

Using Association Aspects to Implement Organisational Contracts

Alan Colman¹ Jun Han²

*Faculty ICT
Swinburne University of Technology
Melbourne, Australia*

Abstract

The maintenance of organisation is a prerequisite for all viable systems in dynamic environments. In many living systems this organisation is, in part, achieved through coordination systems such as the nervous or endocrinic systems that can be seen as *separate* from the functional systems they coordinate. As software systems become more open and complex, the definition of separate organisational structures may prove a useful way to maintain their viability while managing their complexity. In this paper we show how a coordination system can be implemented as a *separate concern*, and *posterior*, to the definition of the functional system it controls and regulates. Such functional systems are loosely coupled collections of roles played by objects. We show how *association-aspects* can be used to create contracts that bind these roles together into an organisation. These contracts regulate the flow of control through a structure of roles in the organisation, and allow performance to be specified and monitored. These contracts also bind clusters of roles into self-managed composites — each composite with its own organiser role. The organiser roles can control, create, abrogate and reassign contracts. This ability enables organisers to reconfigure the system in response to changes in external conditions or changes in performance requirements.

Key words: Contracts, Association Aspects, Coordination.

1 Introduction

In changing environments, where goals of a software system may also change, adaptable software needs to achieve its goals while maintaining its viability. Explicit organisational abstractions are useful for designing, maintaining and regulating systems in complex, open environments. This paper shows

¹ Email: acolman@ict.swin.edu.au

² Email: jhan@ict.swin.edu.au

how to create a coordination system that can help maintain the organisational viability of the system. This coordination-system can be described, implemented and controlled independently from the sub-systems that interact directly with the application domain. This approach is analogous to the coordination-systems that exist both in living things and in man-made organisations. In the realm of biology, the nervous system can be viewed as a system that, in part, coordinates the respiratory, circulatory, and digestive systems. Similarly, the management structure or the financial system in a manufacturing business can also be described at a separate level of abstraction from the functional processes that transform labour and material into products.

This paper describes coordination systems that are built from a hierarchy of organiser roles that control the contracts between functional roles. We have previously described this ROAD (role-oriented adaptive design) framework at a conceptual level in [3], where we suggested that Association-aspects were a suitable mechanism for implementing a coordination system. In this paper we show how this can be done. Such coordination systems can be developed independently and then superimposed *post-facto* on functional systems. These ROAD contracts have both a management and a domain-function level. Management contracts specify the type of communication acts and protocols that are permissible between the two parties. Functional contracts have clauses that specify, among other things, the performance obligations of the parties to each other. Abstracting management-contract aspects from functional-contract aspects makes possible, through contract inheritance, the reuse of their communication-control capability in many types of organisational structure. In this paper we show how such contracts can be implemented using *association-aspects*. Association-aspects as implemented in [9] are an extension to AspectJ [5] language and compiler that enables an aspect *instance* to be associated with a group of objects. This makes such aspects a suitable construct for defining contracts that bind objects together; for defining rules and performance criteria for the interactions between those objects; and for monitoring and enforcing those rules and criteria.

The rest of this paper is structured as follows. Section 2 uses an example to summarise the concepts underlying the ROAD framework that are defined at greater length in [3]. Section 3 gives an overview of the separation of posterior organisational code from anterior functional code, and the separation of domain-specific concerns from general organisational constructs. Section 4 briefly describes how aspects, and in particular association-aspects, can be used to define organisational contracts. We give a very brief description of the concepts of aspect oriented programming (as implemented in AspectJ), and how association-aspects can be used to create a generalisation hierarchy of contracts from which contract instances are created. In Section 5 we show in more detail how to create both management and functional contracts using association aspects. Section 6 provide a model for self-managed composites and their organiser roles (that we defined at a conceptual level in [3]) and

shows how they can be created using the ROAD package. Section 7 discusses some related work and limitations of implementing contracts with aspects. Section 8 concludes with an overview and discussion of further work.

2 Overview of the ROAD framework

To help us discuss the ROAD implementation of a coordination system we will consider an example of a highly simplified business department that makes Widgets and employs Employees with different skills to make them. In such a business organisation an employee can perform a number of roles, sometimes simultaneously. Employees (objects) can perform the roles of Production manager, Foreman, ThingyMaker, DooverMaker and Assembler (who assembles thingies and doovers into widgets). The Foreman’s role is to supervise ThingyMakers, DooverMakers and Assemblers, and to allocate work to them. The WidgetDept Organiser role is responsible for creating the bindings between functional roles and the objects that play them. It creates contracts between the various roles, and then sets performance conditions for those contracts and monitors the conformance to the conditions. The organisation of our Widget Making department is illustrated in Figure 1.

