

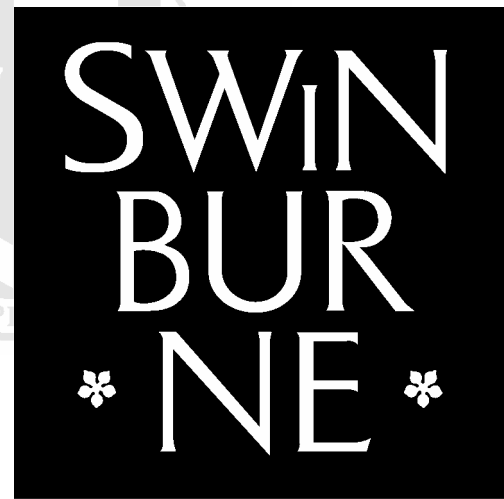
School of Information Technology  
Centre for Component Software and Enterprise Systems

Technical Report No: **SUTIT-TR2004.06/SUT.CeCSES-TR006**

# Operational management contracts for adaptive software organisation

Alan Colman and Jun Han  
{acolman,jhan}@it.swin.edu.au

22 October 2004



SWINBURNE UNIVERSITY  
OF TECHNOLOGY

# Contents

1.	Introduction.....	3
1.1.	Example .....	3
1.2.	Structure of this report .....	4
2.	Overview of the ROAD framework.....	4
3.	Control in a management network.....	6
4.	Operational-management role associations and contracts .....	7
4.1.	Control-Communication Acts .....	9
4.2.	Operational-management contracts.....	9
5.	Using Association-Aspects to implement operational-management contracts.....	11
5.1.	Defining the contract elements.....	13
5.2.	Defining the clauses of the contract.....	13
5.3.	Defining the effect of the contract clauses .....	14
5.4.	Putting the contract together .....	14
5.5.	Creating the contract instance between functional roles .....	16
6.	Related work .....	16
7.	Conclusion and further work .....	17
8.	References.....	18
9.	Appendix – Example Code .....	19
9.1.	Overview .....	19
9.2.	Test Program and Output .....	20
9.3.	mContracts package code.....	23
9.4.	Functional Role Code.....	28

*This paper is an extended version of:*

Colman A and Han, J. 'Organisational Operational management contracts for adaptive software organisation' to be presented at the Australian Software Engineering Conference – ASWEC 2005

# Abstract

As modern computing environments become more open, distributed and pervasive, the software we build for those dynamic environments will need to become more adaptable and adaptive. We have previously introduced the ROAD framework for creating flexible and adaptive software structures. This framework is built on a distinction between functional and management roles. Management roles participate in contracts that regulate the global-flow of control through a structure of objects and roles. This paper shows how these operational-management contracts can be defined. Such contracts specify the permissible interactions between objects playing functional roles within an organisational structure. Association aspects are shown to have the expressiveness needed to represent such management contracts.

## 1. Introduction

As modern computing environments become more open, distributed and pervasive, the software we build for those dynamic environments will need to become more adaptable and adaptive. *Organisational* viability of software is required to make software adaptable and adaptive in changing environments [2]. In order to achieve organisational viability we need to represent organisational aspects of software. This paper addresses how software organisation might be represented, at both design and code levels, so that it is amenable to adaptation. This is a prerequisite if we are to create viable adaptive software systems whose organisational representation can be manipulated.

One way to create adaptable software is to create a loosely coupled structure and to create the relationships between the nodes in that structure as late as possible. The relationships in the structure are regulated according to the changing environmental demands. In this approach, software organisation can be viewed as the maintenance of viable arrangements of elements and the regulation of the flow of control through the structure.

This paper extends our work in [2] on the ROAD framework which models software as a decoupled network of roles and objects. The management of the software is seen as a ‘separate concern’ from the functional aspects of the software. In particular, we show how *management contracts* can be used to regulate the flow of control through a network of roles. The form of such contracts is defined, and we demonstrate how *association aspects* [15] can be used to implement them.

### 1.1. Example

Let us consider an example to illustrate how organisational abstractions can be modelled. This example models 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 varied roles — sometimes simultaneously.

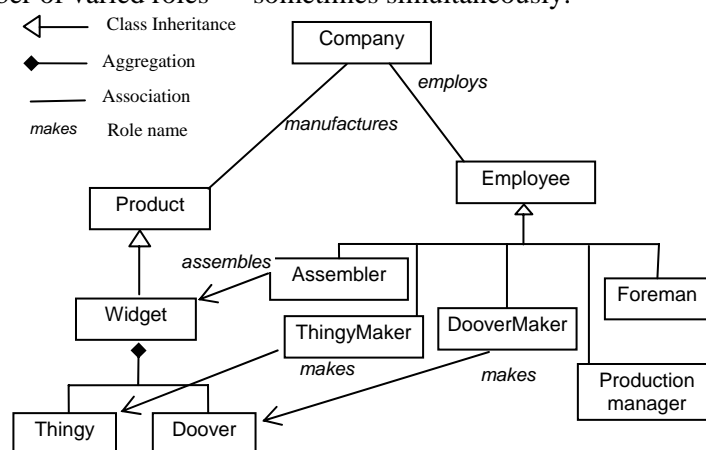


Figure 1. Traditional Object-oriented Class Model

In a traditional class model, as in Figure 1 above, the associations between classes are fixed at design-time in method invocations and inheritance relationships. The associations between classes/objects cannot be dynamically created or richly described. For example, what *types* of interaction are permissible between an Assembler and a Foreman, and do these interactions differ from those types of interaction between an Assembler and a ThingyMaker? Can an Assembler tell a Foreman what to do by invoking its methods? In object-oriented design, there is no organisational level description in terms of the control of the system. The global flow of control through the structure cannot be represented. Only particular sequences of specific interactions can be shown (e.g. in sequence diagrams). Finally, in traditional object-oriented design, roles are implicit to the objects that play them. There can be no dynamic adaptation of the structure of the relationships between objects and roles in response to changing demands on the system.

We will rework the above example to show how management contracts can be created that define and regulate associations between roles in a decoupled structure.

## 1.2. *Structure of this report*

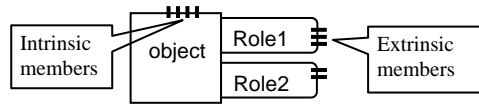
Section 2 gives an overview of the ROAD framework which provides the context for the discussion of role contracts. The model of the network formed by these management contracts is an organisational description of the software system based on the flow of control. Section 3 characterises the different types of control in such a network according to whether it is direct or indirect, and according to the scope of control. In Section 4 we use this characterisation of control to examine in more detail operational-management roles, association and contracts. Operational-management roles form associations. We will define these associations in terms of *control-communication acts* (CCAs) and demonstrate how to formalise them into contracts. We examine the nature of such contracts then define the expressive requirements needed to represent them. Section 5 provides a brief overview of *association aspects*. We then show how association aspects meet the expressive requirements needed to represent operational-management contracts defined in the previous section. Section 6 examines related work and Section 7 draws conclusions and outlines further work.

## 2. Overview of the ROAD framework

The Role-Oriented Adaptive Design (ROAD) framework is made up of a number of parts. A meta-model (ROADsign) that defines the modelling elements the extends object-oriented/UML meta-model with different types of functional and management role; a development methodology (ROADway) that shows how to decouple the object-role structure to make it more flexible and how to create the structure for organisational control; and a mechanism for implementation (ROADworks) that facilitates the modularisation of roles at the code level. This framework should not be confused with the Roadmap [6] agent-oriented methodology.

ROAD is a method for creating adaptable and adaptive object-role software structures. The ROAD framework extends work on role and associative modelling in [1,8-10]. In this section we give a brief contextual overview of our ROAD framework. A more extensive description of the basis of this framework can be found in [2].

A role is an interface of an object that satisfies responsibilities to the system as a whole. Roles can be added to, and removed from, objects. Kristensen [9] provides a definition of roles that is based on the distinction between intrinsic and extrinsic members (methods and data) of an object. Intrinsic members provide the core functionality of the object, while extrinsic members contain the functionality of the role.



