Evolving Requirements through Coordination Contracts

Ana Moreira[1], José Luiz Fiadeiro[2], and Luís Andrade [3]

[1] Dept. Informatics, Faculty of Sciences and Technology, New University of Lisbon, 2829-516 Caparica, Portugal
amm@di.fct.unl.pt

[2] Department of Mathematics and Computer Science, University of Leicester Leicester LE1 7RH, United Kingdom
jose@fiadeiro.org

[3] ATX Software S.A., Alam. António Sérgio 7–1C, 2795-023 Linda-a-Velha, Portugal
landrade@atxsoftware.com

Abstract. Use-case driven software development processes can seriously compromise the ability of systems to evolve if a careful distinction is not made between "structure" and "use", and this distinction is not reflected immediately in the first model and carried through to the implementation. By "structure", we are referring to what derives from the nature of the application domain, i.e. to what are perceived to be the "invariants" or core concepts of the business domain, as opposed to the business rules that apply at a given moment and determine the way the system (solution) will be used.

This paper shows how the notion of coordination contract can be used to support the separation between structure and use at the level of system models, and how this separation supports the evolution of requirements on "use" based on the revision or addition of use cases, with minimal impact on the "structure" of the system.

1 Introduction

Use cases as introduced by Jacobson [11] and incorporated into the UML [3] play a fundamental role in object-oriented system development: they provide a description of the way the system is required to interact with external users. Existing proposals for a software development based on the UML are use case driven. It is not difficult to understand why. Given that the ultimate goal is to produce software that fulfils the expectations of the prospective users, driving the process based on user needs seems to make good sense.

However, our own experience in developing software using object-oriented methods has revealed that this approach is not without dangers. A use-case driven process can compromise the ability of the system to evolve if a careful distinction is not made between “structure” and “use”, and this distinction is not reflected immediately in the first model and carried through to the implementation. By "structure", we are not referring to the architecture of the solution but what derives from the nature of the application domain, i.e. to what are perceived to be the "invariants" or core concepts of the business domain, as opposed to the business rules that apply at a given moment and determine the way the system will be used.

The high degree of volatility of most application domains makes this distinction between core concepts and business rules a fundamental one. Time-to-market, and other business constraints dictated by the fierce competition that characterise the business activities of today, require that information systems be able to accommodate new business rules with minimal impact on the core services that are already implemented.

We believe that use-cases play a fundamental role in software development. However, we must be able to separate core entities from volatile business rules in the initial requirements analysis activity, and ensure that is evolution, and not construction, that is use-case driven. Such a process will be better shaped for supporting continuity and robustness to changes in the business domain. Hence our purpose in this paper is to use a new semantic primitive – coordination contract – for structure to be separated from use, and for evolution to become use-case centred.

The rest of this paper is organised as follows. In Section 2 we set the scenes of the problem we want to discuss by modelling a case study in a traditional way. In Section 3 we initiate a discussion about evolution and show why existing object-oriented approaches are not evolutionary. In Section 4 we give a brief overview of coordination contracts as a semantic primitive. In Section 5 we illustrate how contracts can be used to cope with changes in the business rules. In Section 6 we relate coordination contracts with other work. Finally, in Section 7 we draw some conclusions.

2 Capturing Requirements through Use Cases

2.1 Use Case Driven Approach: An Overview

Use cases, together with a use case driven approach to software development, were first introduced by Jacobson [11] and then adopted by the UML [3]. Since then, use case modelling has become a popular and widely used technique for capturing and describing functional requirements of a software system. It is also used as a technique for bridging the gap between descriptions that are meaningful to software users and descriptions that contain sufficient details for modelling and constructing a software system.

A use case model is represented by a use case diagram and use case descriptions. The diagram provides an overview of actors and use cases, and their interactions. The use cases' descriptions detail the functional requirements. An actor is anything that interfaces with the system. Some examples are people, other software, hardware devices, data stores or networks. Each actor assumes a role in a given use case. A use case represents an interaction between an actor and the system, i.e. it describes the outwardly visible requirements of a system. Use cases are recommended as a primary artefact and contribute to analysis, design as well as planning, estimating, testing and documentation. In addition, a use case model will often be part of a contract between the development organization and the customer regarding the functional requirements of the system to be developed. The quality of the use case model therefore has a large impact on the quality of the rest of the project.

