Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002
©2002 Society for Design and Process Science

COORDINATION CONTRACTS AS CONNECTORS IN COMPONENT-BASED DEVELOPMENT

Leonor Barroca
Deptartment of Computing
The Open University
Milton Keynes MK7 6AA
U.K.
l.barroca@open.ac.uk

José Luiz Fiadeiro
ATX Software and University of Lisbon
Alameda António Sérgio 7,
2795-023 Linda-a-Velha
Portugal
jose@fiadeiro.org

ABSTRACT
Several proposals for component-based development methods have started to appear. However, the emphasis is still very much on the development of components as opposed to the development *with* components. The main focus is on how to generate ideal reusable components not on how to plug existing components and specify their interactions and connections.
The concept of a coordination contract (Andrade and Fiadeiro 1999; Andrade and Fiadeiro 2001; Andrade, Fiadeiro et al. 2001) has been proposed to specify a mechanism of interaction between objects based on the separation between structure, what is stable, and interaction, what is changeable. This separation supports better any change of requirements, as contracts can be replaced, added or removed dynamically, i.e. in run-time, without having to interfere with the components that they coordinate. A coordination contract corresponds to an expressive architectural connector that can be used to plug existing components.
In this paper we integrate the concept of a coordination contract with component-based development and show how coordination contracts can be used to specify the connectors between components.

1. INTRODUCTION

Component-based development (Allen and Frost 1998; D'Souza and Wills 1999; Allen 2001; Cheesman and Daniels 2001) has been put forward with the promise of more flexible systems that can be easily changed and used in an environment where time-to-market is a main constraint. Divide to conquer has been a well accepted strategy in software development for complex systems; smaller components are more manageable and can be more easily designed so that the dependencies are minimised and change in one component does not affect others. There is a lot of expertise and accumulated knowledge in software design that with, for example, the efforts on design patterns cataloguing (Gamma, Helm et al. 1995), have been recognised and lead to a well established discipline. However, component development is not only about creating the perfectly designed components; it is also very much about integrating existing software and building systems from components that sometimes do not exactly match what is initially required. The way these components are put together to achieve a desired effect is today a major part of development. The interactions between the components need to be promoted to first-class citizens if component-based development is to succeed.
Architectural approaches to software development (Shaw and Garlan 1996; Bass, Clements et al. 1998; Barroca, Hall et al. 2000) have always emphasised components and connectors but the concept of connector is often absent from component-based development methods and the emphasis is still pretty much on the development *of* components rather than development *with* components.
Connectors reflect not only how the interactions between components achieve the desired services of a system, but they should also embed what is more volatile in a system, the business rules that are more likely to change without affecting the more stable part of a system, its components.
Coordination contracts have been proposed (Andrade and Fiadeiro 1999; Andrade and Fiadeiro 2001; Andrade, Fiadeiro et al. 2001) based on the separation between the 'core services' and the layer of coordination needed between the service providers to achieve the business requirements. A coordination contract is an architectural connector between a set of partners that defines the interactions superposed on the partners' behaviour.
In this paper we look at a component-based development approach and see how it can be extended using contracts as connectors between components. Section 2 reviews the component-based development proposed by Cheesman and Daniels (Cheesman and Daniels 2001); Section 3 introduces coordination contracts; Section 4 shows through an example how coordination contracts can be used in development to plug existing components; Section 5 concludes.

2. COMPONENT-BASED DEVELOPMENT

The component-based development process followed here is based on the process proposed by Cheesman and Daniels in their book on components

(Cheesman and Daniels 2001). In their approach, the separation of domain modelling from specification modelling is key for the development of models and the clear understanding of their purpose and meaning.

