explicit context dependencies only" [2]. A significant goal of Component Based Software Development[2] (CBSD) at the architectural level is to reuse existing components. "A software component can be deployed independently and is subject to composition by third parties" [2].

Some concerns, such as Persistence, Security, Mobility, etc. crosscut several components, making design and reuse of these concerns more difficult. AO Architecture Design solves this problem by encapsulating these crosscutting and often extra-functional elements in an aspectual component[3] and specifying how its behavior is triggered at the public interface of other components. This specification is achieved through new composition rules which define a crosscutting relationship between a component and an aspectual component.

[1] Requirements for Auction System can be found at [1]. This case study was previously used in [12].

[2] The link between Architectural design and CBSD is explored in [2].

[3] Aspectual components are defined in [3] and [11].

An illustration of the Auction System architecture is provided in Figure 2. This architecture is one of various possible architectures that meets the requirements.

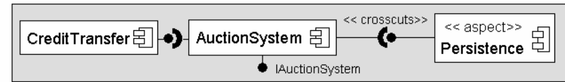


Figure 2 - An architecture for the Auction System

The architecture illustrated in Figure 2 is a composition of three main component roles: AuctionSystem, CreditTransfer and Persistence. AuctionSystem component realizes the role of the same name. The use cases illustrated in Figure 1 are encapsulated in the AuctionSystem component. To realize the TransferHighBid use case, an additional service CreditTransfer for transferring funds is used. Persistence is an extra-functional crosscutting requirement which is separated from the application at the architectural level. Persistence is represented as an aspectual component which is denoted by the <<aspect>> stereotype. The Persistence aspectual component is bound to the Auction System component, which is denoted by the <<crosscuts>> stereotype. Aspectual binding differs from the regular required/provided component binding. This difference can be illustrated through the AuctionSystem architecture. The AuctionSystem component has no dependency on the Persistence aspectual component and can be deployed independently of the persistence component. In contrast, CreditTransfer is subject to the regular required/provided component binding and as such with the AuctionSystem component. This dependency means that the AuctionSystem component cannot be deployed without a CreditTransfer component.

2.3 DAOP-ADL

Software Architectures can be described textually by means of Architectural Description Languages (ADLs). DAOP-ADL [11] is the ADL of the CAM/DAOP approach. CAM/DAOP is an AO approach to CBSD. CAM/DAOP comprises a component and aspect model - CAM, an architectural description language - DAOP-ADL, and a CAM implementation as AO distributed platform - DAOP.

The information described by DAOP-ADL is used by the DAOP platform to compose components and aspects at run-time. This ADL also helps to analyze architectures, verify architectural properties, code generation and traceability. In this paper we focus on the use of the ADL in traceability.

A fragment of a DAOP-ADL description of the architecture illustrated in Figure 2 is presented in Figure 3. We describe this in relation to the elements of DAOP-ADL relevant in this context.

```
1 <applicationArchitecture>
2 <components>
3  <component role="AuctionSystem">
4    <providedInterface>…</providedInterface>
5    <requiredInterface>…</providedInterface>
6    <eventInterface>…</eventInterface>
7    <stateAttributes>…</stateAttributes>
8  </component>
9  <component role="CreditTransfer"> …</component>
10 </components>
11 <aspects>
12   <aspect role="Persistence">
13     <evaluatedInterface>
14        <method name="offer" >
15     </evaluatedInterface>
16   </aspect>
17 </aspects>
18 <aspectEvaluationRules>
19   <sendMessage>
20     <source-comp />
21     <target-comp>
22        <roles>AuctionSystem</roles>
23     </target-comp>
24     <targetMessages>
25        <message name="offer"/>
26     </targetMessages>
27     <AFTER_RECEIVE>…
28         <aspect role="Persistence" />…
29     </AFTER_RECEIVE>
30   <sendMessage>
31  </aspectEvaluationRules>
32 </applicationArchitecture>
```

Figure 3 - DAOP-ADL description for the AuctionSystem architecture of Figure 2.

