A design pattern coupling role and component concepts: 
Application to medical software

Jean-Baptiste Fasquel\textsuperscript{a,}* , Johan Moreaub

\textsuperscript{a} LISA - Angers University, 62 avenue Notre Dame du Lac, 49000 Angers, France
\textsuperscript{b} IRCAD, 1 place de l’hôpital, 67091 Strasbourg, France

\textbf{A R T I C L E   I N F O}

Article history:
Received 9 July 2009
Received in revised form 12 January 2011
Accepted 12 January 2011
Available online 22 January 2011

Keywords:
Design pattern
Dynamic composition
Collaboration
Role
Component
Medical software

\textbf{A B S T R A C T}

One of the challenges in software development regards the appropriate coupling of separated code elements in order to correctly build initially expected high-level software functionalities. In this context, we address issues related to the dynamic composition of such code elements (i.e. how they are dynamically plugged together) as well as their collaboration (i.e. how they work together). We also consider the limitation of build-level dependencies, to avoid the entire re-compilation and re-deployment of a software when modifying it or integrating new functionalities. To solve these issues, we propose a new design pattern coupling role and component concepts and illustrate its relevance for medical software. Compared to most related works focusing on few role concepts while ignoring others, the proposed pattern integrates many role concepts as first-class entities, including in particular a refinement of the notion of collaboration. Another significant contribution of our proposal concerns the coupling of role and component concepts. Roles are related to the functional aspects of a target software program (composition and collaboration of functional units). Components correspond to the physical distribution of code elements with limited build-level dependencies. As illustrated in this paper, such a coupling enables to instantiate a software program using a generic main program together with a description file focusing on software functionalities only. Related code elements are transparently retrieved and composed at run-time before appropriately collaborating, regardless the specificity of their distribution over components.

© 2011 Elsevier Inc. All rights reserved.

1. Introduction

This paper proposes a design pattern addressing three challenging issues, corresponding to the dynamic composition of separated code elements, their collaboration and the reduction of build-level dependencies.

Dynamic composition (i.e. at run-time) is a useful feature for plugging software components together (Bass et al., 1998; Zdun, 2006). Unfortunately, composition is often static (i.e. at compile time). Such lack of dynamicity concerns the aggregation or composition with specific classes or objects, traditional C++ and Java inheritance relationships, generative programming approaches with hard coded template parameterization (Czarnecki and Eisenecker, 1999). The consequence is that a large part of a program must often be adapted (re-written) to composition or API (Application Programming Interface) changes.

Many approaches have been proposed for dynamic composition. Aspect oriented programming (AOP) aims at injecting some code elements at run-time within class methods (class-level composition), changing the behaviour of these methods for all instances of a given class (Villazon et al., in press). Mixin-based composition techniques also focus on run-time composition at class-level, except in the recent proposal coupling both class-level and object-level composition (Zdun et al., 2007). Some other approaches favour composition at object-level, as in our case. The well-known decorator pattern (Gamma et al., 1995) aims at changing the behaviour of object methods at run-time without subclassing. Delegation-based approaches (Lieberman, 1986; Bettini et al., in press) regard dynamic composition as a way to dynamically change or define the behaviour of object methods. Role-based approaches address object-level composition using many concepts being out of the scope of other composition alternatives (Kristensen and Osterbye, 1996; Steimann, 2000; Herrmann, 2007; Baldoni et al., 2007; Colman and Han, 2007). As recently underlined (Herrmann, 2007), due to the richness of these concepts, existing role-based implementation proposals often focus on some of them while ignoring others.

Collaboration represents a challenging issue dealing with the coordination or chaining of the execution of separated code elements. This notion is sometimes called method chaining, method resolution order, linearization of method invocation (Zdun et al., 2007). In many cases, this issue is addressed using a dedicated
entity (e.g. behavioural link (Tanter et al., 2008) or coordination aspect (Fuentes and Sanchez, 2007) in AOP, a mixin list for mixin-based composition (Zdun et al., 2007), a Team (Herrmann, 2007) or a coordination role (Colman and Han, 2007) in recent role-based proposals). As underlined in this paper, the use of a dedicated entity (explicit chaining) is appropriate when the chaining can be foreseen. Unfortunately, this is not necessarily the case in event-based software programs involving a graphical user interface (GUI), as we consider in this paper, where end-user interactions are unforeseen and therefore may become hard to model into a dedicated entity.

The limitation of build-level dependencies also represents a challenging issue. Build-level dependencies can, for instance, result from a link dependency between the main program and dynamic libraries providing functionalities. To integrate any newly developed code element (embedded in new dynamic library), the re-compilation as well as the re-deployment of the entire program can therefore be required, involving the need of a development environment. Component-based development, widely used for software composition (Szyperski, 2002), represents an interesting paradigm to reduce build-level dependencies (i.e. link dependencies). Practically, it can enable to integrate a new dynamic library (provided by a component) without recompiling the main program. A difficulty is to appropriately define components, characterized by well defined interfaces and explicit dependencies (Szyperski, 2002; Lau and Wang, 2007), so that the main program can retrieve, at runtime, compiled code elements which are subject to composition and collaboration with other code elements.

In this paper, we propose a design pattern coupling role and component concepts, for dynamic composition at object-level and collaboration, with limited build-level dependencies. As further detailed in Section 4, we claim four contributions: The integration of many role concepts as first-class entities of the pattern, including the notion of collaboration generally ignored by related role-based approaches (Herrmann, 2007). The refinement of the notion collaboration, involving two types of chaining (explicit and implicit) as first-class entities of the proposed pattern. The integration of the notion of component, where interfaces and dependencies are directly mapped on role entities. A particular component description formalism, involving a graph-based representation, is proposed for the transparent retrieval of functional units of composition (i.e. roles), regardless their distribution over components providing them. To our knowledge, the coupling of both role and component concepts has never been considered, except in the context of a development methodology (Parka et al., 2004), this being out of the scope of this paper. The application of the proposed pattern for developing a set of software programs dedicated to medical applications. These applications are built using a generic main program which, according to a description file, retrieves and composes functionalities provided by software components so that they appropriately collaborate at run-time. As illustrated, composition and collaboration can evolve at run-time, according to end-user interactions.

Section 2 presents the proposed design pattern. Section 3 focuses on its application to medical software. Before concluding, Section 4 compares our proposal with some recent related works.

2. Design pattern

The design pattern is described according to three essential elements considered (Gamma et al., 1995) for specifying a pattern. The first element concerns the problem addressed by the pattern (Section 2.1), indicating when to apply it. The second element, detailed in Section 2.2, regards the proposed solution (i.e. the different elements of the design). The third element is related to the consequences (i.e. the result of applying the pattern), being summarized in Section 2.3.

Note that, according to the classification considered (Gamma et al., 1995), the proposed design pattern has both structural and behavioural purposes, corresponding to composition and collaboration. Moreover it focuses on object (scope criterion; Gamma et al., 1995), because dynamic composition is considered at object-level.

2.1. Problem

The problem regards issues addressed in this paper (i.e. regarding dynamic composition, collaboration and build-level dependencies).

The proposed pattern aims at being applied for developing applications involving many various functionalities that may be unknown to each other and which should be composed at run-time and should correctly collaborate in order to provide initially expected high-level software functionalities. Moreover, this pattern concerns situations where code elements are embedded in components (limitation of build-level dependencies) and should be transparently retrieved, regardless the identity and location of components providing them.

Note that this pattern aims at being particularly appropriate for developing a set of software programs (rather than a single one) sharing common functionalities and data structures to be reused from one application to another. This is illustrated in Section 3 for a set of medical software programs involving various functionalities (e.g. graphical user interface, data reading/writing, 3D visualization and data processing) together with several kind of data structures (e.g. 3D image, video), all of them being provided by components.

2.2. Solution

Fig. 1 provides a class diagram reporting participants involved in the proposed design pattern (Gamma et al., 1995). The ConcreteBase class correspond to data structures. The Role class regards the notion of functionality. The ConcreteRoleType class is related to a functionality type (e.g. notion of data reading or 3D visualization) and the ConcreteRole class represents a particular implementation of a given functionality type (e.g. a specific data reader). The Object class enables to bind functionalities and data structures.
Fig. 2. Design pattern, concepts and issues. Class diagram providing an overview of role («role») and component («component») concepts involved in the proposed design pattern for issues («issue») regarding composition (Composition) and collaboration (Collaboration) with limited build-level dependencies (Build-level dependency). Relationships (explained in Sections 2.2.1 and 2.2.2) summarize how each concept interacts with (or participate to) issues and/or other concept(s).