Identifying and describing the user requirements should be accomplished in a systematic way. After building the use case model we can then draw interaction diagrams to describe the use cases' behavioural part. As new objects are found during this process we can start constructing a class diagram.

2.2 The Toll Collection System Case Study

In order to illustrate modelling and further evolution of requirements, let us use a simplified version of the toll collection system implemented in Portugal [5].

> "In a road traffic pricing system, drivers of authorised vehicles are charged at tollgates automatically. The tolls are placed at special lanes called green lanes. For that, a driver has to install a device (a gizmo) in his/her vehicle. The registration of authorised vehicles includes the owner's personal data, bank account number and vehicle details.
>
> Gizmos are read by the tollgate sensors. The information read is stored by the system and used to debit the respective accounts.
>
> When an authorised vehicle passes through a green lane, a green light is turned on, and the amount being debited is displayed. If an unauthorised vehicle passes through it, a yellow light is turned on and a camera takes a photo of the vehicle's licence plate.
>
> There are green lanes where the same type of vehicles pay a fixed amount (e.g. at a toll bridge), and ones where the amount depends on the type of the vehicle and the distance travelled (e.g. on a motorway)."

Looking at who will get information from the system and who will provide it with information helps identifying the following actors:

- Vehicle owner: this is responsible for registering a vehicle;
- Vehicle gizmo: this comprehends the vehicle and the gizmo installed on it;
- Bank: this represents the entity that holds the vehicle owner's account;
- System clock: represents the internal clock of the system that periodically triggers the calculation of debits.

Asking what are the main tasks of each actor helps identifying use cases. For the actors identified we have the use cases listed below and depicted in Figure 1:

- Register vehicle: is triggered by "vehicle owner"; it is responsible for registering a vehicle and its owner, and communicate with the bank to guarantee a good account;
- Pass single toll: is triggered by "vehicle gizmo"; it is responsible for dealing with tolls where vehicles pay a fixed amount. It reads the vehicle gizmo and checks on whether it is a good one. If the gizmo is ok the light is turned green, and the amount to be paid is calculated and displayed. If the gizmo is not ok, the light is turned yellow and a photo is taken.

– Enter motorway: is triggered by "vehicle gizmo"; it checks the gizmo, turns on the light and registers an entrance. If the gizmo is invalid, a photo is taken.
– Exit motorway: is triggered by "vehicle gizmo"; it checks the gizmo and if the vehicle has an entrance, turns on the light accordingly, calculates the amount to be paid (as a function of the distance travelled), displays it and records this passage. If the gizmo is not ok, or if the vehicle did not enter in a green lane, the light is turned yellow and a photo is taken.
– Pay bill: is triggered periodically by "system clock"; it sums up all passages for each vehicle, issues a debit to be sent to the bank and a copy to the vehicle owner.

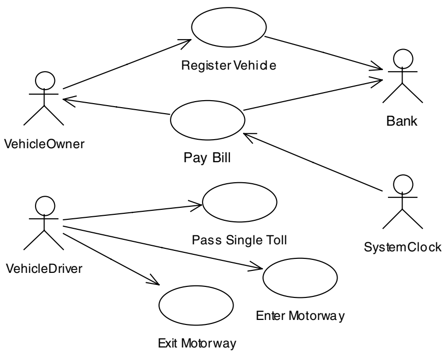

Fig. 1. The use case diagram of the road pricing system

The actors are linked to the use cases through association relationships. The arrow indicates the communication path between an actor and a use case.

The behaviour of each use case can be better defined in a set of interaction diagrams. We favour sequence diagrams, for their simplicity and because it is easier to see the temporal order of the messages. Each message on a sequence diagram corresponds to an operation on a class. So that sequence diagrams are not too complex, we build sequence diagrams to show scenarios, that is, an individual story of a transaction. For the three main use cases `PassSingleToll`, `EnterMotorway` and `ExitMotorway`, we can identify at least two scenarios for each one; one to deal with authorised vehicles and another to deal with non-authorised vehicles. Figure 2 depicts the sequence diagram for an authorised vehicle passing a single toll. (Other scenarios can be drawn in a similar way).