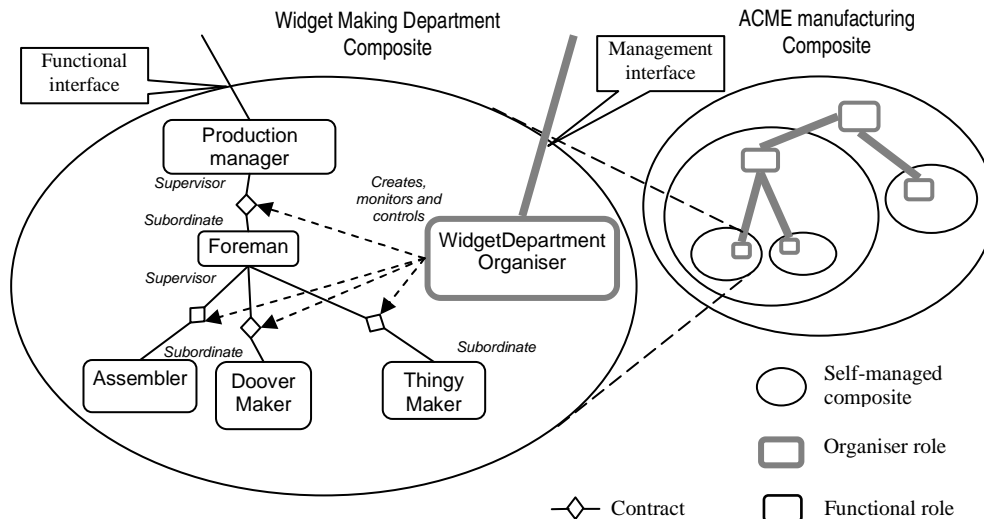


Fig. 1. Functional and Coordination Systems in a Self-managed Composite

In the Figure 1, an organiser-role (Widget-Department organiser) creates, monitors and controls the contracts between functional roles (Foreman, Assembler etc.). The domain of organiser-roles is the system itself rather than the problem-domain. Each organiser role is responsible for a cluster of functional roles called a “self-managed composite”. In terms of a management analogy, a self-managed composite in a business organisation would be a department (e.g. manufacturing department). Such managed composites perform a definable domain function, and can themselves be part of higher-level composites.

A role-based organisation is built from a recursive structure of self-managed composites. This structure is coordinated through a network that connects the organiser roles of each of the composites (the grey lines in Fig. 1). The network of organiser roles and the contracts they control constitute the *coordination-system*.

The organisational contracts in ROAD are more like contracts in the business world, than like contracts that are commonly implemented in software such as design-by-contract (DBC)[7]. ROAD contracts are independent, instantiable software entities that maintain the *state* of the association between two roles played by objects (parties to the contract), rather than being a mechanism for providing interface enforcement for a single class as in DBC; or just as a design concept. ROAD contracts also define rules for *all* interactions between the parties to the contract, rather than being focused on *one* type of interaction between many parties as in [1] and [11]. In ROAD contracts, individual interactions are represented by contract *clauses* that define the respective obligations of the parties during the interaction.

ROAD contracts exist at two levels of abstraction: the management level and functional level. The management-level defines what types of control communication are permissible between parties. Using our example of a Widget-Making department, the control aspects of the relationship between a Foreman and an Assembler could be characterised as a Supervisor-Subordinate relationship. Rules control the interactions between *operational-management* roles such as a supervisor and a subordinate. For example a supervisor can tell a subordinate to do a work related task but a subordinate *cannot* tell a supervisor what to do. These control communications in management contracts can be defined in terms of *control-communication act* (CCA) primitives. In [4], we have defined a simple set of CCAs for direct and indirect control and for information passing – DO, SET_GOAL, REQUEST_RESOURCE and so on. These performatives abstract the *control* aspects of the communication from the functional aspects. Management contracts are more extensively described in [4], and their implementation using abstract aspects is summarised in Section 5.1.

The functional-level contracts specify the requirements for domain-specific interactions. These requirements might include specific sequences of method invocations, and also allow non-functional requirements (NFRs) to be expressed as performance conditions of the contract. These conditions might be expressed in terms of acceptable time of execution, or be evaluated against some other utility function. For example, a time-based performance metric might specify the maximum time allowed for a contracted thingyMaker to produce a thingy. If there are costs associated with the performance of a function such as making a thingy, then the contract might specify the acceptable limit of those costs. These NFRs reside in, and are enforced by, the contract rather than the component itself.