Components and Aspects are named in DAOP-ADL using role names. To detach component interfaces from their final implementations, we assign a unique role name to identify both components and aspect components. A role name identifies a specific functionality and will be played by a component that implements this functionality. These role names are architectural names that will be used for component and aspectual component composition and interaction, allowing loosely coupled communication among them - i.e. no hard-coded references need to be used for exchanging information.

Components: can expose three kinds of interface. These are provided interface, required interface and event interface. Provided interface describes those services and attributes exposed by a component. Required interface specifies context dependencies explicitly, indicating what services are required from specific components. Event interface defines the events that the component can throw. Composition between components is described in terms of provided and required interfaces. Figure 3, Lines 2-8, illustrate the description of the Auction System component.

Aspectual components: define crosscutting behavior. Figure 3, Lines 12-16, show the description of the persistence aspectual component. An aspect has an evaluated interface, which describes how an aspect is executed. An evaluated interface contains the join points, messages, event, creation and finalization of components that an aspect is able to intercept and evaluate. If an aspect is general enough to be able to evaluate any intercepted join point, this interface can be omitted.

Composition specification: this is where specific details about how aspects are applied to component interfaces. Figure 3, Lines 18-32, describe an execution point after the invocation of the offer method as a join point at which persistence is applied. As DAOP-ADL was designed for deal with black-box components, the definition of join points that intercept internal behavior of components is intentionally avoided.

2.4 Theme/UML

Theme/UML [4] is an extension to the UML that facilitates Aspect Oriented Design (AOD). Theme/UML supports a symmetric view of AO, in that it supports the modularization of crosscutting and non-crosscutting concerns in design.

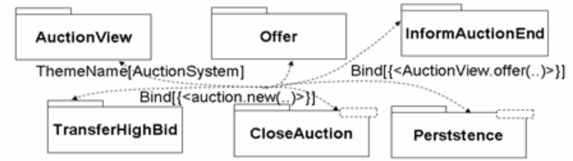
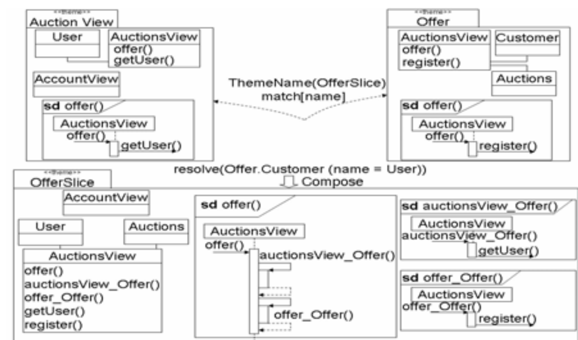


Figure 4 Auction System – Theme Design

Figure 4 presents one possible design (based on the theme approach) that realizes use cases illustrated in Figure 1. We can see that each functional concern is represented as a theme. The non-functional crosscutting persistence requirement is also extracted as a separate theme. Figures 5 and 6 provide more detailed views of this design and are used to describe the four main elements of Theme/UML.

Base Theme: Theme/UML represents non-crosscutting concerns as base themes. Figure 5 illustrates two base themes - AuctionView and Offer. Base themes are extensions of UML packages. Typically[4], these packages contain one class diagram and sequence diagrams to describe the interactions between classes described in the class diagram. The AuctionView and Offer themes illustrated in Figure 5 are examples base themes.

Merge: Base themes may be merged. Merge specifies that common concepts are unified. In Theme/UML, concepts are expressed as classes, methods and attributes. The effect of merge on classes and methods of the same name is their unification. The top of Figure 5 illustrates a merge specification and the bottom presents the resulting composite theme. The name of the composite is specified on the merge specification in this example the name for the composite is OfferSlice. In the composite we can see that classes are unified into one class diagram. AuctionsView is an example of a unified class. The offer method in this class is also unified. The resulting behavior of which is described in the offer() generated sequence diagram fragment.



---

[4] We note that Theme/UML supports the use of other diagrams [4 p160]

Figure 5 Auction System - Base theme composition

Aspect Theme: An aspect theme is an extension of a template package. CloseAuction illustrated in Figure 6 is an example of an aspect theme. CloseAuction exposes template parameters. These parameters represent join points. In aspect themes, sequence diagram fragments are used to describe the crosscutting behavior and its relationship with join points. The close() sequence diagram fragment is an example of this.