Domain modelling is carried through to understand the context of a situation or business and can be carried out independently, whether or not component-based development is to be used. No software solution is assumed and its purpose is to understand the domain concepts and their relationships, and the tasks carried out in the domain, the use cases. The outcomes of domain modelling are a use case model, a conceptual model (also known as the business type model), and a behaviour model. Use cases represent different ways of using the business situation. The conceptual model represents the real world concepts and their structural properties (associations and attributes); no operations are assigned to concepts. Behaviour, when it is relevant, is expressed in terms of event occurrence and broadcast; there is not concept of message passing at this stage.

Specification modelling assumes a software system to address a situation; specification models represent software elements used in a software solution to a problem. The main focus is the definition, at a high level of abstraction, of the services provided by the components seen as far as possible as black boxes. This is where the component architecture starts to be defined and the components specified. The use case model is now refined for the software and the boundaries of the software are fixed. A structural model is defined and the external visible services of the software specified.

Use cases can lead to a first partition of behaviour of the system. Use cases can be ranked to prioritise development but also to carry out parallel development of independent areas of functionality. The external specification of services can be confronted with the specification of any existing components to solve a specific set of services. Focusing on a clear and rigorous specification of the interfaces is a key principle in the development with components. It not only makes a clear separation between the specification and the implementation of a component but it also enforces the principle of encapsulation of data and behaviour.

According to Cheesman and Daniels specification modelling is divided into three stages: *component identification* producing an initial component specification and architecture, *component interaction* discovering the operations needed and allocating responsibilities, and *component specification* creating precise specifications of operations, interfaces, and components.

They define four architecture layers for the application: the User Interface, the User Dialog, the System Services and the Business Services layers. In this paper we are not concerned with the user interface and dialog layers. The System Services layer deals with the functionality identified in the use cases and provides operations that deal with the steps in scenarios, i.e., it consists of the system interactions; the Business Services layer deals with the information and provides the operations that will be needed by the system services, i.e., it provides the stable 'core business' services. The System layer contains what is specific to each particular application; the Business layer provides services that can be common across different applications.

This separation between the system services and the business services is a first step to separate what is stable from what is variable and also what is available (the existing components) from the plugging that needs to be developed.

This is a well defined and sound proposal for component-based development that is, however, geared mainly to the production of new components, components that can be reused in later projects. It does not address in detail the use of existing assets, nor emphasises the role of connectors and their specification. In a previous exercise we followed this process to develop components and a component architecture for a video hire business system. Now we will look at how existing components can be integrated in a component architecture using coordination contracts; we propose the adoption of the concept of a coordination contract as the specification of architectural connectors between components.

3. COORDINATION CONTRACT

The concept of a 'coordination contract' (Andrade and Fiadeiro 1999) is based on the notion of superposition (Katz 1993) as in parallel program design. A coordination contract superposes a behaviour over the direct interaction between its partners by intercepting this interaction. This interception is expressed as a rule in the following way:

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

The event is typically a method invocation and the reaction specifies the set of actions of the contract and its partners that take place as long as the guard is true. The whole interaction is handled as an atomic transaction.

The separation of the business rules from the core business entities is established by shifting to the coordination contract the business constraints that will be more likely to evolve. A coordination contract has the following form.

```
contract <name>
   partners <list of partners>
   invariant <of the association between partners>
   local operations
   coordination rules
end contract
```

Coordination contracts define a set of rules imposed on coordination interfaces (the partners) which tend to deal with single objects (interfaces to an account object, for example).
Here, we will apply this idea but in component-based development, not between classes but between components, or more precisely between their interfaces. These are complex components that typically manage many objects. Coordination contracts will be used to specify the plugging of existing components and impose the constraints in the interactions that will result in the system services.

4. COORDINATION CONTRACTS AS CONNECTORS

The example used is of a video hiring business that keeps a stock of videos for lending to members. The business is operated by cashiers who deal directly with the members' requests. These requests are to borrow a video, to extend a loan, to return a video, or to reserve a film. The business also keeps track of the location of each video – that is, whether it is in the shop or out on loan.
The current constraints on the system services (loans and reservations) are as follows:

- Members cannot have more than three videos out on loan at any one time.
- A loan cannot be extended if the member has any overdue videos, or if there is a reservation for the film of the video that has not yet been allocated a video copy.
- When a reservation is allocated a video copy, the member who made the reservation is informed and the video copy is kept on hold for three days. If the video has not been collected after the three days the reservation is cancelled and the video is no longer on hold. A reservation can be cancelled before the three days after a member's request.

