# Evolution by Contract

**Luís Filipe A. Andrade**
Oblog Software SA
Alameda António Sérgio 7 – 1 A
2795 Linda-a-Velha, Portugal
landrade@oblog.pt

**José Luiz L. Fiadeiro**
LabMAC & Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande, 1700 Lisboa, Portugal
jose@fiadeiro.org

**Abstract**. The volatility of business requirements is putting an increasing emphasis on the ability for systems to accommodate the changes required by new or different organisational needs with a minimum impact on the implemented services. We propose a discipline for software development centred around the separation between what in systems are the basic service-providers (objects) and the mechanisms (contracts) through which the behaviour of these objects is coordinated to fulfil business requirements. We show how this separation can be supported in platforms for component-based development, making it possible for systems to evolve by adding, removing or replacing contracts without having to change the rest of the system.

## 1 Introduction

The problem addressed and the solutions proposed in this paper result from our experience in the last five years in conceiving, developing and applying methods and tools for modelling banking applications. In banking, like in many other business activities, the pace of market evolution and the volatility of requirements have a very deep influence on organisations and their information systems. More and more, large organisations face two important problems in this respect:

- How to conceive and develop information systems in order to support the continuous evolution of the core business and the evolution of system technology?
- How to make development and evolution scalable in the context of highly volatile business domains?

For better or for worse, such organisations seek answers to these problems in the context of object-oriented development techniques, of which the UML [7] is now, *de facto*, a standard. In spite of poor support from formal methods, one has to recognise that the construction of software has become more scalable and controllable thanks to mechanisms like encapsulation, clientship and inheritance. Even if the promised land of software component markets is not exactly around the corner, increased levels of reusability can be recognised in today's development practices.

However, our experience has shown that the benefits that object-oriented techniques have brought to software *construction* cannot be extended directly to software *evolution*. Even if object-oriented techniques make it easier to build systems by putting together components in a way that reflects interactions that take place in the application domain, changes on the implemented systems that result from the need to accommodate new business rules cannot be performed in such a modular way. This is because interactions are too often "hard-wired" in the code that implements the participating objects, making it difficult to change or introduce new interactions without having to change the implementation of the objects as well. Even worse, because such changes may result in new interfaces for the participating objects, a cascade of changes throughout the implementation of the system may well be triggered to account for the other interactions in which the objects participate.

As a consequence, the evolution of an object-oriented system, understood in terms of the need to perform changes on the system after it is released, cannot be supported in a compositional way. In other words, to some extent, object-oriented development is still producing *legacy* systems as far as evolution is concerned. Yet, time-to-market and other business constraints require that information systems be able to accommodate new practices and rules with minimal impact on the core services that are already implemented, thus prompting the need for modelling techniques that enable evolution to be directly compositional over the architecture of the information system.

Our purpose in this paper is to contribute to the solution of this problem by means of a modelling primitive – that we call *contract* – and a design pattern that enables contract-based models to be implemented, in a com-

positional way, over component-based development frameworks like CORBA, EJB and COM.

The rationale for contracts is the realisation that, in highly volatile business domains like banking, one can distinguish between two different kinds of "entities" as far as evolution is concerned. On the one hand, we have classes of objects like *account*, *client,* etc, that correspond to core business entities that are relatively stable in the sense that the organisation would normally prefer not to have to touch them, often because they really constitute *legacy* assets that are important to preserve. On the other hand, we have all the business products like account packages (different types of savings accounts, credits, etc) that keep changing because they determine the competitive edge of the organisation. These products require a layer of coordination to be established over the functionalities of the business entities so that the overall behaviour desired for the system can emerge.

When systems are conceived as collections of interacting objects, the problems that we have just identified require that we be able to express, and make available as first-class citizens, the constraints and the rules that capture the business requirements of the application domain. Because business rules determine the way object behaviour and interaction needs to be coordinated, it is necessary that these coordination aspects be available explicitly in the system models so that they can be changed, as a result of modifications that occur at the level of the business requirements, without having to modify the basic objects that compose the system. The purpose of contracts, as used in this paper, is to provide mechanisms for that layer of coordination to be modelled and implemented in a compositional way. In [4], we have emphasised the static modelling aspects of the concept and shown how contracts can be introduced as an extension of the UML [7]. In this paper, we focus on the dynamic aspects and present contracts as a means of structuring the evolution of systems.

