A modular and evolutive component oriented software architecture for patient modeling

Jean-Baptiste Fasquel*, Guillaume Brocker, Johan Moreau, Nicolas Papier, Vincent Agnus, Christophe Koehl, Luc Soler, Jacques Marescaux

IRCAD, 1 place de l’Hôpital, F-67091 Strasbourg, France

ARTICLE INFO

Article history:
Received 10 January 2006
Received in revised form
29 June 2006
Accepted 6 July 2006

Keywords:
Software architecture
Reusable components
XML
Medical image processing

ABSTRACT

This paper deals with the design aspect of a software aiming at modeling the anatomical and pathological structures of patients from medical images, for diagnosis purposes. In terms of functionalities, it allows to combine image processing algorithms, and to visualize and manipulate 3D models and images. The proposed software uses specific extensible and reusable components and a system managing their combination, thanks to a formal XML-based description of their interfaces. This architecture facilitates the dynamic integration of new functionalities, in particular in terms of image processing algorithms. We describe the structural and behavioral aspects of the proposed reusable component-based architecture. We also discuss the potential of this work for developing other softwares in the field of computer aided surgery.

© 2006 Elsevier Ireland Ltd. All rights reserved.

1. Introduction

In the field of computer aided surgery, current research activity involves the development of an increasing number of softwares for different specific applications: computer aided diagnosis [1,2], surgery planning [3], simulation [4], image guided surgical intervention [5]. Most of these applications share several common functionalities such as image analysis and interactive 3D visualization for instance. Indeed, image analysis is required to model medical structures for diagnosis purposes. It can also be needed for modern and future systems dedicated to image guided intervention [5–7], where it is necessary to detect and analyze, during the intervention, medical information to control the behavior of the system. Three-dimensional visualization is required for patient modeling, to control segmentation process for instance, and for both surgery planning and simulation, where it appears necessary to interact with models using virtual tools [3], sometimes also incorporating force feedback [4]. Moreover, due to the important research activity related to this emerging field, softwares must be modular and evolutive enough to easily support the integration of new functionalities. For instance, in the case of patient modeling on which we focus in this paper, it is crucial to easily incorporate or modify image processing algorithms.

In this context, when developing softwares for such applications, it appears relevant to consider a design methodology facilitating the reusability of the code and the extension of functionalities. Development effort, including refactoring [8] and code maintaining, can be therefore minimized and facilitated.

Component oriented programming [9] is an interesting and useful approach for managing this aspect [10–12]. According to Spzyperski [9], a software component is "a unit of composition * Corresponding author. Tel.: +33 3 88 11 90 79; fax: +33 3 88 11 90 99. E-mail address: Jean-Baptiste.Fasquel@ircad.u-strasbg.fr (J.-B. Fasquel). URL: http://www.ircad.org.
with specified interfaces and describe context dependencies only. A software component can be deployed independently and is subject to composition by third parties." By combining components, one extends their functionalities, therefore leading to the high-level software ones. By standardizing and limiting dependencies between components (restricted to few explicit interfaces), one intends to facilitate refactoring, correction, maintaining and evolution because modifications are localized (e.g. to the code of a given component) and their impact on the rest of the system can be better foreseen and controlled. This programming approach is well known by the community dedicated to software engineering and can be, in our sense, advantageously applied to image processing context.

We developed a cross-platform (Windows, Linux and Mac Os X operating systems) component oriented software, based on the C++ language, for medical image segmentation. In our case, the advantage of this approach essentially concerns the extension of software functionalities in terms of image segmentation algorithms: they are embedded within operator components (similar to “plugins”) which can be easily and dynamically integrated in the software without modifying its code. As previously said, this is highly convenient in this highly evolutive field to facilitate the improvement of existing algorithms and the rapid integration and evaluation of new ones, with minimal programming, as recently underlined by Papajorgji [13] for another application field. Moreover, the use of additional components (dedicated to the graphical user interface, to the rendering of 3D data and to the management of operators for instance) makes our software system highly modular. This is expected to strongly facilitate the future development of additional softwares (associated to some of the above cited surgical applications), by reusing the methodology and most components, with minimal refactoring.

