HAL
open science

Sonata : Flexible connections between interaction and business spaces

Guillaume Godet-Bar, Sophie Dupuy-Chessa, Dominique Rieu

To cite this version:

Guillaume Godet-Bar, Sophie Dupuy-Chessa, Dominique Rieu. Sonata : Flexible connections between interaction and business spaces. Journal of Systems and Software, 2012, 85 (5), pp.1105-1118. 10.1016/j.jss.2011.12.030 . hal-00751780

HAL Id: hal-00751780

https://hal.science/hal-00751780

Submitted on 14 Nov 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Sonata: Flexible Connections between Interaction and Business Spaces

Guillaume Godet-Bar[b,*], Sophie Dupuy-Chessa[b], Dominique Rieu[b]

[a] *University of Grenoble, CNRS, LIG*
385, rue de la Bibliothèque
38041 Grenoble Cedex 9, FRANCE

*Corresponding author
Email addresses: `Guillaume.Godet-Bar@imag.fr` (Guillaume Godet-Bar), `Sophie.Dupuy-Chessa@imag.fr` (Sophie Dupuy-Chessa), `Dominique.Rieu@imag.fr` (Dominique Rieu)

Sonata: Flexible Connections between Interaction and Business Spaces

Guillaume Godet-Bar[b,*], Sophie Dupuy-Chessa[b], Dominique Rieu[b]

[b] *University of Grenoble, CNRS, LIG*
385, rue de la Bibliothèque
38041 Grenoble Cedex 9, FRANCE

Abstract

Every interactive system features a functional core and a user interface. Over the years, several types of software architectures for connecting these conceptual elements have been proposed, all of which fail to conciliate two essential qualities: enabling both business and interaction objects reuse, and limiting the amount of communication-specific code in reusable objects.

We have described in previous work the Symphony Architecture, which bridges the gap between the interaction and business spaces, while requiring no code overhead in either business or interaction objects. Resulting development features minimal coupling between technology-agnostic business and interaction constructs, called Symphony Objects, and improves their reusability by clearly isolating them from the applicative logic and from technical objects.

In this paper, we present an original software framework, called Sonata, which capitalizes on the conventions used for building and organizing Symphony Architecture instances, for minimizing the amount of configuration required for setting up connections between the business and interaction spaces.

Keywords: Framework, information systems, software architecture, HCI, aspects

1. Introduction

The last decade has seen several new types of devices and interaction appear on the high tech market. New interface concepts featuring tactile interaction, augmented reality systems and multimodal mobile devices are now widely accessible at a reasonably low price. Although these technologies are generally aimed at the consumer marker, entreprise applications may largely benefit from the intuitive, mobile and pervasive applications that these new interactions permit.

*Corresponding author

Email addresses: `Guillaume.Godet-Bar@imag.fr` (Guillaume Godet-Bar), `Sophie.Dupuy-Chessa@imag.fr` (Sophie Dupuy-Chessa), `Dominique.Rieu@imag.fr` (Dominique Rieu)

Their development is generally supported by rich interaction libraries and frameworks for easily developing innovative interactive applications. However, these libraries and frameworks are connected with functional cores using software architectures that have seen relatively few evolutions since their description in the 1980s. In fact, most of these architectures can be seen as variations of the classical Model-View-Controller (MVC) pattern [17].

When implementing rich interfaces with several execution threads (e.g., for graphics rendering, sensor integration and audio output), are these architectures satisfactory?

We have proposed in past papers [11, 10, 5, 12] our own contribution to this problem: the Extended Symphony Method, whose relevance for efficiently bridging the gap between Software Engineering, Information Systems and Human-Computer Interaction development practices was demonstrated. At the analysis and design levels, our method suggests using a specific combination of architectural patterns for building the application's business and interaction abstract logics, and for designing clearly distinct technical layers (e.g., for implementing the actual graphics), called the Symphony Architecture.

This paper focuses on the implementation level, in which the Symphony Architecture is directly translated into source code, and is supported by a software framework called Sonata, for managing redundant coding efforts induced by these architectural patterns. Sonata also allows connecting graphics and sensor input-intensive objects with complex lifecycles and technical layers, with entreprise objects. This is the contribution that we are presenting in this paper. In particular, we show how these architectural patterns and framework were extracted from their specific methodological context and generalized to the field of interactive software with complex business and interaction logics.

We present in the next section existing works related to the software architectures for decoupling business and interaction logics, and their integration into libraries and frameworks. In Section 3, we propose a method-agnostic process fragment for designing decoupled business and interaction objects. We describe the Sonata framework, which is the original contribution of this paper, as well as the software patterns it relies on, in Section 4. Finally, Section 5 focuses on the ongoing evaluation of our contributions and discussions on the frameworks comparison with MVC and scalability, before we conclude this paper with some perspective on future work.

2. Related Work

Software architectures and patterns for decoupling the business and interaction spaces are certainly not a recent research topic. Most of the seminal work on this subject was produced during the 1980–1990 decade, and most of today's software architectures of the business–interaction bridge could be described as variations of these initial endeavours. We identified three general types of software architectures: monolithic architectures, distributed architectures and hybrid architectures. We also discuss in this section the existing

frameworks, libraries or tools for supporting or integrating the architectures whenever possible.

Monolithic architectures are based on a tripartite conceptual structure, composed of (1) a business construct; (2) an interaction construct; (3) a control construct, which is responsible for translating method calls between the two other constructs. The Seeheim model [13] was chronologically the first software architecture to feature this decomposition. The Arch model [2] is a refinement of the Seeheim model, where the business and interaction constructs are both subdivided into purely conceptual sets of objects, which act as wrappers for the lower-level sets of technical objects.

Distributed architectures decompose the system's architecture into finely grained agents, which manage a limited set of responsibilities, while still guaranteeing the business/interaction decomposition. The PAC [6] and MVC [17] agents feature tripartite structures, composed of (1) a business object (MVC's Model and PAC's Abstraction); (2) an interaction object (MVC's View and PAC's Presentation) and (3) an object acting as a mediator (MVC's Controller and PAC's Control). Apple's Cocoa graphics framework, which powers desktop and mobile devices, features a MVC-based architecture that resembles the Seeheim model[1] (i.e., Model and View elements communicate only with the Controller). However, due to the dynamic nature of the implementation language (i.e., Objective-C), the actual connection between objects is specified at design time and automatically negotiated by the language runtime during execution.

The communication pattern between MVC's facets is constrained for guaranteeing both the decoupling of the Model from the Controller and the View, and from other agents: any software construct wishing to follow the Model facet's behaviour needs to register itself (following the well-known Observer pattern [9]). Whenever the Model reaches a state that registered agents are *listening* to, notifications are sent to the latter for performing their own translations and reactions.

The communication model of the PAC architectural style is also its main shortcoming. Whereas MVC is characterized by a non-hierarchical organization (its agents may interact with one another through the Controller), PAC-based systems presents a tree-like structure: every agent is itself managed by a parent agent, up to the root agent. Moreover, two agents interact by sending calls through the agent tree, which will be re-emitted by each agent on the call's path. Therefore, this architectural style generates a large volume of message overhead, which is both a development and a runtime burden.

AMF [25] extends the PAC model by integrating facets that address a wider variety of concerns, such as security or auto-diagnostic, which may be dynamically loaded and administered. To this extent, AMF shares several characteristics with service-oriented frameworks, while keeping the tree structure of the original PAC model. Additionally, a tooling support has been developed for

[1] `http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html`

AMF, for facilitating the design and composition of agents and facets.

Apart from Apple's adaptation, there are several other variations of the MVC model, generally with different levels of coupling between the Controller and the View. For example, Sun's Swing library[2] relies on a MVC variation where both the Controller and the View are encapsulated into graphical widgets. The MVC2 alternative, which is adopted by several web application frameworks such as Struts or Ruby on Rails[3], integrates all the controllers into a single entity, which is responsible for dispatching requests between the Models and the Views.

What could be considered as hybrid architectures have been proposed as a solution to some of the issues encountered with the other types of architectures. The PAC-Amodeus model [22] merges the PAC and Arch models, with a PAC tree of agents serving as a fine-grained controller, its Abstraction facets pointing to elements of the business domain, and its Presentation facets pointing to elements of the interaction domain. While the coordination of agents still induces large message overheads, PAC-Amodeus reduces the constraints imposed on software interactions between elements of the same domain.