## 2 An example

We will use an example from banking in order to motivate the notion of contract that we are proposing. The notation that we use in the examples is a shortened version of the Oblog language [http://www.oblog.com] that we have been developing for object-oriented modelling. An example of a class specification is given below for bank accounts.

```
class Account
operations
       class
            Create(client:Customer,iAmount:Integer)
       object
            Deposit(amount:Integer)
            Withdrawal(amount:Integer)
            Balance() : Integer;
            Transfer(amount:Integer, target:Account);
body
       attributes
            number : Integer;
            balance : Integer := 0
       methods
            Deposit is set Balance := Balance+amount
            Withdrawal is set Balance := Balance-amount
            Transfer is {    call target.Deposit(amount);
                             call self.Withdrawal(amount) }
end class
```

In Oblog, a class specification includes a section in which the interface operations are declared. We distinguish between class and object operations: the former are used for managing the population of the class as a whole and the latter apply to each specific instance. Each operation is declared with a list of input and output parameters and a specification of its required behaviour in terms of pre/post conditions (omitted in the example for simplicity). In the case of the bank account, the operations that were chosen are self-explanatory.

The body section of a class specification identifies the attributes that define the state of the instances as well as the implementations of the operations (called methods). Methods can be guarded with state-conditions like in Dijkstra's guarded commands. In fact, one may wonder why a guard has not been specified for `withdrawal` restricting this method to occur in states in which the amount to be withdrawn can be covered by the balance. The answer to this question is a good motivation for contracts.

Assigning the guard `Balance amount` to `withdrawal` can be seen as part of the specification of a business requirement and not necessarily of the functionality of a basic business entity like `account`. Indeed, the circumstances under which a withdrawal will be accepted can change from customer to customer and, even for the same customer, from one account to another depending on its type.

One could argue that, through inheritance, this guard could be changed in order to model these different situations. However, there are two main problems with the use of inheritance for this purpose. On the one hand, it views objects as white boxes in the sense that adaptations like changes to guards are performed on the internal structure of the object. From the point of view of evolution, this is not desirable. On the other hand, from the business point of view, the adaptations that make sense may be required on classes other than the ones in which the restrictions were implemented. In the example above, this is the case when it is the type of client, and not the type of account, that determines the nature of the guard that applies to withdrawals.

Hence, it makes more sense for business requirements of this sort to be modelled explicitly outside the classes that model the basic business entities. Our proposal is that guards like the one discussed above should be modelled as *contracts* that can be established between clients and accounts. In fact, we will provide mechanisms for such contracts to be *superposed* on existing implementations of clients and accounts, considered as black boxes, so that contracts can be added and deleted in a flexible way *(plug and play)*, reflecting the evolution of the business domain.

The example given above motivates the advantage of modelling, as first-class entities, the mechanisms that control the usage of given objects (contracts as controllers). The example below aims at illustrating cases in which a layer of coordination among different objects is required that is active in its own right.

One of the latest products to appear in the banking area can be called "the flexible package". This is a mechanism via which automatic transfers are made between a checking account and a savings account of the same client: from savings to checking when the balance goes below a certain threshold, and from checking to savings when the balance goes above a certain threshold.

Like before, the application of traditional object-oriented techniques for adding this new feature to the system would probably raise a number of problems. The first one concerns the decision on where to place the code that is going to perform the transfers: the probable choice would be the checking account because that is where the balance is kept. Hence, the implementation of account would have to be changed. The "natural" solution would be to assign the code to a new association class between the two accounts but, again, current techniques for implementing association classes require the implementations of the participating classes to be changed because the associations are implemented via attributes.

Another problem is concerned with the handling of the synchronisation of the transfers. If the transfers are not coded in the methods of the accounts, there is no way in which the whole process can be dealt with atomically as a single transaction. Again, what is required is a mechanism via which we can superpose a layer of coordination that is separate from the computations that are performed locally in the objects. This is exactly the purpose of contracts as detailed in the next section.

## 3 Contracts

From a static point of view, a contract defines an *association class* in the sense of the UML (i.e. an association that has all the attributes of a class) but the way interaction is established between the partners is more powerful: it provides a *coordination role* that is closer to what is available for configurable distributed systems and software architectures in general.

Another useful analogy is with architectural connectors [2]. A 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.

In Oblog, contracts are defined as follows:

```
contract <name>
      partners <list-of-partners>
      invariant <the relation between the partners>
      constants
      attributes
      operations
      coordination <interactions-with-partners>  behaviour < behaviour being superposed>
end contract
```

The instances of the partners that can actually become coordinated by instances of the contract are determined through a set of conditions specified as invariants. The typical case is for instances to be required to belong to some association between the partners.

Each interaction under "coordination" is of the form

```
<name> :    when <condition>
            do <set of actions>
            with <condition>
```

The name of the interaction is necessary for establishing an overall coordination among the various interactions and the contract's own actions. This is similar to what happens in parallel program design languages like Interacting Processes [14]. The condition under "when" establishes the trigger of the interaction. Typical triggers are the occurrence of actions 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 (to be further discussed in the following sections) is that, through the "when" clause, the contract intercepts calls to the partners or detects events in the partners to which it has to react. It then checks the "with" clause to determine whether the interaction can proceed and, if so, coordinates the execution of the synchronisation set. All this is done atomically.