Details of the proposed pattern, including in particular those related to Fig. 1, are given in next Sections 2.2.1 and 2.2.2, focusing respectively on role and component concepts on the top of which the pattern is built. Role concepts essentially correspond to the part of the proposed pattern solving dynamic composition and collaboration issues. Component concepts pattern mainly focus on the limitation of the build-level dependencies.

The way these concepts, representing the foundations of the proposed design pattern, contribute to solving considered issues is reported in Fig. 2, being also detailed in Sections 2.2.1 and 2.2.2.

2.2.1. Role concepts

First, we present role concepts mainly focusing on dynamic composition. The Identity feature concerns composition units and result. The Dependency feature regards dependencies between composition units, as well as composition lifetime. Other features also related to dynamic composition correspond to Dynamicity (dynamic composition and de-composition), Multiplicity (composition of several similar functionalities), Transitivity (composition of composition) and Migration (dynamic re-composition).

Then, we consider the two role concepts, namely Explicit chaining and Implicit chaining, focusing on collaboration.

Finally, we present the two last role concepts involved in the proposed pattern. They concern both composition and collaboration issues, and are related to Activation and Abstractivity.

Note that role concepts commonly considered are Identity, Dependency, Dynamicity, Multiplicity, Abstractivity and Transitivity (Kristensen and Osterbye, 1996). Activation and Explicit chaining have been recently proposed for collaboration (Herrmann, 2007). Implicit chaining concerns our proposal for managing unforeseen collaborations.

2.2.1.1. Identity. An object (or subject; Kristensen and Osterbye, 1996) is initially considered as a base-object (ConcreteBase in Fig. 1), being restricted to intrinsic methods and attributes (Kristensen and Osterbye, 1996; Herrmann, 2007). As traditionally considered (Herrmann, 2007), composition regards the fact that the base-object can acquire role(s), including methods and (dynamic) attribute(s) (field in Fig. 1). Acquired methods correspond to methods of Role, ConcreteRoleType and ConcreteRole classes in Fig. 1. The composition result is illustrated by the example of subject reported in Fig. 3.

A dynamically composed subject appears as a monolithic unit (base-object with attached roles, including intrinsic and dynamic attributes), having its own identity, which can present different perspectives or roles to other objects (or subjects) manipulating it or interacting with it. An object has or plays a given role if one instance of such a role is attached to it. As a consequence, a single reference on the base-object is enough to retrieve composition units (get() method of the Object class in Fig. 1), corresponding to attached roles (Fig. 2 ("single reference")).

Within this monolithic entity, the union of all attributes (intrinsic base-object and role attributes, as well as dynamic attributes) constitutes the overall state of the subject (Herrmann, 2007). Base-object’s attributes (both intrinsic and dynamic ones) are used by attached roles to collaborate (Fig. 2 ("base object’s attributes"): if the role execution depends on attributes of another role, this other role assigns these attributes to its base-object, as dynamic attributes. This is further discussed in next paragraphs focusing on collaboration-based role concepts (Implicit chaining and Explicit chaining). It must be pointed out that dynamic attributes can be seen as base-objects (similarly to ConcreteBase) to which roles

Fig. 3. Design pattern and composition: subject example (object diagram).
can be attached: a subject can therefore be composed of subject(s) (Fig. 2 (“subject of subject”)).

As discussed in (Steimann, 2000; Herrmann, 2007), each role keeps its own identity (e.g. the concrete role entity appears distinct from the base-object one in both Figs. 1 and 3). Although appearing contradictory as roles belong to a single monolithic entity (Herrmann, 2007), this is required for collaboration (Fig. 2 (“role identity”)), as underlined in next paragraphs regarding collaboration-based role concepts.

2.2.1.2. Dependency. As reported in Kristensen and Osterbye (1996), this feature regards the fact that a role cannot exist without its base-object, and, to ensure the separation of concerns, a (base) object is independent of any role (Fig. 2 (“base object to role independence”)).

The role to base-object dependency remains strong because a role interacts with its base-object and therefore depends on its methods and attributes. In Fig. 1, this is represented by the dependency between ConcreteRole and ConcreteBase («base object type» dependency) and the navigability of the association between Role and Object classes. This feature has an impact on the management of object lifetime (Fig. 2 (“composition lifetime”)): base-object destruction involves the implicit detachment and destruction of attached roles (together with dynamic attributes). This is represented by composition relationships in Fig. 1.

2.2.1.3. Dynamicity. This feature regards both dynamic composition and decomposition (Fig. 2 (“composition–decomposition”)): roles (and related dynamic attributes) can be attached (composition) and detached (decomposition) during the lifetime of the object (add() and remove() methods of the Object class in Fig. 1).

2.2.1.4. Multiplicity. This feature corresponds to the ability to attach several instances of the same concrete role to a given base-object (Fig. 2 (“multiple similar roles”)), each role instance having its own state (i.e. value of its attributes). In Fig. 3, this could be represented by two instances of ConcreteRole attached to the same ConcreteBase instance.

2.2.1.5. Transitivity. This notion regards the ability to attach a role to another role, being considered as a base-object (Fig. 2 (“role of role”)). In the proposed pattern, a role can be seen a base-object as the ConcreteRole inherits from Object similarly to the ConcreteBase class (see Fig. 1).

2.2.1.6. Migration. This feature regards the ability to move (move method in Fig. 1) one role from a base-object to another (the role being temporarily unattached). This can be seen as a way to recompose (Fig. 2 (“recomposition”)) base-objects and roles without destruction and (explicit) detachment. Compared to the Dynamicity feature (i.e. composition/decomposition), Migration avoids losing the role state.

2.2.1.7. Explicit chaining. This feature regards the collaboration of roles using a specific entity, as considered in (Herrmann, 2007; Colman and Han, 2007). The proposed pattern manages it using a dedicated role (coordination role, being a particular ConcreteRole) having a reference on coordinated roles, as illustrated in Fig. 4(a). The coordination role manages the invocation of update() method (see Role class in Fig. 1) on collaborating roles according to a given order, as illustrated by Fig. 4(b) reporting interactions between involved entities. Role execution (update() communication message) involves the modification (modify() communication message) of the value some base object’s attributes (ConcreteBase instance).

Note that for collaboration purposes, a role can dynamically assign dynamic attributes to its base-object, as previously underlined (paragraph related to the Identity role feature). In Fig. 4(a), ConcreteroleField represents a dynamic attribute which could be assigned and modified by ConcreteRoleA (modification resulting from the update() method execution) and then used by ConcreteRoleB so that both roles appropriately collaborate.

As underlined in Section 1 and further illustrated in next Section 3, such chaining regards situations where collaboration can be foreseen (Fig. 2 (“foreseen”)). Note that this feature also participates to composition through the use of a dedicated coordination role (Fig. 2 (“coordination role”)).

2.2.1.8. Implicit chaining. This concept concerns situations where collaboration cannot be foreseen (Fig. 2 (“unforeseen”)) and therefore not easily modelled using a dedicated entity, this being illustrated and discussed in Section 3.

In the proposed design pattern, Implicit chaining is based on an observation mechanism («observation» relationship in Fig. 1, related to the Observer design pattern (Gamma et al., 1995)). As reported in terms of objects by Fig. 4(c), an observing role has an observation link with its base-object.

The idea is to define how a role behaves according to base-object changes, independently from the composition context (i.e. other attached roles). Roles naturally and implicitly collaborate after attachment: a role performs some computations, which may involve the modification of some base-object attributes (intrinsic and/or dynamic ones). This is represented by the modify() communication message sent by the ConcreteRoleA entity in Fig. 4(d). These modifications are notified to observing roles (update(msg) communication message in Fig. 4(d), and update(msg) method in Fig. 1). Then, these observing roles can perform some other tasks according to modifications reported in msg (modify() communication message sent by the ConcreteRoleB entity in Fig. 4(d)).

2.2.1.9. Activation. This feature is ignored except in Herrmann (2007) to trigger the attachment of roles. Compared to Herrmann (2007), we assume that roles are already attached. In the proposed pattern, this feature is considered for (un)initialization (Fig. 2 (“(un)initialization”)) such as (de)allocation of some resources ((un)initialization of a composition unit).

As reported in Fig. 2 (“enable–disable”), it is also used to enable (or disable) a role so that it processes (or ignores) requests (invocation of update() or update(msg) method, depending on the considered chaining). Practically, this allows to (temporarily) disable the participation of a role to any collaboration without detachment (and therefore destruction), which could otherwise involve the loss of its state (value of its attributes). Conceptually, a subject can have a role without playing it (temporarily). In the proposed pattern, this feature corresponds to both start() and stop() methods of the Role class reported in Fig. 1.