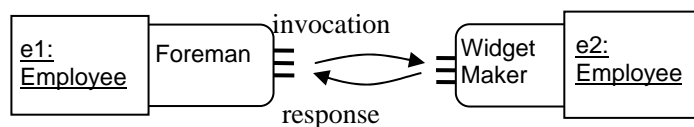
**Figure 2. Object and role members**

In our view, this ‘core functionality’ is the situated computational and communication capabilities of the object. Extrinsic members implement the domain function roles of the object.

Returning to our example in the Introduction, rather than, for example, model a Foreman as a subclass of Employee, Foreman becomes a role an Employee can play. The static inheritance relationships with the Employee class would be removed. These are replaced by potential role-object bindings. Note that Widget would not be treated as a role of Product because in the problem domain Products cannot change roles.

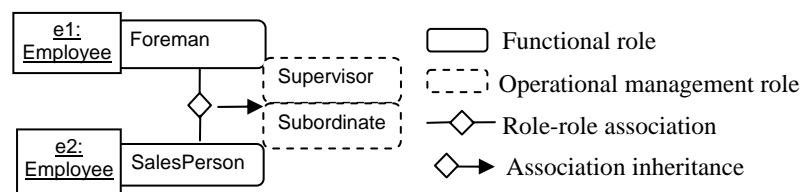
From the basis of decoupled class-role structures, ROAD defines organisational levels of abstraction. The ROAD framework extends previous work on roles by making an explicit distinction between functional and management roles. Three types of role are defined: functional, operational-management and organisational- management roles.

*Functional roles* are focused on first-order goals — on achieving the desired problem-domain output. Functional roles constitute the process as opposed to the control of the system. Some functional roles are coupled to the environment through system i/o. In Figure 3 below, the Employee e1 playing the role of Foreman can invoke action (e.g. ‘make 10 widgets’) in the WidgetMaker role played by the e2 Employee object. The discussion of the binding between functional roles and objects is outside the scope of this paper.



**Figure 3. Association between functional roles**

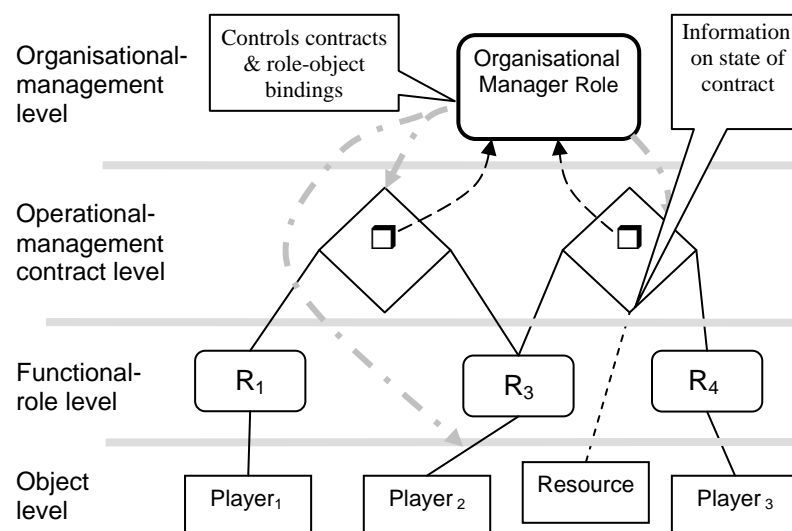
*Operational-management roles*, on the other hand, focus on regulating the relationships between functional roles. They define contracts between functional roles based on a separation of management control from process. Operational management roles have no direct connection with the environment. Extending the example from Figure 3 above, we can characterise the *management* relationship between a Foreman and a WidgetMaker as a Supervisor-Subordinate relationship — the Foreman in the operational-management Supervisor role and the WidgetMaker in the operational-management Subordinate role. The relationship between objects, functional roles, and operational-management roles is illustrated in Figure 4 below.



**Figure 4 Operational-management roles**

A network of operational-management roles represents the global flow of control through the system (as opposed to data flow) — the *organisation* of the software. In this paper, we are focussing on these *operational-management* roles.

*Organisational-management roles* maintain a reflective representation of the system's organisation and mechanisms for restructuring the organisation by creating/destroying role-object bindings and dynamic role-role associations. The controllers (objects/ agents/ humans) that play organisational management roles are responsible for deciding what objects will play the various functional roles, and for the restructuring of the network of operational-management roles. These controllers are linked to the environment and monitor the performance of the software system in terms of its goals. This forms an *adaptive loop*. The discussion of organisational management roles is beyond the scope of this paper.



**Figure 5. Associations in levels of abstraction in ROAD.**

Figure 5 above illustrates the relationships between the different types of roles in the ROAD framework. In summary, the ROAD framework achieves adaptivity through creating decoupled object-role structures. The roles in this initial structure are *functional roles*. A cross-cutting management role-structure is then created from *operational-management* roles. Linking these management roles forms an abstract organisational structure that represents the topology of the organisation and global control flow based on permissible role interactions. The functional roles are then bound to the network of operational-management roles. This binding creates a *domain-specific* organisational structure. An *instantiated* organisational structure can then be created by binding functional roles to objects. Such a structure is adaptable. Run-time adaptivity is achieved by defining *organisational-management roles*. These roles control and maintain the organisation by creating and destroying object-role bindings and dynamic role-role associations.

In this paper, we focus on the characterisation of *operational-management roles* and their association contracts within this framework. We show how these roles can be implemented as *aspects* that *cross-cut* functional roles and objects.

### 3. Control in a management network

We define organisation as the *global flow of control* through a system. Unlike natural systems, where organisation is emergent, software systems are designed to achieve goals. A shortcoming of many supposed organisational descriptions is that they reduce the description of organisation to just the topological structure. In our definition, organisational descriptions

of designed systems are means-end functional descriptions. A complete organisational description would need to indicate how goals are transmitted through the system, how the system changes in response to changing goals and environmental perturbations, and how the system maintains its organisational viability. Such organisational descriptions can be based on the conceptual separation of control from process — the separation of management of the process from the process itself. An organisational/managerial aspect of a structure facilitates the intentional flow of control through a structure of management roles, whereas relations between functional roles define the dataflow through the structure. Management functions can be, to some extent, characterized in a domain-independent way. These functions include coordination, goal-transmission, regulation, accounting, resource-allocation, auditing and reporting. The organisational perspective is one of a number of possible perspectives, but it is a perspective that allows us to explicitly represent and incorporate adaptive mechanisms into a system.

Control in an organisation can be characterised as direct or indirect. *Direct* control is concerned with positive goal propagation through the structure. In a hierarchical structure, direct control is a chain of command. Each node in the structure reinterprets the goal(s) passed down to it. The node then operationalises the goal(s) either by executing a process itself or setting goals for other roles. *Indirect* control is control through constraint – for example the regulation of a process through the allocation of resources to roles performing the process. Resources can be thought of as objects that do not perform management functions within the organisational structure. The scope of what is controlled can be either individual components or parameters that are system/subsystem wide. The table below illustrates various mechanisms for control categorised by *scope* of control and *type* of control.

**Table 1. Control type versus scope of control and example mechanisms of control**

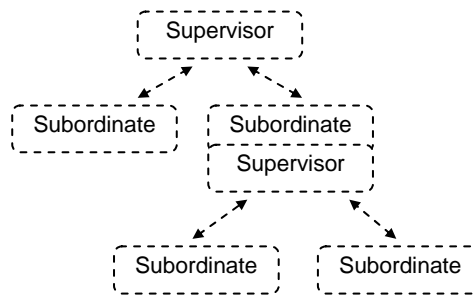
Scope \ Control type	Individual component	(Sub)system
Direct control	Command / goal setting	Prescriptive rules
Indirect control	Resource allocation Individual policies	Proscriptive rules Group policies Norms

In many systems, a combination of these modes of organisation will be present. In this paper we will focus on direct and indirect control of individual components (the shaded area in the above table) through the assignment of organisational responsibilities and the creating of structure for management control.

In the next section we characterise such operational-management associations and contracts in more detail.