An example can be given through the account packages already discussed. The traditional package, by which withdrawals require that the balance be greater than the amount being withdrawn, can be specified as follows:

```
contract Traditional package
      partners x : Account; y : Customer;
      invariants ?owns(x,y)=TRUE;
      coordination
           tp:  when y.calls(x.withdrawal(z))
                do x.withdrawal(z)
                with x.Balance() > z;
end contract
```

Notice that, as specified by the invariant, this contract is based on an ownership association that must have been previously defined. This contract involves only one interaction. It relates calls placed by the customer for withdrawals with the actual withdrawal operation of the corresponding account. The customer is the trigger of the interaction: the interaction requires every call of the customer to synchronise with the withdrawal operation of the account but enables other withdrawals to occur outside the interactions, e.g. by other joint owners of the same account. The constraint is the additional guard already discussed. Notice that the constraint applies only to the identified pair of customer and account, meaning that other owners of the same account may subscribe to different contracts.

The notation involving the interaction in this example is somewhat redundant because the fact that the trigger is a call from the customer to an operation of the account immediately identifies the reaction to be performed. In situations like this, Oblog allows for abbreviated syntactical forms of interaction. However, in the paper, we will consistently present the full syntax to make explicit the various aspects involved in an interaction. In particular, the full syntax makes it explicit that the call put by the client is intercepted by the contract, and the reaction, which includes the call to the supplier, is coordinated by the contract. Again, we stress that such interactions are atomic, implying that the client will not know what kind of coordination is being superposed. From his point of view, it is the supplier that is being called.

As already explained, the purpose of contracts is to externalise the interactions between objects, making them explicit in the conceptual models, thus reflecting the business rules that apply in the current state. Hence, contracts may change as the business rules change, making system evolution compositional with respect to the evolution of the application domain. For instance, new account packages may be introduced that relax the conditions under which accounts may be overdrawn:

```
contract VIP package
      partners x : Account; y : Customer;
      constants CONST_VIP_BALANCE: Integer;
      attributes Credit : Integer;
      invariants
           ?owns(x,y)=TRUE;
           x.AverageBalance() >= CONST_VIP_BALANCE;
```

```
        coordination
            vp:  when y.calls(x.withdrawal(z))
                 do   x.withdrawal(z)
                 with x.Balance() + Credit() > z;
  end contract
```

Notice that, on the one hand, we have strengthened the invariant of the contract, meaning that only a restricted subset of the population can subscribe to this new contract. On the other hand, the contract weakens the guard imposed on withdrawals, meaning that there are now more situations in which clients can withdraw money from their accounts.

In general, we allow for contracts to have features of their own. This is the case of the contract above for which an attribute and a constant were declared to account for the credit facility. It is important to stress that such features (including any additional operations) are all private to the contract: they cannot be made available for interaction with objects other than the partners. Indeed, the contract does not define a public class.

Our last example models the flexible package that we have already motivated.

```
  contract Flexible package
        partners c, s : Account;
        attributes min, max : Integer;
        invariants c.owner=s.owner;
        coordination
            putfunds:when calls(c.Deposit(z))
                     do {    if c.Balance()+z > max then {
                                    c.Deposit(z); c.Transfer(c.Balance()-max,s)} }
            getfunds:when calls(c.Withdrawal(z))
                     do {    if c.Balance()-z < min then {
                                    s.Transfer(min-c.Balance(),c); c.Withdrawal(z)}      }
  end contract
```

## 4 Semantical aspects

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; for instance, in the case of the flexible package, both partners were identified as being of type account but, normally, they will be applied to two subclasses of account: a checking account and a savings account. The actual mechanism of identifying the instances that will instantiate the partners and superposing the contract is outside the scope of the paper. In Oblog, this can be achieved directly as in languages for reconfigurable distributed systems [20], 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 will superpose their coordinating behaviour, achieving a cumulative effect. For instance, in the example of the flexible package, the transfers can also be subject to contracts that regulate the discipline of withdrawals for the particular account and client.

