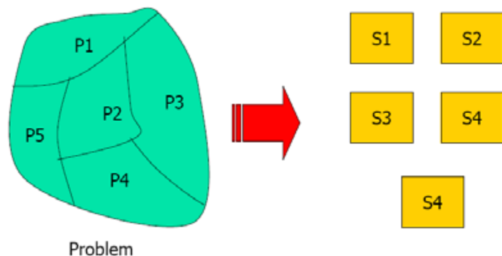
Moving From OOSD towards AOSD

Gurpreet Singh, Manish Mahajan
RIMT-IET, Mandi Gobindgarh

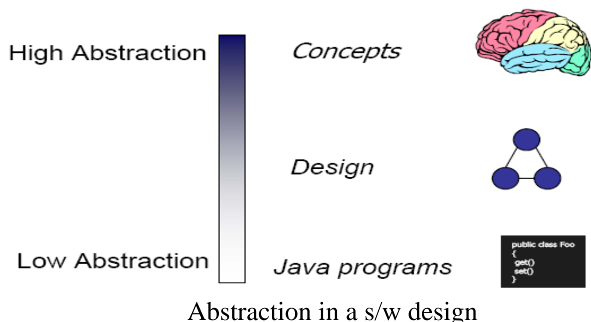
Abstract --Capturing concerns that crosscut the boundaries of multiple components in software architecture descriptions is problematic. Standard description languages, such as UML, do not provide adequate means to understand and modularize such concerns, but AOP (Aspect Oriented Programming) techniques do. This paper explores and analyzes the suitability of UML for aspect-oriented architectural modeling. It takes a bottom-up approach, starting from the code level to the level of software architecture description, via aspect-oriented design, using standard UML

What is the problem?

The problem is to Design a s/w. To do this we can use OOP's concepts. Object Oriented Programming says that instead of designing all the functions in a single module we can divide the functions into different modules depending upon the concerns they fit into. For this we can use divide & conquer principle of s/w engineering, which says that, divide the problem into smaller modules until it cannot be further divided into much smaller modules



OOPs further suggest that design must be such, so that the complexity is hidden from the user, which make it much simpler then the design in which there is no abstraction.



Abstraction in a s/w design

Problem with OOPs concept

There is a well documented problem in the software engineering field relating to a structural mismatch between the specification of requirements for software systems and the specification of object-oriented software systems. The structural mismatch happens because the units of interest during the requirements phase (for example, feature, service, capability, function etc.) are different to the units of interest during object-oriented design and implementation (for example, object, class, method, etc.). The structural mismatch results in support for a single requirement being scattered across the design units and a single design unit supporting multiple requirements - this in turn results in reduced comprehensibility, traceability and reuse of design models. Currently, OOP serves as the methodology of choice for most new software development projects. Indeed, OOP has shown its strength when it comes to modeling common behavior. However, OOP does not adequately address behaviors that span over many -- often unrelated -- modules. Separation of concerns is a basic engineering principle that is also at the core of object-oriented analysis and design methods in the context of UML. Separation of concerns can provide many benefits: additive, rather than invasive, change; improved comprehension and reduction of complexity; adaptability, customizability, and reuse. In contrast, AOP methodology fills this void. AOP quite possibly represents the next big step in the evolution of programming methodologies. However, for aspect-oriented software development (AOSD) to live up to being a software engineering paradigm, there must be support for the separation of crosscutting concerns across the development lifecycle including traceability from one lifecycle phase to another. Concerns that have a crosscutting impact on software (such as distribution, persistence, etc.) present well documented difficulties for software development. Since these difficulties are present throughout the development lifecycle, they must be addressed across its entirety. The separation and encapsulation of crosscutting concerns has been promoted as a means of addressing these difficulties; the standard object-oriented paradigm does not suffice. In order to overcome the difficulties for crosscutting concerns throughout the lifecycle, an approach is required that provides a means to separate and encapsulate both the design and the code of crosscutting behavior.

Aspect Oriented Programming and Design

A gap exists between requirements and designs on one hand and between design and code on the other hand. Aspect oriented programming (AOP) extended to the modeling level where aspects could be explicitly specified during the design process will make it possible to weave these aspects into a final implementation model. Another step could be extension of AOP to the entire software development cycle. Each aspect of design and implementation should be declared during the design phase so that there is a clear traceability from requirements through source code thus using UML as the

design language to provide an aspect oriented design environment.

It is important to work towards a general purpose AOSD design language that meets certain goals including the following:

A. *Implementation language independent*: The final form of AOP language may vary from that of any current one. Thus, any design language that simply mimics the constructs of a particular AOP language is liable to fail to achieve implementation language independence.