A component-based software involves the definition of suitable and reusable components, but also the management of their combination and configuration. As recently underlined in [14], many software systems suffer from missing support for this aspect. This is often hard-coded, leading to tangled composition and configuration code that is hard to understand and maintain. As in [14], we propose to separate this aspect from the rest of the software. We developed a component management system, based on a XML-based formal description of component interfaces (including XML Schema specifications [15]), the role of which being to define how software components are plugged together (or contribute to each other) and to start effective combinations.

The description of the software architecture we give in this paper, involving the above listed modern aspects related to software engineering, is expected to facilitate the work of researchers and computer scientists developing new systems in this emerging field dealing with computer aided surgery, so that they can more easily integrate their work. In our sense, the originality of this paper essentially relies on the use, in our medical context, of the software component approach as well as of the formal XML-based component management system.

The next section is dedicated to the functional description of the software, to facilitate the understanding of the role of the different underlying components. Then, we shortly describe the different components we developed to build the final application. In Section 4, we describe their structure, and how both interfaces and contributions are specified and effectively implemented. In Section 5, a behavioral description of our software is provided. In particular, we detail how components are plugged together when the application starts. We also shortly describe the communication system. Most explanations will focus on image processing which is the central aspect of the considered software dedicated to patient modeling. Before concluding, we discuss about this work and its perspectives in terms of application (extension to additional softwares for computer aided surgery) and software engineering.

2. Functional description of the proposed software

In this section, we shortly describe the standard manipulation mode of our software, and the major functionalities related to image processing.

2.1. The standard manipulation mode

The standard manipulation is described in a recently published work focusing on a specific functionality of this software [16]. Therefore, only main guidelines are reported hereafter. Fig. 1 reports a snapshot of the developed software. In this section, we shortly describe the standard manipulation mode of our software, and the major functionalities related to image processing.

Typically, the user starts a new modeling procedure by loading and preprocessing (e.g. noise removal, rotation) the 3D medical image representing the anatomical and pathological structures of a given patient. For each new sequence, the user first selects the reference sequence (in frame F, Fig. 1), being usually the first one (dedicated to noise removal for instance, and corresponding to “preprocessing” in Fig. 2). This could be another sequence such as “air” (Fig. 2), from which both lungs could be further segmented [16]. Then, he presses the “new sequence” button of the software toolbar, involving the insertion of a new entry in the graphical representation, frame F in Fig. 1, of the image processing graph (corresponding to a new empty pipeline of operators in the effective graph reported in Fig. 2-right). This creates the appropriate connection in the effective image processing graph, between the user selected reference sequence and the new one.

The user can also define the sequence nature (properties editor, frame C in Fig. 1). As detailed in [16], the nature can either be a “target”, for segmentation purpose (e.g. liver or heart), or a “zone”, the goal being therefore to preprocess data so as to facilitate the further segmentation of targets (e.g. extracting air from which both lungs can be further segmented). This is required so that our system automatically
Fig. 1 – Snapshot of the developed patient modeling software. The part surrounded by frame B lists all available image processing operators which can be dynamically created and inserted within the image processing graph surrounded by frame F. This frame allows to visualize the processing line and to select a sequence or an operator. Frame E corresponds to the parameter editor of the operator selected in frame F. Frames A and D surround functionalities related to the visualization of the data associated with the current selection in the processing line (e.g. modifying the orientation of the negatoscope and the displayed slices). Frame C is dedicated to structure properties (type, name, color, etc.).

determines within which region of interest computations can be restricted, according to already segmented medical structures. For instance, if the liver has already been modeled and if the current structure type is a liver tumor, the system automatically determines that processings, for extracting the liver tumor, should be constrained within the liver area, which can strongly facilitate and accelerate segmentations.

After having defined the sequence nature and type, the user can insert (or remove) image processing operators from those available (icons within the frame B of Fig. 1), to build a processing sequence (OPi in Fig. 2-right). Data which is displayed (see frame D in Fig. 1) corresponds to the selected element (within frame F of Fig. 1). This allows to visually control the evolution of the processings. Finally, in case of “target”, the user can validate the sequence (properties editor, frame C in Fig. 1) when he considers that the segmentation is finished. This allows, for instance, to generate its 3D surface representation (which can be superimposed on a 3D representation of the negatoscope, as illustrated in Fig. 1) and to register it within a database from which regions of interest associated with next structures will be determined [16].