3 Supporting Requirements Modelling and Evolution

To promote understandability and reusability and to support requirements evolution, the models that we develop, as well as the design solutions that we derive from them, should provide mechanisms that help us to guarantee separation of concerns at different levels. This section shows what can be achieved using UML.

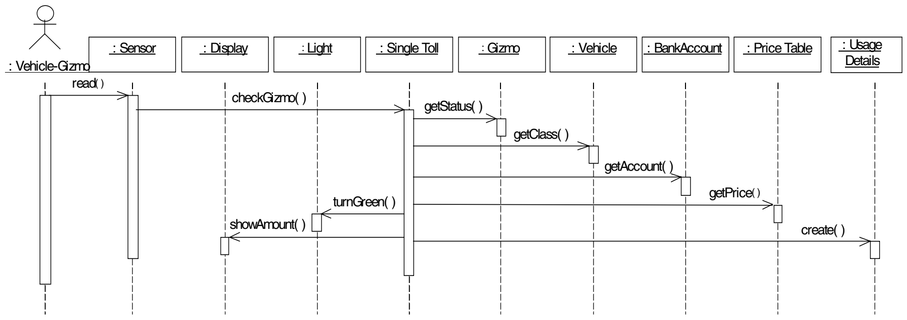

Fig. 2. A sequence diagram for the primary scenario Pass Single Toll

3.1 Building a Class Diagram

In UML we can use different kinds of objects to deal with different categories of information and behaviour. For example, an interface, or boundary, object should only be responsible for defining the structure of the interactions between the environment and the system. The interface object `SingleToll`, depicted in Figure 2, also controls the application, i.e. it is the system's decision maker. It is this object that decides what to do if the vehicle is an unauthorized one, for example. This makes the end system less reusable and understandable, compromising maintainability and making requirements evolution difficult to achieve.

Control objects, as proposed by Jacobson, can solve part of this problem [11]. They provide a means to separate interfacing responsibilities, from the basic system's entities and from functionalities that are hard to distribute among interface and entity objects. Interface objects can then be concerned only with the interactions with the outside world, while control objects drive the application, centralizing the responsibility of coordinating the set of tasks necessary to perform an use case. (And entity objects deal with the core concepts of the problem domain, of course.) Changes in the way actors interact with the system will then just affect the interfaces and the system's functionality does not need to be touched.

Control objects are transient objects that are instantiated to perform a single action, usually being the coordination of several other actions on server (entity) objects, and are destroyed when that action ends. They can be used when the functionality of the use case requires processing information from several objects and this includes decision points where the next alternative to be executed has to be chosen. Therefore, when in a sequence diagram the coordination task is being given to an interface object, as in Figure 2, a control object (in this case `SingleTollProcessor`) should be introduced between `SingleToll` and `Gizmo` to deal with that. Such an approach would result in adding five control objects to our class diagram (see Figure 3). Notice that the dashed arrows represent typical interactions needed to handle the behaviour described in the use cases.