The architectures we have presented conceive the UI as a set of objects subordinated to the business space. However, rich interfaces feature their own execution processes (e.g. for handling the rendering of a 3D scene, as well as multimodal inputs), which may not be synchronized with the business space's. Moreover, because interaction processes are not clearly incarnated in these architectures, the interaction logics are scattered amongst the control and representation constructs. This compromises the maintainability of applications, which is not sustainable for large (entreprise) applications.

In this paper, we demonstrate how the Symphony Architecture, which addresses the problem of business/interaction decoupling, and whose instantiation is guided by the Symphony method, can be efficiently and effectively supported by the software framework Sonata.

3. Symphony Architecture

Our contributions were constructed during our work on the Symphony method. After a summary of the past contributions related to this method, and the presentation of an example application that is used throughout the paper, we focus on the description of the Symphony Architecture.

3.1. History of the Symphony Method

Symphony is an iterative and incremental, user-oriented, business component-driven development process originally proposed by the Umanis company. It has been first extended and formalized by Hassine et al. [14], to improve the reusability of components, in a *by* and *for* reuse approach. Obviously, these components do not reflect operational choices, and are thus **not related to the concept**

[2] `http://java.sun.com/products/jfc/tsc/articles/architecture/`
[3] `http://www.rubyonrails.org`

of component as defined by the CBSE domain. Components used in this context refer to a definition from the Information Systems domain, as "a partial representation of the information system[,] from the requirements engineering to the deployment and the execution[, which] has to be adapted [either] to the domain [or] to the application" [14]. As a matter of fact, from a purely orthogonal Software Engineering perspective, business and interaction components are structured using Object-Oriented concepts, and described using almost standard UML class diagrams.

In [12, 11], we envisaged the involvement of different development cultures in the method's process, and described collaboration mechanisms between development experts. We also described the principles of a method integrating Human-Computer Interaction and business practices for augmented reality systems, focusing on the concept of "Interactional Object" as counterparts of the already existing Business Objects. Our following concern was to present the principles of business evolution, when triggered by interaction choices [10].

Finally, we described some aspects of the method's sub-process for designing the software's technical architecture, as well as its merging with its analysis (i.e., functional) model in [7]. Details concerning HCI design were also addressed in [7]. The quality of the analysis model that the method enables developers to produce was evaluated in [5].

Fig. 1 describes the main phases of the Symphony development cycle. Overall, while past publications focused on the conceptual and organizational steps of the Symphony method (i.e., the left branch of the method), this article focuses on its operational concerns (in particular, the "Design" and "Implementation" phases, where functional constructs are integrated with technical decisions). Therefore, although we summarize in Section 3.3 the main characteristics of the methodology, interested readers are invited to consult given references, in particular [11], for details on its earliest steps.

3.2. Example Application

To facilitate the understanding of our approach, we chose a very simple (and non innovative) example application implemented using the Symphony Architecture, called "TeamAvatars". This toy application consists in displaying the members of a research team along concentric circles and following an arbitrary rule: the team leader is at the center of the wheel, the permanent members on the first circle, PhD students on the second circle, engineers and interns on the third circle. Mentorship relations are materialized using dotted lines. When the user selects a team member, the property box that contains his/her name is extended so as to show his/her office and email, while the other team members are dimmed.

3.3. Symphony Architecture's Characteristics

From the Symphony architecture's point of view, the business and interaction conceptual spaces are described as conceptual constructs, called Business Objects and Interactional Objects.

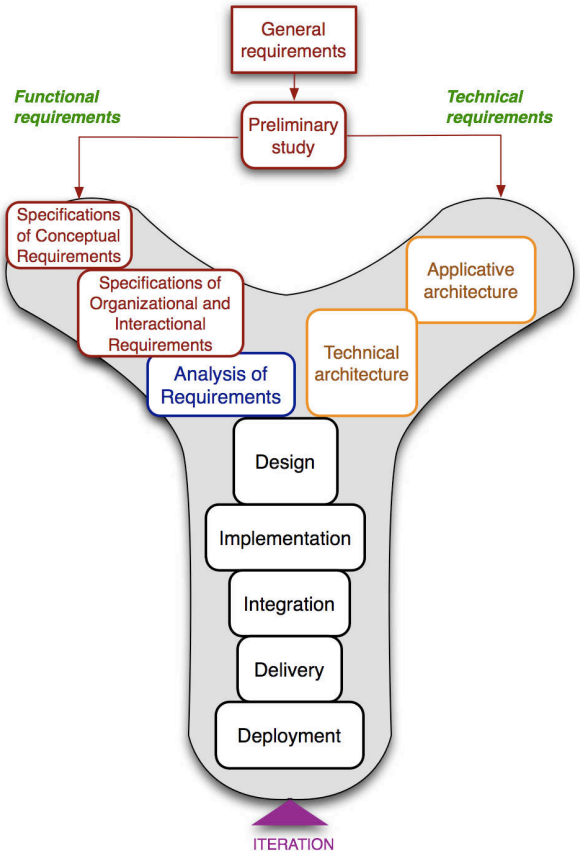


Figure 1: Symphony Development Cycle

We refer to both Business Objects and Interactional Objects under the general term of "Symphony Objects". Specific constructs, called Translation classes, manage the conversion of business semantics into interaction semantics, and conversely. For example, our example application is composed of:

- a **business space** that describes the organization of research teams and provides methods for adding and deleting members. The business space also integrates applicative rules: for instance, a Ph.D student must be mentored by at least one permanent member of the team, interns must be mentored by either a Ph.D student, a permanent member or the team leader.

- an **interaction space** that details the human-computer interface of the application, such as the position of the avatars and their property boxes, or the layout of the scene. It features specific rules, such as the fact that the avatars should be laid out so as not to cover each other, the fact that

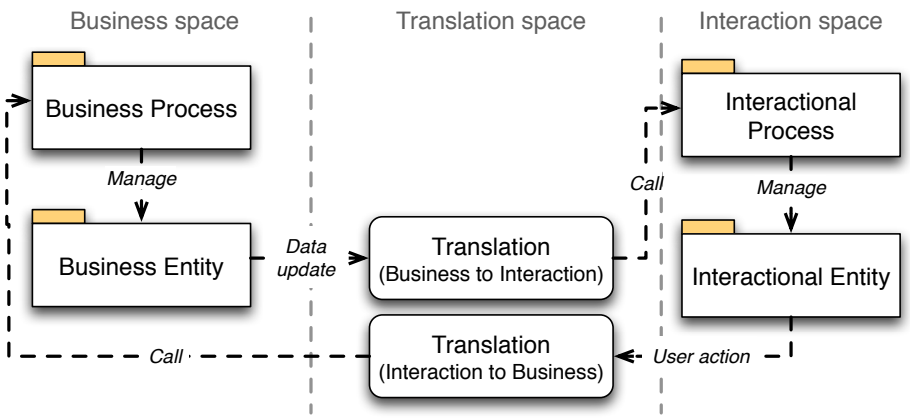


Figure 2: Symphony architecture (conceptual view)

a mouse click on the property box extends it (for displaying additional information), or the description of the animations that may take place when a member is added or deleted.

- a **translation space** that expresses the conversion of the semantics of the triggering event (e.g., the adding of a new member to the team, in the business space) into the other conceptual space's domain language (e.g., the creation of new avatar and of an associated property box, in the interaction space, and the translation of the new member's position in the team into a position on a corresponding circle of the GUI).

Fig. 2 illustrates the three conceptual spaces (business on the left, interaction on the right, translation in the middle), and shows that Symphony Objects may be refined into:

- **Entity Objects**, which describe essential concepts of the application, for instance the "Team" and "Member" from the business space, and the concept of "Avatar" (i.e., the team members' photographs, as well as their attributes such as size, position, orientation, transparency etc.) from the interaction space.

- **Process Objects**, which describe the rules and constraints for managing Entity Objects that are specific to the application. For example, the Interactional Object Entities are managed by the "ManageAvatarScene" Interactional Object Process, and the "ManageTeamMembers" Business Object Process describes the applicative rules for adding or deleting team members.

3.4. Symphony Objects and Translation Classes

The analysis model corresponds to the system's functional architecture that addresses business and interaction rules, and ignores at first technical concerns.