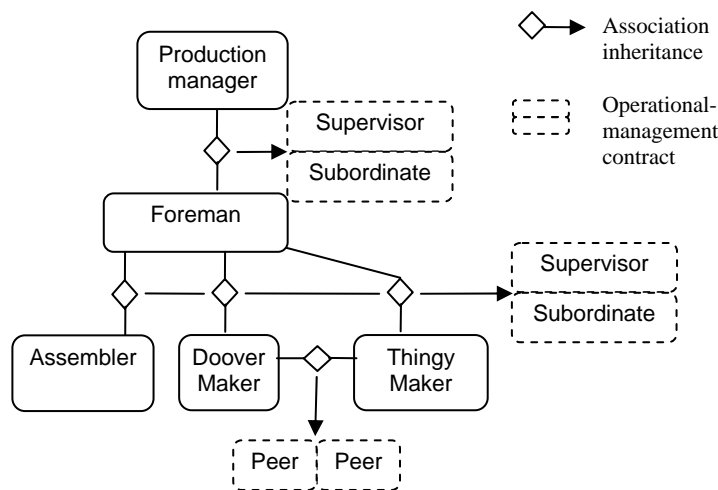
## 4. Operational-management role associations and contracts

The separation of operational-management roles from functional roles gives us a way to describe the organisational topology of the system and the control regime of that structure. Management hierarchies of any complexity, such as that shown in Figure 6 below, can be created with such operational-management roles. The static representation of such a hierarchy would have the appearance of a business's organisational chart.



**Figure 6. Abstract organisation structure of operational-management roles**

A domain-specific organisation is created when functional-roles are bound to operational-management roles. In Figure 7 below, an organisational structure has been created for our Widget department using Supervisor-Subordinate operational-management role associations. The structure is still abstract because no objects have yet been assigned to roles.



**Figure 7. Domain specific abstract organisational structure**

Every functional role in the organisation has a position in the control structure and thus has one or more associated operational-management roles — one for every type of role-role association in which the functional role participates. In other words, there is a correspondence between functional role-role associations and operational-management role-role associations. We call this binding *association inheritance*. Such role-role associations can be viewed as *contracts* that restrict the interactions between objects playing roles. In object-oriented languages such as Java, a target object will respond to any valid invocations of its public methods from objects that have the target in their scope. The scoping of accessibility of such methods can only be structured in a primitive way using accessibility modifiers, program blocks, packages, namespaces etc.

Operational-management role-role contracts *restrict* the type of method that one role can invoke in another role, or restrict what methods it will respond to from another role. From our example above, the Supervisor-Subordinate operational-management role association restricts interactions between the objects playing the WidgetMaker and the Foreman to certain *types* of interaction. For example, a WidgetMaker cannot tell its Foreman Supervisor what to do. These contracts also restrict interaction between particular instances of object playing the roles. For example, the method `WidgetMaker.setProductionTarget()` can only be invoked by the WidgetMaker's own Foreman.

The nodes in a management network are *operational-management roles* referred to above. These roles are always defined in terms of control association with another node in the network. Types of control *association* could include:

- Supervisor-subordinate
- Auditor-auditee
- Peer-peer
- Supply-chain predecessor-successor
- Production-line predecessor-successor

#### 4.1. Control-Communication Acts

We characterize the types of operational-management associations in terms of the control communication between roles in the associations. Such control communication can be defined in terms of *control-communication act* (CCA) primitives. These performatives abstract the control aspects of the communication from the functional aspects. We can define a simple set of CCAs in terms of the direct/indirect control distinction made in the previous section. *Direct control* is direct invocation of action in another role (command) or the setting of a goal state in another role. *Indirect control* is achieved through the allocation or restriction of access to resources. As such indirect control is inherently referential as it involves three entities - the role granting access to the resource, the role consuming the resource, and the resource itself. We assume resources are passive objects – that is, they do not know which roles are able to access them. Indirect control information therefore passes between the controller and the consumer of the resource, rather than to the resource itself. If access to the resource were to be set by the resource itself, it would need to be represented by a node in the operational-management role network.

In addition to direct and indirect control, *control information* needs to be passed between management nodes. This includes responses to commands or requests – such as *accept*, *refuse* etc. It could also include other information relevant to the regulation of the system - e.g. *busy*, *off-line* etc.

**Table 2. Example of Control-Communication Act Primitives**

Type of communication	Communicative control acts
Direct Control	DO, SET_GOAL
Indirect Control	RESOURCE_ALLOC( <i>r</i> ), RESOURCE_REQUEST( <i>r</i> )
Information	ACCEPT, REFUSE, INFORM, QUERY

As an example, Table 2 above defines a set of primitives suitable to a hierarchical organisation. Note that because indirect CCAs express a ternary relationship, they carry a reference to a resource *r*. The above set is not logically complete. For instance it does not capture a referential command relationship (A tells B to tell C to do something), but it is sufficient to allow us to define a number of contracts between operational-management roles. From these contracts we can create organisational structures.

#### 4.2. Operational-management contracts

The concept of a contract is commonly used in software engineering. For example, design-by-contract [11] defines the preconditions, post-conditions and invariants that must hold for a given type of interaction with an object. Such contracts are essentially one-sided in that they only express the conditions that constrain one party. In the real world however, contracts always have at least two parties. They are a type of association that expresses the obligations and responsibilities of parties to the each other. Contracts can be unique (e.g. a contract to build an opera house) or follow a standardized type (e.g. contract for sale for a residence).

Contracts can take a number of incarnations: *form* (à la class), *instantiation* (à la object) and *execution*. The *form* (type) of a contract sets out the mutual obligations and interactions between parties of a particular class (e.g. vendor and purchaser). A contract is *instantiated*

with an identity when values are put against the variables in the contract *schedule* (e.g. vendor and purchaser are named, date of agreed commencement etc.) and the contract is signed. Contracts can also be thought of as having an *execution state* in terms of the fulfilment of the various clauses of the contract.

Operational-management contracts are examples of such associational contracts – they define the form of an ongoing control association between two roles in an abstract organisational structure. The *form* of such contracts contains:

1. Variables defining the parties to the contract. These variables are of a particular type of participant that can enter into the contract.
2. A protocol that defines the allowable *types* of interaction between those parties. In operational-management contracts, these protocols are described in terms of the control relationships between the parties (e.g. A has the power to tell B what to do) rather than the functional relationships (A tells B to do a particular action). In terms of software, the protocols define the allowable types of method invocation one type of operational-management role (e.g. supervisor) can make on another (e.g. subordinate) and the expected response.
3. Other clauses in the form of contract define variables that relate to the execution of the contract. These include conditions relating to commencement, continuation, performance and termination of the contract. Operational-management contracts are on-going in that they define the control relationships between the parties whilst there is an organisational relationship between the parties. In this sense they more like an employment contract than a contract of sale. In a commercial contract such variables are part of the contract *schedule*.

An *instance* of an operational-management contract is created when the variables in the contract schedule are given values — in particular when operational-management roles are bound to functional roles. For example a Supervisor role is bound to a Foreman role. The rules for communication that are defined in the contract protocol are mapped to the method invocations in the functional roles. Performance criteria can also be attached to the execution of various clauses in the contract or to the contract as a whole.

Information on contract *execution* needs to be stored, along with the static information described above. This dynamic information is needed to ensure that the terms of the contract are being met, and includes information on the state of the relationship between the parties (e.g. active, inactive, suspended, in-breach, terminated etc.), and the state of any interaction defined by the protocols (e.g. A has sent a Query to B and is waiting for a response). In the real world, commercial contracts are just passive artefacts so this information is maintained by the parties themselves (or their agents). In software contracts, it makes sense to store this dynamic information in the contract itself.

Using the primitives we defined in the previous section, a Supervisor-Subordinate contract could be defined as in Table 3 below. When a functional role in an organisational structure is bound to an operational-management role using such a contract, all functional role invocations and responses are associated with CCA primitives.

**Table 3. Example form of operational-management contract**

<b>Operational-management Contract</b>			
Name	Supervisor-Subordinate		
Party A	Supervisor		
Party B	Subordinate		
A initiated	DO	→	ACCEPT, INFORM
	SET_GOAL	→	ACCEPT, INFORM
	INFORM	→	—

	QUERY	→	INFORM
	RES_ALLOC	→	ACCEPT
B initiated	INFORM	→	—
	QUERY	→	INFORM or REFUSE
	RES_REQ	→	RES_ALLOC or REFUSE

In summary, the expressive requirements needed to represent operational-management contracts in terms of state and behaviour are:

**State:** Each instance of a contract must include:

- the name of the parties (i.e. the functional roles or objects playing those roles)
- a FSM of the state of communication between the parties as defined by the protocol. It is this representation of the state of the association (viable or otherwise) that allows the organisational manager to maintain a representation of the state of control-flow through the organisation.