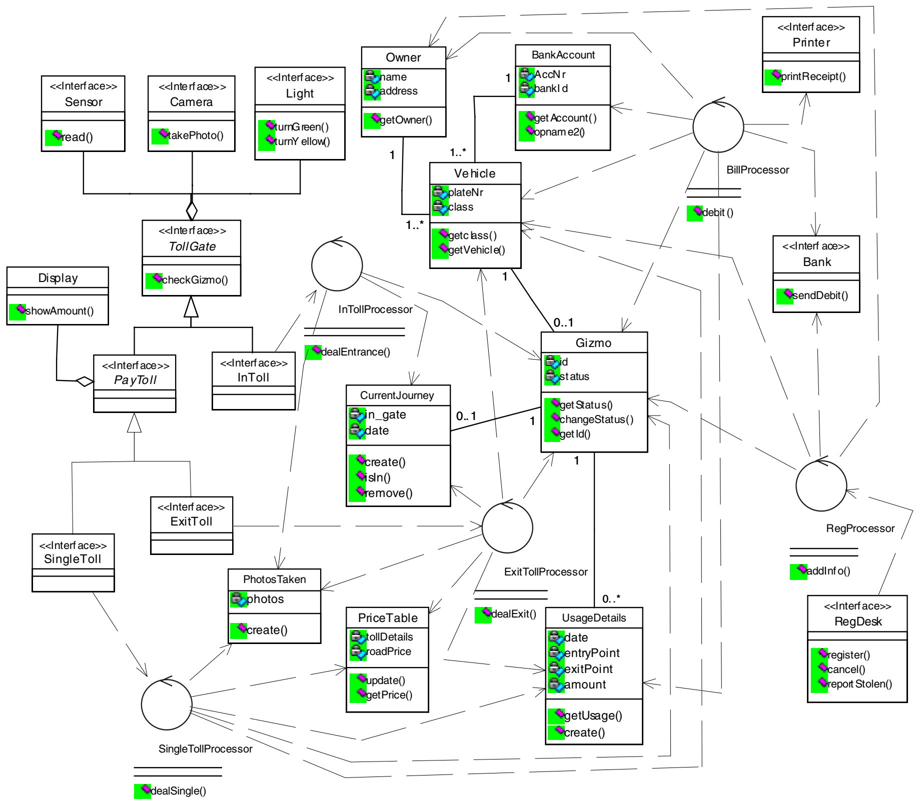

Fig. 3. Class diagram for the road pricing system

3.2 Dealing with Evolving Requirements

Control objects help to derive a good structural model for the system from the use-case analysis, but are they able to deal with the evolution of requirements that derive from new or different ways of using the system? Evolving requirements that reflect changes occurring at the level of business rules should not affect the entities that model the core concepts of the application domain. This cannot be achieved simply using control objects, as these objects are static, i.e. they cannot be reconfigured dynamically. A mechanism to support evolution should allow us to change the business rules without obliging us to change the core concepts already implemented.

To illustrate the problems raised by evolving requirements lets imagine that the owner of the system decides to offer a new set of packages to the users of the already modelled road pricing system.

"Certain public resources have physical limitations. For example, highways and bridges have a limited capacity for free-flowing traffic beyond which traffic congestion dramatically reduces the capacity for traffic flow. Several proposals for differential pricing seek to provide a monetary incentive for drivers to use such limited resources during off-peak hours so alleviating the problems of congestion. In practice, peak hour tolls or other user charges are set at a higher price than the price for off-peak usage. The larger price differential the greater the incentive for a change in driver habits. In this way the differential becomes a variable that can be changed depending on factors that influence the demand for road or bridge facilities, such as season of the year, popular holidays, weekends, etc. Additionally, a differential can be set to discriminate between users based on social factors, such as setting lower rates for health care vehicles, fire engines, handicapped drivers."

These requirements suggest two new use cases, one to deal with the special vehicles package and another to deal with the peak hour traffic package.

In a traditional approach, we must modify the already modelled classes, or even add new classes, to incorporate the new functionality into the system. Let us analyse the changes needed to handle peak hour traffic. In a typical classroom solution the condition `isPeakHour` would be declared on `PriceTable` as a precondition on `getPrice()`. Having the new business rule "hardwired" to `PriceTable` does not seem a very good strategy, for when the owner of the system comes up with a new price-package for customers, other changes will need to be made.

A naïve solution to adapt the existing system to the new business rule would be to enrich `PriceTable` with the new operation `getPeakHourPrice()`. Besides requiring obvious and direct changes to the class `PriceTable`, this solution is also intrusive on the client side because the client classes now have to decide on which operation to call. A further disadvantage of this solution is in the fact that the "business rule" is completely coded in the way the client class calls the price table and, thus, cannot be "managed" explicitly as a business notion.

The typical object-oriented solution to this new situation consists in defining a subclass `PeakHourPriceTable` with new attributes defining the peak hour periods. The operation `getPrice()` would have to be redefined so that the price is calculated according to a new rule. Nevertheless, there are two main drawbacks in this solution: (1) it introduces, in the conceptual model, classes that have no counterpart in the real problem domain (it is time of the day that may be "peak hour", not the price table); (2) this solution is still intrusive because the other classes in the system need to be made aware of the existence of the new specialised class so that links between instances can be established through the new class.