2.2.1.10. Abstractivity. This feature regards the notion of role type (Zdun et al., 2007; Kristensen and Osterbye, 1996), corresponding to the classification of roles in functional groups. In the proposed pattern, this feature corresponds to three levels of the class hierarchy, namely Role, ConcreteRoleType and ConcreteRole (see Fig. 1).

Abstractivity at Role-level enables to summarize several role concepts, considered as first-class entities of the pattern: Migration (Fig. 2 (“Role::move()”)), Explicit chaining (Fig. 2 (“Role::update()”)), Implicit chaining (Fig. 2 (“Role::update(msg)”)), Activation (Fig. 2 (“Role::start()” and Role::stop())).

This facilitates the manipulation of roles in an abstract manner, regardless their type (ConcreteRoleType-level) and particular implementation (ConcreteRole-level).
Fig. 4. Design pattern and collaboration. Explicit chaining (a and b diagrams) and implicit chaining (c and d diagrams) of two roles represented using two object diagrams (a and c) and two communication diagrams (b and d). The considered chaining concerns the execution of a role of type \texttt{ConcreteRoleA} first and then the execution of a role of type \texttt{ConcreteRoleB}, both of them being attached to a base object of type \texttt{ConcreteBase}. In communication diagrams, the \texttt{Client} entity represents a code element in charge of triggering chainings. The \texttt{modify()} communication message regards modifications of a base-object by one of its roles.

Abstractivity at \texttt{ConcreteRoleType}-level regards the management of role concepts for a specific type of functionality (this being illustrated in Section 3). It also concerns interactions with functional groups (Fig. 2 (“functional group”)) regardless concrete roles. For instance, this can correspond to the detachment or (de)activation of an entire functional group, consisting in several roles of given role type (i.e. inheriting from the same \texttt{ConcreteRoleType}).

2.2.2. Component

Role entities reported in Fig. 1 are split over components (see Fig. 5) in order to reduce build-level dependencies. A component is a entity embedding at least a description of its content (description file) and a dynamic library. \texttt{Object} and \texttt{Role} entities are provided by a central \texttt{layer} component. Role types, concrete roles and base-objects are provided by other components, without any particular constraints regarding their distribution.

Details regarding this aspect of the proposed pattern are given hereafter in terms of component interfaces and dependencies (Szyperski, 2002; Lau and Wang, 2007) as well as in terms of description constraints. These three component features participate to the reduction of build-level dependencies, as reported in Fig. 2 by relationships between the three «component»-typed classes and the «build-level dependency» one.

2.2.2.1. Interface.

A component may require and/or provide interfaces (Lau and Wang, 2007), depending on embedded role entities. In the proposed pattern, interfaces are related to \texttt{Role}, \texttt{ConcreteRoleType} and \texttt{ConcreteRole} entities, as reported in Fig. 5(a).

The \texttt{Role} entity is associated with a required interface defined by a set of methods invoked by \texttt{Role} class methods (see Fig. 1). For instance, the \texttt{update()} method invokes an interface method to be implemented by a component providing a concrete role type (\texttt{ConcreteRoleType} entity). This can be seen as a translation of role concepts at concrete role level (\texttt{ConcreteRole} entity).

Fig. 5(b) illustrates how components can be wired for a given composition (Fig. 6(a)).
As a result, component interfaces are associated with the abstractivity role feature, as they correspond to the same three levels in the class hierarchy (i.e., Role, ConcreteRoleType and ConcreteRole). This relationship between both concepts is indicated by Fig. 2 (“interfaces”). Fig. 2 (“implementation”) represents interfaces to be implemented, involving build-level dependencies resulting from the required compatibility between declared interfaces and related implementations.

2.2.2.2. Dependency. A component may have dependencies (Lau and Wang, 2007) with respect to other components (ComponentDependency in Fig. 2), depending on embedded role entities. In the proposed pattern, dependencies result from inheritance relationships and ConcreteRole to ConcreteBase dependency (see Fig. 1).

In Fig. 5(a), the central layer component has no dependencies with respect to other components. Both componentA and base components have a dependency with respect to the layer component, and componentB depends on both componentA and base.

Fig. 6(b) illustrates components involved in the considered composition, where underlying dynamic libraries have to be loaded, taking their dependencies into account (e.g., componentB depends on base), so that both ConcreteBase and ConcreteRole can be instantiated and finally linked to compose the expected subject (Fig. 6(a)).

In terms of concepts, Fig. 2 indicates that both dependency and abstractivity role concepts contribute to component dependencies (Fig. 2 (“inheritance relationship”)) and (“role to base object relationship”)). Fig. 2 (“link”) regards link dependencies between dynamic libraries embedded in components.

2.2.2.3. Description. As previously underlined, except Object and Role entities provided by a central component (layer component in Fig. 5(a)), other role entities can be spread over components in different ways. For instance, a single component could be considered in Fig. 5(a) instead of three ones to embed ConcreteRoleType, ConcreteRole and ConcreteBase entities.

A part of the proposed design pattern concerns the abstraction of the component distribution with respect to embedded base-objects, role types and concrete roles that are involved in compositions. The underlying motivation is to be able to manage compositions (base-objects with attached roles) transparently, regardless of the specificity of the distribution of components. A composition request can therefore remain focused on base-objects and roles, without explicitly indicating required components.

The considered design pattern proposes to support this abstraction by imposing a particular component description model where a description file declares each provided entity (being either a base-object, a concrete role type or a concrete role) as a node of a graph (see Fig. 5(b)). The resulting graph structure binds role and component entities, as reported in Table 1.

<table>
<thead>
<tr>
<th>Graph node</th>
<th>Role</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConcreteBase</td>
<td>base-object</td>
<td>–</td>
</tr>
<tr>
<td>Role</td>
<td>role type</td>
<td>required int</td>
</tr>
<tr>
<td>ConcreteRoleType</td>
<td>concrete role</td>
<td>required int</td>
</tr>
<tr>
<td>ConcreteRole</td>
<td>concrete role</td>
<td>provided int</td>
</tr>
</tbody>
</table>

Graph edge

| ConcreteBase → Object | inheritance     | build-level dep   |
| Role → Object         | inheritance     | build-level dep   |
| ConcreteRoleType → Role | inheritance   | build-level dep   |
| ConcreteRole → ConcreteRoleType | inheritance | build-level dep   |
| ConcreteRole → ConcreteBase | base-object type | build-level dep   |

On one side, according to nodes and edges associated with a given component (each node knows its associated component), we can deduce its required interfaces, those it provides, as well as its dependencies with respect to other components. On the other side, we can deduce whether a given component provides a role type, a concrete role, a base-object or even a mix of all.

In Fig. 6(a), the composition involves a base-object of type ConcreteBase and a concrete role of type ConcreteRole. From graph analysis (see Fig. 5(b)), one can easily retrieve components whose dynamic libraries embed ConcreteBase (base) and ConcreteRole (componentB). According to graph edges, build-level dependencies are identified and appropriate dynamic libraries are loaded (i.e. those provided by layer and componentA components). Additionally, components are appropriately wired (see Fig. 6(b)).

This part of the design pattern is associated with the Description entity in Fig. 2. Related relationships describe the fact that, according to a given composition request (Fig. 2 (“role entity request”)), the retrieval of a role entity (Fig. 2 (“role entity retrieval”)) is performed through the analysis of the graph built from component descriptions.

2.3. Consequences

As further illustrated in next Section 3, using the proposed design pattern has several consequences: An entire application can be defined as a subject being recursively composed of subjects, each subject corresponding to a base-object (data structure) with attached roles (functionalities). This is illustrated in next Section 3, where an application is declared by specifying data structures and functionalities (i.e. subject(s)) using a simple tree-like structure provided as a configuration file to a generic main program. As further underlined in next Section 3, the composition can evolve at run-time. The generic main program has no build-level dependencies with respect to components providing data structures and functionalities. A functionality (or a data structure) can be integrated in an application by simply ensuring that the related component can be accessed by the application (i.e. reading its content and, in particular, loading the dynamic library). Thanks to the graph-based description, the functionality (or the data structure) can be transparently retrieved without explicitly indicating the related component. A functionality can implicitly collaborate with other functionalities by appropriately defining how it reacts according to possible modifications of the state (i.e. value of attributes) of the data structure it is attached to. A specific functionality can be used to manage the explicit collaboration of functionalities (i.e. how they work together). This specific
Fig. 7. Snapshot of the gui of the VR-Render freeware, rendering an image (middle-left and bottom-right rendering areas) and 3D surface models (middle-right rendering area), according to the end-user selection (top panel).

functionality (coordination role) is considered as any other functionality (i.e. can be integrated as any other functionality). The major drawback regards the required understanding of role concepts and their appropriate translation into a specific application domain (i.e. mapping general role concepts to functionality types and particular implementations). Next Section 3 provides examples of such a translation between these concepts and practical functionalities.