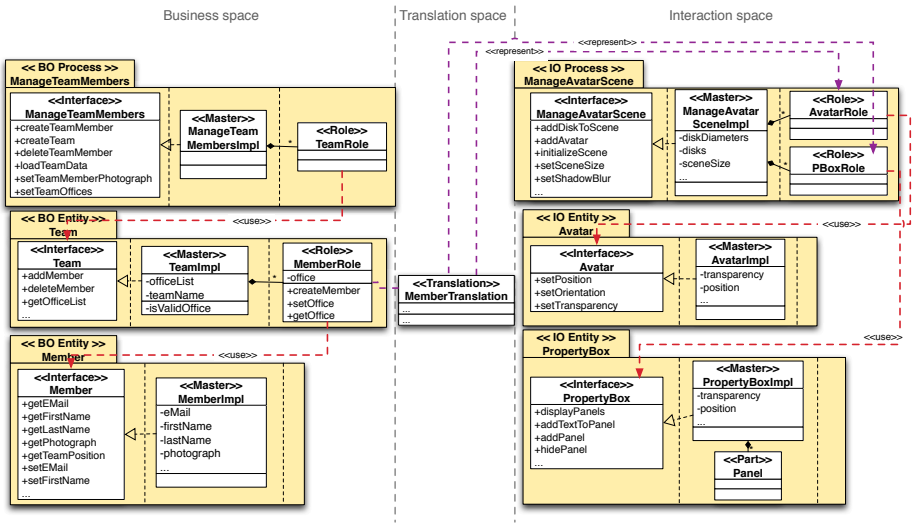


Figure 3: Organization of the Symphony Objects of the TeamAvatars application

This functional architecture is described following systematic rules, as illustrated in Fig. 3.

Symphony Objects are described as tripartite packages. Each package contains conceptual elements whose roles are identified using four stereotypes: <<Interface>>, <<Master>>, <<Part>> and <<Role>>. The left part of the package describes the methods provided by the object, using an <<Interface>> class (e.g., the "Avatar" interface). The central part of the package describes the implementation of these methods (using a <<Master>> class, for instance the "AvatarImpl" class), as well as a subdivision of complementary concepts (using <<Part>> classes, such as the "Panel" class). Finally, the right part is composed of <<Role>> classes, which encapsulate required methods (following a classic delegation pattern [9]) with their adaptation to the object's context (for instance, the "Team" Business Object Entity adapts the "Member" Business Object Entity by overloading it with the concept of office that may be set and retrieved).

Additionally, Symphony Objects can be linked using relations stereotypes as <<use>> or <<represent>>:

- The **use** relationships permit to specify dependency relationships between a Process or Entity Object, and an Entity Object. For example, the "Team" Entity Object depends on the "Member" Entity Object for carrying out its responsibilities.

- The **represent** relationships are drawn between one or several Interactional Object Entities and a single Business Object Entity. They indicate that the Interactional Object(s) correspond to a projection of business concepts into the interaction space. For instance, the "PropertyBox" and

"Avatar" Interactional Objects are projections of the "Member" Business Object.

Although the actual implementation-level architecture of Symphony applications closely matches their functional models, the concept used for describing Symphony Objects in the analysis model (as well as at the specifications level) is related to the organizational stages of the method, and reflects the core motivations of the original Symphony method, which is to build abstractions of business concerns, *by* and *for* reuse. As mentioned previously, one of our contributions to the method was in fact to adapt this approach to human-computer interaction concerns.

This conceptualization of the system as a set of independent and interconnected Symphony Objects encourages their modularity and reuse, as well as that of their specifications.

In order to describe the Translation classes, the <<represent>> relationships are first refined by studying what type of event (e.g. in a Business Object) may trigger a modification (e.g., in an Interactional Object). We adopted a simple grammar inspired by the Aspect-Oriented Programming paradigm [16] for modeling the characteristics of a "connection event", called Symphony Advice. This grammar was also translated into a UML profile. Fig. 4 presents an extract from such a model, in which the "createMember" method from the "MemberRole" Role triggers a connection event identified by the "createAvatarFromMember" name, as soon as it returns ("AFTER" keyword).

Once all the connection events are identified, developers describe their behaviour. A "Translation" class is associated to every Symphony Object that triggers connection events. Likewise, at runtime, a "Translation" object is created for every Symphony Object instance that triggers connection events. For instance, Fig. 5 describes the behaviour of the "createAvatarFromMember" connection event identified in Fig. 4. Whenever a "Member" Business Object is created (through its "MemberRole" Role), a new "Avatar" Interactional Object and its "PropertyBox" are instantiated for representing the actual member in the interactional space (i.e., using its name and photograph).

However, Translation objects should not modify the target Entity Objects, which are considered in this case as "read-only" constructs. Instead, a Process Object is used as a proxy, which guarantees the correct application of the interactional applicative rules (e.g., the relative movement of the avatars and their property boxes). Thus, in Fig. 5, the *trans*1 MemberTranslation object

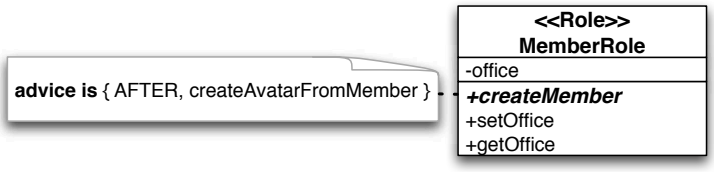


Figure 4: Method identified as triggering a connection event

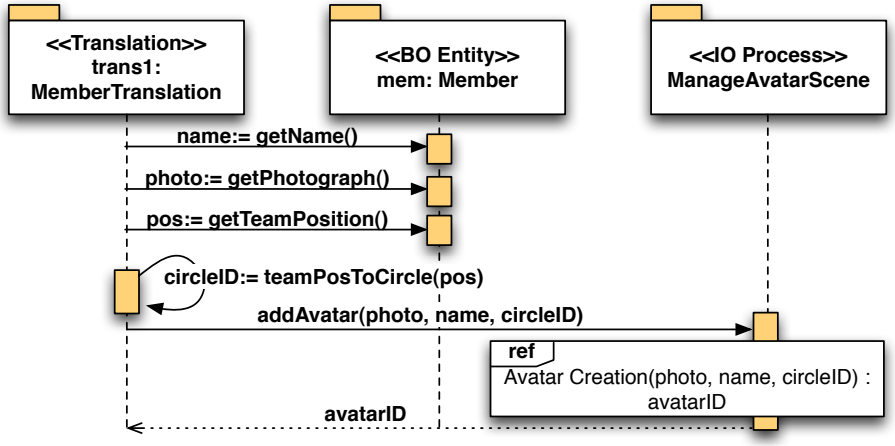


Figure 5: "createAvatarFromMember" connection event semantics, handled by a 'MemberTranslation' instance

calls the "ManageAvatarScene" Process Object for creating the Avatar and the PropertyBox that will represent the "Member" Business Object.

Although the software development practices presented in this section emanate from a specific development method, evaluations made in previous work [5] lead us to believe that the software architecture these practices enable developers to build may be used outside of the method's scope.

3.5. Relevance of the Development Approach

The following properties of the software architectures that our methodology enables developers to construct may be used as the basis for defining architectural patterns:

- The dependencies between elements within the same conceptual space are simple to manage: every relation is mediated by a <<Role>> class, which bears all the weight of the coupling between each pair of objects.
- There is a strong decoupling between business and interaction Symphony Objects: there is no direct relationship between the two conceptual spaces, both at the analysis and implementation levels. There is an indirect dependency, however, which is managed by the Translation classes, but only as an efferent coupling (that is, the Translation class may call either Business or Interactional Process Objects).
- The Translation classes are the only loci in the implementation code that manage the relation between the business and interaction conceptual spaces.

We consider that these elements advocate for a generalization of the Symphony architecture to any type of interactive system that features non-trivial

interaction objects. Indeed, this architecture allows developers to construct the business and interaction spaces in relative isolation, while the bridge between the two spaces is clearly separated from either spaces.

Additionally, our experience shows [11] that structuring the interaction as a consistent organization of elements that are not directly business-dependent, and deferring the description of the interaction-business link, enables HCI developers to explore innovative interaction solutions early in the design process. Once translated into an implementation model, the resulting software architecture facilitates the connection of interaction object with complex lifecycles, with entreprise business objects and information systems.

3.6. Synthesis

The Symphony Architecture addresses the problem of business/interaction decoupling. It features a semi-hierchical organization of business and interaction objects, and capitalizes on the assets of the different types of existing architectures, such as the separation of abstract logics from technical objects (Arch/Pac-Amodeus), the tripartite structure of the business–interaction link, and the clear separation of the logics for converting interaction data to business data (and conversely) from PAC. Our architectural pattern manages to avoid most of the inconveniences of anterior patterns, such as inter-object messaging overhead and the scattering of the application's business and interaction logic.