The user can then start additional sequences, until all structures of interest are segmented. It must be pointed out that the patient model can be saved as a file to be later used for diagnosis, surgical planning or simulation purposes. This file also incorporates a description of the image processing graph interactively built during the modeling procedure.

2.2. Major image processing functionalities

Concerning image processing, two kinds of functionalities have been implemented: high-level ones, close to our medical application and context, and low-level ones, which are less specific. We point out this distinction because it
has an impact on the definition of some of our components, as it will clearly appear in the next sections of this paper.

The first one concerns high-level implicit operator sequence construction and parameterization, according to user actions and medical information. Construction corresponds to the high-level mechanism translating user actions applied on the simplified graphical representation of the processing graph (Fig. 2-left) into the one effectively describing operator connections (Fig. 2-right). Parameterization concerns the determination of the regions of interest according to medical structure types defined by the user [16]. The second kind of image processing functionality deals with the smart mechanism implemented at low-level so that computations are correctly synchronized and optimized. It mainly concerns the automatic updating mechanism (see Fig. 3), similar to the one proposed in ITK [19] or VTK [20].

Fig. 2 – Effective image processing graph and graphical representation. (Left) Graphical representation (the one reported in frame F of Fig. 1). (Right) Associated effective image processing graph, describing the relationship between pipelines and underlying operators OPi.

3. Structural description of the proposed software

We shortly describe the role of the different components, reported in Fig. 4, which are involved in our software.

The ui component represents the graphical user interface. This component is intended to manage the set of graphical user interfaces provided by the other components.

The document component is the central part of our software, as it will be further underlined in Section 5.2. It essentially maintains medical structures and an abstract description of the image processing graph built to achieve segmentations. Previously cited high-level implicit mechanisms (corresponding to high-level functionalities reported in Section 2.2) related to image processing are defined and managed by this component.

Fig. 3 – Updating mechanism on a processing graph. The modification of OPb (left graph) invalidates operators in grey boxes in the central graph. Requesting the update of OPd (surrounded by a dashed frame) involves minimal computations (only OPb, OPc and OPd of the right graph are automatically updated, OPe and OPf remaining invalid).
The render component manages everything concerning visualization of the data centralized by the document component. It provides functionalities for configuring the view mode (e.g. 2D or 3D negatoscope representation, number of simultaneous views, intensity windowing) and manipulating rendered objects (e.g. zoom, translation and rotation of the 3D objects).

The history component is used to store the sequence of user actions. This allows to undo and redo them, which is a highly convenient functionality from the user point of view. This component is not further detailed because it is not central enough in our system.

The processing line component encapsulates the image processing graph of operators. This graph is the implementation of the abstract description maintained in the document component. It synchronizes computations of operators and delivers processed and updated data to the document component. It also manages the low-level image processing mechanism that we shortly described in the previous Section 2.2.

To each operator is associated a specific operator component which provides, to the processing line, the core of the considered image processing algorithm. It can also embed its own graphical interface for entering parameters and is adapted to parameter specificity.

4. Structural description of components and contributions

We first propose to describe component content. Then the formal XML-based description of interfaces and contributions is given. Contributions concern either services or resources that components can provide to other ones, through interfaces, when they are combined. This allows to build the high-level software functionalities which have been previously described in Section 2. In the last part of this section, we focus on contributions related to services (i.e. effective code), achieved thanks to a specific design pattern [21].

4.1. Component content

A component is a bundle containing a XML file, a dynamically loadable library and resources:

- The XML file, which is further detailed in the next Section 4.2, describes component content and contributions.
- The dynamically loadable library implements services, assimilated to functionalities, which can be used to extend those of other components, through contributions. For instance, in the case of operator components, the library provides the image processing algorithm. Similarly, as illustrated by Fig. 5, the library contained in the render component contributes to the ui component one by providing functionalities to the main graphical interface (parts surrounded by frames A and C in Fig. 1). This graphical interface allows to manipulate the visualization scene managed by the render component. The technique used to implement such services is described in Section 4.3.
- Resources correspond to a set of files (related to icons, documentation, etc.) that a component can provide to the system. For instance, each operator component can embed a bitmap image to be incorporated within the graphical interface of the system (icons within the part surrounded by frame B in Fig. 1). The parameter list and default values of the operator component are also provided through textual information. Similarly, the document component provides a textual description of organ properties (i.e. type, color, etc.).