Contracts, as motivated in the previous sections, draw from several mechanisms that have been available for sometime in Software Engineering, which further clarifies the rationale for the semantics that we gave above:

- Contract-based development is based on the idea of separating *computation* from *coordination*, i.e. of making clear what components in a system provide the functionalities on which services are based, and what mechanisms are responsible for coordinating the activities of those components so that the desired behaviour emerges from the interactions that are established. This idea has been promoted by researchers in the area of Programming Languages who have coined the term "Coordination Languages and Models" [e.g. 16].
- The importance of having these coordination mechanisms available as first-class entities and as units of structure in system models and designs was inspired by the role played by connectors in software architectures [23]. This is why in [4] we proposed contracts as a semantic primitive enriching the notion of association class.
- The ability for objects to be treated as black-box components and, hence, for contracts to be dynamically added or removed from a system without having to recompile the partners, is achieved through the mechanism of superposition (or superimposition) developed in the area of Parallel Program Design [9,14,18].

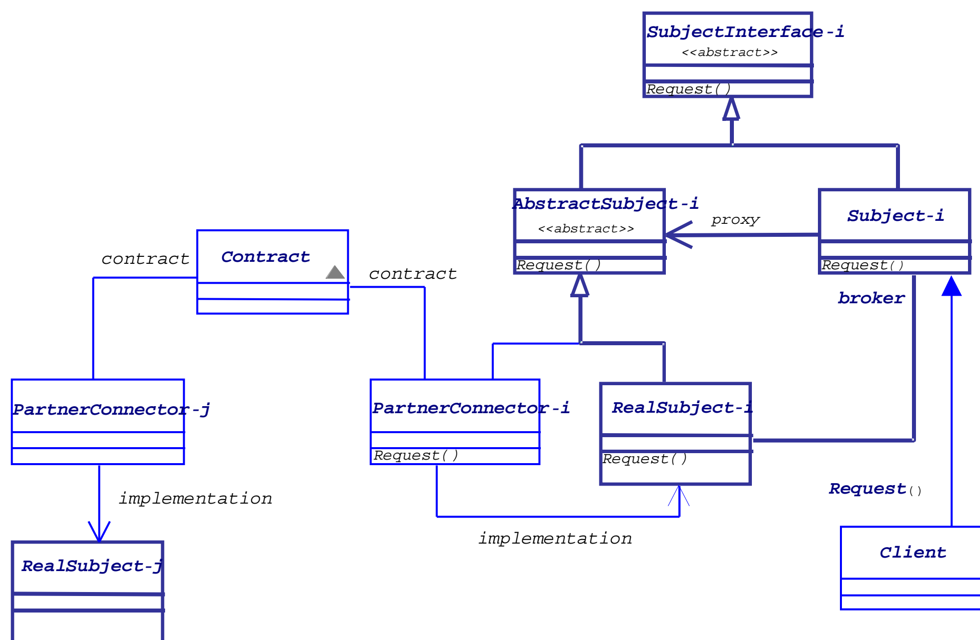
It is important to state that the contribution of superposition techniques for component adaptation in a black-box style has also been recognised in [8]. The relationship between coordination languages and software architectures has also been recognised by several authors, to the point in which joint workshops are now organised by the two communities. Our main contribution has been to integrate these three aspects into a primitive that can be used very effectively for developing and evolving systems in a way that is compositional with respect to the evolution of the business rules in the application domain.

In fact, our contribution has gone a step further in that, over several years, we have developed a mathematical semantics that brings all these aspects together: coordination, superposition, and architectures. The presentation of this semantics is outside the scope of this paper. Please consult [11] for the formalisation of different kinds of superposition that lead to the identification of different kinds of architectural connectors (regulators, monitors, etc); [12] for a formalisation of architectural connectors based on the previous formalisation of superposition, which includes the semantics of instantiation of the partners (roles); [13] for the coordination aspects as related to superposition and the application to software architectures; [25] for the application to dynamic reconfiguration, including the definition of algebraic operations on architectural connectors that are directly applicable to contracts; and [4] for the first formalisation of contracts as presented herein, and their relationship to the UML.

## 5 The contract design pattern

As already explained in the previous sections, a contract works as an active agent that coordinates the contract partners. In this section, we are concerned with the way these coordination mechanisms can be implemented. When defining an implementation, we need to have in mind that, as motivated in the introduction, we should be able to superpose a contract to given objects in a system and coordinate their behaviour as intended *without having to modify the way these objects are implemented*. This degree of flexibility is absolutely necessary when the implementation of these objects is not available or cannot be modified, as in legacy systems. It is also a distinguishing factor of contracts when compared with existing mechanisms for modelling object interaction, and one that makes contracts particularly suited in business domains where the ability to support the definition and dynamic application of new forms of coordination is a significant market advantage.

Different standards for component-based software development have emerged in the last few years, among which CORBA, JavaBeans and COM are the current trend in industry. However, none of these standards provide a convenient and abstract way of supporting superposition as a first-class mechanism. Because of this, we propose our solution as a design pattern. This pattern exploits some widely available properties of object-oriented programming languages such as polymorphism and subtyping, and is based on other well known design patterns, namely the Broker, and the Proxy or Surrogate [15].