Moreover, the implementation of Business and Interactional Objects, of the aspects for interfacing Symphony Objects with their Translation classes, follows regular, systematic rules. For this reason, we have developed two UML profiles for describing Symphony constructs at the specification and analysis levels, and a profile for annotating UML class diagrams (exemplified in Fig. 4) with aspect-related information. This formalization of the Symphony Architecture enabled us to automate large parts of the development cycle and of the runtime. While we rely on model transformations [15] for handling the transition between specification and analysis models, most of the development and runtime automation relies on an original software framework, called Sonata.

In the following section, we describe the main software entities that constitue Sonata, as well as their interactions. We also describe how technical objects may be integrated into the architecture and connected to Symphony Objects (which are functional and library-agnostic), for permitting the rendering of rich, multithreaded user interfaces featuring innovative interaction techniques.

4. The Sonata Framework

4.1. Goals

The Sonata framework was initially designed for facilitating the implementation of Java applications developed with the Symphony method. It was later extracted from the method and built as an independent entity, which nevertheless borrows some concepts from the Symphony method, such as its decomposition of an application into Business and Interactional Objects, mediated by Translation classes.

Sonata permits to easily setup the connections between the business and the interaction parts of an application. It was also designed for directly implementing the Symphony Architecture analysis model, with a minimum effort overhead, while maintaining its properties (i.e. strong decoupling between Business and Interactional Objects, and clear location of the translation and applicative logics).

Sonata also allows designing the abstract and technical objects layers using heterogeneous architectural styles. While the abstract layer adopts the Symphony Architecture described in the previous section, the technical layer may include intensive rendering loops or sensor integration threads that are classical features of complex interfaces (e.g., OpenGL based interfaces with camera input, such as can be found in ARToolkit[4]) [5].

This approach enables developers to adapt the application's architecture to the interactional requirements (i.e. user and environment input, and rendering loops), while keeping an information systems-like integration with the business space.

For achieving these goals, Sonata relies on a "Convention over Configuration" approach, that is, fixed practices for object structure and organization. Provided that these conventions are followed by developers, the framework guarantees decoupling and automatic connection properties.

Sonata is composed of a development-time model-driven tool-chain and a runtime library. The runtime library's architecture is described in Section 4.3, the Software Engineering conventions that should be followed for integrating Sonata into a Java application are described in Section 4.4, and the tool-chain for automating this effort is described in Section 4.5.

4.2. Functional Concepts of the Framework

While some of the Symphony Architecture's concepts are conceptual and do not correspond to any functionality (e.g., the notions of Business Object and Interactional Object, which are purely organizational), 6 of its constructs are operational: (1) Process Object; (2) Entity Object; (3) Symphony Role; (4) Translation class; (5) Symphony Advice; (6) Symphony Connection.

The first five concepts were extensively discussed in the previous section. Symphony Connections gather Entity Objects and Translation classes into a description of all the elements responsible for a given association between one or several source Symphony Objects, and a target Symphony Object, mediated by a Translation class. A Symphony Connection may therefore be defined as a tuple such that:

$[source_1, source_2, \ldots, source_n], target, translation$

These constructs are integrated into the Sonata framework as Java abstract classes, interfaces and/or handled by utility classes. They provide developers

[4] `http://www.hitl.washington.edu/artoolkit/`

[5] Although Sonata does not formally address the integration of low-level languages for handling rendering loops, such as C or C++, experimentations with the Java Native Access library (`http://github.com/twall/jna`) have given good results.

with elegant mechanisms for bridging the business/interaction gap and managing technical objects. Sonata also relies on three entities, which are described in the next paragraphs.

4.3. Main Entities of the Runtime Library

We focus in this section on the three essential objects on which the Sonata runtime library is built: the Initializer, the Entity Factory and the Invoker. Fig. 6 presents a simplified view of the runtime library's architecture, as well as some of the minor classes that are used by the main entities. The development practice that allows designers to interact with these objects is described in the following section.

4.3.1. Initializer

The Initializer extracts all the relevant data from the few configuration files that are required by the framework, i.e.:

- It registers Symphony Objects with the Entity Factory (see below).
- It registers basic properties for each Symphony Object, i.e., attributes that will be set when the Symphony Object is instantiated. The underlying mechanism is similar to the one used for setting JavaBeans[6] properties in that it uses the name of the property (for instance, 'name') for generating its setter method name (for instance, `setName`).
- It loads and registers the technical objects identified in configuration files or in the source code for connecting them with the appropriate Symphony Objects, when the latter are instantiated, using a Dependency Injection mechanism [18].
- It loads the configuration of the Symphony Connections, that is, a set of Symphony Connection tuples.

4.3.2. Entity Factory

The Entity Factory manages the Symphony Objects' lifecycles, as well as common functionality such as searching by object ID, adding and removing objects from a list of registrations. It ensures that the complex initialization process is carried out along the following steps:

1. The Symphony Object's constructor is loaded and executed for producing a new object instance.
2. If any registered technical object implements one of the Symphony Object's technical interfaces, then it is instantiated and its reference is associated to the Symphony Object's instance.

[6] `http://java.sun.com/javase/technologies/desktop/javabeans/`

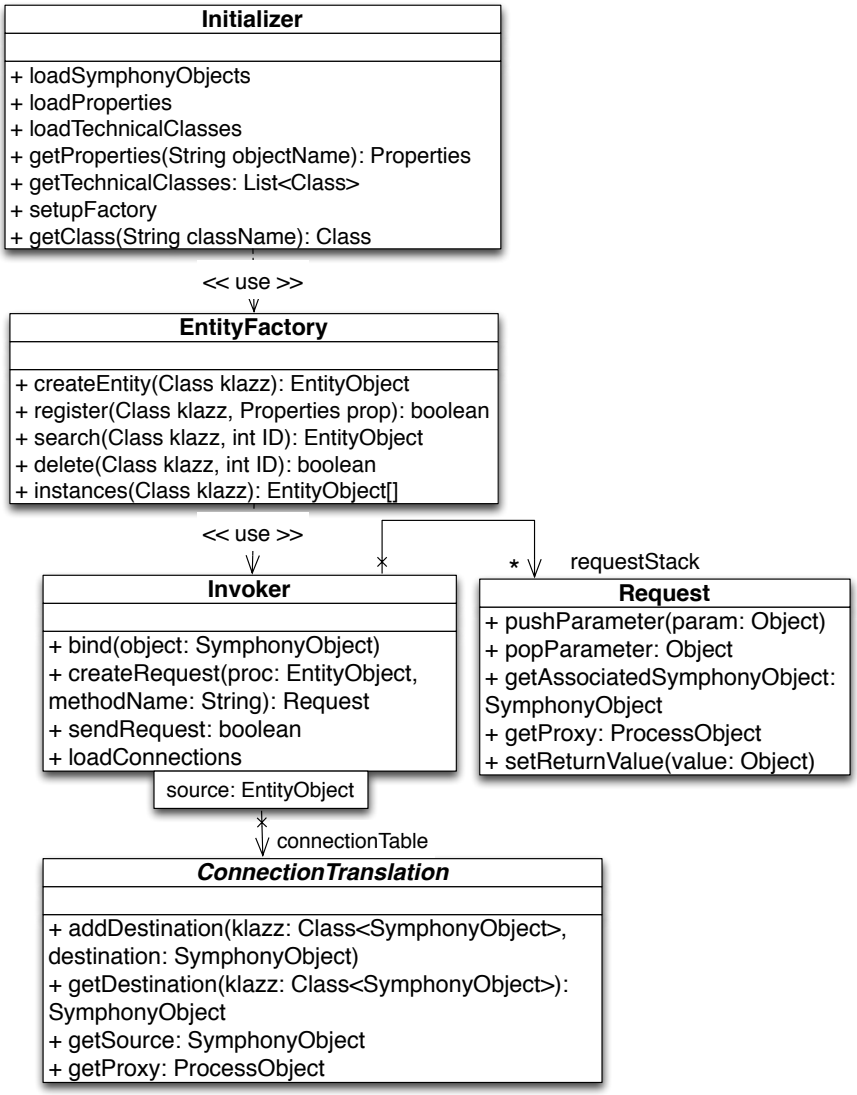


Figure 6: Sonata's simplified architecture

3. If any property was registered with the Initializer, it is injected in the new object instance (and therefore, if necessary, it is transferred to the relevant technical objects; for example, the dimensions of an "Avatar" instance should be transferred to the technical object that manages its graphical rendering).
4. An identifier is computed, whose uniqueness over the object's Java Virtual Machine (JVM) is guaranteed.
5. The object is added to the list of registered instances held by the Entity Factory.
6. The Entity Factory calls the Invoker (using a `bind` method) for checking whether the object should be bound to another Symphony Object for satisfying a Symphony Connection (see below).

