Shared repository pattern

Philippe Lalanda
Thomson-CSF Corporate Research Laboratory
Phone: 33 1 69 33 92 90
Email: lalanda@thomson-lcr.fr
Domaine de Corbeville
91404 Orsay, France

Abstract: The purpose of this paper is to present an architectural pattern named shared repository pattern. This pattern defines a model of communication for software components based on the use of a shared repository. It is a very popular pattern in industrial settings that has been used in numerous and various domains.

The work presented in this paper is part of a project which purpose is to elaborate a system of architectural patterns in order to guide the design of software systems. This project is conducted through the analysis of existing systems in the Thomson-CSF business units.

Keywords: architectural pattern, communication via a shared repository

1 Introduction

Software architecture is defined by a set of components and their relationships, that is the way they communicate to meet a set a requirements, and principles and guidelines governing their design and evolution over time [Perry and Garlan, 1995]. A software component can be defined as a coherent, well-defined structure that encapsulates a given functionality. Granularity is not discriminative, but components are generally bigger than simple objects or functions. It is today recognized that designing a software system starts at the architectural level. The purpose at this stage is to define the components of the system, the way they communicate and collaborate, and the underlying execution model. Decisions taken at the architectural level, like the distribution of responsibilities among components for example, have an important and long lasting impact on the system design and development.

Although the importance of architecture is now recognized, designing software architectures remains a complex, poorly guided activity. We believe that architectural patterns, as introduced in [Buschmann *et al.*, 1996], can provide a major input to the work of software architects. Architectural patterns record validated ways to structure a software system into collaborating components. By definition, patterns have been used in numerous domains and can therefore be trusted and reused. Patterns also come with a set of properties that can drive the design activity, the purpose of which is ultimately to meet a set of pre-defined requirements.

In this paper, we focus on a the definition of communication model for components. This is a major design activity that influences the structures of both the components and the global system, and delivers, or prevents delivery of some non-functional properties at the system level (such as efficiency, adaptability or portability). The purpose of this paper is more precisely to present an architectural pattern defining a model of communication for software components based on the use of a shared repository, and to show how this pattern relates to other architectural patterns.

2 Communication in software architectures

In software architectures, components can communicate directly or indirectly. Direct communication means that components know each other (identification, location) and exchange data by opening point to point communication channels. Different architectural patterns are based on such communication strategies, including:

- The **pipe and filter pattern** [Buschmann *et al.*, 1996] where components send streams of data to each other through pipes,

- The **object style** [Garlan and Shaw, 1993] where components (objects) communicate through method calls.

Indirect communication is more complex. It means that components, in order to communicate, make use of a third party or of a shared memory. Using a third party can take different forms. Some of them have been described as patterns:

- The **mediator pattern** [Gamma *et al*, 1995] which defines a mediator object to control and coordinate the interaction of a group of objects or software components,

- The **Publish/Subscribe pattern** [Dumond and Lalanda, 1998] as implemented in HLA (High Level Architecture) in the simulation domain where a broker receives at run time data produced (published) by software components and sends them to software components that previously expressed their interest (subscription) in these types of data.

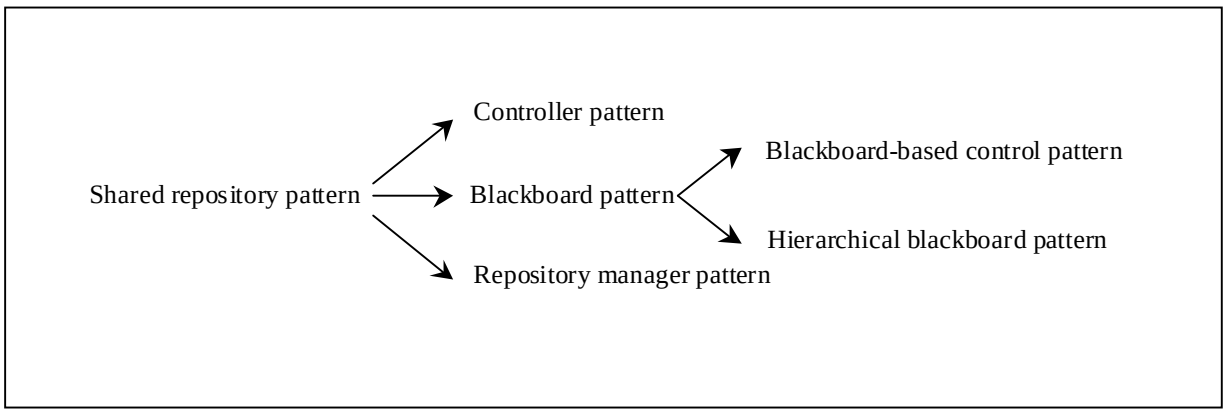

Figure 1: Patterns based on the shared memory pattern.