**Behaviour** represented in the contract includes:

- a protocol definition of permissible types of interaction between the parties
- a mapping from the protocol to the interface signatures of the respective functional-roles. This requires adherence to a naming standard for the methods so that they can be associated with types of invocation
- a mechanism for enforcing the protocol on communications between functional-roles

In the next section, we show how association-aspects [15] can be used to implement operational-management contracts with the above expressive requirements.

## 5. Using Association-Aspects to implement operational-management contracts

This section shows how we can implement operational-management contracts using the *association aspect* extension [15] to AspectJ [3]. We begin with a brief discussion of AspectJ aspects and association aspects and how they can be used to model behaviour between groups of objects. The subsequent subsections outline the steps that are taken to create a contract:

1. The elements of a contract related to *direction* and *restriction* on communication are defined using *pointcuts*.
2. Contract *clauses* are then created from these elements.
3. Actions to be taken when a contract clause is triggered are then defined using aspect *advice*.
4. A contract is constructed from its clauses and other elements in its schedule.
5. An instance(s) of a contract is created.

Aspect-oriented methods and languages seek to maintain the modularity of separate cross-cutting concerns in the design and program structures. As pointed out above, the organisation of software as expressed in a network of *operational-management* roles, cross-cuts the program structure defined by the *functional-roles* and classes.

The AspectJ extension to Java allows the programmer to define *pointcuts* that pick out certain *join points* (well-defined points in the program flow). 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 [8], as far as we are aware they have not been used to implement associations between roles. Aspects as currently implemented in AspectJ do not easily represent the behavioural associations between objects [16]. 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. [15] propose the use of *association-aspects* to allow an aspect instance to be associated with a group of objects. Such association-aspects meet the expressive requirements that we defined in the previous section. *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 and 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 AnAssociationAspect perobjects(o1, o2){
    //aspect variables
    //pointcut declarations
    //advice methods }
```

Figure 8 below is a schema that sets out the relationship between the code-level constructs (such as join points, named pointcuts, advice), and the functional-role and operational-management roles. The contract, as implemented by the association aspect, defines pointcuts that match particular types of communication between particular parties. Actions from the contract (in the form of advice) are woven into the code of the functional role. If a control communication triggers a clause in the contract the respective action is executed.

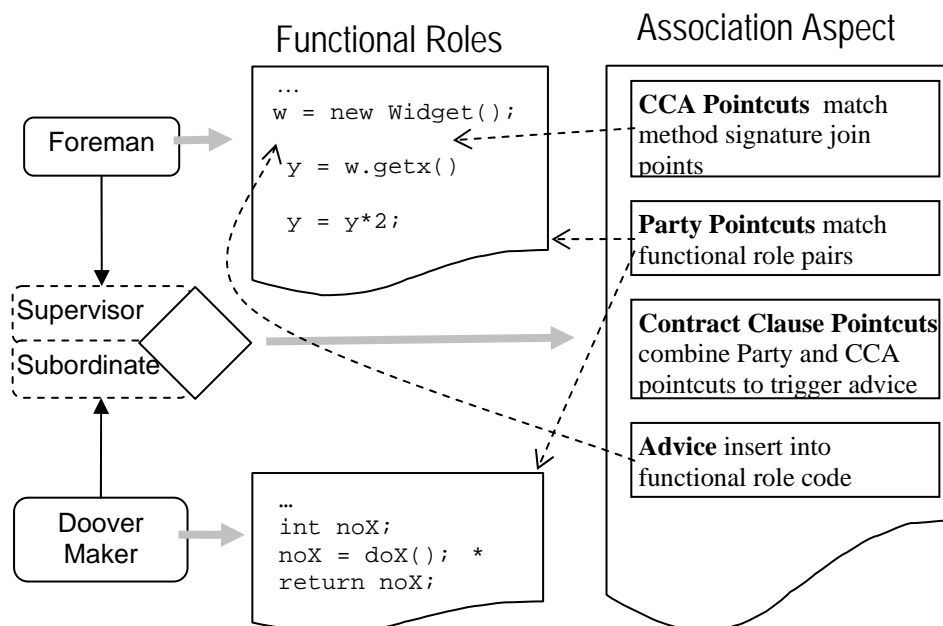


Figure 8. Using association aspects to implement contracts

The following subsections will explain this schema in more detail.

## 5.1. Defining the contract elements

To create an operational-management contract type, as defined in the Section 4, we need define the parties that participate in the contract, and which control-communication acts (CCAs) each of the parties can use. To do this we create three types of pointcuts:

1. Party pointcuts that match the parties to the contract.
2. CCA pointcuts that define the types of message.
3. Instances of these two above basic types of pointcut are then composed into pointcuts that represent particular clauses in the management contract. These composite pointcuts define *who* can say *what*.

**Party pointcuts** represent the parties to the contract and the direction of communication between those parties. 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 [15]. 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 `setG_DailyWidgetQuota()` enables a mapping to be created between the functional method that sets the daily quota of Widgets, and the `SetGoal` CCA primitive defined in the operational-management contract. Where CCAs are referential, as is the case with resource allocation, the method signature is distinguished by the type of parameter. In our example, all resources implement a `Resource` interface. The CCA pointcut `ResourceAllocate` could be defined as follows:

```
pointcut resAlloc() : call(* ra_*(Resource));
```

This pointcut called `resAlloc` matches any method *call* that

- begins with the characters “`ra_`”
- returns any type
- has a variable of type `Resource` as a parameter.

## 5.2. Defining the clauses of the contract

**Contract clause pointcuts** are the combination of a Party pointcut and a CCA pointcut. For example the pointcut below is Clause 1 (`a1`) of the contract. It says that Supervisor `sup` has the authority to allocate resources to the Subordinate `sub`.

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

A clause is defined for every CCA that can be initiated by either party. In the case of the `SupervisorSubordinate` contract as defined in Table 3 above, there are eight clauses required

in all – five governing communication from the Supervisor to the Subordinate (a1..a5), and three governing communication from the Subordinate to the Supervisor (b1..b5). We can define a further clauses a0 and b0 that is the compound of all the aX and bX clauses. For example a0 would be defined as

```
pointcut a0(Supervisor sup, Subordinate sub):
    a1(sup,sub)|| a2(sup,sub)|| a3(sup,sub)|| a4(sup,sub)|| a5(sup,sub);
```

### 5.3. Defining the effect of the contract clauses

Once the clauses of the operational-management contract have been defined, the actions that occur when particular clauses of the contract are triggered need be defined. These actions take the form of aspect advices. For communications between functional roles that are in accord with the clauses of the contract, no modification to the communication is necessary. The state machine that keeps track of the communication within the association contract needs to be updated both when the method call is made (Party A has made a request under the terms of the contract) and when the method returns (Party B has responded in appropriate form). Updating of the contract state can be done with `before()` and `after()` advices either for individual contract clauses (e.g. a1) or a compound clause (a0) :