The class diagram below depicts the proposed pattern. In what follows, we explain, in some detail, its basic features, starting with the participating classes.

*SubjectInterface-i* – as the name indicates, it is an abstract class (type) that defines the common interface of services provided by AbstractSubject-i and Subject-i.

*Subject-i* – This is a concrete class that implements a broker maintaining a reference that lets the subject delegate received requests to the abstract subject (AbstractSubject-i) using the polymorphic entity proxy. At runtime, this entity may point to a RealSubject-i if no contract is involved, or point to a PartnerConnector-i that links the real subject to the contracts that coordinate it.

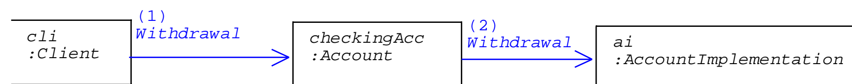
*AbstractSubject-i* – This is an abstract class that defines the common interface of RealSubject-i and PartnerConnector-i. The interface is inherited form SubjectInterface-i to guarantee that all these classes offer the same interface as Subject-i (the broker) with which real subject clients have to interact.

*RealSubject-i* – This is the concrete domain class with the business logic that defines the real object that the broker represents. The concrete implementation of provided services is in this class.

*PartnerConnector-i* – This class maintains the connection between contracts and the real object (RealSubject-i) involved as a partner in the contract. Adding or removing other contracts to coordinate the same real object does not require the creation of a new instance of this class but only of a new association with the new contract and an instantiation link with the existing instance of PartnerConnector-i. This means that there is only one instance of this class associated with one instance of RealSubject-i.

*Contract* – This is a coordination object that is notified and takes decisions whenever a request is invoked on a real subject or when it change its state.

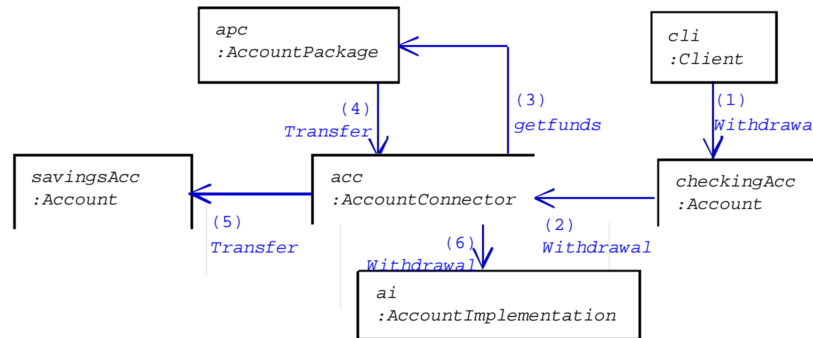
If there are no contracts coordinating a real subject, the contract pattern can be simplified and only the classes that are not dimmed in the figure become necessary. The introduction of a contract implies the creation of instances for the dimmed classes and associations. The following is a possible object diagram when no coordination contract is defined, which means that there are no Contract and no PartnerConnector instances.




In this scenario, where object `checkingAcc` of type `Account` is not under coordination, the only overhead imposed by the pattern is an extra call from the broker to the real object (`AccountImplementation`). Introducing a new contract to coordinate interaction with objects of type `Account` implies only modifications on the object that plays the role of broker, i.e. in the object `checkingAcc`, making its proxy become a reference to the object that plays the role of a contract partner connector, i.e. the object `acc` of type `AccountConnector`, as seen in the following interaction diagram. Doing only this minor modification, neither the code of clients (e.g. object `cli`) nor the code of the broker `checkingAcc` and the real object `ai` need to be modified in order to accom-

modate the new behaviour established by adding the contract `apc` of type `AccountPackage`.

The new behaviour introduced by contract `apc` is described in the object interaction diagram below. This diagram shows how the contract superposes a new behaviour when requests of type `Withdrawal()` are invoked on an object of type `Account`.




In this diagram we can see that once `Withdrawal()` is sent to the object broker `checkingAcc`, it delegates its execution on the proxy reference (in this case on `acc` instead of `checkingAcc`, as seen on the previous figure). In `acc` the implementation of subject services has the following format

```
Request() IS
      IF   (a coordination guard holds)
      THEN Execute the coordination do code
      ELSE Execute the original code of Request
```

That is to say, before the partner connector `acc` gives rights to the real object implementation `ai` to execute the request, it intercepts the request and gives right to the contract `apc` to decide if the request is valid and perform other actions. This interception allows us to impose other contractual obligations on the interaction between the caller and the callee. This is the situation of the first model discussed in section 3 where new preconditions were established between `Account Withdrawals` and their `Customers`. On the other hand, it allows the contract to perform other actions before or after the real object executes the request. This is the situation of the the coordination equation named `getfunds` established by the contract `Flexible package` in section 3. Only if the contract authorises can the connector ask the object implementation `ai` to execute and commit, or undo execution because of violation of post-conditions established by the contract.