B. *Design-level composability*: Design level composability is a desirable property for two reasons. First designers may check the result of composition prior to implementation, for validation purposes. Second, some projects will continue to require the use of a non-aspect oriented implementation language because of pragmatic constraints, such as the presence of legacy code written in languages without aspect oriented extensions; these projects could still benefit from separating the design of crosscutting concerns.

C. *Compatibility with existing design approaches*: An AOSD design-level language should also build existing design languages such as UML, to provide a bridge from old techniques to new, so that software engineering realities such as incremental adoption and legacy support are possible.

Why do we need Aspect Oriented Design in Software Development?

The identification of the mapping and influence of a requirement level aspect promotes traceability of broadly scoped requirements and constraints throughout system development, maintenance and evolution. The improved modularization and traceability obtained through early separation of crosscutting concerns can play a central role in building systems resilient to unanticipated changes hence meeting the adaptability needs of volatile domains such as banking, telecommunications and commerce. These crosscutting concerns are responsible for producing tangled representations that are difficult to understand and maintain. Examples of such concerns at the requirements level are compatibility, availability and security requirements that cannot be encapsulated by a use case and are typically spread across several of them. With increasing support for aspects at the design and implementation level, the inclusion of aspects as fundamental modeling primitives at the requirements level and identification of their mappings also helps to ensure homogeneity in an aspect oriented software development project. The main drive behind aspect oriented design language research is the idea of developing design constructs (elements) that exhibit a degree of flexibility and Customizability that is only known from programmable end systems. While new design language constructs based on aspect oriented programming are being designed they are still tied to a particular platform where by the vendor provides both the software tool and the design language tool as a complete package with additional proprietary tools. Thus, new design language aspect constructs can only be tested or utilized to individual specific requirements after the vendor has released a software upgrade. The development of new functionality is typically preceded by a long and awkward standardization process. These different paradigms have created an increasing gap between the functions and capabilities of these constructs in an aspect oriented development environment. Reconsidering the system architecture of object oriented software applications is therefore a crucial step in aspect oriented software development.

Aspect Oriented Software Development Design Language

AspectJ is a popular and well established AOP language that provides support for specifying and composing crosscutting code into a core system. It supports the AOP paradigm by providing a special unit, called “aspect”, which encapsulates crosscutting code. Other compositional implementation languages and mechanisms also exist. At the design level, an AOSD design language with extensions to UML in its capabilities relating to decomposition and modularization is required that would map to a particular AOSD implementation. Further, a standard AOSD design language must be capable of supporting many of these aspect programming languages. A graphical notation helps developers to design and comprehend aspect-oriented programs. Further, it would facilitate the perception of aspect-orientation. A design notation helps developers to assess the crosscutting effects of aspects on their base classes. Its application carries over the advantages of aspect-orientation to the design level and facilitates adaptation and reuse of existing design constructs.

Supporting Aspect-Oriented Modeling

This section addresses UML support for object-oriented modeling, and discusses the appropriateness of current UML for modeling of aspect-oriented software systems.

a. Using UML for Object-Oriented Modeling

Modeling object-oriented concepts is well supported by UML. Consider, for example, a simple banking system, which the following Java classes Account and Customer are part of:

public class Account {
private int balance = 0;
public void withdraw (**int** amount) {...};

public void deposit (**int** amount) {...};
public int getBalance() {...};
}
public class Customer {
public String name;
// inside some method (a is an account)
a.withdraw(50);}

To model the static and dynamic structure shown in the Java code, two types of UML
diagrams could be used, i.e., a collaboration diagram and a class diagram, each providing a different view. The collaboration diagram shown in Figure 1 illustrates a behavioral view that focuses on the interaction between a Customer object c invoking a method of an Account object a. Collaboration diagrams typically give you an idea about how objects work together, showing both the participant objects (part of the structural characteristics) and the interactions between them (the messages they exchange). In our example, c calls the method withdraw of a, which is shown in the figure by the arrow labeled with withdraw (50).

Fig. 1: A UML Collaboration Diagram

As in this case no further information is given on how the customer c gets to know the account a, we must assume that they are statically linked together. This results in an association link between the two classes, as depicted by the class diagram in Figure 2. Thus, Figure 2 shows a static structure view of (this part of) the system with a particular focus on how the classes relate to each other.