Using a global shared memory is another way to realize indirect communication. The principle is that components put all of their outputs in a memory accessible by the other components and retrieve their inputs from this shared memory (also called a repository). This approach obviously raises a control issue, that is how and when components are activated in order to read and write in the shared memory. Various control strategies, with specific advantages and liabilities, have been proposed and could be described as patterns (see figure 1) :

- The **controller pattern** introduces a control software component in the system. This control component rules the system and schedules the activation of the other components. This pattern is applicable to deterministic problems where sequences of components activation can be determined off line and coded in the controller (using diverse techniques).

- The **repository manager pattern** is applicable in a distributed environment. It introduces a repository manager which sends notification of data creation or modification to the software components.

- The **blackboard pattern** refines the controller pattern to deal with non-deterministic problems [Buschmann *et al.*, 1996]. It also presents several variants [Lalanda, 1997].

The Controller and Repository Manager patterns are, at this time of writing, only candidate patterns which have yet to be fully drafted and workshopped.

It appears that the use of a shared memory is the basis of these different patterns. We actually believe that using a shared memory to communicate is very important in itself, apart from control considerations. The purpose of this paper is then to define a shared memory pattern and to show that, depending on the control strategies and the execution platforms, complementary patterns have to be defined.

Shared repository pattern

Name

Shared repository pattern.

Example

Consider a software system for mission planning in the avionics domain. One of the main requirements of such system is to compute a detailed trajectory that is possibly transmitted to a plane automatic pilot. A trajectory consists of thousands of multi-dimension points, that is points with several application-specific attributes including longitude, latitude, altitude, speed, time, cape, etc. An illustration is provided by figure 2.

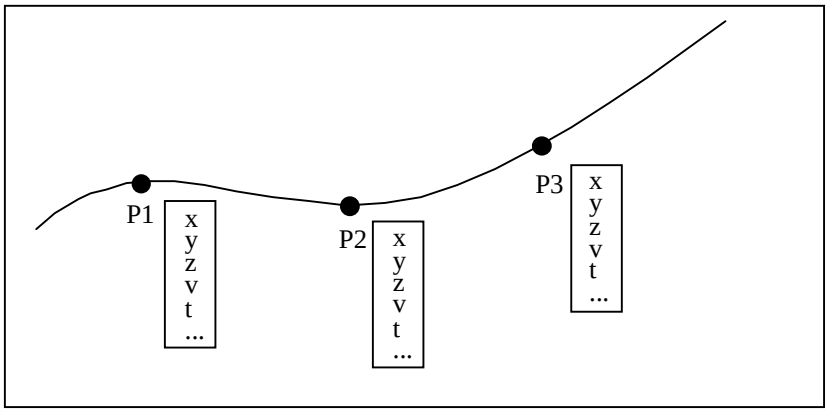

Figure 2: Multi-dimension trajectory.

Computing a trajectory is a complex task that involves various techniques and resources (maps, databases, …). A software system for mission planning is thus distributed into several software components specialized in the computation of different dimensions of a trajectory. Generally, a first component provides a simple, two-dimension trajectory (sequence of points with longitude and latitude). Then, depending on specific requirements, other components successively provide additional attributes like altitude, speed, or cape. Companies specialized in mission planning systems own many such components and reuse them in different applications depending on specific requirements.

This solving process involves heavy and frequent communication. The trajectory, which represents a large, structured data, is exchanged between several components that successively refine it. Designing and implementing adequate communication mechanisms become a central issue in such systems upon which performance depends.

Context

A software system made of several software components that need to communicate, that is exchange potentially large and evolving data, in order to meet the system requirements.

Problem

Establishing a communication model for the software components is a major design activity that leads to, or prevents, some non-functional properties at the system level. In the mission planning system example, this activity corresponds to the issue of defining how the software components computing the various parts of an architecture collaborate, and how the trajectory is actually exchanged between them.

When designing such mechanisms, the following, sometimes contradictory, forces have therefore to be balanced:

- Reliability and robustness. Data have to be completely and correctly transmitted to the targeted components. In many systems, fault tolerance mechanisms allowing error recovery during transmission are necessary.