Aspect Merge: Aspect merge is an extension of the merge specification. The top of Figure 6 illustrates the merge specification for the CloseAuction and Offer themes. In the aspect merge, join points to be bound to the aspect themes exposed parameters are specified in a bind statement. In the resulting composite we see that each join point bound to the parameter results in a new sequence diagram fragment in which crosscutting behavior is described in relation to specific join points.

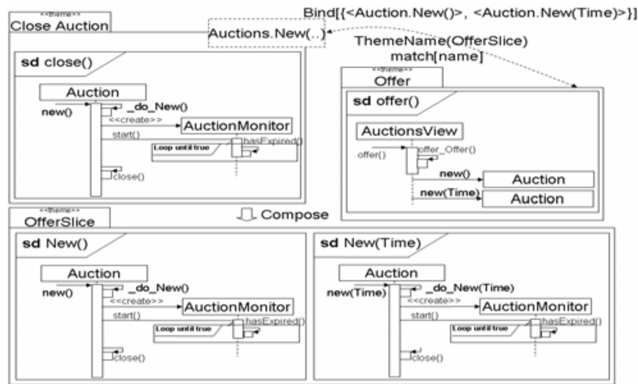


Figure 6 Auction System - Aspect theme

3 MAPPING AO ARCHITECTURE TO AO DESIGN

A prerequisite to providing traceability support between AO architecture an AO design is mapping AO concepts from architecture to design. To concretely illustrate traceability support, we consider a mapping and traceability between DAOP-ADL (AOAD) and Theme/UML (AOD).

In the development of an application, a set of requirements is initially mapped into an architectural design. In this approach, a set of uses cases are modularized requirements that are mapped into the architectural design. Each component participates in the realization of one or more use cases. A use case can be realized by more than one component. Figure 7 illustrates this mapping. Metadata information about which uses cases are realizing a component is stored in the traceability file. Each use case behavior or relationship is mapped to a set of provided/required methods of the components interface.

Once an architecture design is developed, we need an implementation for each component in it. As we commented before, an important goal of CBSD, and CAM/DAOP, is to reuse components. Therefore, at design time we look for previously developed components, in a component (and aspect) repository. Implementations for CreditTransfer and Persistence roles are found in this repository.

If we are unable to find implementations for a component in any repository, that component needs to be designed. Theme/UML is used for this purpose.

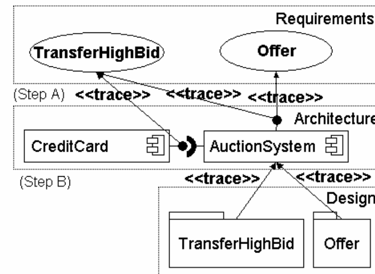


Figure 7 Mappings from requirements to architecture and from architecture to design

In Table 1 we present a high-level mapping between these approaches, this is then expanded based on our scenario.

Table 1 Description of how design element map to architectural elements

	Architecture	Design
1	Component (or an aspectual component) designed from scratch	Maps to a composition of themes and aspect themes specified through a merge – each theme contains one composite structure diagram in which the concerns are encapsulated.
2	Reused component	Not mapped to any specific Theme element, but it can be referenced in themes as a black-box component.
3	Reused aspectual component	Maps to an aspectual theme. The sequence diagram that describes the interaction between the aspectual component and the base components is mapped from the composition rules, as described below.
4	Composition specification (defined by the aspect evaluation rules)	Mapped to two elements. (1) The specification of how aspectual behavior should affect the join point (i.e. BEFORE_SEND, AFTER_RECEIVE, etc.) is mapped to the aspect theme sequence diagram which describes interaction between base and aspectual components. (2) The information about the source and target roles and the join point specification of DAOP-ADL , is used to construct the “bind” relationship between the aspect theme, containing the aspect, and the base themes, in Theme/UML.