3. Case study

The proposed design pattern has been applied to medical software. Section 3.1 describes the application domain and the set of developed software programs. Section 3.2 presents an overview of the implementation of the proposed design pattern and three illustrating examples regarding functionalities which have been widely used in considered software programs. Section 3.3 discusses the relevance of the proposed pattern coupling both role and component concepts for developed applications.

3.1. Application domain and developed software programs

This application domain concerns a large community of computer scientists developing medical systems and software for computer aided diagnosis and therapy (DiMaio et al., 2007; Brown and McNitt-Gray, 2000; Duncan and Ayache, 2000; Taylor and Stoianovici, 2003). Considered software programs are often expected to integrate many kind of data structures (e.g. patients, images, videos and 3D surface models) and to combine various functionalities such as graphical user interface (gui), data reading/writing (io), 3D rendering (visualization), processing algorithms (e.g. medical image analysis for diagnosis purposes), video-based tracking (e.g. tracking of surgical tools in computer aided surgery) and medical robotics (see Fasquel et al., 2008, 2009 for examples).

Several medical applications have been developed on the top of this pattern. The VR-Render¹ freeware (see Fig. 7) is an advanced software focusing on visualization, together with gui, io and some processing functionalities. Involved data structures are patients, medical images and 3D surface models. The term “advanced” means that the application provides a large number of sophisticated functionalities with a specific effort on usability (International Organization for Standardization, 2010).

In addition to this advanced software, several basic applications (i.e. without specific effort on usability (International Organization for Standardization, 2010)) have also been developed. They were prototypes developed for research purposes or for testing many functional compositions covering our application field: BasicViewer focuses on the visualization of 3D medical images, together with gui and io functionalities (illustrated in next Section 3.2.3), BasicProcessor concerns medical image processing, together with BasicViewer’s functionalities (illustrated in next Section 3.2.4), BasicAugmentedReality involves 3D surface models, video processing, io, gui, visualization and tracking (see Fig. 8). BasicRobot aims at driving a medical robot (see Fig. 9), including several data structures (video, 3D surface models and robot) and functionalities (video processing, io, gui, visualization, tracking and command).

3.2. Design pattern implementation: overview and examples

3.2.1. Implementation overview

The proposed design pattern has been implemented using the C++ language, and represents the core of the fw4spl² open source project. The long-term purpose of this project is to provide a software product line for medical applications (fw4spl stands for framework for software product line). Note that some implementation details provided in next sections may slightly differ from the code available on the fw4spl website, either for clarity purposes or because fw4spl’s code regularly evolves.

Several concrete base-objects (ConcreteBase in Fig. 1) have been developed, for the different data structures previously mentioned.

Several concrete role types (ConcreteRoleType in Fig. 1) have been implemented to manage the different functionalities supported in developed software programs. Next sections provide

examples regarding reading (Section 3.2.2), rendering (Sections 3.2.3 and 3.2.4), processing (Section 3.2.4) and gui (Section 3.2.4).

Several concrete roles (ConcreteRole in Fig. 1) have been developed for the different concrete role types and concrete base-objects. The largest number of concrete role per concrete role type have been developed for image, as it represents a central data structure of our application domain.

In the proposed implementation, a component is a directory containing a XML-based description file (examples are provided in next Section 3.2.2), a dynamic library as well as optional elements such as icons.

A central layer component, required whatever the medical software, has been implemented (similar to the layer component reported in Fig. 5). The underlying dynamic library (represented by the layer package in Fig. 5) implements all previously presented concepts regarding roles and components. For practical reasons, it also provides a container keeping a reference on all instantiated subjects to facilitate the retrieval of the instantiated roles, regardless their related base-object.

Other components provide the different concrete base-objects, concrete role types and concrete roles.

A software instance is defined by a root subject instance, declared in a XML-based description file provided to a generic launcher (or generic main program) depending on the layer package only.

3.2.2. Example: reading

This example regards the reading, from the filesystem, of a 3D medical image. Medical images are often stored in DICOM\(^3\) format (DICOM stands for Digital Imaging and Communications in Medicine). Practically, a 3D medical image is stored in a repository as a set of 2D image files, as illustrated in Fig. 10. In addition to pure image information, these files provide medical information about the patient, the hospital and the acquisition protocol for instance. Due to the variability of the information and the specificity of imaging devices, many various reading algorithms exist.

Components involved in this example are reported in Fig. 11. The io component regards the role type (the ConcreteRoleType corresponds to the IReader class) dedicated to data reading (and writing although not reported for clarity). The associated library (corresponding to the io package) embeds common functionalities required by concrete reading roles (e.g. management of interactions with the filesystem regardless the specificity of both concrete reading roles and concrete base objects). It provides an interface compliant with the Role one. It also declares a required interface as a contract to be fulfilled by components providing specific reading roles. The dicom component provides a concrete reading role and therefore an interface fulfilling the IReader required one declared by the io component. This interface is associated with the internal Dicom class, which depends on the appropriate base-object class (Image class, provided by the data component, because the DICOM-based reading role is dedicated to images).

As previously said, interfaces map role concepts related to Role methods at both concrete role type and concrete role levels. For instance, the updateRole invokes its pure virtual updating() method (part of the required interface) if the role is activated (start must have been called). The IReader class implements this updating() method (part of the provided interface) which, in turn, invokes its pure virtual read() method to be implemented by the Dicom class (part of the provided interface). The read() method is called only if, in this case, the location of 2D image files is valid (this being checked by the IReader::updating() method).

The dicom component depends (build-level dependency) on the data component as well as on the component regarding its associated concrete role type (i.e. io component in this case).

Component description files are also reported in Fig. 11. As illustrated, a description file may report several information: \(<\text{plugin id="..."/> regards the component identifier (e.g. io). \(<\text{library name="..."/> concerns the name of the C++ dynamic library (e.g. libio). \(<\text{point.../> focuses on the declaration of Object, Role, concrete base-object type (e.g. Image), concrete role type (e.g. IReader) or concrete role (e.g. Dicom). Static inheritance relationships and role to base-object dependency are declared using implements XML elements.

The graph-based representation recommended by the proposed design pattern results from point XML elements, as illustrated by Fig. 12. Point XML elements are the nodes of the graph, where edges are defined by implements element values. As previously said, nodes could be spread over components in different ways, according to the component distribution. For instance, the io component could also provide a default image reader (default concrete role), which would lead to an additional node in the graph.

An example of root subject declaration, to be provided to the generic launcher, is reported in Fig. 13(a). This example involves a base-object of type Image to which one wishes to attach one role (instance of the Dicom Class). The reading functionality is seen as a role played by an image, as illustrated by the object diagram reported in Fig. 13(b), resulting from the instantiation of the subject. Fig. 13(c) provides an additional diagram showing components involved in the composition, including their appropriate wiring.

---

\(^3\) http://medical.nema.org/ (January 2011).
Hereafter, we propose to shortly describe the instantiation procedure of a software in the case of this example. Before instantiating the subject, the graph representation previously described is built from the analysis of the component description files, where components are organized as a set a directories on a filesystem. From the analysis of the graph, build-level dependencies are identified in order to start appropriate components. In this example, it concerns the data component because the subject’s concrete base-object type is Image and the dicom component because the concrete role is of type Dicom. Note that starting a component means loading a dynamic library and performing some registrations in a factory maintained by the layer component, so that classes declared by components can be instantiated (e.g. an image can be instantiated after having started the data component). As the graph indicates the (build-level) dependency of the dicom component with respect to the io one (Dicom to IReader edge), the io component is started first (the layer component being already started). After having started appropriate components, both base-object and concrete role can be instantiated and attached.

Note that the state or configuration (value of attributes) of base-objects and roles can be specified in root subject declarations. This is illustrated in Fig. 13(a) for the reading role with location XML element indicating the location of medical image files to be read. Although not detailed in this paper, such a configuration is performed by passing an abstract parameter to the Role entity.