- Performance. A given level of performance, in term of transmission time for example, is often part of the basic requirements of a system.

- Understandability. Relationships between components and the way they are implemented represent major design decisions, and an important communication vehicle for the various stakeholders of the system. Understanding the communication model is also essential during the maintenance phase where new engineers have to learn the system in order to debug it or to meet new requirements.

- Flexibility. Relationships between components can evolve, statically and dynamically. The communication mechanisms must therefore be flexible enough to meet changing requirements.

- Maintainability. Maintenance is an important and often costly phase in a system life cycle. Among other things, it is today essential, for both economical and technical reasons to be able to swap in new components in order to infuse new technology or simply to replace obsolete or failing components. The communication model influences the components structure, and has to be such that it allows easy replacement of components.

Various mechanisms, balancing in different manners the previous forces, have been proposed to make components communicate in a system. The pattern presented in this paper records a model of communication for software components based on the use of a shared repository.

Solution

The solution described by this pattern consists in using a data repository for communication. The repository is known and accessible by all the software components of the system and represents the only means of communication for components. When a component produces some information that is of interest for other components, it stores it in the shared repository. The other components will retrieve it there if need be.

This pattern presents an indirect form of communication where components do not know each other. A component has no knowledge of which components have produced the data it uses, and which components will use its outputs.

In our avionics example, instances of software components are the components capable of computing the various dimensions of a trajectory. The shared repository is a global structure that is able to store, at least, the trajectory under construction.

This pattern is summarized in figure 3.

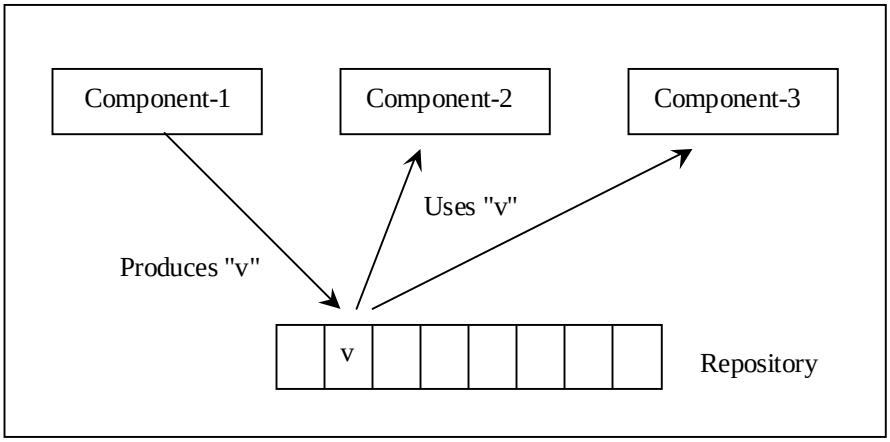

Figure 3: Communication through a shared repository.

This pattern is only about communication. It does not tackle control issues that can be very complex and which gave rise to various solutions with associated advantages and liabilities. Such solutions, because of their complexity and impact on the overall system properties, are patterns in themselves. References to these additional patterns are given in the “see also” section.

Structure

This pattern structures a software system into:

- A set of software components. These are computing modules containing the knowledge of the domain. They have to communicate with each other in order to meet the system requirements. These components do not know each other: they are only defined by their needs to perform the computation (their inputs) and the results they can provide (their outputs).

- A structured repository accessible by every component (read and write accesses). This repository can store all the data that need to be exchanged by components during system execution. It represents the only vector of communication for the system components.

Structural relationships between these components are summarized in UML notations in Figure 4.

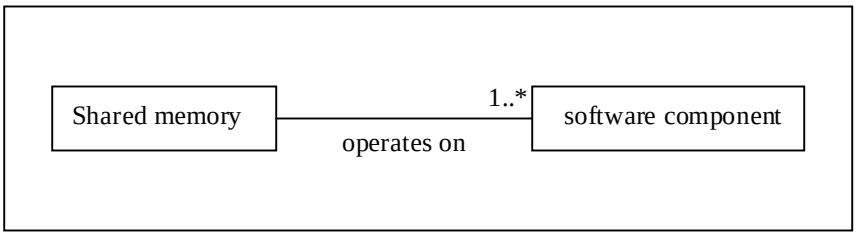

Figure 4: Structural relationships.

Additional structures or mechanisms need to be provided to implement the control part of a system. They depend both on the application characteristics (expected performance, determinism, …) and the execution platform. In a single processor environment, a control component is generally added to schedule the activation of the software components. On the other hand, in a distributed environment where software components are executed on different processors, notification mechanisms are generally implemented to notify the concurrent components of any change in the shared repository.