3 Separating the implementation of organisation from function

Our implementation of the ROAD framework separates code into three (largely) independent parts: domain-specific functional code; domain-independent organisational code; and domain-specific organisational code. Figure 2 below illustrates these three parts:

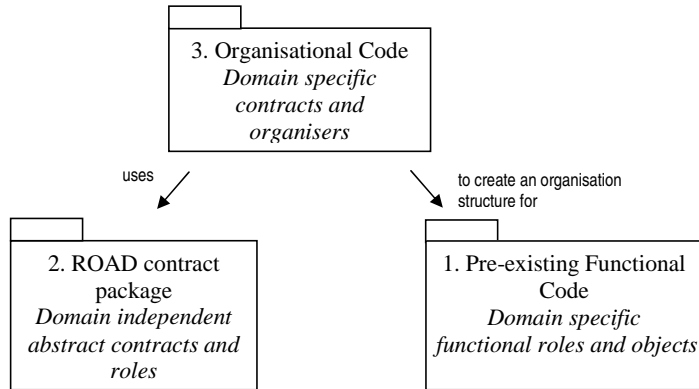


Fig. 2. Separation of Organisational code from Functional code

Domain-specific functional code. Classes that define the functional roles of the organisation and the objects that can play those roles can be defined separately from the organisational constructs. Classes representing these roles can be written without defining the configuration of organisation in which they will participate. These functional roles (or more properly *domain-function* roles) are focused on achieving/maintaining the first-order goals the system — on achieving the desired application-domain output. Functional roles constitute the *process* as opposed to the *control* of the system. In ROAD these functional roles are decoupled; they do not directly reference each other.

Domain-independent organisational code. A reusable library (see Figure 4 below) that specifies the general form of ROAD contracts is used. This library defines contracts at a management-level of abstraction. These contract types characterise the type of *control (or management) relationship* between the parties to the contract. For example, the acceptable forms of interaction between a supervisor and a subordinate are captured in a Supervisor-Subordinate relationship. A Peer-Peer management contract would specify a different set of permissible types of interaction. The library also defines abstract classes and interfaces for the different types of ROAD role (functional, operational-management and organiser).

Domain-dependent organisational code. The third ‘package’ of code defines the organisation of the domain-specific system. This coordination system is based on the abstract contract library. The programmer creates domain-specific functional contracts between the pre-existing functional roles. These functional contracts allow performance requirements to be specified

and enforced for interactions between the particular functional roles that are bound by the contract.

4 Association aspects

Aspect-oriented methods and languages seek to maintain the modularity of separate cross-cutting concerns in the design and source-code structures. Examples of cross-cutting concerns that have been modularised into aspects include security, logging, transaction management and the application of business rules. The AspectJ [5] extension to Java allows the programmer to define *pointcuts* that pick out certain *join points* (well-defined points in the program flow such as a *call* to method). An *advice* is code that is executed when a join point that matches a pointcut is reached. *Aspects* encapsulate such pointcuts and advices. These units of modularity can model various cross-cutting concerns.

While AspectJ-like aspects have previously been used to add role behaviour to a single object [6], as far as we are aware they have not been used to implement associations between roles. Aspects, as they are currently implemented in AspectJ, do not easily represent the behavioural associations between objects [10]. Current implementations of AspectJ provide *per-object* aspects. These can be used to associate a unique aspect instance to either the executing object (*perthis*) or the target object (*pertarget*). When an advice execution is triggered in an object, the system looks up the aspect instance associated with that object and executes that instance. This allows the aspect to maintain a unique state for each object, but not for associations of groups of objects.

Sakurai et al. [9] propose the use of *association-aspects* to allow an aspect instance to be associated with a group of objects. *Association-aspects* are implemented with a modification to the AspectJ compiler to handle an additional pointcut primitive. Association-aspects allow aspect *instances* to be created in the form

```
MyAssAspt a1 = new MyAssAspt (o1, o2, ... , oN);
```

where *a1* is an aspect instance and *o1* to *oN* are a tuple of two or more objects associated with that instance. Association-aspects are declared with a *perobjects* modifier that takes as an argument a tuple of the associated objects.

```
aspect MyAssAspt perobjects(o1, o2){
. ...//aspect variables, methods, pointcut declarations}
```

The ability to represent the associative state between objects in a group makes association-aspects suitable for representing contracts as we have defined them. Figure 3 below schematically shows how an instance of ROAD contract (*ft1* of type *FTContract*), implemented as an association-aspect, mediates the interaction between the two functional roles (*Foreman f* and *ThingyMaker t*). The contract intercepts method calls that match pointcuts defined in the contract aspect and that are between parties bound by the con-

tract. In the case below, calls from f to t that start with a method name prefix `do_*` are intercepted. Pointcuts can also be defined that prevent methods unauthorised by the contract (e.g. `wash_car()`) calls either between the parties to the contract, or from external entities - that is, any method call that are not specified in the contract.