4.3.3. Invoker

The Invoker essentially manages Symphony Connections, that is, Translation class instances. Fig. 6 presents the `ConnectionTranslation` class that Translation classes must extend.

Symphony Objects interact with the Invoker using Request objects, which are created whenever a Symphony Connection is triggered. The creation of Request objects occurs within an aspect [16], which is woven into the corresponding Symphony Objects at runtime. A Request object keeps track of the Symphony Object that triggered the Symphony Connection, as well as the name of the Translation method that should be called.

Upon receiving the Request, the Invoker either creates or completes a Symphony Connection, depending on the following criteria: (1) when no Request is currently being treated, a Symphony Connection instance is created for every new Symphony Object instance (through the call to the `bind` method). The latter is considered as the source element of the Symphony Connection instance; (2) if a Request is currently being treated, then the Symphony Object was created during the execution of a method from a Translation class (this is actually verified by inspecting the JVM's call stack). Therefore, it is considered as the target element of the Symphony Connection instance.

The Invoker borrows some aspects of its behaviour from the Broker pattern [4], as it uses the Request object for determining which Translation object should be called, using the introspection mechanisms integrated into the Java language.

The execution of the Translation methods occurs in the same thread as the Symphony Object that triggered the Symphony Connection. Thus all the Symphony Object instances exist within the same thread; this guarantees a certain level of consistency between the lifecycles of Business and Interactional Objects: any connection event must be fully resolved before another one may be taken into account. This restriction does not apply to technical objects, which may operate on several threads (including their parent Symphony Object's), for instance for using graphics rendering loops or multimodal integration engines.

In our example, let us suppose that a Request reaches the Invoker with a reference to a "Member" Business Object Entity, the `createAvatarFromMember`

Translation method name and a list of arguments that should be transferred to the Translation's method. The Invoker first identifies the Translation class instance that is associated to the "Member" instance. By introspecting the Translation class, the Invoker generates a call to the method identified in the Request object by its name, using the list of arguments that are encapsulated in the Request. During the execution of the `createAvatarFromMember` method, a new "Avatar" instance is created (using the Entity Factory), which is bound to the "Member" instance by the Invoker.

4.4. Application Development for Sonata

From the developper's point of view, the Sonata framework stands as a set of helpers and tools, and a Java runtime library whose API allows connecting with the Sonata system. We describe in this section every step of the development process, as well as the services that are offered by Sonata for each of these steps.

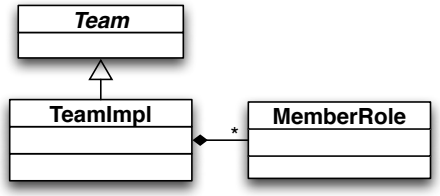
4.4.1. Symphony Objects Implementation

Figure 3 shows that Symphony Objects should feature at least an Interface class and a Master class, and that it may optionally feature an unlimited number of Role classes. For implementing these features, Sonata provides developers with marker interfaces, that is, empty Java interfaces whose only function is to specify the role that the implementing class should play, in order for software frameworks to apply specific behaviours when manipulating instances of these classes. For instance, the `Serializable` Java interface is a marker interface, which indicates to the JVM that its implementing classes should be part of the serialization processes.

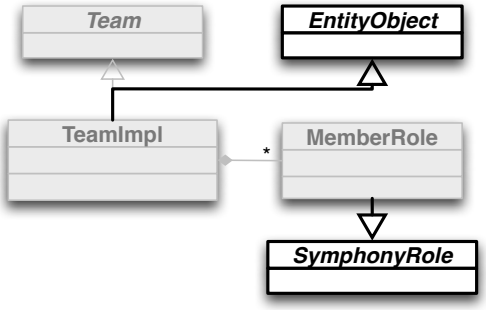
Concerning the Sonata framework, the marker interfaces enable the framework to add several sets of helper methods to the object instances. The rules for implementing Symphony Objects are as follows:

- The Interface class is implemented using the Java language's construct for interfaces. The name of the interface must match the name of the Symphony Object.
- The Master class must implement the interface identified as the Interface class, as well as either the `EntityObject` or `ProcessObject` marker interfaces, depending on the Symphony Object's applicative role. Additionally, the Master class's name must match the Interface class's, suffixed by `Impl`. For instance, given the "Avatar" Entity Object, then its Interface class should be named `Avatar` and its Master class `AvatarImpl`.
- Each Role class must implement the `SymphonyRole` marker interface.

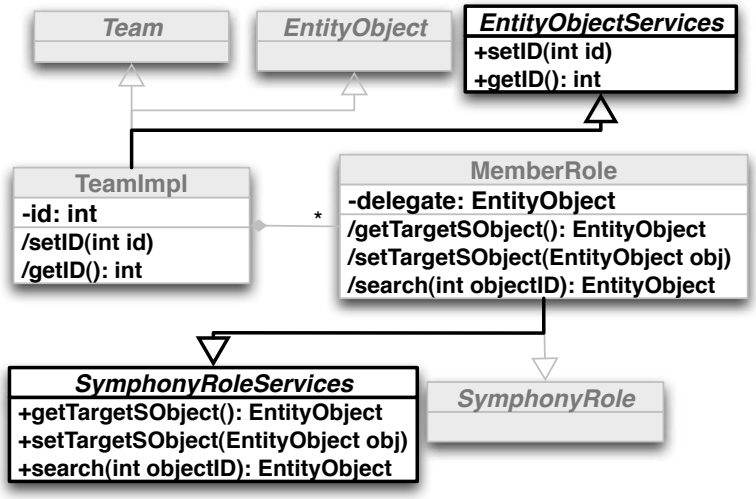
Fig. 7 illustrates these rules, using an excerpt from Fig. 3. Figure 7(a) shows the analysis level description of the "Team" Entity Object; Figure 7(b) shows its design level description and Figure 7(c) shows the services that are consequently woven into the Symphony Object instances.



(a) Initial structure of the "Team" Symphony Object (Analysis level)



(b) Marker interfaces (manually added by the designer, at Design/Implementation level)



(c) Automatic weaving of services

Figure 7: Weaving of services into Symphony Object instances

The latter mechanism corresponds to an aspect-based Java implementation of the "Mixin" mechanism, which was introduced in the Flavors language [20]

and that may now be found in Object-Oriented languages such as Ruby or C++. It is used for adding sets of methods and attributes to object instances. The following code sample describes how an "ObjectA" instance may be created, and how its identifier may be obtained, using the services that are accessed through the `EntityObjectServices` interface that is woven into it.

```
ObjectA obj = (ObjectA)EntityFactory
                  .createEntity(ObjectA.class) ;
int objId = ((EntityObjectServices)obj).getID() ;
```

Likewise, several methods are woven into Role class instances, and accessed through the `SymphonyRoleServices` interface, which are essentially used by Sonata for managing Symphony Connections (see List. 2).

Symphony Objects are interfaced with technical objects through Java interface classes, following a classic dependency injection pattern [18]. In order for Sonata to be able to recognize these interface classes, it is required that they extend the framework's `TechnicalObject` marker interface. This allows the framework to recognize and manage these technical objects, as shown previously. Finally, technical object instances are associated to Symphony Objects at runtime, based on a file or source code configuration.

For instance, the "Avatar" Interactional Object Entity includes an interface called `AvatarTechnicalLayer`, which describes methods for displaying and moving the team member's photographs. Therefore, we built a `SwingAvatar` class, which implements the `AvatarTechnicalLayer` interface, and permits integrating the avatars in Swing-based GUIs.

4.4.2. Symphony Connection Setup

Configuration of the Connections. Developers first describe all the Symphony Connection tuples (Section 4.2), before they are loaded by the Initializer. These descriptions may currently be saved as XML or JSON objects, or directly implemented in the application's source code. Developers may extend the loading mechanisms by implementing a specific `InitializerDAO` Java interface. List. 1 presents an excerpt from an XML configuration file, between the "Member" Business Object Entity and the "Avatar" and "PropertyBox" Interactional Object Entities.