4.2. A formal XML-based description of interfaces and contributions

As previously said, the XML file describes contributions and interfaces the component provides. Contributions are formalized by specific a XML element that we called extension: the corresponding component proposes to extend or contribute to specific interfaces (associated to a XML element we called extension point) provided by other components.
Fig. 5 – Graphical representation, associated to a formal XML-based description, of the contribution relationship between the various components (also called bundles) of the architecture. Render, document and processing line components provide graphical user interface contributions to the ui component thanks to extensions matching the views extension point. An extensible number of operator components (components operator 1 to operator n) can contribute to the processing line component through another kind of extension point differing from the one proposed by the ui component.

To facilitate the understanding and for illustration purposes, we propose to describe both notions of extension and extension point in a particular case associated to the contribution of a given operator to the processing line. This example is of great interest because it concerns the extension of system functionalities in terms of image processing operators, which is essential because the software is dedicated to medical image segmentation. This can be assimilated to a formal description of traditional plugins, similar to those described by Papajorgji [13] who directly focused on their technical implementation for another application field, without considering the proposed formal description.

The processing line component declares its intention to use external image processing operators. Thus, it defines an extension point (see Fig. 6), called pl::operators in our case, and a schema (see Fig. 7).

This schema, bundled to the processing line component as a resource file, follows the XML Schema specification [15] and imposes that an operator must provide, for instance, a human-readable name as well as an icon. The parameter list (“parameter-list” in Fig. 7), being optional (“minOccures = 0”), can contain an unbounded number of parameters (“maxOccures = unbounded”), each one having an identifier as well as a default value (could be extended to an acceptance range for instance). To contribute to this extension point, the given operator component declares an extension (see Fig. 8), which contains the required XML elements icon and name, as imposed by the schema file of the processing line component (verification being achieved when contributions are started by our runtime system, described in Section 5.1). Moreover, it provides an icon (which is a resource) and a dynamic library (associated to the XML element library name), which are bundled in the given operator component. The class XML attribute is related to a reference class type used to effectively incorporate the code embedded with the library, as detailed in Section 4.3. In our example, the given operator has one parameter (identified as “threshold” in Fig. 8. Parameters, initially considered as strings, are interpreted (appropriate types such as float or integer) by the instantiated operator at assignment (thanks to the identifier). Data and parameter management is further detailed in Section 5.2.

From this example, we clearly understand that an extension point is the formal declaration of an interface proposed by a component to allow others to extend its features. It can be seen as a contract establishing the requirements to contribute to the extension point (the schema). An extension must fulfill the contract defined by the corresponding extension point.

```
<plugin id="pl">
  <library name="pl"/>
  <extension-point id="pl::operators" schema="pl.operators.xsd"/>
</plugin>
```

Fig. 6 – An extension point declaration. pl is the short cut of processing line. This represents a part of the XML description file of the processing line component (or pl plugin). This file provides the library name (associated to the XML element library name) of this component and an extension point.
Fig. 7 – Part of the XML Schema related to the pl::operators extension-point (file pl.operators.xsd considered in Fig. 6).

Fig. 8 – An extension declaration (validated by the schema reported in Fig. 7).

It facilitates the management of component combination and configuration, as underlined in [14]. Indeed, the visualization of effective contributions becomes really clearer than when they are hard-coded, therefore facilitating the maintaining and the evolution of the structuration of the software. Moreover, contributions can be easily configured. For instance, as it will be detailed in Section 5.1, thanks to such textual descriptions, it becomes possible to easily define the set of components and connections we wish when starting the application (therefore configuring the set of high-level functionalities). The proposed software could therefore be easily converted into a single medical image viewer by disabling components related to image processing (ignoring contributions associated with image processing). Similarly, extensions and extension points can be easily configured. For instance, resources such as default parameter values of operators could be easily adapted, by simply configuring the XML description file.

4.3. Service

If contributing to an extension-point, a component can provide a library implementing a specified interface, finally leading to high-level software functionalities. Technically, this is managed by the use of the Strategy Design Pattern [21], as illustrated by Fig. 9 in the case of two particular contributions.