Dealing with special vehicles package would require similar changes in `PriceTable`, but also the creation of a new association class between classes `Vehicle` and `Owner` to establish the set of people driving special vehicles. In this case, the solution would affect even more the components in the system and, as before, is still intrusive as all the existing components have to be made aware that the new association class has become available.

The above two new business packages would oblige us to change the analysis models, the designs, and, of course, compile the corresponding code in an already implemented system. What would be interesting is to deal with new business rules without having to change an existing solution.

What we propose is to have two different kinds of concepts to handle evolution:

– classes of objects (e.g. Vehicle and Owner) that correspond to core business entities that are relatively stable in the sense that the organisation would normally prefer not to touch them;
– business products (e.g. special deals for off-peak usage) that may change according to the business policies of the organization.

Ideally, the business products should be added or removed from the system without requiring modifications in the core objects already implemented.

4 Coordination Contracts

Coordination contracts are the mechanism that will help achieving the dynamic evolution we have been arguing for. They provide simple modelling mechanisms, encapsulating all the design issues regarding dynamic reconfiguration of the system. Therefore the model is focused only on the rules that may change the behaviour of an existing system.

The notion of coordination contract was proposed in [1] for representing explicitly, as first class citizens, the rules that determine the way object interaction needs to be coordinated to satisfy business requirements. Our perception has been that this information is too often dispersed between the different diagrams that support models in the UML, both static and dynamic. For instance, a class diagram may contain associations whose only purpose is to provide the relationships that are necessary for given objects to be coordinated. The specific rules that enforce the required coordination may be found coded in one or more interaction diagrams, for instance in terms of message sequencing, or in state machines for controlling the operations being coordinated. The end result is that this dispersion makes it difficult for these interactions to be changed when new business requirements have to be added or old ones need to be modified.

Contracts allow for such coordination mechanisms to figure explicitly in class diagrams as special association classes whose semantics is similar to that of connectors in software architectures. A coordination contract (or, for simplicity, just "contract") consists, essentially, of a collection of role classes (the partners in the contract) and the prescription of the coordination effects (the glue in the terminology of software architectures) that will be superposed on the partners to coordinate their joint behaviour.

As a semantic primitive, coordination contracts are basically independent of any conceptual modelling approach in the same sense as, say, entities, associations or attributes are. The general form of a contract is:

```
contract <name>
   partners <list-of-participants>
   invariant <invariant properties>
   constants <local constants>
   attributes <local attributes>
   operations <local methods>
   coordination <coordination rules>
end contract
```

A contract has a name, a list of partners (the objects whose interactions we may want to coordinate), describe the invariant properties that the partners of the contract have to satisfy, a list of local constants, attributes and operations that may be useful to help specify the coordination mechanisms and, finally, the coordination section under which we describe the behaviour to be superposed. Each interaction under "coordination" has the form:

```
when <event>
  with <guard>
do <reaction>
```

The condition under "when" establishes the trigger of the interaction. Typical triggers are the occurrence of actions or state changes in the partners. The "do" clause identifies the reactions to be performed, usually in terms of actions of the partners and some of the contract's own actions. Together with the trigger, the reactions of the partners constitute what we call the synchronisation set associated with the interaction. Finally, the "with" clause puts further constraints on the actions involved in the interaction, typically further preconditions.

The intuitive semantics of contracts can be summarised as follows:

- Contracts are added to a system by identifying the instances of the partner classes to which they apply; these instances may belong to subclasses of the partners. The actual mechanism of identifying the instances that will instantiate the partners and superposing the contract is outside the scope of the paper. This can be achieved directly as in languages for reconfigurable distributed systems [13], or implicitly by declaring the conditions that define the set of those instances.
- Contracts are superposed on the partners taken as black-boxes: the partners in the contract are not even aware that they are being coordinated by a third party. In a client-supplier mode of interaction, instead of interacting with a mediator that then delegates execution on the supplier, the client calls directly the supplier; however, the contract "intercepts" the call and superposes whatever forms of behaviour are prescribed; this means that it is not possible to bypass the coordination being imposed through the contract because the calls are intercepted.
- The same transparency applies to all other clients of the same supplier: no changes are required on the other interactions that involve either partner in the contract. Hence, contracts may be added, modified or deleted without any need for the partners, or their clients, to be modified as a consequence.
- The interaction clauses in a contract identify points of rendez-vous in which actions of the partners and of the contract itself are synchronised; the resulting synchronisation set is guarded by the conjunction of the guards of the actions in the set and requires the execution of all the actions in the set.
- The effect of superposing a contract is cumulative; because the superposition of the contract consists, essentially, of synchronous interactions, different contracts may apply simultaneously to the same partners for the same trigger, which means that the synchronisation set that is going to be executed as a reaction to the trigger consists of the union of the synchronisation sets of the applicable triggers. Furthermore, the resulting synchronisation set is guarded by the conjunction of the guards of the