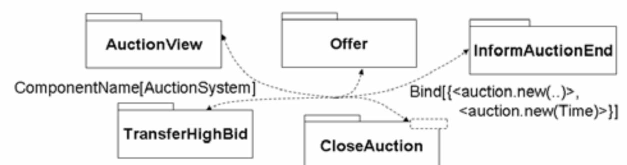


Figure 8 Component - theme & merge

In Figure 8, we illustrate rule one in our mapping. Here we see that the concerns (use cases) that the AuctionSystem component has to realize, and stored as metadata in the traceability file, are defined as themes. The merge specification indicates that the composition will result in a component named AuctionSystem. The themes presented in Figure 8 show that in each theme we use a composite structure diagram to show the internal structure of the component being designed.

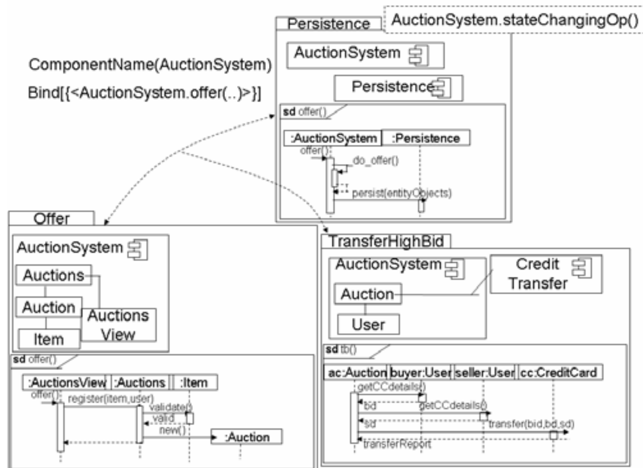


Figure 9 Aspectual component - aspectual theme

Rules two to four of our mapping are illustrated, in Figure 9. We can see that the CreditCard component is referenced inside the TransferHighBid theme. Internal of this component are not shown (Rule 2). Persistence, an aspectual component, is mapped to the persistence aspect theme (Rule 3). The aspect evaluation rules described in Figure 3, Lines 18-32 map to: (1) the sequence diagram shown in the persistence theme, which indicates that Persistence is applied AFTER_RECEIVE(ing) a generic message. (2) a bind relationship that specifies that Persistence has to be applied to the "offer()" message of the AuctionSystem component (Rule 4).

4 TRACEABILITTY

Traceability support is provided by recording instances where elements in design have been developed based on architectural elements. Several alternatives may exist when designing concerns based architectural designs. These alternatives and a justification for choosing a specific alternative are also recorded.

In the first iteration we will demonstrate how this information is recorded. In the second iteration we will continue to show how information to traceability is recorded. We will also illustrate how information recorded in the first iteration may be useful when dealing with both anticipated and unanticipated change.

4.1 First Iteration

In the first iteration of the Auction System development, components to realize the CreditTransfer and Persistence roles are reused, and AuctionSystem component is developed from scratch.

```
15 <component role="AuctionSystem">
16  <alternative id="1">
17   <designed byModule="AuctionSystem" href=".../AuctionSystem(1).xmi"/>
18    <justification>We map each concern to a theme</justification>
19  </alternative>
20  <alternative id="2">
21    <designed byModule="AuctionSystem" href=".../AuctionSystem(2).xmi" />
22    <justification>
23    We map each concern to a theme, but TransferHighBid and InformAcutionEnd
24    do not contain enough functionality to be described as separate themes.
25    </justification>
26  </alternative>
27  <alternativeSelected id="2">
28  <justification>Alternative 1 is using too many themes, making composition
     more complicated. Alternative 2 looks easier to maintain. This may change
     is new use cases are added, and separate functionality of TransferHighBid
29     and InformAcutionEnd would make sense</justification>
30  </alternativeSelected>
31 <forwardNotes>All concerns where this component is involved must be
32               implemented in design</forwardNotes>
33 <backwardNotes>....</backwardNotes>
34 </component>
```