The constraints defined above are likely to change as the business adapts to customer requests and competition. In the rest of this paper we will be looking at how we can use this knowledge to specify an architecture where the changeable elements are easily identifiable and replaceable.
The following use cases were identified: *Make a loan*, *Extend a loan*, *Terminate loan*, *Make a Reservation*, *Cancel a Reservation*. Each use case corresponds to a system service, i.e., a specific service provided by the video hire business system. We will look into the interactions needed to perform each of these services starting from a set of existing components. The existing components correspond to the stable 'core business services' which will be plugged to satisfy the system services.
The existing components (see Figures 1, 2, and 3) are: a component with a IVideoMgt interface that manages all the videos, a component with a IFilmMgt interface that manages all the , and a component with IMemberMgt interface that manages all the members. These interfaces provide the typical operations of retrieving, setting and modifying attributes to manage instances of the 'core business types' Member, Video, and Film. We omit here their specification but hope that the names of the operations used below are self-evident.

4.1 COMPONENT ARCHITECTURE

Given the three component interfaces we need to revisit the use cases for the system services and specify the interactions between the existing components that will achieve these use cases. Each use case corresponds to an interface of a system component. We decided to associate all the resulting interfaces with a single system component as they are related and depend on the interactions between the existing core business components.
The component architecture in Figure 1 shows the dependencies between a component that satisfies the system services and the existing components for the core business services.
Now we need to specify the interactions between the core components that fulfil the interfaces of the system component. We show how this can be achieved using coordination contracts.
This system component can be seen as a connector for the other components. It acts as a dispatcher of requests delegating requests for system services to the core business components. It also plays a role of coordinator and it imposes over the connected components conditions that these need to observe. We could have had several components each for a use case. However, as all these use cases are of related services we decided to have a single component.

4.2 COORDINATION CONTRACTS TO SPECIFY COMPONENT INTERACTION

Coordination contracts have been proposed to address interaction between instances of classes. Here we will be using the same concept but at a higher level of abstraction. The interactions are to be defined between complex components that manage many object instances. These components correspond to the core business services. They can be reused for different systems in different configurations.
To achieve the system services required by a video hire business system their interactions need to be specified. We define a special component whose role is only to dispatch any request for a service to any of the existing components. A coordination contract will be imposed on each of the services of this special component. This component acts as a connector to the core business components and satisfies the following interface. Each

coordination contract will have as its partners the interface of this system component and the interfaces of the existing core components.

```
coordination interface Video-Hire-Business
import types Id, Date
services
    makeLoan(memberCode:Id,videoCode:Id, loanPeriod:
Date)
    terminateLoan(videoCode: Id)
    extendLoan(memberCode: Id, videoCode: Id,
loanPeriod: Date)
    makeReservation(memberCode: Id, filmCode: Id)
    cancelReservation(memberCode: Id, filmCode: Id)
end interface
```

The coordination contract specifies the business requirements for a system which are imposed over the behaviour of the components participating in the resulting component architecture. These business requirements are expressed in the following ways:

- as services provided by the resulting component architecture, and
- by events that trigger some change within the components, like time events or change of state.

An example of an event is *end-of-day*; it triggers the cancellation of reservations that have exceeded the three days holding a video while the video hasn't been taken out.

To be able to impose conditions on services we need to rely on a model of the component supporting an interface. Note that this may seem to equate to a break of information hiding; however, the models used are not implementation models; we don't know the internal representation of the component but we need to know how it behaves. We need to use UML models provided for each component.