Listing 1: XML configuration example

```
  <SOConnection>
    <source name="business.entity.Member" />
    <destination name="interactional.entity.Avatar" />
    <destination name="interactional.entity.PropertyBox" />
    <translation name="control.translation.MemberTranslation
        " />
  </SOConnection>
...
```

Description of the aspects for connecting Symphony Objects with the Invoker. Once the Symphony Connections are described for the Initalizer to load at runtime, the methods that trigger connection events should be marked. An aspect is described for each Symphony Object that acts as the source object of a Symphony Connnection. For instance, as the "MemberRole" object acts as a source object (see Fig. 3), a "MemberRoleAspect" aspect should be described that captures all the method calls that should be translated and transmitted to the corresponding "Avatar" and "PropertyBox" Interactional Object instances. This aspect is based on the conceptual advice defined during the analysis phase (for instance as illustrated in Fig. 4). List. 2 presents an excerpt of the "MemberRoleAspect" aspect, using AspectJ[7].

Listing 2: Excerpt of aspect "MemberRoleAspect"

```
1  public aspect MemberRoleAspect {
2
3    pointcut MemberRoleCalls(): execution(public * MemberRole
         .*(..)) ;
4  ...
5    after() returning(MemberRole target): execution(MemberRole
         createMember(firstName,lastName,office)) {
6      try{
7        Request req = Invoker.getInstance().createRequest(((
             SymphonyObjectServices)target).getTargetSObject(),
8            "createAvatarFromMember");
9        req.pushParameter(firstName) ;
10       req.pushParameter(lastName) ;
11       req.pushParameter(office) ;
12       Invoker.getInstance().sendRequest();
13     } catch (Exception e) {...}
14   }
15 }
```

Every aspect first describes a pointcut (line 3) that limits the weaving to instances of the Role class (here, the `MemberRole` class). The advice *per se* (lines 5–14) are built following a systematic structure:

- The first line of the advice captures the method that should trigger a connection event (e.g., the `createMember` method).
- The Invoker should be called for creating a new Request object, which should contain the name of the Translation method that will bridge the business–interaction semantic gap.
- All the method's arguments should be passed to the Request object.
- The Request should finally be executed (i.e., sent to the Invoker).

[7] `http://www.eclipse.org/aspectj`

Implementation of the Translation classes. Finally, the developer should implement the method that will enable the business space's semantics to be converted to the interaction space's (or conversely), into a Translation class. A Translation class is essentially a class that extends the `ConnectionTranslation` abstract class (see Fig. 6). List. 3 presents an excerpt from the implementation of the `MemberTranslation` class (as described in Fig. 5), which manages all the translations between "Member" Business Object instances and "Avatar" and "PropertyBox" Interactional Object instances.

Listing 3: Excerpt from the "MemberTranslation" implementation

```
public class MemberTranslation extends
    ConnectionTranslation {
  ...
  public int createAvatarFromMember() {
    String name= ((Member)source).getName();
    Image photo= ((Member)source).getPhotograph();
    TeamPosition pos= ((Member)source).getTeamPosition();
    int circleID= teamPosToCircle(pos);
    int avatarID= ((ManageAvatarScene)proxy).addAvatar(
        photo, name, circleID);
    return avatarID;
  }
  int teamPosToCircle(TeamPosition pos) {...}
}
```

As mentioned earlier, the calls to the source Symphony Object (i.e., the "Member" instance) are read-only actions, while the side-effect inducing code goes through a Process Object (i.e., a "ManageAvatarScene" instance, which acts as a proxy) that maintains applicative rules. The actual translation is realized in the `teamPosToCircle` method. Additionally, the code contained in the `MemberTranslation` class is the only locus in the whole application where the "Member", "Avatar" and "PropertyBox" implementation code meet, thus reducing the coupling between the interaction and business spaces to an acceptable minimum.

Fig. 8 summarizes (and condenses) the main events that occur when the `createMemberRole` method is called. The part of the sequence diagram that is above the dashed box represents the set of calls that are triggered in the business space, for creating a new team member. The calls within the dashed box are woven automatically to the "MemberRole" class, following the constraints described in Fig. 4, and correspond to the triggering of a connection event: a request is sent to the Invoker, which calls the "MemberTranslation" class for creating an "Avatar" and a "PropertyBox" Interactional Objects, as described in Fig. 5.

4.5. Development Process Automation Using Sonata's Model-Driven Tool-Chain

The development of the Sonata framework reflects the global automation efforts undertaken to facilitate the execution of the Symphony methodology. This

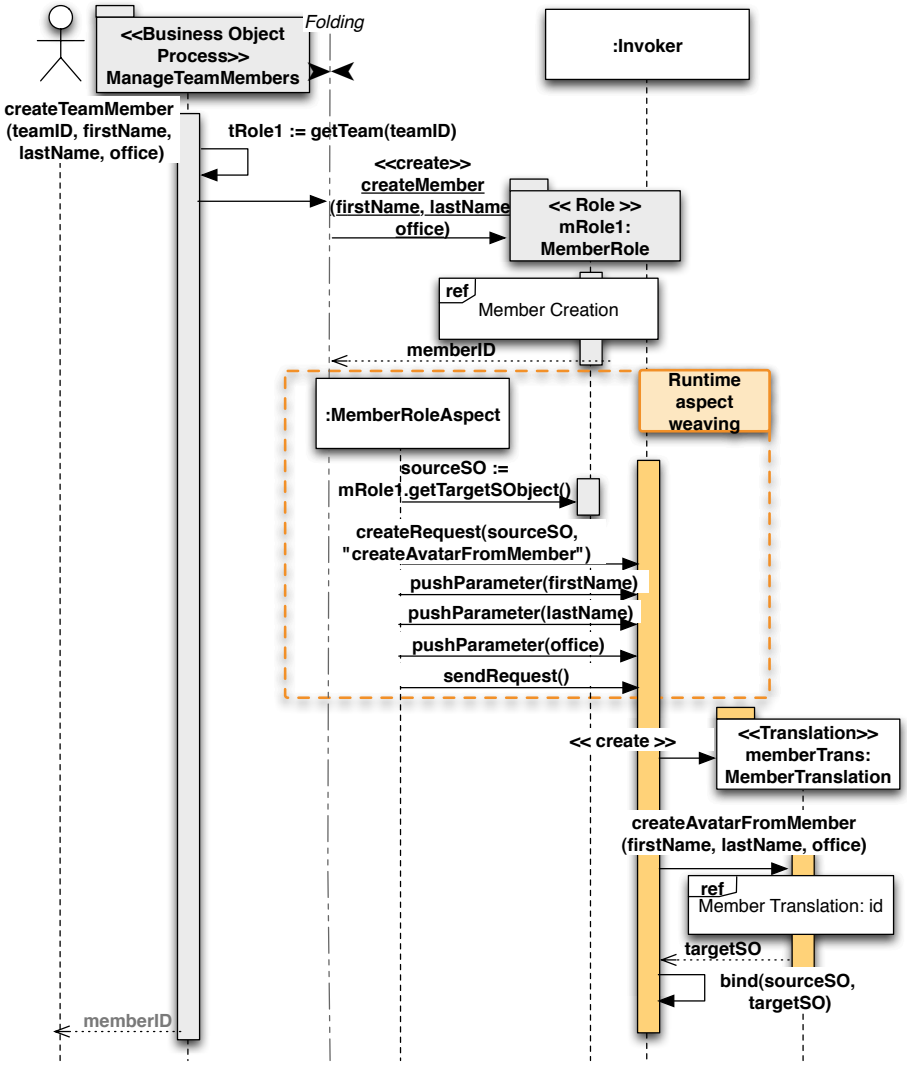


Figure 8: Runtime weaving realized by Sonata when the `createMemberRole` method is called

includes capitalizing on the models constructed by designers in order to automate model or code generation, and verify model (or, more generally, method product) traceability and consistency. In this respect, our approach follows the principles of Model Driven Engineering [24, 3].

From a technological point of view, we chose to base our tooling on the Eclise Modeling Framework[8] technology: besides being mature and largely adopted, its integration into the Eclipse IDE allows developers to benefit from modeling tools, from programming languages, such as Java, and from the cooperation of modeling tools and programming languages.

Section 4.5.1 provides some technological context surrounding Sonata, and presents the MDE tools used during the method's early steps (and outside of Sonata's scope), which enable developers to semi-automatically produce models such as shown in Fig. 3. Section 4.5.2 details the actual model-to-text (M2T) tool-chains provided by Sonata.