An image can finally be read (according to the specified location) by invoking the update() method of the reading role: this leads to the updating of image attributes such as the data buffer.

3.2.3. Example: reading and rendering

This example concerns the combination of image reading and rendering functionalities, as illustrated in Fig. 14. In this application domain, different techniques can be used to display 3D medical images. The multi-planar rendering (Mpr for short) is one of the most classical technique. It consists of rendering slice(s) (i.e. 2D image(s)) of the full set of 3D points, each rendered slice being defined by a slice index the end-user can interactively modify to get across the full 3D data set. Other rendering techniques exist such as the volume rendering one (Vr for short), for displaying an entire 3D medical image. Note that both rendering techniques are supported by the VR-Render freeware. In this example, it is
Fig. 14. Example combining reading and rendering functionalities: principle. Rendering functionalities (roles) encapsulate end-user interactions allowing to change rendering properties such as the slice index of the horizontal plane of Mpr roles.

expected that rendering areas automatically refresh after reading. As illustrated in Fig. 14, it is also expected that both Mpr-based rendering roles are synchronized with respect to the horizontal plane which can be modified through end-user interactions. If the end-user modifies the slice index of the horizontal plane to be rendered in one area, it is expected that the associated plane in the other area automatically changes according to this modification. Note that the end-user can perform modifications either in the Mpr-based rendering area configured (role state) to render three planes or in the one configured to render the horizontal plane only.

Fig. 15 reports the XML declaration of the subject (Fig. 15(a)), together with the resulting object diagram (Fig. 15(b)). Observation links are set using the com XML element (com stands for communication). These links allow to automatically refresh rendering areas after image reading: an event is raised by the reading role and caught by rendering ones which can refresh according to the new image buffer content. Observation links also enable to synchronize rendered horizontal planes according to end-user interactions. Planes are defined by a common slice index being stored in an Integer attribute assigned by Mpr-rendering roles to the base-object with a common identifier (sliceIndex in Fig. 14(b)).

Note that, besides implicit chaining, this example also illustrates the multiplicity feature as two similar rendering roles (Mpr ones, differently configured) are attached to the same base-object.

3.2.4. Example: reading, rendering, processing and graphical user interface

This example concerns the combination of a graphical user interface with 3D medical image reading, rendering and processing. In the considered application domain, many various image processing algorithms exist. An image processing algorithm aims at extracting or transforming image information in order to facilitate its understanding and therefore make a more efficient diagnosis for instance. The considered combination of functionalities is illustrated in Fig. 16. The algorithm considered in the example performs a morphological erosion of the image (Pratt, 2007), which involves some noise removal and an enlargement of dark areas (not detailed as out of scope of the present paper).

As illustrated by Figs. 17 and 18, the algorithm is seen as a role (Erode, identified by algo) to be played by the root base-object of type Object. This algorithm is configured to process the input0 image (Fig. 16 (“Initial 3D image”)) and input0 in Figs. 17 and 18). This image is a dynamic attribute of the root base-object. This dynamic attribute has a Dicom role for loading the image to be processed, and a Mpr role for rendering purposes. This situation illustrates subject composition (i.e. a subject being composed of subject(s)). The output image (Fig. 16 (“Processed 3D image”)) and output in Figs. 17 and 18) has also a rendering role.

An editor (Editor role in Figs. 17 and 18) embeds text controls to enter algorithm’s parameters (dynamic attributes of type Float). This editor corresponds to the left panel appearing in the snapshot of the gui in Fig. 16.

The Aspect role (see Figs. 17 and 18) manages gui elements such as windows to be used by the Editor and both Mpr roles to install the gui-based functionalities they provide (win XML elements reported in Fig. 17).

Action defines an additional role appearing as a menu item in the gui (see the snapshot of the gui in Fig. 16). This role is in
charge of triggering the processing: when the end-user presses this menu item, the \texttt{Action} invokes the \texttt{update()} method of the \texttt{algo} role. This procedure illustrates the concept of explicit chaining in a basic situation where the \texttt{Action} role is a coordination role configured to trigger \texttt{algo} (see subject declaration reported in Fig. 17). After having run \texttt{algo}, the output image is updated, and its rendering role naturally refreshes (implicit chaining through observation).

Note that optional information can also be provided together with the root subject declaration, corresponding to \texttt{start} and \texttt{stop} XML tags in Fig. 17. The \texttt{start} tag indicates role(s) to be activated (specifying the concrete role type), according to a given order chosen to take functional dependencies into account. For instance, \texttt{IAAspect} must be activated before \texttt{IRender}. Indeed, the layout of windows, within which rendering is expected to be performed, must be initialized first. The \texttt{stop} tag allows to specify the deactivation order of roles when one quits the application. Note that, when quitting a software, roles are naturally deactivated and destroyed consequently to automatic (root) subject destruction, even if not explicitly specified in the subject declaration.

3.3. Discussion

The purpose of the proposed pattern is to facilitate dynamic composition and collaboration of separated code elements without code rewriting or glue code. In particular, this includes the ability to reuse code elements developed for one software to another, as summarized in Section 3.3.1. Section 3.3.2 focuses on role aspects. Additionally to observed advantages of considered role features, this section details the meaning of some of them in this application domain. Section 3.3.3 summarizes observed advantages regarding the integration of the component orientation for developed applications.

3.3.1. Software

In the case of basic applications, no specific code (i.e. glue code) has been required for the various compositions. Moreover, the code has not been modified to reuse roles from one application to another. Only specific XML descriptions (root subjects) have been written, in order to appropriately combine and configure roles as well as base-objects, all of them being transparently provided by components.

In the case of the advanced VR-Render software, roles developed for basic applications have been reused. No adaptation (i.e. (partial) rewriting of the code) has been necessary to meet the specific functional requirements of VR-Render. Only few additional concrete roles have been specifically developed, concerning some specific gui elements. For instance, an image selector panel has been developed for choosing the medical image to be visualized among a set of loaded images (provided by patients). The amount of specific code (specific roles) appeared almost negligible compared to the code reused (the one developed for basic applications).

```
<object type="Object">
  <object id="input0" type="Image">
    <role type="IRender" implementation="Mpr" com="yes" win="900"/>
    ...
  </object>
  <object id="input1" type="Float"/>
  ...
  <role id="algo" type="IProcessor" implementation="Erode">
    <input image id="input0"/>
    ...
  </role>
  <role id="editor" type="IEditor" implementation="Editor" com="yes" win="901"/>
  <role id="action" type="IAction" implementation="Action" target="algo" />
  <role id="aspect" type="IAAspect">
    <win id="900" ...
    <win id="901" ...
    ...
  </role>
  <start type="IAAspect" />
  ...
  <start type="IRender" />
  <stop type="IRender" />
  ...
  <stop type="IAAspect" />
</object>
```

Fig. 16. Example combining a graphical user interface (gui) with reading, rendering and processing functionalities: principle (left area) and snapshot of the gui (right area).

Fig. 17. Example combining a graphical user interface (gui) with reading, rendering and processing functionalities: subject declaration.
To summarize, many code elements have been reused from one software to another, with few specific developments. In particular, no glue code has been required to build software programs, as only XML-based description files have been used. Although a quantitative evaluation of the degree of reuse would be interesting (e.g., amount of code reused from one application to another), the variety and the functional complexity of generated software programs underline the overall relevance of the proposed pattern for composition and collaboration.

### 3.3.2. Role

Observed advantages are discussed for each role concepts introduced in Section 2. The meaning of some of them for the considered application domain is also underlined.

#### 3.3.2.1. Identity

This feature enabled to associate data structures and functionalities using a single reference on one entity only (data structure), without requiring a dedicated container which would be in charge of handling functionalities independently from the (base) objects to be processed. This facilitated the use of simple description files declaring subjects in a simple tree-like structure (functionalities appearing as a simple list of sub-elements of base-objects). Besides, the fact that each role keeps its own identity has been useful for explicit chaining with coordination roles retrieving collaborating roles using their identifier. The use of dynamic attributes appeared crucial to prevent unwanted inter-relations between functionalities (limited to dynamic attributes), and to ensure a good separation of concerns.

#### 3.3.2.2. Dependency

This feature enabled to preserve the traditional separation between data (base-objects) and functionalities (roles) [Szyperski, 2002; Sridhar, 2007], although the composition result was seen and manipulated as a single entity. It also ensured the implicit detachment and destruction of functionalities attached to a given base-object, without requiring glue code. Besides, as roles directly depend on their base-objects, functionalities have a direct access to data structure, which appeared useful in terms of runtime performances, this being crucial in our application domain. For medical image and video processing as well as for rendering purposes, this regards the fact that the data buffer (of images or videos) can be directly accessed. As base-objects do not depend on functionalities (no a priori knowledge about possible functionalities to be attached to base-objects), any new initially unforeseen functionality has been integrated without requiring the modification of data structures (code rewriting).