Coordination contracts model business rules that are superposed over the services that are provided by the partners. A typical case is the enforcement of business invariants such as the number of loans that each member is permitted to make. This is the purpose of the with-guard of *makeLoan*. One advantage of externalising this "rule" in an explicit contract is that it allows for the rule to be changed (e.g. increasing or decreasing the number of loans, introducing temporary campaigns for attracting new customers, etc) without having to intrude in the way the service is implemented in the partner.

The *makeLoan* service also enforces the verification of the member and video codes and the non-existence of overdue videos by the member making the loan.

We show the coordination contract for the *makeLoan* service. The notation used is OCL (Warmer and Kleppe 1999) and the OCL expressions are based on the UML models for the components that are in Figures 1, 2, and 3.

```
contract Loans
    partners
    mc: IMemberMgt, vc: IVideoMgt, fc: IFilmMgt,
    vhc: Video-Hire-Business
    invariants
    local operations
    overdueVideos(memberCode:Id): Boolean is
    "there is a loan for the member with memberCode,
        and the loanPeriod has expired"
    vc.videoCopy.loan->exists(l |l.member.memberCode
              = memberCode and l.loanPeriod < today)
    coordination <interactions-with-partners>
        services
        makeLoan:
          when vhc.makeLoan(memberCode, videoCode,
loanPeriod)
          do
    vc.createLoan(memberCode,videoCode, loanPeriod)
    mc.incrementNumberLoans(memberCode)
    "if video on hold, it is on hold for a reservation for a
member with memberCode; the reservation is cancelled
and video no longer on hold"
            if (fc.checkReservation(memberCode,
vc.getReservationCode(videoCode))
            then
fc.cancelReservation(vc.getReservationCode(videoCode))
and
    vc.releaseVideoOnHold(vc.getReservationCode(video
Code)) )
          with mc.verifyMemberCode(memberCode) and
          (mc.getNumberLoans(memberCode) < 3) and
                not (overdueVideos(memberCode)) and
                vc.verifyVideoCode(videoCode)
end contract
```

5. CONCLUSIONS

In this paper we used the concept of a 'coordination contract' for a connector in a component architecture. We have also shown how use cases lead to the specification of the coordination contract. This approach complements component-based development (Cheesman and Daniels 2001) in providing the tools to reuse and integrate components in a component-based architecture.

An advantage of using coordination contracts instead of, for example, interaction diagrams to specify these interactions is that we have not imposed any sequencing of actions that could be premature and we have gathered in a single place all the interactions. Each coordination contract details a use case and allows us to represent in a single place all the business rules that relate to that use case. If any of these rules changes it will be easy to identify where to change the interactions as the rules have been separated from the components; they reside in the connectors.

There are other business requirements that are not easily represented in use cases as they are not triggered by actors (state changes, times events). These can also be expressed as part of the coordination contract between components. There is a balance that needs to be established between what are the preconditions on the business interfaces of the components and the preconditions on a coordination contract. This is a difficult balance to establish when developing components but it is no longer a choice when assembling existing components. In this case we have to rely on existing specifications for the components and the contract superimposes the required behaviour over these specifications.

The coordination contracts approach defines external coordination mechanisms to the coordinated objects. This mechanism has the advantage of being usable from the conceptual model through to design and implementation. What we did in this paper was to promote coordination contracts for objects to coordination contracts for complex components that manage many objects. Coordination contracts are used as large architectural connectors between components.

6. REFERENCES

Allen, P. (2001), "Realising e-Business with Components", Addison-Wesley Publishing Company.

Allen, P. and S. Frost (1998), "Component-Based Development for Enterprise Systems: Applying the SELECT Perspective", Cambridge University Press.

Andrade, L. and J. L. Fiadeiro (1999), "Interconnecting Objects via Contracts". *UML'99-Beyond the Standard*. R. France and B. Rumpe, Springer Verlag. LNCS 1723**:** 566-583.