4.5.1. MDE Tools and Techniques for the Method's Early Steps

In the early steps of the Symphony methodology, the use of model generations, transformations and queries is encouraged. Indeed, one of the goals of this approach is to enable an end-to-end traceability of all development products [1], in order to facilitate the maintenance of these products, signal inter-model inconsistencies and provide means for developers to easily trace the impact of a change on the whole development flow. In this context, we used the following tools:

- The UML profiles that are mentioned in Section 3 were described using TOPCASED[9]. Being based on the EMF framework, we were able to use the resulting models in our model transformation flows.
- Model transformations (e.g., for generating the different stages of the Symphony Objects class diagrams) and consistency checks (e.g., for validating the consistency of each stage of the Symphony Objects diagram with the previous development steps) were implemented using the Atlas Transformation Language (ATL) [15].

The latter point deserves some details:

- Consistency checks were implemented using ATL queries, which execute the equivalent of an OCL check and return predefined strings used as booleans. Although technically akward, this approach enabled us to easily integrate the results of individual checks in a global diagnostic, as a first step towards maintaining inter-model traceability throughout the development process. A more mature approach would rather directly make use of Eclipse's OCL interpreter [10].

[8] `http://www.eclipse.org/modeling/emf/`
[9] `http://www.topcased.org/`
[10] `http://www.eclipse.org/projects/project.php?id=modeling.mdt.ocl`

- ATL model-to-model (M2M) transformations are used for transforming very early specifications models into analysis models stubs, which are then completed by designers to produce complete analysis models, such as the one presented in Fig. 3.

- At the implementation level, ATL queries were first used to generate Sonata XML configurations (see List. 1), from the analysis model (Fig. 3), as well as the corresponding aspect files (see List. 2). As demonstrated below, we adopted the Xtext 2[11] framework for later M2T transformations.

4.5.2. Generation of the Aspect and Translation code

Based on what we presented in the previous sections, one may notice that the implementation of the aspect code for enabling Symphony Connections (List. 2) and of the Translation classes (List. 3) is both systematic and tedious: it is essentially a matter of reproducing the same code patterns, with only a few variations. Concerning the aspect code, these variations can be summed up to the Symphony Advice (see Fig. 4).

Consequently, we chose to provide developers using Sonata with two development options:

- Developers using model-driven tools for the whole development process (i.e., including the specifications and analysis phases) may capitalize on ATL queries (as described earlier) for generating the aspect code from the analysis model.

- Developers that would rather start working with model-driven tools later in the development cycle (either because their organization does not support model-driven processes, or because their early development process is informal) may use a small DSL aimed at describing Symphony Connections in a preexisting Java project.

We elaborate on the latter option in the following paragraphs.

This concise textual language expresses the same concerns as the advice described in Fig. 4. We developed its grammar and its editing environment using Xtext 2, a powerful framework for designing languages/DSLs, integrated into the Eclipse IDE. Fig. 9 provides an example of this simple language. In this example, the `connect` statement declares a new Symphony Connection, between the `MemberRole` and `Avatar` classes. The `createMemberRole()` token corresponds to the signature of the method from the `MemberRole` class that triggers the Symphony Connection, while the `createAvatarFromMember` token corresponds to the name of the method from the Translation class that will get called once the Symphony Connection is triggered, as is described in Fig. 8.

The small DSL excerpt presented above is all that's required for generating all the aspect code presented in List. 2, as well as the skeleton (i.e, class, method signatures, and import statements) of the Translation class presented in List. 3.

[11] `http://www.eclipse.org/Xtext`

```
import business.entity.team.MemberRole
import interactional.entity.avatar.Avatar

connect MemberRole to Avatar
     after createMemberRole() -> createAvatarFromMember
end
```

Figure 9: Runtime weaving realized by Sonata when the `createMemberRole` method is called.

Xtext's advanced import mechanism allowed us to define the scope of method signatures that may be referenced by the user when describing the connections, and to integrate the corresponding content assist elements.

Additionally, the editing environment provided by Xtext reconstructs the EMF model that corresponds to the textual representation defined by the user (the Ecore metamodel is deduced by the Xtext framework, from the grammar initially defined by the user). This in-memory model may then be used as the input of Xtext's generation (M2T) tool: Xtend (formerly known as Xpand). The latter interpolates the model's attributes with metamodel-aware templates, to produce the final aspect code. The end result is identical to what is achieved with ATL queries, although the Xtext-based solution is obviously far more efficient.

Consequently, in the small example described above, we managed to reduce the number of manually written lines of source code describing the Symphony Connection from approximately 20 to approximately 10.

We describe in the following section the empirical validation process we adopted for Sonata and the symphony architecture.

5. Evaluation and Discussions

5.1. Early Results

We presented in Cret et al. [5] an evaluation of an early version of Sonata. Three versions of the same application were developed, two of which used a standard and an optimized implementation of MVC, while the third implementation used Sonata. We were then focusing on object coupling and object complexity, with the assumption that, given the same functional scope and Human-Computer Interface, variations of these properties would reflect the quality of the underlying implementation patterns or frameworks.

The results presented in the latter article showed that Sonata enabled applications to feature lower total, average and maximum coupling values than equivalent MVC implementations. Our implementations' cyclomatic complexities (i.e., McCabe complexity [19]) shared the same order of magnitude, between 1.46 and 1.79, which are considered as low (i.e., below a value of 4) complexity applications.

Since the publication of these results, we focused on further reducing the coupling and complexity induced by Sonata, as well as on reducing the amount of code necessary for setting up the framework.

5.2. Comparing MVC and Sonata

The prototypes' and Sonata's evolutions makes any comparison with previous MVC implementations inherently flawed. Instead, we chose to compare the amount of work necessary for setting up Sonata with an MVC implementation.

We compared the smallest functionality possible: the handling of a textual value. The latter value is defined by the user from an HCI, then transformed[12] before it can be handled by the business space. Obviously, any direct change on the business space requires updating the interaction space with the new, translated value. For the sake of simplicity, this functionality would not be supported by any technical (in particular, graphical) framework. Interaction space modifications would simply be emulated by writing to an appropriate method in the interaction space, which would then trigger the appropriate callbacks.

We used Nguyen et al.'s Unified CodeCount tool [21] for acquiring source code metrics. In the following paragraphs, we do not consider the source code related to the instantiation of MVC's facets or the instantiation of Sonata's Process Objects. Indeed, Sonata's initialization relies on reading configuration files, the evaluation of which would be irrelevant. Additionally, we do not consider the code related to the benchmarks' statistics.

From the MVC point of view, this functionality was implemented as a simple triad, with the business space being implemented as a Model, and the interaction space being implemented as a View. Also, we chose R. Eckstein's recommended Java implementation [8]. While we did not try to evaluate this implementation's qualities, we consider that this should be the one considered as the *de facto* standard by the Java community.

Our implementation of the MVC triad is 49 logical lines of code (lLOC) long, out of which 10 lLOCs consist in communication-specific code distributed between the three facets (Controller registration of the Model and View, notification calls to the Controller, following the Observer pattern). Additionally, because the MVC pattern provides no prescription as to how the translations between the Model and the View should be handled, the corresponding source code was implemented in the Controller. The latter thus assumes several responsibilities, which are prone to be interlaced with one another at the source code level.

The granularity of the implementation pattern supported by Sonata is larger than MVC: our Sonata implementation of the same functionality features a Business Object Entity, a Business Object Process, an Interactional Object Entity and an Interactional Object Process. The connection between the two conceptual spaces is handled by two Translation classes: the first handling methods triggered by the Business Object Entity, the second methods triggered by the Interactional Object Entity. The aspect code for binding the two Entity Objects was generated automatically, using the small DSL described in Section 4.5.2. Based on the 8 lines that were required for describing the bidirectional con-

[12] Actually, the string is simply reversed, for the purpose of illustrating the semantic gap between the business and interaction spaces.

nection, 40 lines of AspectJ code and 25 lines of Translation class code were generated.

With its 77 lLOCs, the remaining manual implementation is about 1.5 times the size of the previous one. However, no communication code was implemented by hand. Moreover, while the MVC implementation defined three software constructs in isolation from any operational process, the Sonata implementation encourages envisaging and defining business and interaction processes that integrate all classes into a consistent whole.

5.3. Sonata Scalability

We implemented several applications using Sonata, which enabled us to examine whether the framework allowed applications to scale.

Apart from the TeamAvatar example, two applications were developed using Sonata: an Augmented Inventory of Fixtures (AIoF), whose development process was extensively described in other articles [10, 7, 11], and an application for rendering airport security test campaigns, called EDEMOI.