individual synchronisation sets. Determining which contracts should apply in which states (of a system) is another matter, one that relates to the management of the evolution of the system, not to its computational behaviour. We are currently developing a language for addressing such aspects, which relates to existing proposals for dynamic reconfiguration of distributed systems and software architectures [13].

Coordination contracts can be represented in UML as a new stereotype <<contract>> of classifier whose icon is the scroll symbol. Figure 4 shows the idea, with a simplified example taken from our case study.

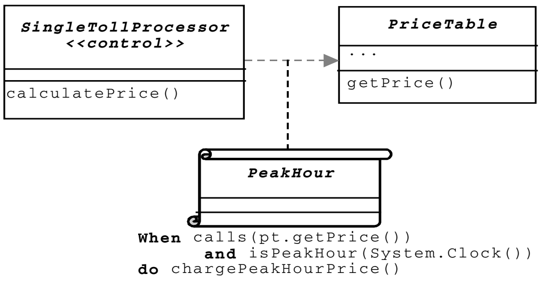

Fig. 4. PeakHour contract

A contract is a persistent entity that, through the *when* clause, intercepts interactions with the partners or detects events in the partners to which it has to react. In our example, whenever `SingleTollProcessor` calls `getPrice()` in `PriceTable` the contract `PeakHour` intercepts the call and if the time is within a peak hour period, it superposes a different way to calculate the price to be charged.

Notice that contracts do not define classes, with public properties. They do not offer services; instead, they coordinate services provided by the core classes.

5 Evolving with Coordination Contracts

The purpose of coordination contracts is to provide mechanisms that allow business rules to be added or removed from a system without affecting what is already specified and implemented. The two new use cases can be modelled through contracts, one for each use case, as shown below. As a contract details the business rules of a use case, changes in the use case only imply changes in the contract. Since contracts can be dynamically added and removed from a system, changes in the requirements are easily propagated to a solution.

The contract `PeakHour` is responsible for coordinating peak hour traffic and the contract `SpecialVehicle` coordinates the prices for special vehicles. We can complement the class diagram with a scroll symbol for each contract (and a note describing the trigger, the condition, if any, and the actions to be superposed). For example, as `PeakHour` intercepts calls to the operation `getPrice()`, we can "hang" the new symbol on the dependencies (dashed arrows) reaching PriceTable (see Fig. 4).

The full specification of `PeakHour` contract is:

```
contract PeakHour
   partners pt: PriceTable;
   constants peakCharge: Float:= 0.5;
             t1am: Time:= 7; t2am: Time:= 9;
             t1pm: Time:= 17; t2pm: Time:= 19;
   operations
      ?isPeakHour(t: Time): Bool :=
         (t>t1am and t<t2am) or (t>t1pm and t<t2pm)
   coordination
   peakHour:
      when calls(pt.getPrice(classVehicle,local)) and
             isPeakHour(System.clock())
      do {return(pt.getPrice(classVehicle,local)* (1 +
                    peakCharge))}
end contract
```

In our example, `PeakHour` defines several constants and the operation `isPeak-Hour()` that will be used in the *when* clause to select the subset of interactions to `PriceTable` that will be intercepted. If a vehicle uses a tollgate during peak hours, the contract will coordinate the behaviour superposing a different algorithm to calculate the amount to be paid. Otherwise, if the vehicle uses the system during off-peak hours, the price paid is the regular one and the contract will have no effect. Notice that the core classes previously modelled are seen as black boxes. As we said before, they do not know that their interactions are being intercepted and coordinated by a third party, which implies that they do not have to be modified.