Figure 10 Differing component design alternatives

Figure 10 depicts the first of several fragments of XML which describe the alternatives (identified in the first iteration) available when mapping architecture to design. In Figure 10, Lines 16-26, we see the specification of the alternatives available for designing the AuctionSystem component. The two alternatives are illustrated in Figure 11. The alternatives here relate to the level of theme decomposition of the AuctionSystem Component. In Lines 18 and 22-25, justification for choosing each alternative is recorded. AuctionSystem component needs some messaging support. However, the messaging needs of the AuctionSystem component do not justify the introduction of an aspectual component for messaging. If new uses cases were added in future iterations, to separate InformAuctionEnd and TransferHighBid themes would make sense, because this decomposition could get benefits from the messaging aspectual component.

The selected alternative and its justification are recorded (see Lines 27-30). Here we can see that alternative two has been chosen and this choice has been justified based on composition complexity and maintainability. Other information such as forward/backward notes and hyperlinks to other documents may also be recorded.

The alternatives for designing or reusing existing components to realize the CreditTransfer component are described in Figure 12, Lines 37-50. In this description we can see that the CreditCard COTS component is selected from a specific COTS repository to realize this component (see alternative selected on lines 51-54). The justification for this decision is that it offers the cheapest reuse option and it is supportive of current business needs.

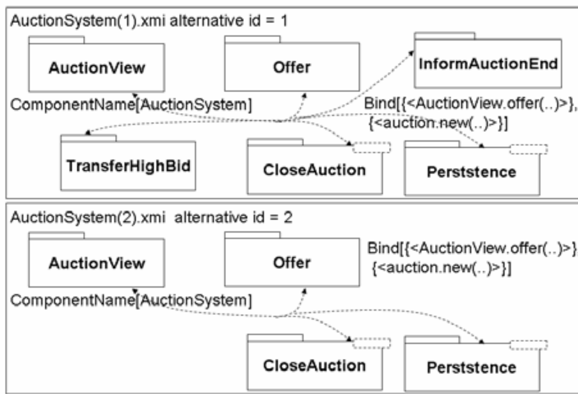


Figure 11 Design alternatives

```
37 <component role="CreditTransfer">
38  <alternative id="1"><designed byModule="CreditCard(1).xmi"/>...<alternative>
39  <alternative id="2">
40    <reused from="CreditCard">
41      <COTSrepository uri="....." />
42      ...
43    </reused>
44  </alternative>
45  <alternative id="3">
46      <reused from="VISA">
47        ....
48        ....
49      </reused>
50  </alternative>
51  <alternativeSelected id="2">
52    <justification>We can reuse a COTS component, it is more cost effective
53    </justification>
54  </alternativeSelected>
55 </component>
```

Figure 12 Component reuse alternatives

The Persistence aspectual component and alternatives for its realization are described in Figure 13 lines 57-59. The first alternative is to design the persistence module. The next alternative is to reuse an OraclePersistence component and the last alternative is to reuse a SybasePersistence component. The decision taken in this instance is the selection of alternative where the OraclePersistence component is reused, because Oracle is already presented in the application enviroment. Justification for this decision appears in lines 60-63.

```
56 <aspect role="Persistence">
57   <alternative id="1"><designed byModule="Persistence">...</alternative>
58   <alternative id="2"><reused from="OraclePersistence">...</alternative>
59   <alternative id="3"><reused from="SybasePersistence">...</alternative>
60   <alternativeSelected id="2">
61      <justification>The application is going to be deployed in an environment
62      that is already using Oracle</justification>
63   </alternativeSelected>
64 </aspect>
```

Figure13 Design or reuse alternatives

At the end of the first iteration the recorded information ensures that the relationship between the architectural and design elements is explicit. This improves the clarity of architecture-design relationship.

4.2 Second Iteration

Figure 14 illustrates both anticipated change and unanticipated changes that occur at the architectural level in the second development iteration.