As stated at the beginning of this section, current component-based technology does not provide a convenient way for coordinating components. The benefit of having this form of coordination available as a primitive construction when specifying components and their interactions is that it avoids the burden of having to code such a pattern. In the meanwhile, tools which, like Oblog, provide automatic code generation from high level specifications, must hide the implementation complexity of coordination, allowing the developer just to specify the contract itself.

## 6 Contract-based development

The object technology market is growing very rapidly, increasing the offer of low-level tools and technologies for the fine-grain parts of information system development – *development in the small*. However, the current practice of object-oriented software development methods in what concerns the large-grain parts, namely architectures – *development in the large* – still involves the sequential use of analysis and design techniques and tools. For most large-scale systems, this is quite a wasteful process to follow because it ignores the existence of software components, design models and environment frameworks.

A new trend is emerging in software development that is based on a growing belief that analysis and design should be based on predefined frameworks of skeletal applications, components, and design patterns that can be easily customised and integrated. The only hope for organisations to be able to face the challenges of the very fast market evolution that we are already witnessing is, clearly, to adopt such a strategy. Therefore, it seems clear that the integration and coordination of the different parts of information systems, at all levels of granularity, is one of the major challenges that software development needs to face.

We believe that a good basis for meeting this challenge can be provided by adopting a development strategy based on three subprocesses:

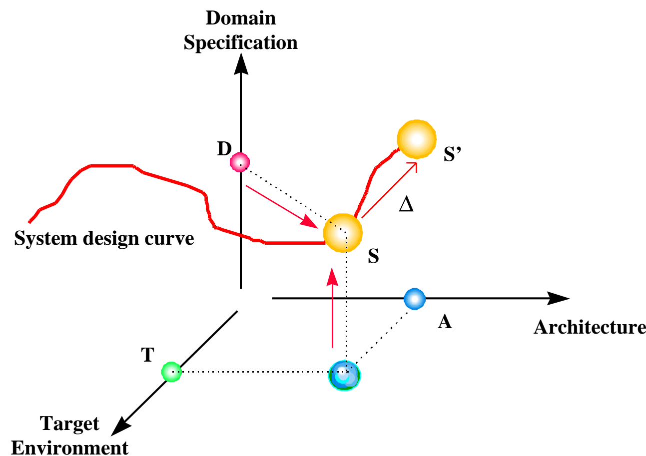
1. the construction of individual components implementing the domain objects;
2. the construction of contracts that play the roles of coordination agents responsible for implementing the configurable business rules for specific domains;
3. the construction of generic component connectors for specific architectures.

Given that contracts can be dynamically superposed on existing components, the word 'construction' above can be replaced by 'identification'. Indeed, because contracts treat the partner instances as black-boxes, this approach promotes reuse at all levels. Moreover, it facilitates the integration of third-party, closed components, namely legacy systems.

The advantages that we have identified above in terms of supporting evolution apply, as stated in the introduction, to changes that are triggered by the evolution of the business rules that apply to the system at hand. As we all know, changes to the system can be triggered by many other factors. Our experience has shown that implementing an information system involves making decisions that may be decomposed according to three different dimensions, as depicted in the figure below.

- **domain of the problem being solved** – this dimension is concerned with the description of the problem at hand, leading to an ideal model, free from any details concerning implementation. This model is defined using a specification language.
- **system architecture description** – this dimension is concerned with the way the system is structured in terms of components and interconnections. Modules are vital for dividing large specifications into parts, and to specify these parts with sufficient precision that one can construct each part knowing only the specification of the other parts, which points to a component-based approach [24]. The nature of the components that are needed, and the form in which they need to be interconnected, are influenced by infrastructural constraints like the distribution strategy or the type of interaction with the environment;
- **software chosen for the implementation** – this dimension is concerned with the technology used to implement the business problem according to the chosen architecture.

Ideally, a choice taken with respect to one of these dimensions should not interfere with the decisions related to the other. This independence enables designers to change a previous decision in one of the dimensions without having to reconsider the decisions taken in the other, thus achieving a significant degree of flexibility.




As part of future work, we intend to study the way a contract-based approach can influence development and evolution not only along the Domain Specification, but also the Architecture dimension. For that purpose, we can capitalise on work already done in using notions similar to contracts at these lower levels of development. In the next section, we identify some of these approaches.