Andrade, L. and J. L. Fiadeiro (2001), "Coordination: the Evolutionary Dimension". *Technology of Object-Oriented Languages and Systems-TOOLS 38*. W. Pree, IEEE Computer Society Press**:** 136-147.

Andrade, L., J. L. Fiadeiro, et al. (2001), "Patterns for Coordination". *COORDINATION'00*. G. Catalin-Roman and A. Porto, Springer Verlag. LNCS 1906**:** 317-322.

Barroca, L., J. Hall, et al., Eds. (2000), "Software Architectures, Advances and Application", Springer-Verlag.

Bass, L., P. Clements, et al. (1998), "Software Architecture in Practice", Addison Wesley.

Cheesman, J. and J. Daniels (2001), "UML Components, A Simple Process for Specifying Component-Based Software", Addison-Wesley.

D'Souza, D. and A. Wills (1999), "Objects Components, and Frameworks: The Catalysis Approach", Addison-Wesley.

Gamma, E., R. Helm, et al. (1995), "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley.

Katz, S. (1993), "A Superimposition Control Construct for Distributed Systems." *ACM TOPLAS* 15(2): 337-356.

Shaw, M. and D. Garlan (1996), "Software Architecture: Perspectives on an emerging discipline", Prentice Hall.

Warmer, J. and A. Kleppe (1999), "The Object Constraint Language, Precise Modeling with UML", Addison-Wesley.

FIGURES

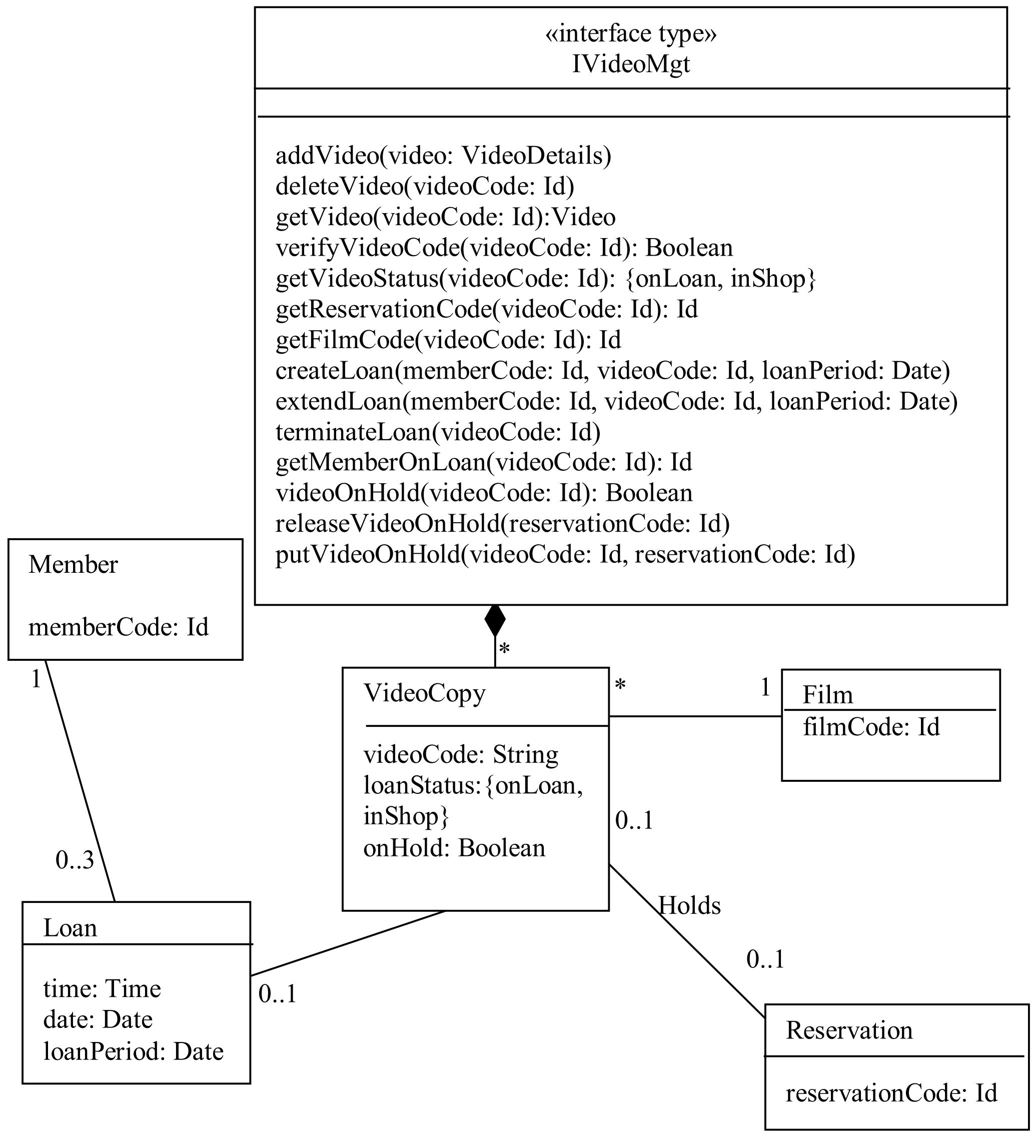
«interface type»
IVideoMgt
addVideo(video: VideoDetails)
deleteVideo(videoCode: Id)
getVideo(videoCode: Id):Video
verifyVideoCode(videoCode: Id): Boolean
getVideoStatus(videoCode: Id): {onLoan, inShop}
getReservationCode(videoCode: Id): Id
getFilmCode(videoCode: Id): Id
createLoan(memberCode: Id, videoCode: Id, loanPeriod: Date)
extendLoan(memberCode: Id, videoCode: Id, loanPeriod: Date)
terminateLoan(videoCode: Id)
getMemberOnLoan(videoCode: Id): Id
videoOnHold(videoCode: Id): Boolean
releaseVideoOnHold(reservationCode: Id)
putVideoOnHold(videoCode: Id, reservationCode: Id)
Member
memberCode: Id
1
0..3
Loan
time: Time
date: Date
loanPeriod: Date
0..1
*
VideoCopy
videoCode: String
loanStatus:{onLoan, inShop}
onHold: Boolean
*
1
Film
filmCode: Id
0..1
Holds
0..1
Reservation
reservationCode: Id