#### 3.3.2.3. Dynamicity

This feature enabled to achieve dynamic composition and decomposition of functionalities. In some cases, dynamic composition appeared similar to load-time composition (or early run-time composition), as for root subject instantiation (software instantiation). Nevertheless, real run-time composition and decomposition has also been considered. It has been used for processing, rendering or reading functionalities which could be interactively changed by the end-user after software instantiation (i.e., changing the processing, reading or rendering role). For instance, the reading algorithm could be changed each time the end-user decided to read a new image or video, by pressing the appropriate menu item (typically the “File/Open” menu). The list of available reading roles for the considered data structure (e.g., image or video) was first proposed to the end-user. The concrete role was then selected by the end-user, instantiated and finally attached to the appropriate image before being run. Note that any already attached reading role was removed (decomposition) if different from the selected one.

#### 3.3.2.4. Multiplicity

This feature has been widely used in developed applications, in particular for assigning multiple similar concrete rendering roles to the same image, each one being specifically configured (e.g., example related to Fig. 14).

#### 3.3.2.5. Activation

This feature has been used for all applications, as for the initialization of gui-based roles (e.g., preparation of a window or installation of menu items) and rendering-based ones (allocation of some resources for 3D rendering). It has also been used for temporarily enabling/disabling rendering roles depending on windows which could be hidden or shown according to end-user interactions (as illustrated for VR-Render). For instance, the Mpr rendering was disabled if the window was hidden (not visible by the end-user). In such a case, any attempt for refreshing the window content (invocation of update or update(message) methods) was ignored, therefore saving computation time and/or avoiding system failure. This feature also facilitated the management of the control flow at software instantiation time or when quitting applications. For instance, when quitting an application, systematically deactivating roles (in the appropriate order, as illustrated by the example related to Fig. 17) before detaching and destroying them preserved from unforeseen behaviour or software failure (e.g., deleting a role managing a window while a rendering role using it still receives refreshing requests).

#### 3.3.2.6. Migration

This feature has been used in VR-Render to manage image selection changes, involving the migration of attached roles to the newly selected image. Migration appeared useful for rendering roles in particular, to avoiding losing their state (e.g., rendering windows within which renderings were performed) as well as to save computation time (no need to reallocate rendering resources).

#### 3.3.2.7. Explicit chaining

This feature appeared useful to explicitly control the collaboration between roles for some specific functionalities. For instance, it has always been used to trigger the execution (invocation of the update() method) of particular roles (e.g., processing, reading or rendering) from buttons and menu items (being themselves roles). Such gui controls were seen as coordination roles (e.g., Fig. 18 (“Action”)). Explicit chaining also appeared a relevant alternative for ordering particular sequences of tasks avoiding indirect interactions which would result from observations (implicit chaining).
in order to ensure a limited response time. For basicRobot (see Fig. 9) in particular, explicit chaining enabled to completely control the execution flow (video tracking and command application) so that the control loop cycle was shorter than 40 ms (Fasquel et al., 2009). In such a case, a coordination role was in charge of the control loop, triggering video rendering only if enough time was remaining.

3.3.2.8. Implicit chaining. This feature appeared absolutely required to manage collaboration between functionalities in case of unforeseen end-user interactions as well as in case of unforeseen compositions encountered in considered software programs. This has been observed for the collaboration of rendering functionalities with reading and processing ones (illustrated in Fig. 18), the number of rendering roles involved in the composition being a priori unknown (unforeseen composition). This also appeared convenient for the collaboration, according to (unforeseen) end-user interactions, between several rendering roles (synchronized illustrated in Fig. 14).

3.3.2.9. Abstractivity. At Role-level, this feature appeared crucial to manipulated roles (i.e. functionalities) in an abstract manner. For instance, this facilitated the development of an application using a simple description file, involving a limited number of key-works, corresponding, in particular, to the well-identified (i.e. mapped on well-identified role concepts) and limited number of methods provided by this Role entity.

At ConcreteRoleType-level, this facilitated the manipulation of groups of roles, corresponding to concrete functionalities classified into functional groups. Thanks to a central container registering all instantiated subjects (and therefore functionalities), it was possible to access and interact with all functionalities of a given type. For instance, this enabled to activate all rendering roles only after those managing windows (e.g. Fig. 17 (“IAspect”)) within which renderings were expected to be performed, regardless their related data structure (rendering roles being attached to two distinct images in Fig. 16).

3.3.2.10. Transitivity. This feature has been used to configure image readers (being a role) using a graphical editor (i.e. selection of filenames to be read) acting as a gui-based edition role (similar to Fig. 18 (“Editor’’)) attached to the reading role.

3.3.3. Component

The integration of the component orientation, coupled with role concepts, appeared useful for the reduction of build-level dependencies, as any new functionality or functionality type has been incorporated without entirely recompiling and redeploying an application.

Thanks to the graph-based component description in particular, one remained focused on functional composition (i.e. composition of base-objects and roles, corresponding to data structures and functionalities) regardless the specificity of the distribution of both data structures and functionalities over the different components. For load-time (or early run-time) composition, subject declarations did not explicitly indicate components to use and wire; only base-objects and roles were reported (see Figs. 13(a), 15(a) and 17). Required components were automatically and transparently retrieved through the analysis of the graph resulting from their descriptions. For run-time composition, this also appeared useful to integrate new functionalities without taking care of the identity and the location of components providing them. In the case of reading functionality previously discussed, the graph enabled to easily propose to the end-user the list of reading role available for images for instance; this list was automatically built from graph analysis by searching for nodes connected simultaneously to both IReader and Image nodes. After selection, the appropriate component was started if necessary, to finally instantiate and attach the concrete role.

4. Related works

Hereafter, we propose to compare our proposal to some well-known approaches addressing issues considered in this paper. Considered alternatives are those mentioned in Section 1, namely those based on aspect oriented programming (AOP), role, mixin and delegation. Section 4.1 focuses on the composition issue. The collaboration issue is discussed in Section 4.2. Section 4.3 is dedicated to roles and Section 4.4 focuses on the reduction of build-level dependencies through the use of components. Table 2, the content of which being detailed hereafter, aims at providing an overview of the proposed comparison.

4.1. Composition

For comparison purposes, composition is considered in terms of composition type (i.e composition at class-level or object-level), composition time (i.e. run-time or compile-time) and composition context. Composition context regards the context of use of the composition for the different alternatives.

4.1.1. Composition type

Class-level composition involves that all instances of a given class are concerned by composition, compared to object-level composition focusing on instances. The proposed design pattern considers composition at object-level. For instance, in our application domain, a rendering or a reading role is attached to an image, and not to all images. Table 2 reports the type of composition favoured by the different considered related works, this being detailed hereafter.

As recently underlined in Baldoni et al. (in press), aspect oriented programming (AOP) focuses on class-level composition. This paradigm, representing an active research area (Tanter et al., 2008), aims at inserting some code into class methods of a target program, the position in the target program being defined by jointpoints (Bettini et al., in press; Tartler et al., 2010). The inserted code is an advice and a pointcut identifies or captures jointpoints in the program flow. Advices and pointcuts are declared using aspects, and code insertion corresponds to aspect weaving. AOP is widely used for cross-cutting concerns (i.e. common tasks independent from the business logic) such as logging, monitoring, persistence and security purposes (Villazon et al., in press; Fuentes and Sanchez, 2007).

As recently underlined in Baldoni et al. (2007), mixin-based approaches mainly focus on class-based composition (mixing separated class hierarchies), rather that on object-base one, although the ability to consider composition at both object-level and class-level has been recently studied (Zdun et al., 2007).

Approaches based on roles (Herrmann, 2007; Baldoni et al., 2007) and delegation (Gamma et al., 1995; Lieberman, 1986; Bettini et al., in press) focus on composition at object level. As an example of delegation-based object composition approach, a recent proposal introduces the notion of incomplete objects, which focuses on the dynamic assignment of method implementation to objects (objects becoming complete after composition) (Bettini et al., in press). Delegation-like design patterns such as decorator aims to changing object method behaviour through dynamic composition (Gamma et al., 1995). Recent role-based proposals such as ObjectTeams/Java (OT/J for short) (Herrmann, 2007) and powerJava (Baldoni et al., 2007) aim at modifying the behaviour of some existing object methods through role attachment (e.g. notion of callin in OT/J (Herrmann, 2007)).
The counterpart is that the behaviour of existing methods can be provided through composition and are therefore not modified in our case, as it is not the intention of our proposal. This difference is reported in Table 2 (Adding behaviour and Changing behaviour).