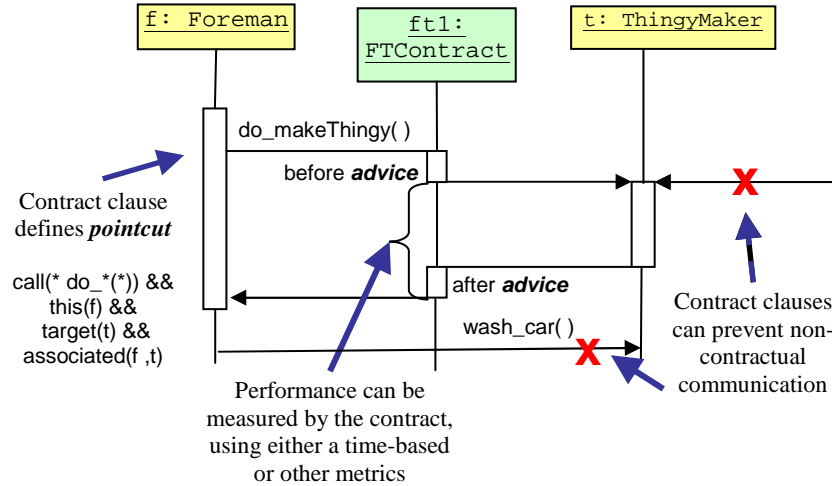


Fig. 3. Interaction between roles under contractual control

Advice in an aspect is code that is executed when a *pointcut* is reached in the execution flow. AspectJ supports a number of types of advice:

- Before (executed just before the *join point* is reached)
- After (after returning normally or after returning from an error)
- Around (allows alternate code to the invoked method to be executed instead of the invoked method)

Rules for communication can be applied in a *before* advice. As illustrated above, contracts make use of the change of state between *before* and *after* advice to measure performance of a contract clause. For example, the time elapsed between before and after advice can be used to calculate time-based metrics such as rate of production. Alternatively, some other utility function (such as cost) could be evaluated by accessing the execution context of the advice (either the state of the object or the environment).

The next section shows in more detail how association-aspects can be used to create contracts.

5 Management and Functional Contracts

As we have defined in [3], contracts are defined at two levels of abstraction – a management level that controls the *types* of interactions allowed between parties, and a functional level that allows us to define, monitor and enforce non-functional requirements (NFRs) or QoS characteristics of the interactions.

In [4] we have previously described the implementation of the management level of contracts using association contracts. In this section we will briefly summarise the implementation of management contracts, before introducing a model for the implementation of functional contracts and QoS performance criteria.

5.1 Management contracts

In brief, management contracts define what types of communication are permissible between parties. To create an operational-management contract type, such as a Supervisor-Subordinate, we need to define the types of parties that participate in the contract, and then determine which control-communication acts (CCAs) each of the parties can use. To do this we create three types of pointcuts: Party, CCA and Contract Clause pointcuts. Party and CCA pointcuts are composed into pointcuts that represent particular clauses in the management contract. These composite Contract Clause pointcuts define *who* can say *what*.

Party pointcuts match the direction of communication between the parties to the contract. For example, in a contract between two operational-management roles (of type `MRole`) there would be two party pointcuts: a `aToB` pointcut that represents communication from party A to party B, and a `bToA` pointcut that represents communication the other way. The definition in AspectJ is as follows:

```
pointcut aToB(MRole a,MRole b):associated(a,b)&& this(a)&& target(b);
```

The `associated(a,b)` condition is an AspectJ extension from [9]. In this case it ensures that the parties, represented by the particular `MRole` variables `a` and `b`, are associated in a contract. The `this(a)` condition ensures `a` is making the call. The `target(b)` condition ensures that `b` is the target of the communication.

CCA pointcuts use a mixture of primitive pointcuts provided by AspectJ and pattern matching on the method signatures to enforce the communication protocol between the functional roles. If the CCA types cannot be distinguished by primitive pointcuts alone, a naming-convention is required that identifies the method signature with particular CCAs in the contract. To achieve this in our example we define the convention that: ‘an abbreviation of the CCA prefixes the method’. For example, the name `do_makeWidget(..)` enables a mapping to be created between the functional method that orders a `Widget`, and the `DO` CCA primitive defined in the operational-management contract. The CCA pointcut `DO` could be defined as follows:

```
pointcut do_it() : call(* do_*(..));
```

This pointcut called `do_it()` matches any method *call* that begins with the characters “do_”; returns any type; and has any parameters.

Because CCA pointcuts define abstract types of control communication between the parties, these pointcuts are common to many types of management

contract. For this reason these pointcut patterns can be defined in the general form of contract. This has been implemented in the `MContract` abstract association-aspects illustrated in Figure 4 below.