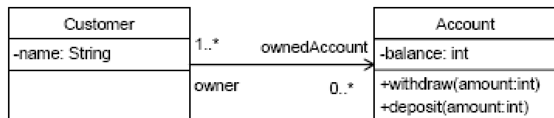


Fig. 2: A UML Class Diagram

As the simple banking system evolves, the requirements change. Developers might
be asked to add the following new feature to the system: every access to an account
object should be logged, recording the name of the accessing customer and the type of access on a log file. This logging feature is a typical example of a crosscutting concern, which can not easily be represented in an object-oriented design. The concern will inevitably be scattered throughout the model and / or entangled with other features. Adding such a feature is best supported by aspect-oriented software development. The most intuitive join points defined by AspectJ are method calls (not including super calls). The pointcut designator that allows a programmer to pick out a set of method calls based on the static signature of a method has the form:

pointcut someName : *call*(Signature);

Using the pointcut construct, it becomes straightforward to write an aspect that intercepts all calls to object instances of a certain class, performs some preprocessing, proceeds with the call, and finally does some postprocessing. In the example below, the Logging aspect intercepts all method calls made by customers on an account object and logs the access to a file:

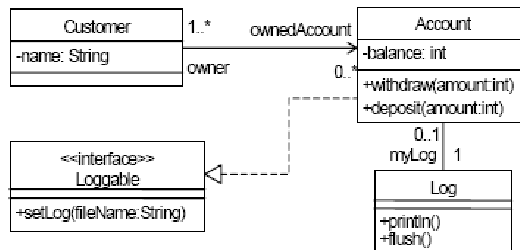


Fig. 3: Class Diagram for Account Logging Aspect

A typical way of modeling interception of method calls, when using UML, is to
add a new class, whose instance will be interposed between the interacting objects.
This technique is illustrated in Figure 4. An interceptor objects i is placed between the customer and the account objects. In addition to implementing the logging feature itself, the object i must:

• Offer an interface that supports all methods of the account object that are invoked by the customer object.
• Provide a mechanism to forward intercepted calls to the account object.

Instead of calling the account object directly, the customer object now actually calls the interceptor object (1: withdraw (50)). In our example, the logging action is performed in the after advice. Therefore, the message 1: withdraw (50) is first forwarded to the account object as 1.1: withdraw (50), and upon return the call is logged by with the account object.

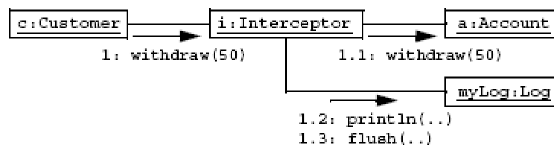


Fig. 4: Collaboration Diagram With an Interceptor

Extending UML for AOSD

Code-driven design, as presented in the previous section, limits the ability to understand various kinds of concerns, since it enables expressing the crosscutting nature of software concerns from only one single perspective (a low-level, static and textual view of the system). This makes it difficult to integrate aspects with other software artifacts and to reason about modules of crosscutting concerns from different perspectives or viewpoints. Indeed, developing aspect-oriented software requires thinking of an aspect as an abstraction that defines a certain interaction context, and offers behavior that can vary depending on certain conditions at run-time. Fulfilling such a requirement necessitates the ability to understand and describe the system from multiple viewpoints. To overcome the shortcomings of current modeling

techniques, aspects need to be treated as first-class citizens in advanced modeling languages. We propose to define an aspect as a UML model element that modularizes crosscutting concerns at various levels of abstraction, not only at the code-level. Consider, for example, the logging aspect introduced previously. To capture the logging aspect in a single module using UML, we need to introduce a representational unit that encapsulates the role of the interceptor object as well as the interactions between the participant objects of the classes Customer, Account and Log.

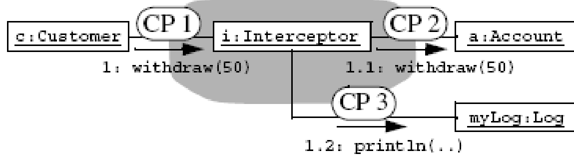


Fig. 5: Identifying Connection Points

Let's go back to our collaboration diagram. Figure 5 is similar to Figure 4, except that the additional structure introduced by the logging aspect is highlighted in grey. We will now proceed and make this grey part a first-class citizen, and determine what must be part of this aspect, and what not. Clearly, the participants c, a, and myLog are not part of the aspect, since they perform computation on the outside. It is the role of the interceptor object to mediate the interactions between these objects, linking them together. The interceptor therefore should be part of the aspect. The mediation itself happens at the connection points highlighted in Figure 5 by CP 1, CP 2 and CP 3.