In our opinion, a motivation of approaches focusing on the modification of existing methods is to adapt the behaviour (at object or class level) of these methods from the point of view of the client program invoking them. This clearly appears for AOP approaches focusing on cross-cutting concerns rather than on business logic. This underlines that AOP-based composition is used to extend an existing program (existing program in Table 2) and not to build an entire one. In our case, the entire program (entire program in Table 2) is concerned by the composition (the entire program being defined by the root subject).

In our sense, due to this difference regarding the composition context, considered related works appear complementary to our proposal.

4.2. Collaboration

In AOP-based approaches, collaboration regards the execution order of pieces of code (advice) injected at a given point of the target program. Controlling the chaining, and therefore interactions between advices woven to a given joinpoint, remains a challenging issue (Tantzer et al., 2008). In approaches comparable to dynamic inheritance (Zdun et al., 2007), chaining corresponds to the navigation across object or class methods (notion of navigation path), according to the composition. For instance, in the case of incomplete objects (Bettini et al., in press), the implementation of some methods can be provided through composition and are therefore dynamically integrated in the navigation path. For mixin-based composition, navigation depends on methods dynamically added at class level, or at both class and object levels in Zdun et al. (2007). For role-based composition, the notion of chaining regards the coordination of roles (Herrmann, 2007; Colman and Han, 2007).

4.1.2. Composition time

As underlined in Table 2, all considered related works support run-time (dynamic) composition (see e.g. Bettini et al., in press for delegation and Zdun et al., 2007 for mixins). AOP supports dynamic weaving, even though static weaving remains widely used (e.g. AspectJ extension of the Java language for AOP) (Bachara et al., 2010).

Recent proposals for role-based composition consider composition at run-time (Herrmann, 2007; Baldoni et al., 2007). Note that an earlier work considered the implementation of roles in C++ (VanHilst and Notkin, 1996), the language we consider in the case of mixin-based approaches. In our opinion, a motivation of approaches focusing on the modification of existing methods is to adapt the behaviour (at object or class level) of these methods from the point of view of the client program invoking them. This clearly appears for AOP approaches focusing on cross-cutting concerns rather than on business logic. This underlines that AOP-based composition is used to extend an existing program (existing program in Table 2) and not to build an entire one. In our case, the entire program (entire program in Table 2) is concerned by the composition (the entire program being defined by the root subject).

In our sense, due to this difference regarding the composition context, considered related works appear complementary to our proposal.

4.1.3. Composition context

This aspect regards the context of use, or the intention, of considered proposals with respect to dynamic composition.

In our sense, considered related works regard dynamic composition as a way to modify or provide the behaviour of existing method at class-level (e.g. AOP; Villazon et al., in press) or object-level (e.g. incomplete object, Bettini et al., in press; OT/J, Herrmann, 2007; and PowerJava Baldoni et al., 2007). In our case, one focuses on adding functionalities, initially unknown to the base-object. Practically, instead of changing the behaviour of, for instance, a read method which could have been declared for images, one favours the ability to dynamically add the notion of reading. In our case, base-objects have no knowledge about the kind of functionalities which could be attached to it after composition. An advantage is that it is not required to update the declaration of a base-object in order to integrate a new type of functionality. Note that, similarly, mixin-based approaches allow to add methods (mainly at class-level). The counterpart is that the behaviour of existing methods are not modified in our case, as it is not the intention of our proposal. This difference is reported in Table 2 (Adding behaviour and Changing behaviour).

In our opinion, a motivation of approaches focusing on the modification of existing methods is to adapt the behaviour (at object or class level) of these methods from the point of view of the client program invoking them. This clearly appears for AOP approaches focusing on cross-cutting concerns rather than on business logic. This underlines that AOP-based composition is used to extend an existing program (existing program in Table 2) and not to build an entire one. In our case, the entire program (entire program in Table 2) is concerned by the composition (the entire program being defined by the root subject).

In our sense, due to this difference regarding the composition context, considered related works appear complementary to our proposal.

4.2. Collaboration

In AOP-based approaches, collaboration regards the execution order of pieces of code (advice) injected at a given point of the target program. Controlling the chaining, and therefore interactions between advices woven to a given joinpoint, remains a challenging issue (Tantzer et al., 2008). In approaches comparable to dynamic inheritance (Zdun et al., 2007), chaining corresponds to the navigation across object or class methods (notion of navigation path), according to the composition. For instance, in the case of incomplete objects (Bettini et al., in press), the implementation of some methods can be provided through composition and are therefore dynamically integrated in the navigation path. For mixin-based composition, navigation depends on methods dynamically added at class level, or at both class and object levels in Zdun et al. (2007). For role-based composition, the notion of chaining regards the coordination of roles (Herrmann, 2007; Colman and Han, 2007).

Table 2

Comparison with some recent works (see references therein for earlier works). “Out of scope” means that, to our knowledge, no specific study has been performed to evaluate whether (or detail how) the approach could support the notion.

<table>
<thead>
<tr>
<th>Notion</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed pattern</td>
<td></td>
</tr>
<tr>
<td>Role</td>
<td></td>
</tr>
<tr>
<td>(Herrmann, 2007; Baldoni et al., 2007; Colman and Han, 2007)</td>
<td></td>
</tr>
<tr>
<td>AOP</td>
<td></td>
</tr>
<tr>
<td>(Tanter et al., 2008; Fuentes and Sanchez, 2007; Bachara et al., 2010; Tizzei et al., in press)</td>
<td></td>
</tr>
<tr>
<td>Mixin</td>
<td></td>
</tr>
<tr>
<td>(Zdun et al., 2007)</td>
<td></td>
</tr>
<tr>
<td>Delegation</td>
<td></td>
</tr>
<tr>
<td>(Gamma et al., 1995; Lieberman, 1986; Bettini et al., in press)</td>
<td></td>
</tr>
</tbody>
</table>

Composition type

(Section 4.1.1) Object-level | Class-level | Class-level | Object-level

Composition time

(Section 4.1.2) Run-time | Run-time | Run-time | Run-time

Composition context

(Section 4.1.3) Adding behaviour (Entire program) | Changing behaviour | Changing behaviour (existing program) | Changing behaviour

Collaboration

(Section 4.2) Explicit (team; Herrmann, 2007); (coordinator; Colman and Han, 2007) | Explicit (behaviour links; Tanter et al., 2008) | Explicit (mixin list; Zdun et al., 2007) | Out of scope

Role

(Section 4.3) Many supported concepts, advanced functionalities | Few supported concepts, many supported concepts (Herrmann, 2007), basic functionalities (Herrmann, 2007; Baldoni et al., 2007) | Out of scope | Out of scope | Out of scope

Build-level dependencies

(Section 4.4) Component | Component (Tizzei et al., in press) | Out of scope | Out of scope, component (Gamma et al., 1995)
As underlined in Section 1, considered related works favour the management of collaboration using a dedicated entity. For instance, in AOP, a behavioural link has been recently proposed to control the program flow, according the configuration of links (e.g. activation condition) (Tanter et al., 2008). Another recently studied approach is to consider a specific aspect in charge of the coordination (Fuentes and Sanchez, 2007). For mixins, this issue has also been addressed by providing a mixin list (Zdun et al., 2007). The recent OT/J proposal, based on role concepts, manages role collaboration using a dedicated entity (Herrmann, 2007), the Team which encapsulates collaborating roles. In Colman and Han (2007), a specific role is in charge of the coordination.

In our case, as previously underlined, we propose to consider two collaboration modes (see Table 2). The first one regards the use of a coordination role, as in Colman and Han (2007) (Explicit chaining). The second collaboration mode corresponds to the notion of implicit chaining, being, in our sense, ignored by related works addressing dynamic composition, and in particular by role-based approaches. Note that a strength of our proposal is that both collaboration modes appear as first-class entities of the proposed pattern. For explicit chaining, this regards the update() method of the Role class together with a coordination role. For implicit chaining, this regards the update(msg) method of the Role class coupled with the observation link. A perspective is to facilitate the reuse of a given role in different collaboration modes. For instance, by removing the observation link of a given role with respect to its base-object (implicit chaining mode), the given role could be used in an explicit chaining mode, where coordination would be managed by a dedicated role, as in Colman and Han (2007).