The interactions established through contracts are atomic, i.e. the synchronisation set determined by each coordination entry of the contract is executed as a transaction. In particular, the object that calls `PriceTable` will not know what kind of coordination is being superposed. From its point of view, it is `PriceTable` that is being called.

`SpecialVehicle` contract is specified as follows:

```
contract SpecialVehicle
  partners st: SingleToll; et: ExitToll; v: Vehicle;
           g: Gizmo; pt: PriceTable;
  constants discount: Float:= 0.2;
  attributes p: Integer; special: Boolean;
  coordination
  VIVehicles:
      when(calls(st.checkGizmo(id)) or
           calls(et.checkGizmo(id) and special)
         local g:= Gizmo.getById(id);
      do
      {if g.getStatus() then
         p:= pt.getPrice(v.getClass(id),local)*
              (1-discount);
         st.Light.turnGreen();
         st.Display.showAmount(p);
         st.usageDetails.create(local,System.date(),p);
else st.Light.turnYellow();
          st.photosTaken.create(st.Camera.takePhoto());}
end contract
```

As only a subset of vehicles can benefit from a lower price, interactions are intercepted earlier than in the previous contract. Now we intercept calls, triggered by (special) vehicles, sent by pay tolls to the corresponding control objects and superpose a different behaviour. As we can see, when a contract is put "over" a control object superposing its single operation behaviour, the end result is replacing the control object in some situations. In a situation where we are developing a system for the first time, we propose to scrap control objects and use coordination contracts instead. This means that in our class diagram, each control object would be substituted by a regular coordination contract where the partners would be the classes with dependencies to that object.

We have not yet included in the specification the conditions under which a vehicle subscribes a given contract. This is left to the "coordination context" through which we can configure the system by using the services it makes available [2]. For example, for a vehicle to subscribe the `SpecialVehicle` contract the coordination context must offer the service `subscribeSpecialVehicle()`, as follows:

```
Coordination context Vehicle (v: Vehicle)
  workspace
     component types SingleToll; ExitToll; Vehicle;
                     Gizmo; PriceTable
     contract types SpecialVehicle;
      …
     services
        subscribeSpecialVehicle(g: Gizmo, s: Special):
           pre exist v and v.owns(g)
           post exists'SpecialVehicle(any,any,v,g,any) and
              SpecialVehicle(any,any,v,g,any)'.special=s
end context
```

Each context is "anchored" to a component or set of components. In this example the anchor is a vehicle instance. The workspace section lists the component and contract types involved in that context. The services are specified in terms of their pre and post conditions. When instantiating the contract, its partners have to be instantiated as well. In our example we use the keyword **any** to signify that this contract is valid for all objects of the type of the corresponding partner. The service is available to vehicles and its parameters are the gizmo owned by the vehicle and a boolean variable that states if the vehicle is special.

As `PeakHour` contract is valid for all vehicles, this means that special vehicles using the system during off-peak hours will get both discounts, resulting in a cumulative effect. In situations where several contracts are superposing conflicting behaviour to the same trigger, we can establish priorities among contracts to define which is the one that performs the superposition.

6 Related Work

What we propose through coordination contracts is a set of modelling primitives to support the development and evolution of software systems. What distinguishes our approach and takes it beyond what object-oriented and component-based methodologies can provide is its separation between computation and coordination. The computation level is concerned with the functionalities of the core entities of the domain while the coordination level is responsible for the volatile business rules that describe how a system is to be used.

For this purpose we brought together concepts and techniques from software architectures (the notion of connector [14]), parallel program design (the notion of superposition (or superimposition) [4, 7, 12]), distributed systems (the notion of techniques for supporting dynamic reconfiguration [13]) and programming languages, which have coined the term "Coordination Languages and Models" (the idea of separating computation from coordination [9]) that are now integrated in the concept of coordination contract. The coordination contracts can be dynamically superposed on the system, i.e. at run-time. The basic idea is to model the collaborations outside the components as coordination contracts that can be applied at run time to coordinate their behaviour.