Contract clause pointcuts are the combination of a Party pointcut and a CCA pointcut. For example, the Clause `a1` of the contract can be represented by the pointcut below. It says that Supervisor `sup` has the authority to tell the Subordinate `sub` to do something.

```
pointcut a1(Supervisor sup,Subordinate sub): aToB(sup,sub)&&do_it();
```

In a management contract, a clause is defined for every CCA that can be initiated by either party. Contract clauses are specific to types of management contract such as the Supervisor-Subordinate contract (the `SuperSub` aspect in Figure 4 below).

Management contracts are enforced by checking in a *before advice* that the method invocations between the parties match appropriate CCAs. If they do not, an `InvalidCCAException` is thrown. For example, in the code snippet below, the composite pointcut `a0` defines all valid CCAs a supervisor can invoke. If the method is invoked by the supervisor (`aToB(a, b)`), and is not one of these CCAs (`!a0()`), then an exception is thrown. The `thisJoinPoint` is a special variable provided by AspectJ that provides reflective information about the execution context of the advice. In the example below, the identity of the parties to the contract and the signature of the invalid method is exposed.

```
before(Supervisor a, Subordinate b): !a0() && aToB(a,b){...
    throw new InvalidCCAException(s);}
```

Figure 4 below expands the three ‘packages’ schematised in Figure 2 above: domain-dependent functional code; domain-independent organisational code; and domain-dependent organisational code. Because the abstract management contracts and roles described in this section are domain-independent, they provide the basis for a contract framework that has the potential to be reused in a number of domains.

Contracts are implemented as an inheritance hierarchy of aspects (the shaded classes in Figure 4 above). The most general form of contract in this library (the `MContract` aspect) specifies general *patterns* (CCAs and protocols) that allow the control-aspects of communication between the parties to be monitored and enforced by sub-aspects. The `MContract` aspect also provides mechanisms by which parties and clauses can be added to (and removed from) instances of contracts. Abstract management contracts (such as Supervisor-Subordinate, Peer-Peer, Auditor-Auditee etc.) inherit these common CCA patterns and functionalities from `MContract`. Such management contracts express various types of control relationships between roles by defining *contract clause pointcuts* as described above.

In order for the functional-role classes to be able to work with management contracts, they need to implement the empty interfaces that represent any applicable operational-management roles such as `Supervisor` or `Subordinate`.

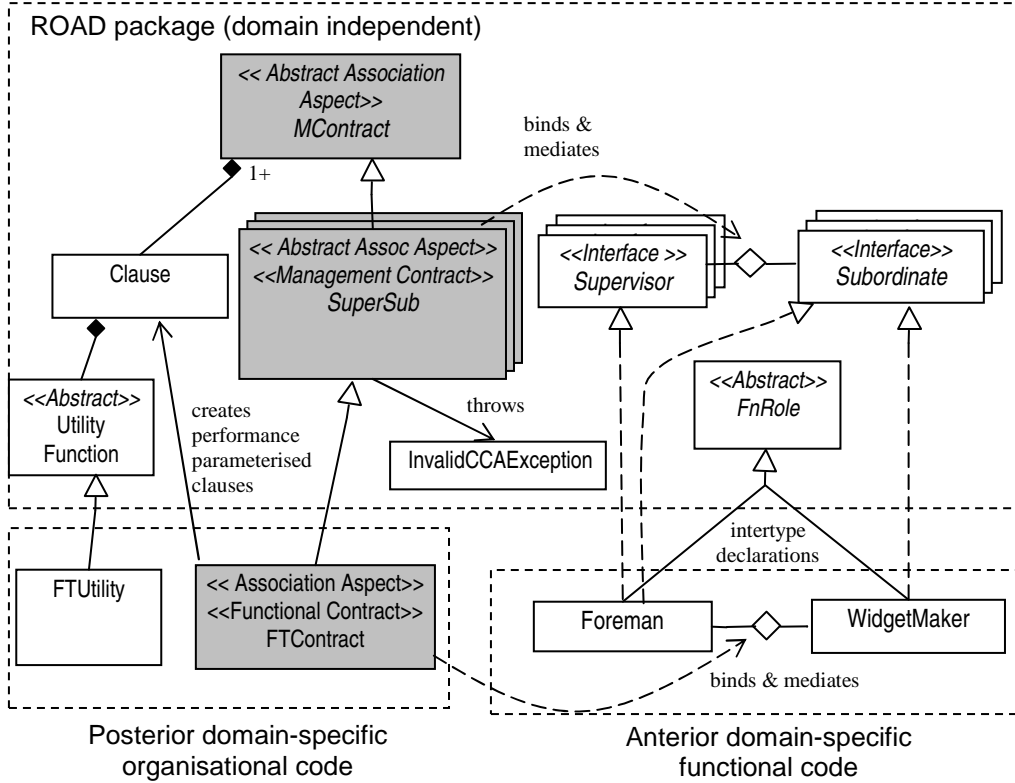


Fig. 4. Classes and aspects that define management and functional ROAD contracts (Contracts implemented as aspects are shaded)