At build-level [22] (i.e. when compiling the source code leading to the library), the library only includes the abstract class (e.g. IOperator), but does not need to be linked with the library associated to extension-point (e.g. pl::operators). This ensures limited dependencies, only restricted to interfaces.

Both Figs. 5 and 9 report a simplified representation of the contribution network (respectively in a formal manner and in terms of effective service implementation) used to combine our components so that high-level functionalities and information are provided.

5. Behavioral description of the proposed software

In this section, we first describe how contributions are initialized, leading to high-level software functionalities. In particular, we explain how services related operator components are effectively started. Then, we shortly describe the technique used to manage communication between components, focusing on the document and processing line ones.

5.1. Effective contributions: a formal XML-based management system

Effective contributions are achieved by our component management system [14], which we called runtime, dedicated to the combination and the configuration of our components according to extension points and extensions they declare. It is a library using third party ones such as libxml2 [23].
Fig. 9 – Simplified UML-like representation of the Strategy Design Pattern used to implement effective contribution, related to services, between components. IOperator and IView are abstract classes corresponding to component interfaces (related to extension-points pl::operators and ui::views). Inherited classes are concrete implementations extending component functionalities.

Fig. 10 shows the code invoked when the application starts. As we can see, runtime is asked to browse a given directory where components might be located (m_bundlePath in Fig. 10). Each time a component is found, it parses the description file and updates a knowledge base of component extension points and extensions.

The document component is started first because next components (processing line, render, etc.) observe it (see next Section 5.2 for details).

Application’s user interface component ui is started at last, because all the previous ones contribute to it. In this case, the runtime retrieves all extensions (declared by previously started components) associated to the view extension-point declared by ui and starts contributions (in terms of both services and resources), therefore providing views, toolbar buttons, menus, icons, etc., to ui.

Concerning operator components, the behavior slightly differs because contributions related to the code of algorithms are not started at application initialization. Indeed, the instantiation of a given operator is required only if the user explicitly inserts it when processing images. For these components, initialization only concerns resources such as the insertion of buttons (see Fig. 1, frame B), thanks to appropriate contributions to the ui component: the processing line queries runtime to retrieve all extensions contributing to its pl::operators extension point. Then, the processing line only uses the icon and name elements of each operator bundle (declared in extensions, as reported in 8) to create a button in the user interface. It will also associate each extension, in terms of service, to the relevant button for later use.

After this initialization step, the application’s graphical user interface is shown to the user who can start using the software (e.g. performing a patient modeling from an initial image, as described in Section 2).

Thanks to this runtime system, any new component is automatically detected and incorporated in our system, without describing it neither at code level nor in a configuration file. For instance, any new operator component located in the appropriate directory (browsed by the runtime system) becomes automatically available for the user in the toolbar of the application. This avoids the maintaining of a list of the available components. Indeed, the processing line component does not require to know the list of operator components which can contribute to it. As it has been partially discussed in the previous Section 4.2, our runtime system could be easily configured to filter contributions to start. For instance, if it is preferred to start the application in a basic viewer mode, contributions to pl::operators (see extension example 8) could be disabled by providing a simple XML configuration file to the runtime.

5.2. Communication between components

In this section, we propose to shortly describe how the communication between components is managed. When the application runs, most components communicate through an observation mechanism centered on the document component. Technically, the Observer Design Pattern is used [21]: the doc-
ument component is the subject and the other components observe it. In practice, most user actions are applied to document objects which are subjects that send notification messages. These messages can be received by objects (observers) belonging to the other components observing these document subjects. Then, these observers perform appropriate operations with respect to message nature (i.e. according to the initial user action). Therefore, the document component plays the role of a manager with respect to render and processing line components. Such a mechanism requires a preliminary registration step of components (initial attachment).

For illustration purposes, we propose to detail this communication mechanism in the particular case of the processing line component observing the document one.

As illustrated by Fig. 11, in the case of the processing line component, initial attachment requires the creation of a Processing Manager object which is then attached to the Modeling Manager. Both Processing Manager and Modeling Manager are singletons (Singleton Design Pattern [21], only one instance in the system). Singleton instantiation and attachment (if required) are managed by components encapsulating them, by executing an initialization code they embed. This execution is triggered by our runtime system (startBundle() in Fig. 10). This is the reason why the document component must be started before those observing it (in particular processing line, as illustrated by Fig. 10).