## 7 Related work

Several authors have already made similar observations about the need to make explicit and available, as first-class citizens, the rules that govern the behaviour of systems, namely in [19], which became the subject matter of the ISO General Relationship Model (ISO/IEC 10165-7). The notion of contract that we proposed in the paper is in the spirit of this work but adds to it the evolutionary aspects that it inherits from the architectural

approaches, and the concurrency and synchronisation aspects that it inherits from the notion of superposition as used for parallel program design.

In section 4, we have also mentioned, and gave evidence to the effect, that our approach capitalises on a number of proposals that have been made in the areas of Programming Languages (coordination languages and models), Parallel Program Design (superposition), and Software Engineering (software architectures). In fact, we view the notion of contract as a synthesis of these three aspects which, together with the experience that we have gathered in information system development, justify an investment in methodologies and tools for supporting contract-based development.

Having acknowledged the sources on which our proposal is grounded we must, nevertheless, emphasise that other approaches can be found in the literature that, in many ways, are similar in spirit or capture some of the intuitions that we had when developing contracts.

We should start by saying that the term 'contract' itself has been quite overloaded:

- there are, of course, contracts in the sense of Meyer [21]; their purpose is to support the development of object methods in the context of client-supplier relationships between objects. Therefore, they apply, essentially, to the construction of systems. In fact, in what concerns evolution, Meyer's contracts are at the origin of some of the problems that we have identified in the introduction and which made us point out that even object-oriented development techniques are producing legacy systems: by adopting clientship as the basic discipline for object interconnection, a bias is introduced in the way business rules get coded up, assigning to the supplier side the reponsibility for accommodating changes that, from the point of view of the business rules, belong to the client. This was the case of the flexible withdrawals for VIP customers: by placing the contract on the supplier side (the account), the new rules are more easily modelled as specialisations of account whereas, in the application domain, they reflect specialisations of the client. In [4] we have discussed this matter with more detail. In summary, our opinion is that Meyer's notion of contract does not scale up to system evolution.
- A notion of contract that applies to behaviours and not to individual operations or methods is the one developed in [17]. The aim of contracts as developed therein is to model collaboration and behavioural relationships between objects that are jointly required to accomplish some task. The emphasis, however, is in software construction, not so much in evolution.
- A notion of contract can also be found in [5] that emerged in the context of the action-systems approach. Like in our approach, it promotes the separation between the specification of what actors can do in a system and how they need to be coordinated so that the computations required of the system are indeed part of the global behaviour. The architectural and evolutionary dimensions are not explored as such.

Besides these related notions of contract, other approaches can be found in the literature that explore the use of architectures in evolution:

- The importance of architectures for run-time software evolution has been recently pointed out [22]. The authors highlight the role of architectural connectors in mediating and governing interactions among components, which is essential for facilitating system evolution in the sense that we motivated in the paper. However, the authors concentrate too much in presenting design and implementation solutions in the form of a tool set for a Java-C2 combination, missing the abstraction power that superposition provides as a general technique for enabling contract (or connector) based evolution, and the setting-up of semantic primitive which, like contracts, can be used in conjunction with general languages for object-oriented development like the UML.
- Another recent architectural-approach to evolution that is important to mention is the one developed by [10] in which a notion of contract is explicitly developed for managing exchanges between units of evolution. These units, called zones, encapsulate processes, objects and types at design and run-time so that the contents of one zone can be evolved without affecting the code in another zone. Contracts make the inter-relationships and communication between zones explicit by providing mechanisms – "change absorbers" – for transforming calls made on a pre-evolution type from outside the zone into a call onto the evolved type. The emphasis is, therefore, very much on "lower-level" aspects of managing change during run-time evolution which, nevertheless, is a good indication of the importance of architecture-based approaches for all levels of evolution.

We have also mentioned already that we share with [8] the belief that superposition (or superimposition) is the enabling technique for a truly "black-box", "plug-and-play" approach to evolution. A basic difference between the proposed models is that, whereas our purpose is to support connectors, i.e. mechanisms for inter-

connecting components, the goal in [8] is to adapt software components. It results that [8] proposes a layered model of wrappers reflecting the successive adaptations to which the component is subjected. Instead, our proposal does not rely on wrappers but on proxys and brokers that facilitate the dynamic reconfiguration of the context in which the component is being used.

To summarise, it is clear that, by bringing to bear techniques from Coordination Languages and Models, Software Architectures, and Parallel Program Design, the notion of contract-based software evolution that we proposed in this paper shares many of the advantages of similar approaches that have been proposed in the literature exploring some of these combinations (but not all of them). The experience that we have gathered in applying the concept in real-world projects suggests that contracts are not only an effective mechanism for enabling a flexible evolution of business products, but also a powerful modelling primitive that analyst find useful at the earlier stages of domain modelling, at least in the business areas that we have already mentioned.