Other technologies, such as aspect-oriented programming [6] and design patterns [8], also contribute to making software more amenable to change. However, they do this at a lower level of abstraction, at the design or even at the implementation level. What we claim is that we can do that earlier in the software development life cycle, using more abstract, higher level, primitives. Coordination contracts are used at a specification level (and in this paper at a requirements level). A good design can be changed at run-time, but that is more difficult to do, and usually requires a very good understanding of the existing solution. Design patterns, for instance, offer solutions that are too low level to be able to support an evolution process that takes place at the much higher level of abstraction in which business strategies and rules are (re)defined: they are useful for providing the design infrastructure that will support the required levels of adaptability, but they cannot be used for modelling and controlling the evolution process by themselves.

7 Concluding Remarks

We illustrated how coordination contracts can be used for handling requirements evolution, by means of a case study. We started by modelling the requirements using an approach based on the UML and then introduced new requirements to illustrate how contracts can be used to deal with evolution.

Coordination contracts are just a "technology", i.e. a means to an end. How to use these means for the end purpose is another matter, and one that we have not addressed in the paper. Knowing which concepts from the application domain correspond to core entities and which correspond to volatile business rules requires a good deal of expertise that is not necessarily available to everyone. Therefore, we should also work on the requirements engineering methods in a way that this distinction becomes apparent when interacting with the clients. That is, use-case analysis needs to be in a context where the distinction between core entities and business rules is more clearly promoted. We believe that, through coordination contracts, we have the modelling primitive that will allow for this distinction to be enforced. Although we have focused only on modelling, we have already developed an implementation of coordination patterns based on design patterns that can be deployed on current platforms for component-based development like CORBA, EJB and COM [10].

Acknowledgements. We thank M. Wermelinger, J. Gouveia and G. Koutsoukos for insightful discussions during the development of this work.

References

1. Andrade, L.F. and Fiadeiro, J.L.: “Interconnecting Objects via Contracts”, in UML'99 – Beyond the Standard, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag, 1999, pp. 566–583
2. Andrade, L.F., Fiadeiro, J.L. and Wermelinger, M.: “Enforcing Business Policies through Automated Reconfiguration”, in 16th Int. Conf. On Automated Software Engineering, IEEE Computer Society Press 2001, 426–429.
3. Booch, G., Rumbaugh, J., and Jacobson, I.: *The Unified Modeling Language User Guide*, Addison-Wesley, 1999
4. Chandy, K. and Misra, J.: *Parallel Program Design – A Foundation*, Addison-Wesley, 1988
5. Clark, R. and Moreira, A.: “Constructing Formal Specifications from Informal Requirements”, in proc. Software Technology and Engineering Practice, IEEE Computer Society, Los Alamitos, California, 1997, pp. 68–75
6. Elrad, T., Filman, R., and Bader, A.: "Theme Section on Aspect-Oriented Programming", Communications of ACM, Vol. 44, No. 10, 2001
7. Francez, N. and Forman, I.: *Interacting Processes*, Addison-Wesley, 1996
8. E., Gamma, Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley 1995
9. Gelernter, D. and Carriero, N.: "Coordination Languages and their Significance", Communications ACM 35(2), 1992, pp. 97–107
10. Gouveia, J., Koutsoukos, G., Andrade, L. and Fiadeiro, J., “Tool Support for Coordination-Based Software Evolution”, in *Technology of Object-Oriented Languages and Systems – TOOLS 38*, W.Pree (ed), IEEE Computer Society Press 2001, 184–196
11. Jacobson, I.: *Object-Oriented Software Engineering – a Use Case Driven Approach*, Addison-Wesley, Reading Massachusetts, 1992
12. Katz, S.: “A Superimposition Control Construct for Distributed Systems”, ACM TOPLAS 15(2), 1993, pp. 337–356
13. Magee, J. and Kramer, J.: “Dynamic Structure in Software Architectures”, in 4th Symp. on Foundations of Software Engineering, ACM Press, 1996, pp. 3–14
14. Perry, D. and Wolf, A., “Foundations for the Study of Software Architectures”, ACM SIGSOFT Software Engineering Notes, 17(4), 1992, pp. 40–52