When the user starts modeling a set of medical structures from an initial image, it involves the creation of a Modeling Procedure object in the document component. The Modeling Manager notifies this user action (i.e. corresponding to the starting of a new modeling procedure). The Processing Manager receives the related signal and creates an Effective Pipeline which is automatically attached to the Modeling Procedure (observation O1 in Fig. 11). The process is iterated according to user actions directly affecting document thanks to the ui component with which the user interacts (e.g. starting a new sequence or inserting a new operator). This is automatically notified to the processing line component in order to update the effective image processing graph.

Practically, each sequence (e.g. Sequence 1, Sequence 2 and Sequence 3 in Fig. 11) of document encapsulates a pipeline of operators as well as additional notions such as the sequence type shortly described in Section 2 (i.e. either a “target” such as Organ 1, Organ 2 or a “zone” such as Preprocessing). When the user starts a new sequence, this involves the creation of a pipeline in document to be observed by its dual in processing line (e.g. observation O2 in Fig. 11). Operator insertion (i.e. in a pipeline, associated with a given sequence, of document) is notified, thanks to this observation O2, to its dual in processing line, leading to the observation O3 in Fig. 11. This also involves the instantiation of the effective image processing operator (according to the mechanism presented in Section 4.3), as well as its appropriate incorporation within the image processing graph.

Hereafter, we shortly describe how data and operator parameters are managed by our architecture and how the execution of an inserted operator is launched. When a parameter of an inserted operator is modified, a specific notification is sent by the considered operator of the document and received by its dual in processing line (e.g. observation O3 in Fig. 11). This dual operator assigns the parameter to the effective algorithm provided by the appropriate operator component, which translates it into the appropriate type (according to the operator specificity), checks its validity (this can consist in a comparison with a range of acceptation) and raises an exception if relevant. The operator execution can also be requested using this communication mechanism. Input data is retrieved according to operator connections (i.e. the input data being the output one of the previous operator), and computations are started. Finally, the result (output data) is assigned by the dual operator (i.e. in processing line) to the one it observes in document. We use our own data structures (e.g. images supporting various point types) which are shared between our different components to avoid useless buffer copies. They are also shared according to image processing connections, similarly to the design considered in ITK [19] (i.e. the input port of an operator points to the appropriate output port of another operator, according to connections).

The major advantage of this observation based approach is that both image processing graphs (corresponding to the “high-level” and “low-level” functionalities reported in Section 2.2) are synchronized without building a too large component which would incorporate both of them. This avoids polluting the core component (document component) with too much code, which would make refactoring, correction and evolution more complex. Moreover, the processing line component, managing effective operator connections and execution (represented by “gears” in Fig. 11), can therefore be more easily reused for other softwares dealing with another context: marker detection and 2D/3D registration instead of organ segmentation, as in the case of the augmented reality based application presented in [7], for instance. In such a case, this processing line component would only require to be attached to another appropriate core component (similar to our document one) which would be specific to this new software.

6. Discussion and perspectives

The proposed architecture has been proved to be relevant in the sense that the software provides the required functionalities described in Section 2. This has been evaluated by performing several coarse modelings of sets of medical structures, as illustrated in [16]. Some image processing algorithms have been easily developed without any modification or adaptation of other components.

Our patient modeling software could be compared with modular systems such as MeVisLab [17], which seems to be highly modular and evolutive in terms of medical image processing algorithms. This appears to be the main focus of this system, as illustrated by publications involving medical image analysis (see [24,25] for instance). In our sense, one of the main originalities of our approach relies on the use of dynamic components for the various aspects (not solely restricted to image processing algorithms) of our application field: visualization, graphical user interface, processing graph and algorithms. We expect that the implemented foundations, including the XML-based contribution mechanism in particular, will facilitate the
development of additional softwares which do not necessarily involve image processing (e.g. surgical planning and simulation, as later discussed).

Concerning image processing operators, it is possible to integrate algorithms issued from third party libraries such as ITK [19] for instance. One only requires appropriate data structure converters. ITK-based algorithms we implemented process our data which are converted into ITK structures (the result being converted back to our data structure to deliver operator output data to our system). Although ITK pipelines can therefore be integrated, they are not directly connected to our pipeline as part of the graph (can basically be seen as “functions”). The advantage is that our image processing components do not depend on third party libraries. Dependencies can occur at algorithm level, resulting from the choice of the developer to use third party frameworks (requiring appropriate data structure converters), instead of directly interacting with our own data structures.