They also need to extend the abstract functional role class `FnRole`. This abstract class provides some methods that are common to functional roles, such as keeping track of which contracts the role is party to. The creation of these dependencies does not require the alteration of the pre-existing functional classes but can be achieved by using a static aspect with *inter-type declarations*. Such declarations can create, at compile-time, the inheritance and interface relationship to functional roles. For instance the following line creates the inheritance relationships for the `ThingyMaker` and `DooverMaker` classes:

```
declare parents:(ThingyMaker||DooverMaker) extends FnRole implements Subordinate;
```

In the next subsection we will introduce the *domain-specific organisational* code that inherits from the domain-independent ROAD package, and that is superimposed on top of the pre-existing functional code.

5.2 Functional Contracts and Performance

A functional contract is a concrete sub-class (sub-aspect) of a management-contract aspect. It maintains references to the participating parties and includes methods for assigning the contract, revoking, and preventing the duplication of contracts between objects. For example, the code below shows the

declaration of a functional contract type between a Foreman and a Thingy-Maker (`FTContract`):

```
public aspect FTContract extends SuperSub perobjects(Supervisor, Subordinate)
```

Functional contracts, such as the `FTContract` aspect in Figure 4 above, allow us to define performance characteristics for each clause of a contract object. As currently implemented, all clauses are of the same type. Domain-specific characteristics are passed as parameters to the `Clause` class constructor (below) when the clause is created. These parameters include a reference to the contract that the clause belongs to, the method signature, the direction of the invocation (`aToB` or `bToA`), and a domain-specific utility function object that defines the performance metrics of the clause.

```
public Clause(MContract c, String sig, int dir, UtilityFunction util){...}
```

The abstract `UtilityFunction` class has a `calculatePerformance()` method that can be overridden by a concrete domain-specific utility sub-class (such as `FTUtility` in Figure 4 above). This `calculatePerformance()` method is invoked by the *after* advice of the contract and used to calculate the performance of the clause. The `calculatePerformance()` method takes as a parameter the timing information on when the *before* and *after* advices are executed. In addition, a reference to the execution context (the `thisJoinPoint` variable of the advice) is passed as a parameter. Once the performance of a clause is calculated, its state will be reported to the contract, if it is in breach or is underperforming. The contract, in turn, notifies any underperformance to the organiser that manages the contract. The implementation of organisers, and the self-managed composites they manage, is discussed in the next section.

6 Self-managed Composites and Organiser Roles

A self-managed composite is a type of functional role that contains *contracts* and an *organiser* role. The organiser role manages the contracts in the composite. From the enclosing system's perspective, a self-managed composite looks like a functional role with a management interface. It can be bound in contracts in the same way as a standard functional role. The classes and aspects involved in the self-managed composite and organiser are shown in Figure 5 below. This arrangement allows the development of a recursive structure of self-managed composites that contain composites (as illustrated in the schema of our manufacturing company in Figure 1). A composite contains contracts which bind roles. These roles may, themselves, be composites.

We refer to these entities as composites rather than components because the roles, and the objects playing those roles, are not *necessarily* encapsulated in a package, nor necessarily exclusively used by the composite.

Self-managed composites have two interfaces – a *functional* interface and a *management* interface. The *functional* interface of a composite is the aggregation of all the methods of its member roles that interact with roles external to the composite. These roles are stereotyped as *delegates* because the calls