AIoF was developped as a small (about 2,000 lLOCs) prototype for validating whether Sonata could scale with a multithreaded, input-intensive application. AIoF enables a user's movements to be translated into high-frequency accelerometer data, which is input into AIoF for identifying the user's position in an OpenGL-rendered virtual model. The user can also execute vocal commands on AIoF for creating and locking virtual markers that identify housing damages. Although this application's business process is very simple (i.e., adding damages to the housing's rooms), Sonata enabled us to implement complex technical layers (where sensory inputs are handled) for the Interactional Object Entities, while designing simple translations with the much simpler business space.

EDEMOI is a larger application (about 5,000 lLOCs), with equally complex business and interaction spaces (3 Business Object Entities and 5 Interactional Object Entities). The application's technical layer (XML parsing and SVG graphics handling) counts about 1,200 lLOCs and 3,800 lLOCs for the functional layer. 4 Translation classes were implemented (3 of which were a interaction to business space translation). We used our small connection DSL for generating the aspect code, as well as translation class stubs. 4 advice files were defined, amounting to 33 lLOCs. From these descriptions, our generator automatically produced 135 lLOCs of aspect code (which did not require any posterior modifcation) and 61 lLOCs of translation code. Once manually completed, the translation classes amounted to 258 lLOCs (about 5% of the total application size). This would tend to show that:

- While a basic Sonata implementation starts with what may seem a large number of lLOCs, this number grows very slowly with the application's size.
- Sonata's MDE tool-chain is an efficient asset for the development of applications built for running with the Sonata runtime library.

Our methodology, as well as Sonata, enabled us to design and implement the business and interactional processes as almost independent systems (in fact, parts of both processes were implemented at different periods by different teams). We were also able to build and test the Translations independently from the rest of the system.

Altough no formal conclusion may be deduced from the latter considerations, we may nevertheless note that Sonata could be considered as an asset with application scaling issues.

6. Conclusion and future works

We have described in this paper a software architecture for efficiently building modular business and interactional objects. These practices rely on patterns for ensuring the reusability of objects, called Symphony Architecture, and on a software framework called Sonata.

Sonata abstracts developers from common tasks related to the instantiation and management of Symphony Objects and technical objects, and proposes a powerful mechanism for transparently connecting the business and interaction spaces, as well as synchronizing and translating events occurring in either spaces.

Even though its adoption presents several advantages, it is neither the only way to construct applications following the Symphony method, nor is Java the only language that can be adopted with the methodology. Reversely, the framework does not substitute itself to the development process: although the interconnection of Business and Interactional Objects is facilitated, it is advised to define and describe these constructs following the principles proposed in Symphony.

Implementing parts or variations of MVC using aspects is not a new idea [23]. However, our contribution includes both the creation of the link as well as its realization using a message broker. We consider that this is a powerful advantage for separating business and interaction concerns, and for connecting rich user interfaces with information systems. Indeed, Sonata allows distinguishing the business processes from the interactional processes, while maintaining an efficient communication strategy between the two sets of constructs. We consider that this advantage facilitates the construction of innovative user interfaces, connected to entreprise information systems.

Furthermore, the organization of the application's objects narrows volatile code, such as the applicative rules and the code for connecting the interactional and business spaces, into well-defined constructs: the Process Objects and the Translation classes. Consequently, Sonata enables developers to build software objects with very low coupling, thus encouraging their reuse, maintainability and legibility.

We have presented some elements of past evaluations realized on previous versions of Sonata, that already showed that the framework provides much lower coupling values than comparable approaches, such as MVC. Additionally, we presented several considerations regarding the coding overhead required for

implementing a Sonata-driven application, compared to MVC, as well as the framework's scalability.

As the Sonata framework gains in maturity, future work should focus on two interdependent aspects: increasing the tooling support of the framework and (thus) encouraging its use by developers. The Java libraries for setting up Sonata applications may be found online at `https://github.com/ggodet-bar/Sonata`. Additionally, this tooling support should also include the model transformations and queries used in the early steps of the method.

Concerning Sonata's instrumentation, we have presented a small DSL aimed at automating the generation of aspect code and of elements of the Translation classes, using model-driven engineering (MDE) technologies. Our next effort in this direction will be the integration of this contribution, and of existing profiles and model transformations, into a larger scale environment for handling models, and to a larger extents all products, produced during the development of applications following the Symphony methodology.

References

[1] Asuncion, H.U., François, F., Taylor, R.N., 2007. An end-to-end industrial software traceability tool, in: ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM, New York, NY, USA. pp. 115–124.

[2] Bass, L., Pellegrino, R., Reed, S., Seacord, R., Sheppard, R., Szezur, M.R., 1991. The Arch model: Seeheim revisited, in: Proceedings of the User Interface Developper's workshop.

[3] Bézivin, J., 2005. On the unification power of models. Software and Systems Modeling 4, 171–188. 10.1007/s10270-005-0079-0.

[4] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. Pattern-Oriented Software Architecture: A System Of Patterns. John Wiley & Sons Ltd.

[5] Céret, E., Dupuy-Chessa, S., Godet-Bar, G., 2010. Using software metrics in the evaluation of a conceptual component, in: 4th Int. Conf. On research Challenge in Information Science RCIS'2010.

[6] Coutaz, J., 1987. PAC, an object oriented model for dialog design, in: Buillinger, H.J., Shackel, B. (Eds.), Proceedings of INTERACT'87.

[7] Dupuy-Chessa, S., Godet-Bar, G., Pérez-Medina, J.L., Rieu, D., 2009. A software engineering method for the design of mixed reality systems. Engineering of Mixed Reality .

[8] Eckstein, R., 2007. Java SE Application Design With MVC. Sun Microsystems Inc.

[9] Gamma, E., Helm, R., Johnson, R.E., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

[10] Godet-Bar, G., Dupuy-Chessa, S., Rieu, D., 2008. Advanced Information Systems Engineering. Springer-Verlag Berlin / Heidelberg. volume 5074/2008 of *Lecture Notes in Computer Science*. chapter When Interaction Choices Trigger Business Evolution. pp. 144–147.

[11] Godet-Bar, G., Rieu, D., Dupuy-Chessa, S., 2010. HCI and business practices in a collaborative method for augmented reality systems. Information and Software Technology, Elsevier 52, 492–505.

[12] Godet-Bar, G., Rieu, D., Dupuy-Chessa, S., Juras, D., 2007. Interactional Objects: HCI concerns in the analysis phase of the Symphony method, in: 9th International Conference on Enterprise Information Systems ICEIS'2007, Funchal, Madeira. pp. 37–44.

[13] Green, M., 1985. Report on dialogue specification tools, in: Pfaff, G. (Ed.), User Interface Management Systems, Springer-Verlag. pp. 9–20.

[14] Hassine, I., Rieu, D., Bounaas, F., Seghrouchni, O., 2002. Symphony: a conceptual model based on business components, in: SMC'02, IEEE International Conference on Systems, Man, and Cybernetics.

[15] Jouault, F., Kurtev, I., 2006. On the architectural alignment of ATL and QVT, in: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, ACM, New York, NY, USA. pp. 1188–1195.

[16] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., 1997. Aspect-oriented programming, in: European Conference on Object-Oriented Programming (ECOOP), pp. 220–242.

[17] Krasner, G., Pope, S., 1988. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. Journal of Object Oriented Programming 1, 26–49.

[18] Martin, R.C., 1995. Object Oriented Design Quality Metrics: An Analysis of Dependencies. volume 2. ROAD.

[19] McCabe, T., 1976. A complexity measure. IEEE Transactions on Software Engineering SE-2, 308–320.

[20] Moon, D.A., 1986. Object-oriented programming with flavors, in: OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications, ACM, New York, NY, USA. pp. 1–8.

[21] Nguyen, V., Deeds-Rubin, S., Tan, T., Boehm, B.W., 2007. A SLOC Counting Standard. Technical Report. COCOMO Forum.

[22] Nigay, L., Coutaz, J., 1995. A generic platform for addressing the multimodal challenge, in: CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. pp. 98–105.

[23] Piveta, E.K., Zancanella, L.C., 2003. Observer pattern using aspect-oriented programming, in: Hanmer, R., Andrade, R.M.C. (Eds.), Proceedings of 3rd Latin American Conference on Pattern Languages of Programming SugarloafPLoP'2003, pp. 313–324.

[24] Stahl, T., Völter, M., Bettin, J., von Stockfleth, B., 2006. Model-Driven Software Development. John Wiley.

[25] Tarpin-Bernard, F., David, B.T., 1997. AMF: A new design pattern for complex interactive software?, in: HCI (2), pp. 351–354.