It must be pointed that implicit chaining (similar to an endogeneous coordination model; Fuentes and Sanchez, 2007) may present a limitation resulting from the underlying observation mechanism. In such a case, roles manage both computations (i.e. the functionality (Fuentes and Sanchez, 2007), associated with the update() method of the Role class in the proposed pattern) and interactions (according to received messages, associated with the update(msg) method the Role class in the proposed pattern). As a consequence, any modification of the coordination protocol (e.g. changing the identifiers of dynamic attributes including base-object modifications in our case) may involve some rewriting of concrete roles (regarding the update(msg) method). In the case of explicit chaining (similar to an exogeneous coordination model; Fuentes and Sanchez, 2007), only the coordination role has to be modified, therefore preserving concrete roles.

4.3. Role

As recently underlined (Herrmann, 2007), there is no well accepted standard for applying roles to a programming language, as it relies on concepts (Kristensen and Osterbye, 1996; Steimann, 2000) rather than on well established implementation constraints or rules. Therefore, a difficulty is to find an appropriate mapping between these high-level concepts and low-level programming languages, as addressed in our case with the proposed design pattern. A consequence is that, as underlined in Herrmann (2007), almost all existing implementation proposals focus on some concepts while ignoring others (see recent papers Herrmann, 2007; Baldoni et al., 2007, and references therein).

As underlined in Table 2, we claim that a strength of the proposed pattern regards the fact that we support many role concepts. In our sense, the set of supported concepts is comparable to those considered in the recent OT/J proposal which covers more aspects than previous proposals (Herrmann, 2007) (Few supported concepts, except for Herrmann, 2007 in Table 2).

Concepts being supported in our case as well as by OT/J are identity, dependency, dynamicity, multiplicity, abstractivity and transitivity. Notions of activation and chaining are associated with the notion of collaboration managed by the Team entity in OT/J. A first difference regards the notion of activation which has a different meaning in our case, as underlined in Section 2.2.1. A second difference concerns the concept of visibility, regarding the ability to restrict the access to some roles (Kristensen and Osterbye, 1996). This feature is supported by OT/J but not in our case because all attached roles can be accessed without any restriction. A third difference regards the migration feature, being supported in our case but not by OT/J. In our sense, a major limitation of OT/J regards the fact that roles are hard-coded inner classes of Team entities. This does not facilitate the reuse of a role in different contexts (a given role being constrained to a particular Team). This also affects the ability to easily support the multiplicity feature, as underlined in Herrmann (2007), roles of similar types and attached to the same base-object instance must be provided by different contexts (i.e. different Team instances). Note that differences between our proposal and OT/J are not detailed in Table 2 because it aims at providing an overview not limited to role concepts. For clarity, only the overall degree of coverage of role concepts is reported in Table 2 (i.e. many or few supported concepts), details being provided in this section.

Among other recent alternatives supporting few role concepts (Herrmann, 2007), powerjava (Baldoni et al., 2007) breaks the dependency feature at code level, because roles are declared as inner classes of the base-object, as also underlined in Herrmann (2007). The recent ROAD proposal (Colman and Han, 2007) focuses on collaboration, rather than on other concepts, using a dedicated role, similarly to our proposal of explicit chaining (discussed in the previous Section 4.2), and does not suffers the limitation of Team as roles are independent for the coordination role.

In our sense, additionally to the good coverage of role concepts, another contribution of our proposal, compared to related role-based approaches, regards the refinement of the notion of collaboration by considering two complementary modes (discussed in previous Section 4.2).

Another strength of our pattern is that several concepts are considered as explicit first-class entities of the pattern. This regards the fact that the Role class defines an interface (to be implemented by entities related to both ConcreteRoleType and ConcreteRole- Type in Fig. 1) consisting in five abstract methods corresponding to well identified role concepts (activation, implicit chaining, explicit chaining and migration). This guides the developer in declaring new concrete role types as well as new concrete roles, while respecting associated role concepts appearing as contract to be fulfilled (interface to be implemented). This also facilitates the manipulation of roles in an abstract manner, through the concepts related to Role class methods regardless the specificity of role types and concrete roles.

Note that another interesting aspect of our work is the proposed mapping between some role concepts and a particular application domain which has never been considered in related works regarding roles. It illustrates both the meaning and the relevance of role concepts in a new practical situation, involving complete software programs with advanced functionalities (Advanced functionalities in Table 2) compared to related works such as (Herrmann, 2007; Baldoni et al., 2007), limited to basic use cases (Basic functionalities in Table 2).

4.4. Build-level dependencies

As reported in Table 2, coupling dynamic composition and component has been considered in the case of AOP, where aspects can be provided by a component to another (Tizzi et al., in press) (see references therein for other works coupling both notions). To our knowledge, there is no specific study focusing on the cou-
pling of component orientation with other dynamic composition approaches such as mixin and specific delegation alternatives. In Table 2, we only consider that approaches based on standard design patterns such as decorator (Gamma et al., 1995) have been widely used for software development involving components.

To our knowledge, there is no proposal coupling both role and component concepts. Nevertheless, it is interesting to mention the work of Parka et al. (2004), focusing on a methodology, coupling both notions, for the organization of the development of collaborative software systems. The goal is to reduce the development cycle time. The role perspective focuses on capturing business requirements for the c-commerce community business functions. The result is the logical-level specification of reusable components (for storing code elements). This work does not consider roles for composition purposes at programming level as in our case. For this reason, the coupling of roles with the issue related to the reduction of build-level dependencies using components is considered as out of scope in Table 2. Although out of the scope, their idea can also be found in our case. Indeed, the role orientation focuses on the composition of functionalities (at implementation level in our case), hiding the specificity of their (physical) distribution over components. In our sense, this observation underlines the relevance of coupling both role and component concepts, although the context of use differs from Parka et al. (2004).

5. Conclusion

The proposed design pattern offers a good coverage of role concepts for dynamic composition and collaboration purposes. Compared to related works, it integrates a refinement of the notion collaboration by considering two complementary approaches, depending on whether collaborations can be foreseen or not. The presented design pattern also proposes a solution for coupling both role and component concepts, where the component orientation aims at reducing build-level dependencies. As a result, the proposed design pattern proposes a solution for managing composition by focusing on functionalities only, components providing them being transparently retrieved.

The pattern has been successfully implemented in C++ and used to develop a set of medical software programs, involving many complex and advanced functionalities. Each role concept appeared perfectly appropriate to manage considered functionalities. The proposed coupling with the component orientation enabled to build an entire software using a generic program, independent from dynamic libraries embedding data structures and functionalities, together with description file focusing on functionalities regardless the specificity of the component distribution.

The perspective and future steps of that work is to further evaluate the relevance of the proposed design pattern for managing a reliable software product line, involving even more advanced medical software programs, as well as for other application domains.

Acknowledgements

A part of this work has been achieved in the context of the eHealth PASSPORT project of the 7th Framework Program of the European Community, funded by the ICT program.

We would like to thank all the research and development teams of IRCAD. In particular, we thank Guillaume Brocker and Vincent Agnus for their technical contributions to the implementation of the proposed design pattern. We also thank all medical partners who have provided us the medical images of patients, and in particular, Pr. Afshin Gangi and Pr. Catherine Roy from the radiological department of the University Hospital of Strasbourg. Finally, we thank the entire digestive and endocrine surgical team of Prof. Marescaux and in particular Prof. Didier Mutter, Prof. Bernard Dallemagne, Prof. Jol Leroy and Dr. Michel Vix.

Authors thank reviewers for their helpful suggestions regarding concepts and models.

References


Jean-Baptiste Fasquel was born in 1975. He received his diploma from the general engineering school ENSPS in France, and his diploma for advanced studies in applied optics and computer vision from the University of Strasbourg in 1998. He obtained his PhD in computer science from the University of Strasbourg, France, in 2002. From 2002 to 2009, he has been a permanent researcher in computer sciences at the Research Institute Against Digestive Cancer IRCAD, Strasbourg. Since September 2009, he has been an Associate Professor at the University of Angers. His main areas of interest are software engineering and medical image processing for computer-aided diagnosis and surgery.

Johan Moreau received his M.S. in computer science from the University of Rennes in 2001. Since 2001, he has been a permanent engineer in computer sciences at the Research Institute Against Digestive Cancer IRCAD, Strasbourg. His main areas of interest are network and software engineering for computer-aided diagnosis and surgery.