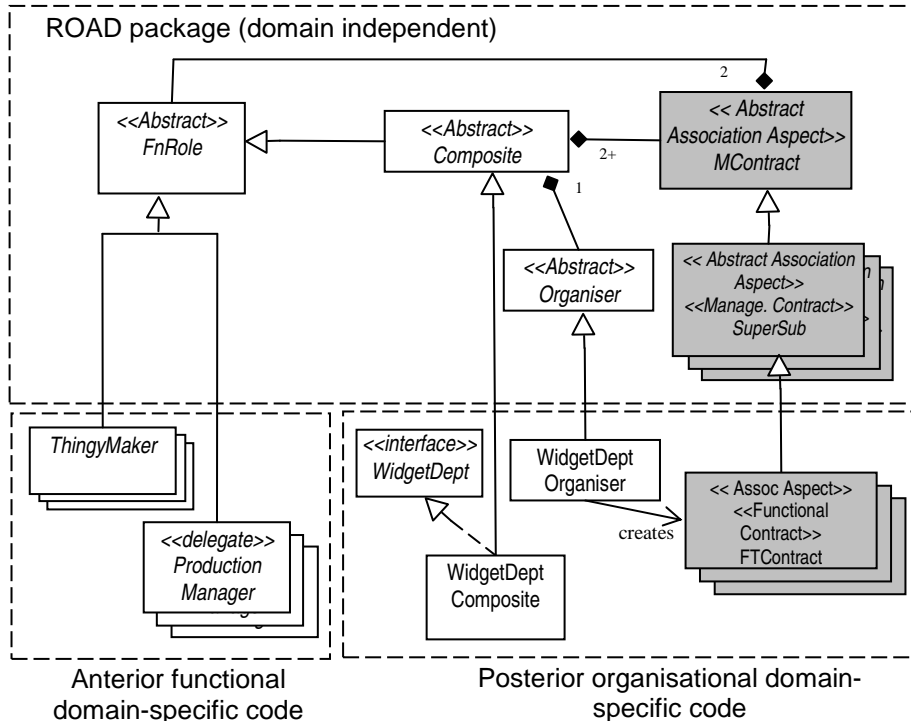


Fig. 5. Implementation of Widget Department Self-managed Composite

to or from the composite are delegated to these roles. In other words, the composite performs no domain function in itself, but relies on its nested functional roles to perform tasks. In our Widget Department example in Figure 5 above, the only functional role with external associations is the Production Manager role. All i/o of the Widget Department composite, such as an order for Widgets, is delegated to this Production Manager.

As shown in Figure 5 above, each Composite has an Organiser. The interface of the *organiser* role is the *management* interface of the composite. The organiser role is responsible for monitoring the performance of the composite's contracts. It also reconfigures the composite if environmental perturbation, or a change of requirements, leads to the composite not meeting (or potentially not meeting) its contractual performance obligations. The organiser can achieve reconfiguration by assigning and revoking contracts, and by changing the binding between functional roles and potential role-players. For an organiser to be able to respond adaptively to changing situations, it needs to have available at its disposal a range of role/role-players of various performance characteristics. The mechanisms for achieving such an open system 'service discovery' will vary with the type of domain. The discussion of such mechanisms is beyond the scope of this paper.

An organiser can create a contract simply by instantiating a functional contract, with the functional roles that are to be parties to the contract, passed as parameters to the aspect constructor.

```

Foreman f = new Foreman();
ThingyMaker w = new ThingyMaker();
FTContract ft1 = new FTContract(this, f, t);

```

The organiser also passes a reference of itself to the contract, so that the contract can notify it of any underperformance as defined by the contract clause’s utility function objects.

The management interface of a composite is the external interface of its organiser role. Required and actual performance measures, as defined by the various utility function objects, pass back and forth over this link. If this organiser has difficulty organising the composite to meet the performance targets set by the enclosing composite, it escalates the warning up the chain of the coordination system hierarchy. Such escalation can be viewed as an organised form of exception-handling where performance messages and warnings flow through the *coordination*-system. These messages allow organisers, at the appropriate level, to reconfigure the organisational structure before error-messages propagate through the *functional*-system’s (sometimes ill-defined) exception-handling structure.

7 Discussion and Related Work

A couple of limitations in the current implementation of the association-aspect compiler [9] have become apparent during implementation. Firstly, different pointcuts within the same aspect generalisation hierarchy (such those in Figure 4) cannot match the same join point. This limitation prevents the specialisation of contract clauses using pointcuts: in other words, a functional contract cannot add extra advice to that already defined in the management contract by means of defining its own pointcuts. Instead the advice in management contract must invoke abstract methods that are over-ridden in the functional contract. The second limitation of association-aspects is that all pointcuts in an association-aspect must bind all objects in the association. It follows that to restrict interaction between a functional role and other types of object (as illustrated in Figure 3 above), we need to define a standard (non-association) aspect that prevents this type of communication. This is easy to do but, like the previous limitation, lacks conceptual clarity because a single contract type should be able to define all its terms.

Work related the concepts behind the ROAD framework can be found in [3,4] and is not repeated here. The coordination model outlined here adopts a control-oriented [2] architectural approach, primarily focused on adaptivity rather than synchronisation. ROAD has many similarities and some major differences with work by Andrade, Wermelinger and colleagues [1,11]. Both approaches represent contracts as first-class entities, and both use a layered architecture. In [1,11] the layers are Computation, Coordination and Configuration (‘3C’). This is broadly similar to ROAD’s four layer architecture (Computational-object, Functional-role, Management-contract, Organisation)

with 3C's Computation layer similar to ROAD's Object and Functional role layers. 3C's contracts are method-centric rather than role-association-centric. They define a single interaction sequence that might involve many parties, whereas ROAD contracts are currently limited to two roles and involve many types of interaction. Both approaches use contracts to model unstable aspects of the system, but 3C's focus is on business rules whereas ROAD focuses on performance variability. In 3C, there is no concept of a coordination network through which regulatory control messages pass.