Dynamics

The following scenario illustrates the behavior of a system where communication is made through a shared repository. It is based on a simplified mission planning system, as introduced in the example section. We consider the following software components: *compute_xy*, *compute_z*, *compute_v*, and *compute_t* which respectively compute a set of two-dimension points (longitude and latitude), the altitude of each point, the expected speed at each point, and the expected time at each point. We also consider a *display* component, the purpose of which is to present the complete trajectory on various devices. The shared repository is accessible by the five components previously defined and is structured so as to store the trajectory under construction.

We present hereafter an execution scenario (in this scenario we do not show how components are activated):

- Compute_xy is activated. It calculates a two-dimension trajectory (given a two-dimension map and the plane characteristics) and stores it in the shared repository.
- Compute_z is activated. It retrieves the two-dimension trajectory in the shared repository and computes the altitude of each point (given a three-dimension map and the plane characteristics). It stores the result, that is a three-dimension trajectory, in the shared repository.
- Compute_v is activated. It retrieves the trajectory in the shared repository and adds speed information to each point. It stores the result in the shared repository.
- Compute_t is activated. It retrieves the trajectory in the shared repository and brings timing information to each point. It stores the result in the shared repository.
- Display is activated. It retrieves the complete trajectory and takes care of its presentation.

This scenario is illustrated by figure 5 with a UML-like sequence diagram. It is clear on this figure that there is no direct communication between components. It is also clear that the issue of activating software components at the right time and in the right order (that is the global control of the system) is central.

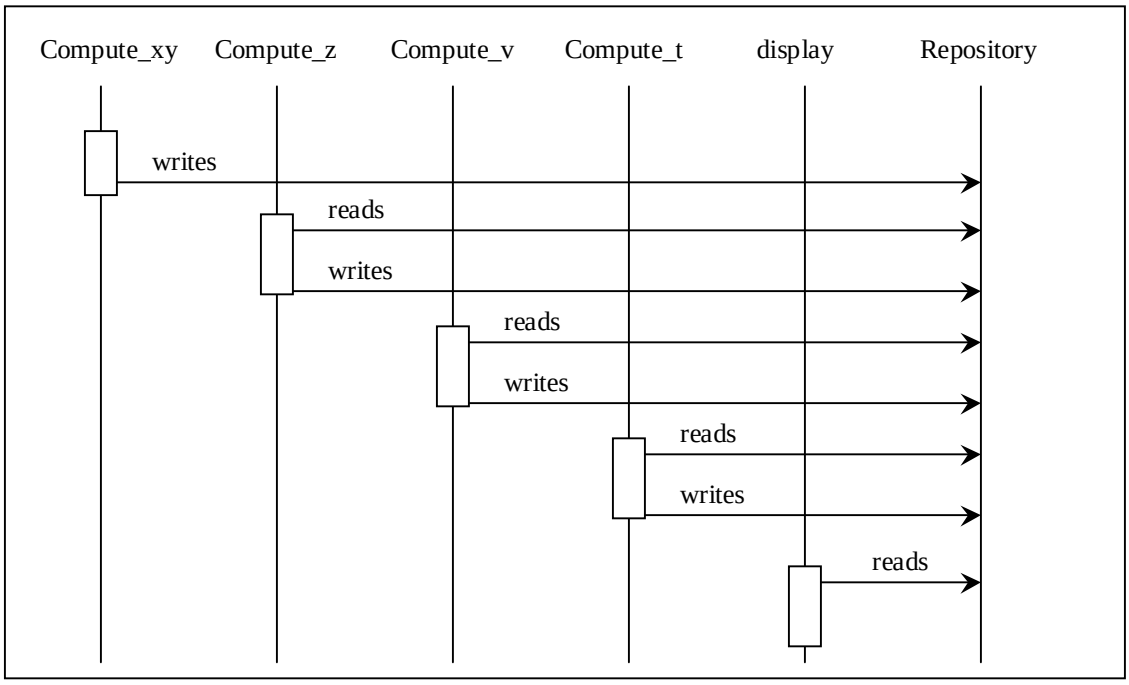

Figure 5: UML sequence diagram.

Implementation

In order to build a software system using the repository pattern, a designer has to perform the following, possibly interleaved, activities:

- Separate the system into software components. Both existing legacy code that has to be reused and the need to rationally structure the system into coherent, well-defined modules generally influences this activity. The set of components has to be complete, in the sense that it includes all the functions necessary to meet the system requirements, and complementary, in the sense that each component can be provided with the necessary inputs.