It would be useful to extend image processing functionalities by creating interfaces for interpreted languages, as considered by ITK [26] with CableSwig for instance. The fast integration and evaluation of algorithms would therefore be further facilitated, as well as the scripting of image processing sequences, as considered in MeVisLab [17]. In our sense, our framework is generic and modular enough to support such an extension.

Another interesting extension of our system would concern the integration of interactivity in image processing operators, with an approach similar to the one proposed in [27]. This would be useful to directly interact with the 3D data for guiding algorithms: selecting image points as seeds for propagation based algorithms, selecting image intensities, manually deforming 3D models … Interactive image segmentation would be further facilitated. Our system is conceived to support such an extension by simply contributing to the render component using the Strategy Design Pattern. In such a case, this component must provide an extension-point, associated with a Strategy interface we could name IInteractor, to which some operator components could contribute (declaration of an extension).

Another point concerns the image processing graph shared between the document component and the processing line one. In our sense, the descriptive representation, close to the medical aspect, could be associated to a specific dedicated component (e.g. a processing manager component). Indeed, incorporating it within the general purpose processing line component does not seem pertinent, as previously underlined. Moreover, pure medical structures and associated general information (e.g. color, type) incorporated within the document should be independent from the processing aspect. For instance, for surgery planning, modeled medical structures are required while image processing is useless. This processing manager

Fig. 11 – Observation based communication between document and processing line components. This illustrates our communication system, which is centered on document and based on the Observer Design Pattern.
could be dedicated to the translation of user actions and to implicit mechanisms which could be specific to patient modeling (similarly to the current one determining regions of interest according to medical structure types), transforming our software into an expert system (the “expert” being the processing manager component).

The next software to be developed with the proposed component oriented architecture will be dedicated to surgery planning. Functionalities related to such a software concerns, for instance, the incorporation of manipulation tools to interact with modeled medical structures (e.g. for liver resection [3]). These functionalities could be provided by a specific component contributing to the ui and render components. Both Observer and Strategy Design Patterns could be advantageously used for this. In our sense, most existing components could be kept, with minimal refactoring (e.g. splitting the document component). This is really more convenient than rewriting an entire software, including a graphical user interface, the management of the rendering of 3D structures, the definition of medical structures... Similarly, our current software architecture could be extended to incorporate virtual surgical tools for simulation purposes.

Concerning software engineering aspects, despite component construction [22] and modeling [22], combination/configuration is essential to reuse components for several softwares. In our sense, our runtime system provides the required foundations, allowing to easily make it evolve so that this aspect can be defined with a high level of abstraction, thanks to formal XML descriptions. For instance, we can define the set of required contributions according to the manipulation mode of the software, as underlined in Section 5: working in research mode (e.g. with an extended view of the processing graph), in basic view mode (only appropriate operators are started)... In our sense, a limited improvement effort is required to manage this evolution.

7. Conclusion

This paper presents a modular component oriented architecture of a software dedicated to medical image processing. Additionally to the efficient programming techniques involved, the proposed formal XML-based runtime system constitutes a powerful approach facilitating the management of contributions between components. This work has been proved relevant to easily extend software functionalities in terms of image processing algorithms.

Besides the considered application, an originality of this work is the application of efficient and modern methods, related to software engineering, to the emerging field of computer aided surgery, involving surgical planning, simulation, image guided intervention... In our sense, it is highly useful for computer scientists working in this field and trying to optimize the reusability of their work.

The next step will consist in extending our software functionalities so that operators can be directly configured or guided from the part dedicated to data visualization in order to further facilitate interactive modeling procedures. Additionally, we expect to prove the evolutivity and the reusability of our component oriented architecture by developing, with minimal refactoring, new softwares dedicated to surgical planning and simulation.

Acknowledgements

First of all, we would like to thank all our medical partners, who have provided us medical images of patients, and in particular Prof. Afshin Gangi and Prof. Catherine Roy from the radiological department of the Civil Hospital in Strasbourg. We also thank the surgical team of Prof. Marescaux for their help in our works, in particular Prof. Mutter, Prof. Leroy, Dr. Vix.

References