```
before() : a1{...}/* update contract state machine and add any extra
    management functions e.g.accounting */
after() returning : a1{... } //ditto
```

All communication that does not conform to a contract clause should be prohibited. This is done with `before()` advice that throws an error if any method that does not correspond to terms of the contract (e.g. !a0 as defined above),.

### 5.4. Putting the contract together

The Party and CCA basic pointcuts are the common basis for all two-party operational-management contracts. Given this we can define in an abstract `ManagementContract` (`MContract`) aspect that contains these basic pointcuts, rather than having to define each management contract from scratch. Likewise all operational-management roles such as Supervisor and Subordinate implement the `ManagementRole` (`MRole`) interface. Figure 9 below shows these relationships.

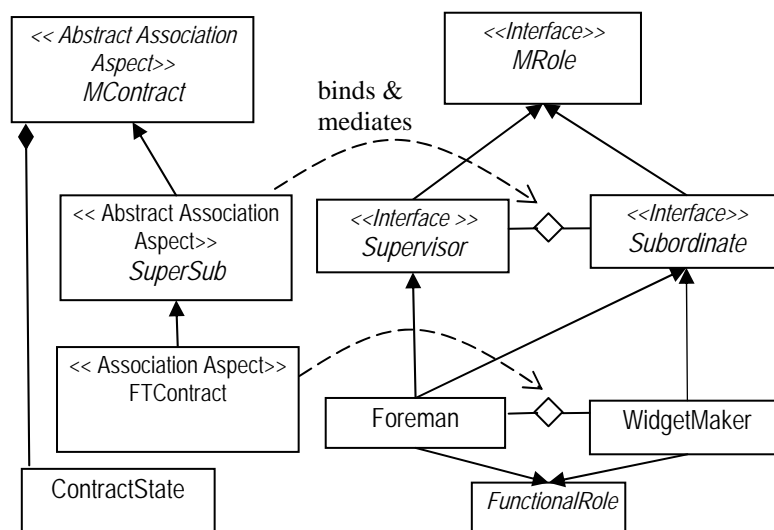


Figure 9. Inheritance Diagram of Aspects and Roles

The definition of a basic abstract two-party contract is as follows:

```

public abstract aspect MContract {
    ...
    protected ContractState cs; /* could be overridden by sub-aspect */
    ...
    //Party communication direction pointcuts
    abstract pointcut aToB (MRole a, MRole b);
    abstract pointcut bToA (MRole a, MRole b);
        //CCA pointcuts
    pointcut doIt() : call(* do_*(*));
    pointcut setGoal() : set(void setG_*(*));
    pointcut inform() : set(void inf_*(*));
    pointcut query() : call(* qry_*(*));
    pointcut resAlloc(): call(* ra_*(Resource));
        //returns reference to resource
    pointcut resReq():call(Resource rr_*(String));
        //parameter name of requested resource
        //returns reference to resource or null}

```

The form of the Supervisor-Subordinate contract aspect can now inherit from the general management contract as below. The contract from Table 3 above is replicated with numbered clauses (a1..a5, b1..b3).

Direction	Clause	CCA
Supervisor initiated	a1	DO → ACCEPT, INFORM
	a2	SET_GOAL → ACCEPT, INFORM
	a3	INFORM → —
	a4	QUERY → INFORM
	a5	RES_ALLOC → ACCEPT
Subordinate initiated	b1	INFORM → —
	b2	QUERY → INFORM, REFUSE
	b3	RES_REQ → RES_ALLOC, REFUSE

```

public abstract aspect SuperSub extends MContract{
    ...
    /*define contract clauses from directional Party and CCA pointcuts defined
    in the abstract ManagementContract parent class */
    pointcut a1(Supervisor sup, Subordinate sub)
        : aToB(sup, sub) && doIt();
    pointcut a2(Supervisor sup, Subordinate sub)
        : aToB(sup, sub) && setGoal();
    ...
    pointcut b1(Supervisor sup, Subordinate sub)
        : bToA(sup, sub) && inform();
    ...
    //all valid clauses
    pointcut a0(Supervisor sup,Subordinate sub):
        a1(sup,sub)||a2(sup,sub)||a3(sup,sub)|| a4(sup,sub)||a5(sup,sub);
    pointcut b0(Supervisor sup,Subordinate sub):
        b1(sup,sub)||b2(sup,sub)||b3(sup,sub);
    pointcut c0(Supervisor sup,Subordinate sub):
        a0(sup, sub) || b0(sup, sub);

    /*advices that define the actions when a contract clause is triggered*/
    before() : c0{
        cs.update(thisJoinPoint);}
    /* update contract state machine. Add any extra management functions e.g.accounting*/
    after() returning : c0{
        cs.update(thisJoinPoint);}
    before(): !c0{
        throw new InvalidCAA(thisJoinPoint);}
}

```

We can now define a concrete aspect based on the abstract Supervisor-Subordinate contract. This functional contract that defines the association between a Foreman and a WidgetMaker. In addition to monitoring and controlling the CCAs between parties to the contract, functional contracts can define performance requirements specific to the parties. In real-world terms,

functional contracts fill in the details of the contract schedule attached to the *form* of contract defined by the operational-management contract. For example, a functional contract may require a WidgetMaker to make a Widget with x seconds.

```
public aspect FTContract extends SuperSub perobjects(Supervisor,
    Subordinate) {
    ...
    public FTContract(Foreman f, ThingyMaker t){
        ...
        associate(f, t); //creates association
    }
    //instantiate directional pointcuts
    pointcut aToB(Supervisor f, Subordinate t): this(f) && target(t) &&
        associated(f, t);
    pointcut bToA(Supervisor f, Subordinate t): this(t) && target(f) &&
        associated(f, t);
    ...
    //define methods for revoking and reassigning contract
    //define functional performance pointcuts
}
```

## 5.5. Creating the contract instance between functional roles

We now apply these programming constructs to operational-management contracts using the Supervisor-Subordinate contract as an example. The creation of an instance is done as follows:

```
class Foreman implements Supervisor, Subordinate {... }
class WidgetMaker implements Subordinate {... }

//create instances of functional-roles
Foreman f = new Foreman();
WidgetMaker w = new WidgetMaker();