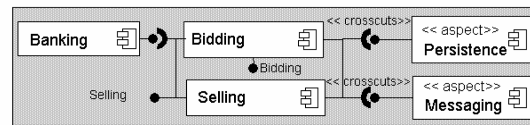


Figure 14 Changes at the architectural level

Anticipated change: In Figure 14 a messaging component is introduced at the architectural level. This new component is introduced because new use cases addressed in the second iteration demand support for more complex messaging. In the first iteration, this was anticipated. In the first iteration, AuctionSystem was identified as a component demanding some messaging support. However, its messaging needs were not enough to justify the introduction of a new component at architectural level. Some alternatives of AuctionSystem design were marked as more suitable than the selected currently if messaging were introduced (at architectural level) finally.

The introduction of this new messaging component makes Alternative 1 (see Figures 10 & 11) for AuctionSystem component more attractive.

Figure 15, Lines 73-84 show how the selection of this alternative may be recorded as traceability support information. Figure 15 also illustrates the way in which this architecture design change is recorded. Lines 65-70 show the possible alternatives for the realizing the new messaging component, the alternative selected, and associated justifications, etc.

```
65 <component role="Messaging">
66  <alternative id="1">
67    <reused from="MessageService">....</reused>
68  </alternative>
69  <alternativeSelected id="1">...</alternativeSelected>
70 </component>

71 <component role="AuctionSystem">
72   <!-- See Figure 6 for alternatives -->
73   <history>
74     <iteration id="1">
75       <alternativeSelected id="2">
76         <justification> Too many themes are used in Alternative 1, making
77         composition more complicated. Alternative 2 looks easier to
78         maintain.</justification>
79       </alternativeSelected>
80       <reasonOfChange>A Messaging Component has been introduced at
81       architectural level. Separating InformAuctionEnd makes sense now because
82       this theme can benefit of the new component.</reasonOfChange>
83     </iteration>
84   </history>
85 </component>
```

Figure 15 Introducing messaging

In Figure 14 a Banking component replaces the CreditTransfer component. They are different architectural components, with different interfaces, protocols and semantics. Banking is transferring real funds between accounts, whereas CreditTransfer is more flexible because it is managing credits (which will be associated to a real bank account later). Figure 16 illustrates the introduction of a Banking component to replace the CreditTransfer component. In Figure 16, Lines 90-91 the component introduction is defined along with alternatives for realizing this component. In Figure 16, Lines 94-A4 the replacement, reason for

the replacement and impact of the replacement are described.

```
90 <component role="Banking">
91  <alternative id="1">...</alternative>
92  ...
93 </component>

94 <component role="CreditTransfer">
95  <history>
95   <iteration id="1">
96    <component role="CreditCard">
97      <alternativeSelected id="3">
98      <reasonOfChange>We must deal with Banking instead of CreditCard because
99      of it is provide by a new  partner of the company</reasonOfChange>
A0      <impact>We have to check interface compatibility. Adapters for Baking
A1      component could be required</impact>
A2      </component>
A3   </iteration>
A4  </history>
```

Figure 16 Replace CreditCard with Banking

Unanticipated change: In the scenario illustrated in Figure 14, new use cases related to bidding are addressed. With the addition of these use cases an unanticipated decision is taken to decompose the application into two components, bidding & selling. From Figure 10 & 11, we can see that the designer can easily see which themes are associated to the initial AuctionSystem component. Some of these themes may be reused in the design of bidding and selling components. By using the information recorded in Figure 10 and illustrated in Figure 11 the designer can quickly assess and plan for the pending change.

```
A5 <component name="Selling">...</component>
A6 <component name="Buying">...</component>

A7 <history>
A8  <iteration id="1">
A9    <component role="AuctionSystem">
B0     <split>
B1        <component role="Bidding" />
B2        <component role="Selling" />
B3     </split>
B4     <reasonOfChange>Splitting up is done to leverage reuse opportunities
B5     of the selling component in other commercial applications</reasonOfChange>
B6     <impact>...</impact>
B7    </component>
B8  </iteration>
B9 </history>
```

Figure 17 Splitting up Auction System

Figure 17 illustrates how a new decomposition of the Auction System into two new components – Bidding and Selling. The introduction of the Bidding and Selling are shown in Figure 17, Lines A5-A6. Figure 17, Lines A7-B9 describe how the component is split up, the reasons of change and the possible impacts on design indicating all the modules that could be affected.