Fig. 1 Video component

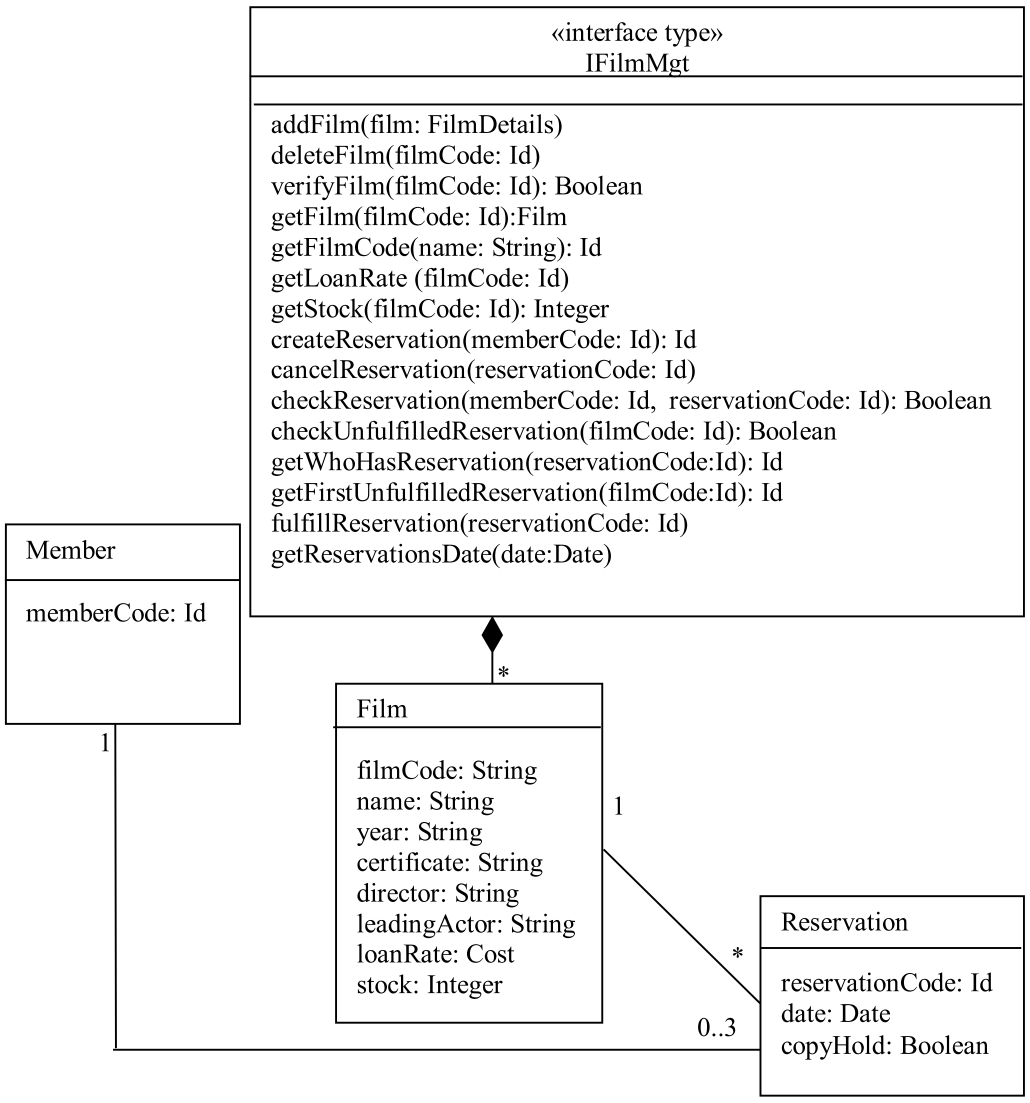
«interface type»
IFilmMgt
addFilm(film: FilmDetails)
deleteFilm(filmCode: Id)
verifyFilm(filmCode: Id): Boolean
getFilm(filmCode: Id):Film
getFilmCode(name: String): Id
getLoanRate (filmCode: Id)
getStock(filmCode: Id): Integer
createReservation(memberCode: Id): Id
cancelReservation(reservationCode: Id)
checkReservation(memberCode: Id, reservationCode: Id): Boolean
checkUnfulfilledReservation(filmCode: Id): Boolean
getWhoHasReservation(reservationCode:Id): Id
getFirstUnfulfilledReservation(filmCode:Id): Id
fulfillReservation(reservationCode: Id)
getReservationsDate(date:Date)
Member
memberCode: Id
1
*
Film
filmCode: String
name: String
year: String
certificate: String
director: String
leadingActor: String
loanRate: Cost
stock: Integer
1
*
Reservation
reservationCode: Id
date: Date
copyHold: Boolean
0..3