/* create the Foreman-WidgetMaket(FWContract) contract instance that binds
the functional-roles also passes reference to organisational-manager creator
*/
FTContract ft1 = new FTContract(this, f, w);
```

Communications between the Foreman and the WidgetMaker now conform to the SuperSub contract.

## 6. Related work

Our methodology extends work on role and associative modelling in [1,7-10]. Fowler [5] compares a number of object-oriented patterns for implementing roles including separate role types, role sub-typing, and the Role-Object pattern [1]. Kendall [8] has shown how aspect-oriented approaches can be used to introduce role-behaviour to objects. Roles are encapsulated in aspects that are woven into the class structure. While these role-oriented approaches decouple the class structure, they do not explicitly define an organisational level of abstraction by defining management roles. They are concerned with role-object bindings rather than role associations.

[14] and [15] 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 [15], on the other hand, modifies the AspectJ compiler to handle the additional *associated* pointcut primitive. We have used the later approach because it allows selection of the aspect instance based on any of the objects in the association.

The notion of CCA in this paper is derived from the concept of a *communication act* in multi-agent systems (MAS) agent communication languages such as FIPA-ACL [4]. CCAs, as defined here, are much more restricted in their extent. CCAs deal only with control communication, and do not have to take intentionality of the other parties into account[18].

Work on roles has also been undertaken in MAS [6,13,18]. In particular, [17] extends the concept of a role model to an organisational model. MAS systems, however, rely on components that have deliberative capability and more autonomy than the objects and roles discussed here. These agents negotiate interactions with other agents to achieve system level goals. These negotiations occur within a more amorphous structure than is defined here.

## 7. Conclusion and further work

In this paper we have shown how association aspects can be used to implement operational-management contracts. These contracts create relationships between functional roles and regulate the flow of control communication between them. In the ROAD framework, such contracts form a flexible organisational network that is designed to make software structures more adaptive to changing environments and goals.

Here we will limit the comments on further work to the area of operational-management contracts. There are a number of open issues. This paper has used the supervisor-subordinate association as an exemplar. Contract protocols need to be developed for other operational-management associations, such as those listed in Section 4. The set of CCAs that defined our example protocol is not complete and somewhat arbitrarily defined. This informality may suffice if operational-management contracts are only application or domain specific. However, if CCAs are to be generalised, a more rigorous approach may be needed. The UML 2.0 Superstructure Specification [12] provides a list of *primitive actions* which may provide the basis for a more formal definition of CCAs. Alternatively, agent communication languages such as [4] may provide the basis of a more rigorous definition.

The discussion in this paper has been limited to two-party contracts. Examples need to be developed of protocols that involve more than two parties. Examples also need to be developed of protocol sequences (e.g. negotiation protocols) that are more than just a single invocation-reply pair. In our example, the resource allocation clause gave permission to the Superordinate to access any subtype of Resource. In practice, different roles are likely to have access to different resources. It follows that we need to develop some scheme of resource ownership or access rights.

There are unresolved questions relating to the execution phase of contracts. Are the contracts managed externally by observing the state of the contract, or is breach/failure in the contract report triggered by the contract itself? For example, in the case of failure of a subordinate to meet the terms of a contract, presumably a supervisor would try to find another subordinate to perform the task. If that is not possible, the supervisor would report to the next higher level. Such issues will need to be resolved in the context of the ROAD framework.

## 8. References

- [1] Bäumer, D., Riehle, D., Siberski, W., and Wulf, M. "Role Object" in Pattern languages of program design 4, eds. Harrison, N., Foote, B., and Rohnert, H. Addison-Wesley, 2000, pp. 15-32.
- [2] Colman, A. and Han, J., "Organizational abstractions for adaptive systems," Proceedings of the 38th Hawaii International Conference of System Sciences, Hawaii, USA, 2005.
- [3] Eclipse Foundation, AspectJ <http://eclipse.org/aspectj/>, 2004, last accessed 7 Oct 2004
- [4] The Foundation for Physical Intelligent Agents, FIPA Communicative Act Library Specification <http://www.fipa.org/specs/fipa00037/>, 2002, accessed 27 Aug 2004
- [5] Fowler, M., "Dealing with Roles," Proceedings of the 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, USA, 1997.
- [6] Juan, T., Pearce, A., and Sterling, L., "ROADMAP: extending the Gaia methodology for complex open systems" Proceedings of the first international joint conference on Autonomous agents and multiagent systems, Bologna, Italy, ACM, 2002, pp. 3-10.
- [7] Kendall, E. A., "Role model designs and implementations with aspect-oriented programming." Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications, Denver, CO, 1999, pp. 353-369.
- [8] Kendall, E. A., "Role Modelling for Agents System Analysis, Design and Implementation" First International Symposium on Agent Systems and Applications IEEE CS Press, 1999
- [9] Kristensen, B. B. and Osterbye, K., "Roles: Conceptual Abstraction Theory & Practical Language Issues" Special Issue of Theory and Practice of Object Systems (TAPOS) on Subjectivity in Object-Oriented Systems, 1996
- [10] Lee, J. S. and Bae, D. H., "An enhanced role model for alleviating the role-binding anomaly" Software: practice and experience, vol.32, 2002, pp. 1317-1344.
- [11] Meyer, B. Object-oriented software construction, New York: Prentice-Hall, 1988.
- [12] Object Management Group, UML 2.0 Superstructure (Final Adopted specification) [www.uml.org/#UML2.0](http://www.uml.org/#UML2.0), 2004, accessed 13 Oct 2004
- [13] Odell, J., Parunak, H. V. D., Brueckner, S., and Sauter, J., "Changing Roles: Dynamic Role Assignment" Journal of Object Tech., ETH Zurich, vol.2(5), 2003, pp77-86.
- [14] 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.
- [15] Sakurai, K. , Masuharat, H., Ubayashi, N., Matsuura, S., and Komiya, S., "Association Aspects," Proceedings of the Aspect-Oriented Software Development '04, Lancaster U.K, 2004.
- [16] Sullivan, K., Gu, L., and Cai, Y., "Non-modularity in aspect-oriented languages: integration as a crosscutting concern for AspectJ," Proceedings of the 1st international conference on Aspect-oriented software development, AOSD 02 , Enschede, The Netherlands, 2002.
- [17] Zambonelli, F., Jennings, N. R., and Wooldridge, M. J., "Organisational Abstractions for the Analysis and Design of Multi-Agent Systems," Workshop on Agent-oriented Software Engineering ICSE 2000, 2000. Notes: Extension to Gaia for open systems
- [18] Zambonelli, F., Jennings, N. R., and Wooldridge, M., "Developing multiagent systems: The Gaia methodology" ACM Transactions on Software Engineering and Methodology (TOSEM), vol.12(3) , 2003, pp. 317-370.

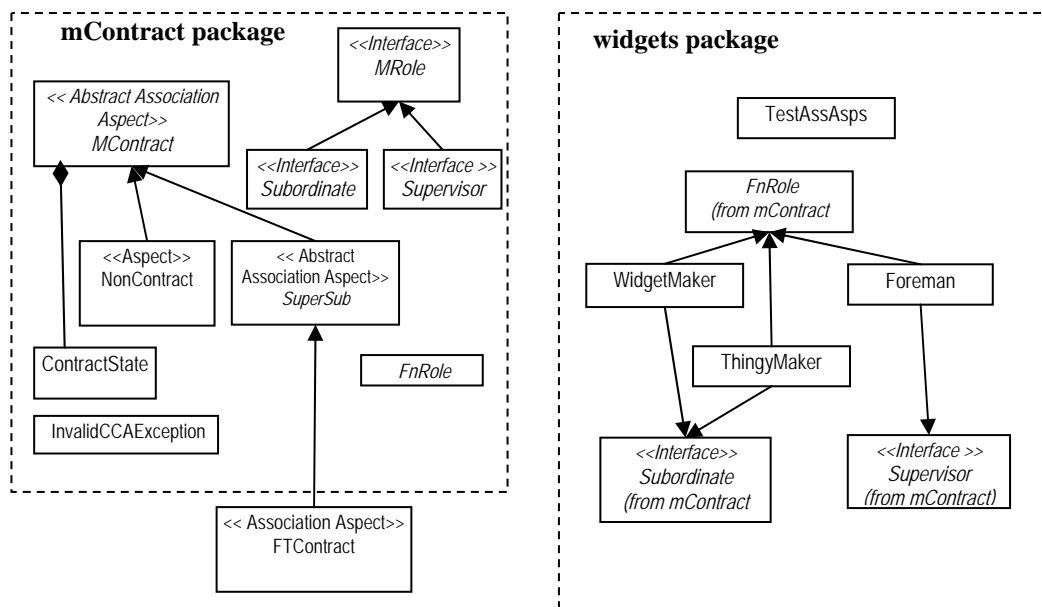
## 9. Appendix – Example Code

### 9.1. Overview

To compile the following code, an extension to the AspectJ compiler (ajc.exe) that handles association aspects is the required. This available from its author's web site [15] at <http://www.komiya.ise.shibaura-it.ac.jp/~sakurai>.

The example code below is divided into two packages – a **widgets** package that contains the functional code and a **mContracts** package that contains the operational-management roles, and management contracts in the form of aspects.

In order for weaving to occur, classes in both packages must be compiled simultaneously. An example make file “aa.lst” is listed below. To test the independence of the aspect contracts from the functional roles, it is possible to compile (using the standard AspectJ compiler) and run the code (with the exception of the couple of references to contracts in the test program), simply by removing the references to the association-aspect contracts from the make file.



Dependencies between the functional roles and management code exist in that functional-roles need to implement a management role interface (**Supervisor**, **Subordinate** etc) and inherit from an abstract class **fnRole**. The abstract functional role class keeps track of the contracts that the functional role is party to. This list is checked by the contract when a message is sent to the role. If the message is from, for example, a Supervisor who is *not* the role's supervisor, the message will be rejected. It would be possible to increase the modularity of the functional-role code in the widgets package by using an aspect with *inter-type declarations* that 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 WidgetMaker classes:

```
declare parents: {ThingyMaker || WidgetMaker} extends FnRole implements Subordinate;
```

The other dependency that impacts the widget package is the requirement that method call signatures follow arbitrary code conventions, so that pointcuts can be pattern-matched according to the type of interaction.

The only dependency that impacts the mContract package is obvious — that functional contracts such as FTContract need to be written with particular functional roles in mind. All other code in the mContract package could serve as a domain-independent, reusable library.

**Listing 1 aa.lst**

```
..\mContract\InvalidCCAException.java
..\mContract\Resource.java
..\mContract\Supervisor.java
..\mContract\Subordinate.java
..\mContract\FnRole.java
Foreman.java
ThingyMaker.java
WidgetMaker.java
NonFnRole.java
..\mContract\MContract.aj
..\mContract\NonContract.aj
..\mContract\SuperSub.aj
..\mContract\FTContract.aj
TestAssAsps.java //could be compiled separately
```

## 9.2. *Test Program and Output*

The test program demonstrates the functions:

- Contract creation
- Contract revocation
- Contract reassignment
- Interception and execution/prevention of various types of method-call between different types of party

The following cases involving authorised and unauthorised parties are tested for both valid and non-valid CCAs:

- Interaction between contracted parties
- Interaction between uncontracted functional roles
- Interaction between a functional role (of type FnRole) and class that does not inherit from FnRole.

---

**Listing 2 TestAssAsps.java**

---