- Define the data that are exchanged by the software components during the solving process. For each component, required and provided data are specified.

- Define the computing environment (operating system, middleware, hardware, …).

- Define how the components are executed in the computing environment and define the system control. Control mechanisms take into account the system requirements and characteristics, and the computing environment.

- Define and structure the shared repository. Depending on the computing environment, a repository can be implemented as a simple global data, a shared memory accessible by several tasks in a real-time environment, or a component with services. Services may go from simple access functions with possible concurrent access management, to more complex services like persistence management for example (that may involve the coupling with a DBMS).

Known Uses

The repository pattern is a very popular pattern crossing domain boundaries. An informal study in the Thomson-CSF group has shown that this pattern, or variants of it, is currently used in numerous and various domains including:

- Avionics systems like the mission planning system previously introduced,

- Simulation systems,

- Communication and control systems where data are gathered by various sources and processed by several components.

Consequences

In this section, we show how the forces presented in the “problem” section are resolved by this pattern.

Regarding **performance**, this pattern has the following properties:

- Reduction of communication flows. Data modification generates communication with the repository and then, only on request, with some components. This entails a gain in performance, especially in systems with frequently evolving data that are needed only occasionally by some components.

- In a distributed environment, the shared repository appears as a bottleneck. Performance of the system can be seriously affected if the addition of specific mechanisms for concurrent access to the shared repository or consistency maintenance is necessary.

- Control indirection has a slight cost on the system efficiency.

Regarding **robustness**, this pattern has the following properties:

- Data shared by components are kept in the repository. If a transmission fails, it is simple to make a new access to the repository to get the expected data.

- Data kept in the repository represents the system current state. It can be used during a fault recovery transaction to put the system back into a coherent state.

- In a distributed environment, the node containing the repository represents a very fragile location. If this node goes down, the whole system is immediately affected.

Regarding **understandability**, this pattern has the following properties:

- At execution time, the repository represents the system current state at a given level of abstraction. Accessing the repository can therefore allow to check the system evolution and coherence at run time. Such a feature is particularly interesting for debugging and measuring the system performances.

- Because of the indirection, communication between two components does not appear clearly. This requires extra documentation to make clear the relationships between components.

Regarding **maintainability**, this pattern has the following properties:

- Evolution/replacement of software components is relatively easy. A component is only concerned with its expertise (or domain knowledge) and the input/output with the shared repository. It does not implement any control structure related to a specific application.

- Changing the structure of the shared repository has a strong impact on a system: it implies changing the interfaces of some components.

Regarding **flexibility**, this pattern has the following properties:

- Since there is no direct communication between components, the architecture topology is not fixed in the components code and is most of the time easy to change (though it is related to the system control and its implementation).

See Also

The **blackboard pattern**, as defined in [Buschmann *et al.*, 1996], is based on a shared memory and can be seen as an extension of this pattern. The blackboard pattern is characterized by a very sophisticated controller which schedules in an opportunistic fashion the activation of the software components. The description of software components is extended in order to provide appropriate information to the controller. It is to be noted that various control strategies have been proposed in blackboard systems [Lalanda, 1997].

Mediator [Gamma *et al*, 1995] differs in the shared memory pattern in that it takes the responsibility of controlling and coordinating the interaction of a group of objects or software components. The shared memory pattern does not include control strategies.

Acknowledgements

I would like to thank my colleagues at the Thomson-CSF research center for their contribution to the writing of this pattern. I am also grateful to Hans Rohnert from Siemens and Sazi Temel from BEA Systems for their comments on this paper.

I also wish to thank my PloP'98 shepherd, Alan O'Callaghan, for his valuable comments on the first drafts of this paper.

3 References

[Buschmann *et al.*, 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-oriented software architecture – A system of patterns. Wiley, 1996.

[Dumond and Lalanda, 1998] Régis Dumond and Philippe Lalanda. The publish/subscribe pattern. SISO'98, Orlando, Florida, 1998.

[Garlan and Shaw, 1993] David Garlan and Mary Shaw. An introduction to Software Architecture. Advances in Software Engineering and Knowledge Engineering, vol. 1, World Scientific Publishing Company, 1993.

[Gamma *et al*, 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[Lalanda, 1997] Philippe Lalanda. Two complementary patterns to build multi-expert systems. PloP'97, Monticello, Illinois, 1997.