Fig. 2 Film component

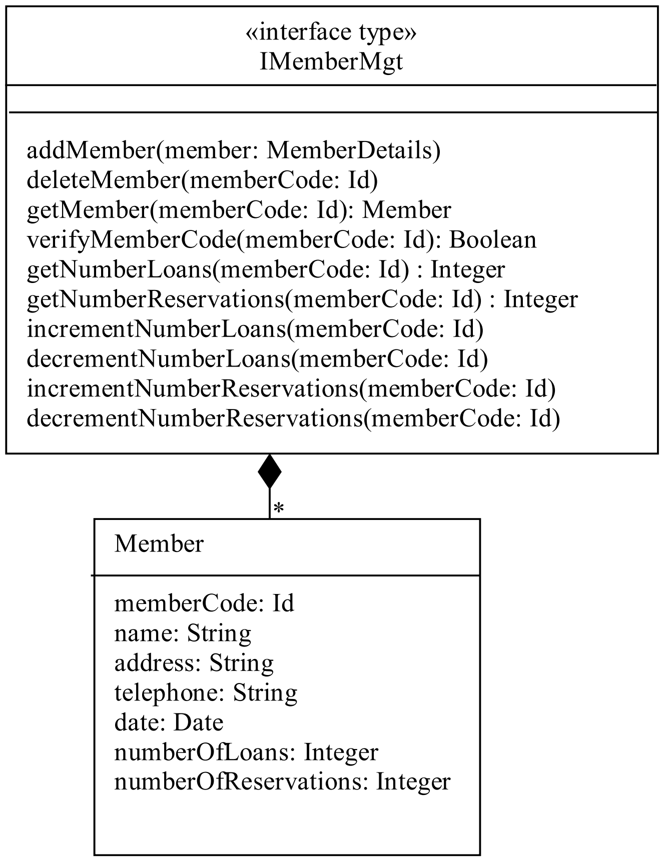
«interface type»
IMemberMgt
addMember(member: MemberDetails)
deleteMember(memberCode: Id)
getMember(memberCode: Id): Member
verifyMemberCode(memberCode: Id): Boolean
getNumberLoans(memberCode: Id) : Integer
getNumberReservations(memberCode: Id) : Integer
incrementNumberLoans(memberCode: Id)
decrementNumberLoans(memberCode: Id)
incrementNumberReservations(memberCode: Id)
decrementNumberReservations(memberCode: Id)
*
Member
memberCode: Id
name: String
address: String
telephone: String
date: Date
numberOfLoans: Integer
numberOfReservations: Integer

Fig. 3 Member component

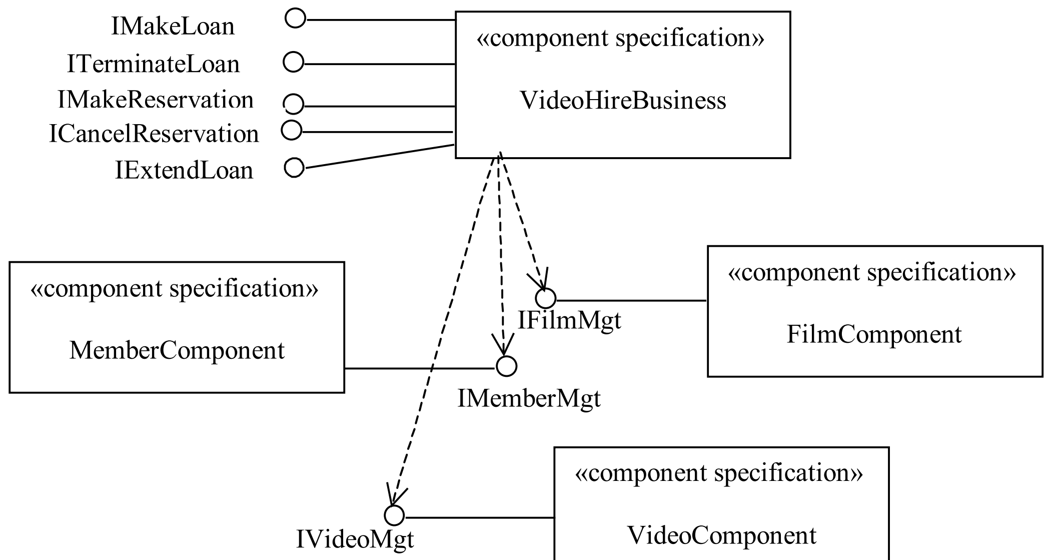

Fig. 4 Component architecture