```
package widgets;
import mContract.*;

public class TestAssAsps {
    public static void main(String[] args) {
        Foreman f = new Foreman("f1");
        Foreman f2 = new Foreman("f2");
        ThingyMaker t = new ThingyMaker("t1");
        ThingyMaker t2 = new ThingyMaker("t2");
        NonFnRole nfr = new NonFnRole(); //not of type FnRole

        //create a contract
        FTContract ft = new FTContract(f, t);

        //check the contract exists
        if(ft.checkParties(f,t))
            System.out.println(f + " & " + t + " are contracted");
        else
            System.out.println(f + " & " + t + " are NOT contracted");

        //Invoke valid request between contracted parties
        System.out.println("\nBefore valid authorised request");
        f.valid_request();

        //Invoke valid CCA but between uncontracted parties
        System.out.println("\nBefore valid UNauthorised request");
        f2.setThingyMaker(t);
        f2.valid_request();

        //CCAs cannot be invoked by non-functional roles
        System.out.println("\nBefore request from non FnRole");
        nfr.t = t;
        nfr.makeOutsideRequest();

        //ThigyMakers can not tell Foreman what to do
        System.out.println("\nBefore invalid unauthorised request");
        t.unauthorisedRequest();

        /*Foreman cannot ask a ThigyMaker to do something that is not in the
        *job description - ie wash car*/
        System.out.println("\nBefore invalid request");
        f.invalid_Request();

        //Destroy the contract
        ft.revoke(f, t); //revoke assocaition
        System.out.println("\nContract revoked - UNauthorised");
        f.valid_request(); //but invoker not authorised by contract

        //Reassign the contract
        System.out.println("\nBefore reassign");
        ft.reassign(f, t); //reassign the existing contract
        if(ft.checkParties(f,t))
            System.out.println(f + " & " + t + " are contracted");
        else
            System.out.println(f + " & " + t + " are NOT contracted");

        //Test if new contract works
        System.out.println("\nBefore valid authorised request");
        f.valid_request();
        //Test invocation from B to A
        System.out.println("\nBefore authorised inf_ request from reassigned
party 2");
        t.authorisedValidRequest();
    }
}
```

The output from the above program is shown below. The symbols generated by the contract aspects have the following meaning:

```
---> Valid authorised call from A to B
<--- Valid authorised call form B to A
X--X Call from uncontracted or other unauthorised party
-X-> Invalid call from contracted A to B
<-X- Invalid call from contracted B to A
```

### Listing 3 Output from TestAssAsps

```
widgets.Foreman:f1 & widgets.ThingyMaker:t1 are contracted

Before valid authorised request
---> before a0 : call(void widgets.ThingyMaker.do_makeThingy())
Update contract state
Thingy made
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
Update contract state

Before valid UNauthorised request
X--X CCA call from uncontracted functional role: call(void
widgets.ThingyMaker.do_makeThingy())

Before request from non FnRole
X--X CCA call from non-functional role: call(void
widgets.ThingyMaker.do_makeThingy())

Before invalid unauthorised request
<-X- Unauthorised method-call "void widgets.Foreman.do_somethingElse()" from
widgets.ThingyMaker to widgets.Foreman

Before invalid request
-X-> Unauthorised method-call "void widgets.ThingyMaker.washCar()" from
widgets.Foreman to widgets.ThingyMaker

Contract revoked - UNauthorised
X--X CCA call from uncontracted functional role: call(void
widgets.ThingyMaker.do_makeThingy())

Before reassign
widgets.Foreman:f1 & widgets.ThingyMaker:t1 are contracted

Before valid authorised request
---> before a0 : call(void widgets.ThingyMaker.do_makeThingy())
Update contract state
Thingy made
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
Update contract state

Before authorised inf_ request from reassigned party 2
<--- before b0 : call(void widgets.Foreman.qry_name())
Update contract state
My name is widgets.Foreman:f1
<--- after b0 : call(void widgets.Foreman.qry_name())
Update contract state
```

### 9.3. *mContracts* package code

A number of limitations of Sakurai's [15] implementation of association aspects became apparent during coding. In particular, *around advice* could not be made to work in a *perobjects* aspect. In addition, all pointcuts in a *perobjects* aspect must bind to all of the objects in the association aspect's parameter list. This made the implementation of pointcuts that filtered communication between non-contracted and contracted parties impossible from within the association aspect. While it would be nice to keep all rules relating to a contract in the one aspect, this limitation can be worked-around by putting pointcut and advice for non-contracted communication in a standard aspect – in this case **NonContract.aj**.

The inability to implement *around advice* did not prove a disadvantage as it is much cleaner and quicker to prevent unauthorised/invalid communication through the use of exceptions. In Java, primitive types and objects do not share a common parent. Separate *around advice* must therefore be implemented for each primitive type, for type void and for type Object, as it can not be known in advance what the return-type of a method being intercepted might be.

Below is code from some of the more interesting classes in the *mContracts* package. The functional role class, **FnRole**, enables all contract parties to be treated polymorphically and enables each functional role to keep track of what contracts it is party to.

#### Listing 4 FnRole.java

```
package mContract;
import java.util.*;

public abstract class FnRole {
    protected String name;
    protected boolean orgSlave = true;    //if true only repond to contracted
    invocations

    private Vector contracts = new Vector();

    public void addContract(MContract mc){
        contracts.add(mc);
    }
    public void removeContract(MContract mc){
        contracts.remove(mc);
    }
    public boolean isContractedTo(FnRole f){
        //check vector
        MContract mc;
        if (contracts.size() == 0)
            return false;
        for ( int i=0; i< contracts.size(); i++){
            mc = (MContract)contracts.elementAt(i);
            if (mc.isParty(f))
                return true;
        }
        return false;
    }
    public void setOrgSlave(boolean os){
        orgSlave = os;
    }
    public boolean getOrgSlave(){
        return orgSlave;
    }

    public String toString(){
        return this.getClass().getName() + ":" + name;
    }
}
```

All contract aspects inherit from the abstract aspect **MContract.aj**. The MContract class provides the definitions of the CCA pointcut patterns and provides the ability for contract instances to keep track of who are the parties to the contract. (It would be nice if this was a standard method of *perobject* association aspects.)

#### Listing 5 MContract.aj

```
package mContract;
import java.util.*;
public abstract aspect MContract {

    pointcut doIt() : call(public * do_*());
    pointcut setGoal() : call(public void setG_*(..));
    pointcut inform() : call(public void inf_*(..));
    pointcut query() : call(public * qry_*(..));
    pointcut resAlloc() : call(public * ra_*(Resource));
    pointcut resReq() : call(public Resource rr_*(String));

    pointcut allCCAs() : doIt() || setGoal() || inform() || query() ||
resAlloc() || resReq();
    /* Not really needed as aa.associated should do this*/
    Vector parties = new Vector();

    void addParty(FnRole fr){
        parties.add(fr);
        //System.out.println("Party added:" + fr);
        //System.out.println("Number of parties: " + parties.size());
    }
    void removeParty(FnRole fr){
        parties.remove(fr);
    }
    public boolean isParty(FnRole fr){
        FnRole party;
        for ( int i=0; i< parties.size(); i++){
            party = (FnRole)parties.elementAt(i);
            if (party == fr)
                return true;
        }
        return false;
    }

    public String toString(){
        String s = "Contract between ";
        for ( int i=0; i< parties.size(); i++)
            s = s + (FnRole)parties.elementAt(i) + ", ";
        return s;
    }
}
```

The abstract SuperSub contract defines the relationship between contracted Supervisor and Subordinates. The abstract pointcuts aToB and bToA allow communication direction pointcuts to be defined without knowing in advance which functional roles will be bound to the respective management roles.

### Listing 6 SperSub.aj

```

package mContract;

public abstract aspect SuperSub extends MContract {

    abstract pointcut aToB(Supervisor sup, Subordinate sub);
    abstract pointcut bToA(Supervisor sup, Subordinate sub);