Though these examples we have seen that the effects of both anticipated and unanticipated architectural change can be identified though examining the information recorded in the first iteration. We have also seen that by analyzing the justifications for selection of alternatives in the first iteration can aid the designer to change design decisions in the second iteration.

5 RELATED WORK

In [6] and [4] surveys of traceability approaches are presented. Although AOSD is gaining maturity, there is little work addressing traceability between AO approaches at different stages in the software development lifecycle.

In [6], a component oriented method which provides a mapping from AO architecture and AO design. In [8] an AO method driven by uses cases is presented. This approach also provides a mapping from AO architecture and AO design. However, these do not explicitly address traceability. In addition, these proposals have been developed based on specific AOP platforms and consequently constructs used at architectural and design levels neatly map to one another and to their AOP implementation target.

Traceability from AO requirement analysis to AO architecture is investigated in [3]. We take a similar approach in this paper and hope of integrate this work in the future.

6 SUMMARY AND FUTURE WORK

AO is a software development paradigm that is maturing. AO approaches which address crosscutting concerns at different stages of the lifecycle have been proposed.

In this paper we have investigated the provision of traceability support from AO architecture design to AO design. We have identified integrated AO approaches that provide support means for addressing crosscutting concerns both architecture and design stages in the software development lifecycle.

A mapping between the constructs used to represent similar AO concepts at the architecture and design level is a prerequisite to traceability. In this paper, we have used architecture and design approaches that are completely independent and based on different models of AO. This has been done as this is more representative of the existing AO architecture and AO design research.

In this paper we firstly create a mapping between DAOP-ADL and Theme/UML. On this basis, we illustrate both how to record information relevant when supporting traceability from AO architecture design to detailed AO design. We also illustrate how this information may be used to aid the design in the face of architectural change.

In our future work we will continue to expand this initial investigation. Our goal is to extend traceability support from requirements to implementation. Our next steps to achieving this include the integration of the existing work done in [3] and the investigation of traceability from AO requirements engineering and AO design.

ACKNOWLEDGMENTS

This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

REFERENCES

[1] Auction System Requirement Description. Available on line: http://lgl.epfl.ch/research/fondue/case-studies/auction/problem-description.

[2] Bass L., Clemens P., Kazman R., "Software Architecture in Practice. 2nd Edition", Addison Wesley Professional. 2003.

[3] Chitchyan R., Pinto M., Fuentes L., Rashid A., "Relating AO Requirements to AO Architecture." Early Aspects Workshop, OOPSLA 2005. San Francisco, (California, USA), September 2005.

[4] Clarke S., Baniassad E., "Aspect-Oriented Analysis and Design: The Theme Approach", Addison-Wesley Professional, February 2005.

[5] Gills M., Survey of Traceability Model in IT projects. Workshop on Traceability, EC-MDA, November 8, Nurnberg, Germany, 2005.

[6] Gotel O. C. Z., Finkelstein A. C. W., "An Analysis of the Requirements Traceability Problem", Software Change Impact Analysis, 1996.

[7] Grundy J., "Multi-perspective specification, design and implementation of software components using aspects", Journal of Software Engineering and Knowledge Engineering. 20 (6), 713-734. 2000.

[8] Harrison W. H., H. Ossher L., Tarr P. L., "Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition" Tech-Report, IBM-RC22685. 2002.

[9] Jacobson I., Ng P, "Aspect-Oriented Software Development with Use Cases", Addison Wesley Professional. December 2004.

[10] Jackson A., Clarke S., AO Design Section of "Survey of Aspect Oriented Analysis and Design Approaches" p163-232. http://www.aosdeurope.net/documents, 2005.

[11] Pinto M., Fuentes L., Troya J. M., "DAOP-ADL: an architecture description language for dynamic component and aspect-based development", In Proc. of 2nd Int. Conf. GPCE, pp 118-137, Erfurt (Germany). 2003.

[12] Sendall S., Strohmeier A., "Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML", In Proc. of 4th Int. Conference, UML 2001, Toronto (Canada), October, 2001.