[8] and [9] propose different solutions to modelling the behaviour between groups of objects. The former's AOP language Eos aspects can be created to represent behavioural relationships, however it selects advice execution associated with a target object. Sakurai [9], on the other hand, modifies the AspectJ compiler to handle the additional *associated* pointcut primitive. We have used the latter approach because it allows aspect instances to be created independent of the objects in the association.

8 Conclusion

In this paper we have shown how association-aspects can be used to create an organisational structure with a coordination-system that controls the interactions between functional roles. These coordination systems can be developed independently and imposed on functional systems. They are built from a hierarchy of organiser roles that control the contracts between functional roles. ROAD contracts have management and domain function levels. Management contracts specify the type of communication acts and protocols that are permissible between the two parties. Functional contracts specify, among other things, the performance obligations. Abstracting management-contract aspects makes possible, through contract inheritance, the reuse of their communication-control capability in many types of organisational structure.

In [3] we have raised a number of open questions and further work to be done at a conceptual level, whereas here we will limit the comments here to implementation issues. In terms of implementation, we have shown how association-aspects can be used to implement contracts that define and control the interaction between roles in an organisational structure, and how this structure can be superimposed after the functional code has been written. The caveat to this independence is the requirement that method signatures in the functional code follow arbitrary code conventions defined by the CCAs (e.g. `do_makeWidget(...)`), so that pointcuts can be pattern-matched according to the type of interaction.

A demonstration system using association-aspects has been implemented. The code has been written in the three packages described above. The functional code package can be compiled either with or without the organisational code. If compiled with the organisational code, the functional role interactions

are placed under the control of contracts that can be created, revoked and re-assigned by organisers at run-time. Time-based utility functions that measure the performance of contract clauses have also been implemented. Work is currently being undertaken on extending the system to demonstrate adaptive restructuring to response to simulated load changes.

The impact on system performance of superimposing a coordination system on the functional system also needs to be investigated. While Sakurai [9] has shown that there is little additional overhead in the use of *association-aspects* viz. ordinary AspectJ aspects, it remains to be seen whether a coordination system based on association-aspects would impose a significant run-time penalty — particularly if complex utility functions are introduced and need to be calculated with each *after* advice. Further work also needs doing on elaborating and implementing suitable performance measures.

References

- [1] Andrade, L., Fiadeiro, J. L., Gouveia, J., and Koutsoukos, G., “Separating computation, coordination and configuration” *Journal of Software Maintenance and Evolution: Research and Practice* , vol.14(5), 2002, pp. 353-369.
- [2] Arbab, F., “What Do You Mean, Coordination?” *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, March, 1998
- [3] Colman, A. and Han, J., “Coordination Systems in Role-based Adaptive Software,” *Proc. Coordination 2005 (Coord’05)*, Lecture Notes in Computer Science, Vol. 3454, 2005, pp. 63-78.
- [4] Colman, A. and Han, J., “Operational management contracts for adaptive software organisation,” *Proc. Australian Software Engineering Conf. (ASWEC 2005)*, 2005, pp.170-179
- [5] Eclipse Foundation, *AspectJ* <http://eclipse.org/aspectj/>, 2004.
- [6] Kendall, E. A., “Role Modelling for Agents System Analysis, Design and Implementation” *1st International Symp. on Agent Systems and Applications IEEE CS Press*, 1999
- [7] Meyer, B. “Object-oriented software construction”, New York: Prentice-Hall, 1988.
- [8] Rajan, H. and Sullivan, K., “Eos:instance-level aspects for integrated system design” *ACM SIGSOFT Software Engineering Notes* , vol.28 (5), 2003, pp. 297-306 .
- [9] Sakurai, K., Masuharat, H., Ubayashi, N., Matsuura, S., and Komiya, S., “Association Aspects,” *Proc. Aspect-Oriented Software Development ’04*, Lancaster U.K, 2004.

- [10] Sullivan, K., Gu, L., and Cai, Y., “Non-modularity in aspect-oriented languages: integration as a crosscutting concern for *AspectJ*,” *Proc. 1st Inter. Conf. on Aspect-oriented software development, AOSD 02*, 2002.
- [11] Wermelinger, M., Fiadeiro, J. L., Andrade, L., Koutsoukos, G., and Gouveia, J., “Separation of Core Concerns: Computation, Coordination, and Configuration,” *Proc. of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.