## 8 Concluding remarks

In this paper, we presented a discipline for software development and evolution centred around the notion of contract. Contracts were motivated by the need that we have experienced, when developing and applying tools in business areas, to separate the domain concepts (objects) from the business rules that regulate their behaviour. This separation recognises that there are two different dynamics in system evolution: changes to the way components operate and changes to the way components are interconnected. The former requires a "white box" view of components and is supported by object-oriented techniques like inheritance. The latter requires a "black box" view of components and, so far, has been lacking adequate support. The aim of contracts, as proposed in the paper, is to support this latter view towards more flexible ways of system evolution.

Contracts bring to bear techniques developed in the area of Coordination Languages and Models, Reconfigurable Distributed Systems, Software Architectures and Parallel Program Design. More precisely, contracts promote the separation of the coordination aspects that regulate the way objects interact in a system, from the way objects behave internally; they fulfil a role similar to architectural connectors in the sense that they make available these coordination features as first-class citizens. Contracts are based on superposition mechanisms for supporting forms of dynamic reconfiguration of systems. These mechanisms enable contracts to be added or replaced without the need to change the objects to which they apply.

As a consequence, we can claim that new levels of flexibility have been added to the development process, promoting *plug and play*, and a better integration and coordination of third-party, closed components (e.g. legacy systems). This, we believe, will bring us one step closer to a real industry of software components.

## References


1. G.Agha, *ACTORS: A model of Concurrent Computation in Distributed Systems*, MIT Press 1986.
2. R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM,* 6(3), 1997, 213-249.
3. L.F.Andrade, J.Gouveia, P.Xardone and J.Camara, "Architectural Concerns in Automating Code Generation", in *Proc. TC2 First Working IFIP Conference on Software Architecture*, P. Donohoe (ed), Kluwer Academic Publishers.
4. L.F.Andrade and J.L.Fiadeiro, "Interconnecting Objects via Contracts", in *UML'99 – Beyond the Standard*, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 566-583.
5. RJ.Back, L.Petre and I.Paltor, "Analysing UML Use Cases as Contracts", in *UML'99 – Beyond the Standard*, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 518-533.
6. L. Bass, P.Clements and Rick Kasman, *Software Architecture in Practice*, Addison Wesley 1998
7. G.Booch, J.Rumbaugh and I.Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley 1998.
8. J.Bosch, "Superimposition: A Component Adaptation Technique", *Information and Software Technology* 1999
9. K.Chandy and J.Misra,*Parallel Program Design - A Foundation*, Addison-Wesley 1988.

10. H.Evans and P.Dickman, "Zones, Contracts and Absorbing Change: An Approach to Software Evolution", in *Proc. OOPSLA'99*, ACM Press 1999, 415-434.
11. J.L.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming* 28, 1997, 111-138.
12. J.L.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", in *TAPSOFT'97*, LNCS 1214, Springer-Verlag 1997, 505-519.
13. J.L.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination, or what is in a signature?", in *AMAST'98*, A.Haeberer (ed), Springer-Verlag 1999
14. N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley 1996.
15. E.Gamma, R.Helm, R.Johnson and J.Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley 1995.
16. D.Gelernter and N.Carriero, "Coordination Languages and their Significance", *Communications ACM* 35, 2, pp. 97-107, 1992.
17. R.Helm, I.Holland and D.Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", in *Proc. OOPSLA'90/ECOOP'90,* ACM Press 1990, 169-180
18. S.Katz, "A Superimposition Control Construct for Distributed Systems", ACM TOPLAS 15(2), 1993, 337-356.
19. H.Kilov and J.Ross, *Information Modeling: an Object-oriented Approach*, Prentice-Hall 1994.
20. J.Magee and J.Kramer, "Dynamic Structure in Software Architectures", *in 4th Symp. on Foundations of Software Engineering*, ACM Press 1996, 3-14.
21. B.Meyer, "Applying Design by Contract",*IEEE Computer*, Oct.1992, 40-51.
22. P.Oreizy, N.Medvidovic and R.Taylor, "Architecture-based Runtime Software Evolution", in *Proc. ICSE'98*, IEEE Computer Science Press 1998.
23. D.Perry and A.Wolf, "Foundations for the Study of Software Architectures", *ACM SIGSOFT Software Engineering Notes* 17(4), 1992, 40-52.
24. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison Wesley 1998
25. M.Wermelinger and J.L.Fiadeiro, "Towards an Algebra of Architectural Connectors: a Case Study on Synchronisation for Mobility", in *Proc. 9th International Workshop on Software Specification and Design*, IEEE Computer Society Press 1998, 135-142.