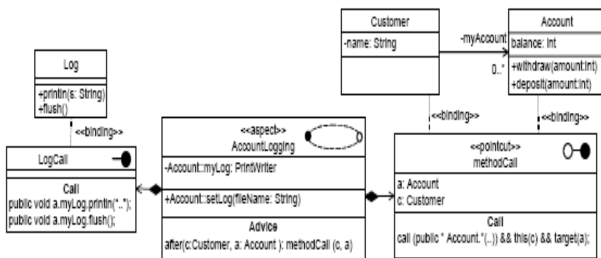


Fig. 6: The Logging Aspect Model using a UML Collaboration Stereotype

By making the notion of connection points explicit, we are now able to define stand-alone aspects and reason about them as a particular kind of UML collaboration. They clearly separate crosscutting interaction concerns from computation as performed by participant objects. The connection points being well-defined parts of the aspect, it is possible to bind them to individual objects. In UML, a binding corresponds to the concept of attaching a classifier role to an association role end. In our case, the association role end corresponds to a connection point. However, in contrast to standard UML, our model focuses on explicit modeling of the association role end as part of the aspect construct, allowing one to clearly separate modeling of interaction concerns from the "dominant" structure of classifier roles. We propose to define a new stereotype of UML collaboration to support aspect-oriented modeling with UML. This enables us to instantiate the new *aspect classifier* to model interaction concerns in an explicit and reusable way. Figure 6 illustrates this idea. It consists of four types of elements: the *aspect* itself, the associated *connection points*, normal UML classes, and the *binding* relationship. The aspect itself, highlighted in the figure by a dotted oval, specifies the actions to be performed at the connection points and along the interconnections, along with static information that will be woven into the objects of the classes it binds to. For example, in Figure 6, the first two compartments of the aspect represent two elements of the logging feature, an attribute and a method, which need to be woven into the Account class at binding time. The *Advice* compartment defines an action to be executed after returning from a method call to an account object instance. This action encapsulates the actual code calling the log.

The connection points are shown in Figure 6 using white and black circles. White circles are incoming connection points. Black circles are outgoing connection points. A white and a black circle connected by a line means that the connection points are conjugated. A connection point is the construct designed to model pointcuts, but it is general enough to cover also other kinds of points of interactions. At weave-time, objects bind to the connection points offered by aspects. The binding relationship specifies what class of objects an instance of the aspect can be bound to. In our example, the aspect must be bound to instances of the classes Customer, Account and Log. Therefore, our aspect could, at weave-time, interconnect customer c with account a and the log object named mylog

Aspect Design Model

The aspect design model is based on one presented in section above, but this time, each participant object is treated as a separate black-box component.

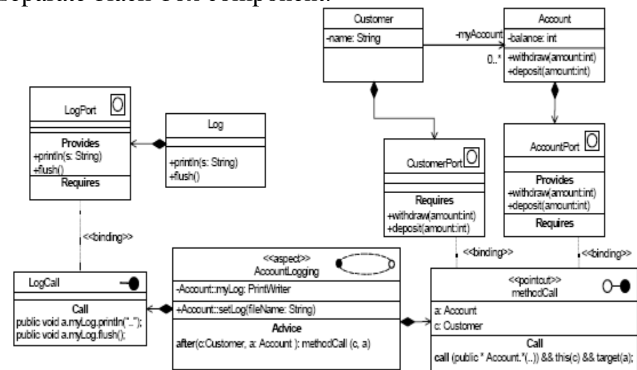


Fig. 7: Aspect Design Model

Figure 7 shows the aspect design model of our Logging aspect. In addition to Figure 6, it now contains three ports. They are named CustomerPort, AccountPort and LogPort. Ports have strong similarities with UML Interfaces. However, like connection points, ports may have attributes and can be instantiated. Also, each port defines two compartments, specifying the services it provides to, but also the ones it requires from the environment, unlike UML Interfaces.

CONCLUSION

Since UML does not define the idea of weaving, the nicely separated concerns in the aspect-oriented program ended up scattered throughout the design model. In addition, we showed that by making aspects first-class citizens, we can nicely separate crosscutting concerns. To capture different facets of an aspect, we proposed an architectural model named aspect design model. The Aspect Design Model shows the static structure of the aspect at type level. It specifies well-defined connection points, which are the basis for pluggability, since they specify the aspect interface. Likewise, ports are added to participant components, stating both provided and required services, and exposing possible extension points. Connection points and ports together determine what components the aspect can connect.

REFERENCES

[1] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher: *"Discussing Aspects of AOP*". *Communications of the ACM **44**(10)*, pp. 33–38, October 2001.

[2] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr.: "**N** *Degrees of Separation: Multi-Dimensional Separation of Concerns*". In *Proceedings of the 1999 International Conference on Software Engineering*, pp. 107 – 119, Los Angeles, CA, USA, 1999, IEEE Computer Society Press.

[3] J. Rumbaugh, I. Jacobson, and G. Booch: *The Unified Modeling Language Reference Manual*. Object Technology Series, Addison Wesley Longman, Reading, MA, USA, 1999.

[4] G. Booch, J. Rumbaugh, and I. Jacobson: *The Unified Modeling Language User Guide*. Addison–Wesley, Reading, Massachusetts, USA, 1 ed., 1999.

[5] *Uniform Support for Modeling Crosscutting Structure by Maria Tkatchenko, Gregor Kiczales.University of British Columbia Software Practices Lab*

[6] MOVING FROM AOP TO AOSD DESIGN LANGUAGE by Deepak Dahiya, Rajinder k Sachdeva