    //Supervisor actions
    pointcut a1(Supervisor sup, Subordinate sub): doIt() && aToB(sup, sub);
    pointcut a2(Supervisor sup, Subordinate sub): setGoal() && aToB(sup, sub);
    pointcut a3(Supervisor sup, Subordinate sub): inform() && aToB(sup, sub);
    pointcut a4(Supervisor sup, Subordinate sub): query() && aToB(sup, sub);
    pointcut a5(Supervisor sup, Subordinate sub): resAlloc() && aToB(sup, sub);

    //subordinate actions
    pointcut b1(Supervisor sup, Subordinate sub): inform() && bToA(sup, sub);
    pointcut b2(Supervisor sup, Subordinate sub): query() && bToA(sup, sub);
    pointcut b3(Supervisor sup, Subordinate sub): resReq() && bToA(sup, sub);

    //compound actions
    pointcut a0() : doIt() || setGoal() || inform() || query() || resAlloc();
    pointcut b0() : inform() || query() || resReq();
    pointcut c0() : a0() || b0();

    //INVALID CALLS BETWEEN PARTIES to the contract
    before(Supervisor a, Subordinate b): !a0() && aToB(a, b) {
        String s = "-X-> Unauthorised "+ thisJoinPoint.getKind() + " \" +
            thisJoinPoint.getSignature() + "\" from "+
            thisJoinPoint.getThis().getClass().getName() + " to " +
            thisJoinPoint.getTarget().getClass().getName();
        throw new InvalidCCAException(s);
    }
    before(Supervisor a, Subordinate b): !b0() && bToA(a, b) {
        String s = "<-X- Unauthorised "+ thisJoinPoint.getKind() + " \" +
            thisJoinPoint.getSignature() + "\" from "+
            thisJoinPoint.getThis().getClass().getName() + " to " +
            thisJoinPoint.getTarget().getClass().getName();
        throw new InvalidCCAException(s);
    }
    //VALID CALLS BETWEEN PARTIES
    before(Supervisor a, Subordinate b): a0() && aToB(a, b) {
        System.out.println("---> before a0 : " + thisJoinPoint );
        System.out.println("Update contract state");
    }
    before(Supervisor a, Subordinate b): b0() && bToA(a, b) {
        System.out.println("<--- before b0 : " + thisJoinPoint );
        System.out.println("Update contract state");
    }
    after(Supervisor a, Subordinate b): a0() && aToB(a, b) {
        System.out.println("---> after a0 : " + thisJoinPoint );
        System.out.println("Update contract state");
    }
    after(Supervisor a, Subordinate b): b0() && bToA(a, b) {
        System.out.println("<--- after b0 : " + thisJoinPoint );
        System.out.println("Update contract state");
    }
}

```

The FTContract aspect is a subclass of SuperSub. It provides the constructor for the contract, as well as methods for revoking and reassigning the contract. Domain specific advice would also be added at this point. In real-world code, such domain functional contracts would be kept aside the mContract library. The reassign and revoke methods should ideally be defined in a more abstract class, and overridden.

**Listing 7: FTContract.aj**

```

package mContract;
import widgets.*;

public aspect FTContract extends SuperSub perobjects(Supervisor,
Subordinate){
    private Foreman f;
    private ThingyMaker t;

    public FTContract(Foreman f, ThingyMaker t) {
        f.setThingyMaker(t);
        t.setForeman(f);
        associate(f, t);
        addParty(f); //inherited from MContract
        addParty(t);
        f.addContract(this);
        t.addContract(this);
        this.f = f; this.t = t;
    }
    public Foreman getForeman(){
        return f;
    }
    public ThingyMaker getThingyMaker(){
        return t;
    }
    public boolean checkParties(Foreman f, ThingyMaker t){
        if (this.f == f && this.t == t)
            return true;
        else
            return false;
    }
    //prevent duplicate contracts being formed
    /* around throws runtime error*/
    before(Foreman f, ThingyMaker t):call(public FTContract.new
        (Foreman,ThingyMaker)) && args(f, t) && associated(f, t){
        System.out.println("FTContract between these objects already exists");
    }

    //make the directional pointcuts concrete
    pointcut aToB(Supervisor f,Subordinate t):
        this(f)&&target(t)&&associated(f,t);
    pointcut bToA(Supervisor f,Subordinate t):
        this(t)&&target(f)&&associated(f,t);

    public void revoke(Foreman f, ThingyMaker t) //a safer delete {
        if (this.f == f && this.t == t){
            removeParty(f); removeParty(t);
            f.removeContract(this); t.removeContract(this);
            delete(); //Association aspect built-in method
        }
    }
    public void reassign(Foreman f, ThingyMaker t) {
        f.setThingyMaker(t);
        t.setForeman(f);
        associate(f, t);
        addParty(f); addParty(t);
        f.addContract(this); t.addContract(this);
        this.f = f; this.t = t;
    }
}

```

The NonContract aspect is a standard AspectJ aspect that controls communication between functional roles and non-contract parties. This aspect could be refined to discriminated between functional-roles that are happy to accept instructions from non-functional roles, and those that are not.

It is important to allow non-functional roles to make methods calls to methods that are not CCAs. For example, if this was restricted `System.out.println` could not invoked the `toString()` method in a functional role.

**Listing 8: NonContract.aj**

```
package mContract;

public aspect NonContract extends MContract {

//UNAUTHORISED CALL FROM NON-PARTY
//stop all calls from non-contracted functional roles
before(FnRole a, FnRole b): call(public * mContract.*(..)) && this(a) &&
target(b) {
    if (!((FnRole)thisJoinPoint.getTarget()).
        isContractedTo((FnRole)thisJoinPoint.getThis()))
        throw new InvalidCCAException("X--X Call from uncontracted
            functional role: " + thisJoinPoint );
}

//stop all CCA pattern calls from non-functional roles
before(Object a, FnRole b): allCCAs() && this(a) && target(b) &&
!within(NonContract){
    if (a instanceof FnRole){
        if (!((FnRole)thisJoinPoint.getTarget()).
            isContractedTo((FnRole)thisJoinPoint.getThis()))
            throw new InvalidCCAException("X--X CCA call from uncontracted
                functional role: " + thisJoinPoint );
    }
    else
        throw new InvalidCCAException("X--X CCA call from non-functional
            role: " + thisJoinPoint );
}
}
```

## 9.4. Functional Role Code

The code from the functional roles is provided below. Note that all calls to contracted parties must be in try blocks.

### Listing 9: Foreman.java

```
package widgets;
import mContract.*;

public class Foreman extends FnRole implements Supervisor{
    private ThingyMaker t;

    public Foreman(){
        super.name = "anonymous";
    }
    public Foreman(String name){
        super.name = name;
    }
    public void setThingyMaker(ThingyMaker t) {
        this.t = t;
    }
    public void valid_request() {
        try{
            t.do_makeThingy();
        }
        catch (Exception e){
            System.out.println(e.getMessage());
        }
    }
    public void do_somethingElse() {
        System.out.println("foreman.do_somethingElse executing");
    }
    public void qry_name() {
        System.out.println("My name is " + this);
    }
    public void invalid_Request() {
        try{
            t.washCar();
        }
        catch (Exception e){
            System.out.println(e.getMessage());
        }
    }
}}
```

### Listing 10: ThingyMaker.java

```
package widgets;
import mContract.*;

public class ThingyMaker extends FnRole implements Subordinate {
    private Foreman f;

    public ThingyMaker(){
        super.name = "anonymous";
    }
    public ThingyMaker(String name){
        super.name = name;
    }
    public void setForeman(Foreman f) {
        this.f = f;
    }
    public void do_makeThingy() {
        System.out.println("Thingy made");
    }
    //Widgetmaker CANNOT ask Foreman to do things
    public void unauthorisedRequest() {
        try{
            f.do_somethingElse();
        }
        catch(Exception e){
            System.out.println(e.getMessage());
        }
    }
    //But Widgetmaker CAN ask Foreman for information
    public void authorisedValidRequest() {
        try{
            f.qry_name();
        }
        catch(Exception e){
            System.out.println(e.getMessage());
        }
    }
    //No one in organisation can ask a ThingyMaker to wash their car
    public void washCar() {
        System.out.println("Car Washed");
    }
}
```