Shared Repository Pattern

Shared Repository Introduction

- Some applications are inherently data-driven.
  - Interactions between components do not follow a specific business process, but depend on the data on which they operate.
  - However, despite the lack of a functional means to connect the components of such applications, they must still interact in a controlled manner.
- In the shared repository pattern components put all of their outputs in memory accessible by the other components (the *repository*) and retrieve their inputs from this shared memory.
- This approach raises a control issue, that is how and when components are activated in order to read and write the shared memory.
  - Various control strategies, with specific benefits and liabilities, have been described in additional patterns.

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's 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, etc.
- 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.

Example (2)

- 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 time.
- This solving process involves heavy and frequent communication. The trajectory, which represents a large, structured data set, 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.

Example (3)

- To compute and display a trajectory, several components are activated in sequence:
  1. *Compute_xy* component calculates a two-dimension trajectory (given a two-dimension map and the plane characteristics) and stores it in the shared repository.
  2. *Compute_z* component retrieves the two-dimension trajectory from 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.
  3. *Compute_v* component retrieves the trajectory from the shared repository and adds speed information to each point. It stores the result in the shared repository.
  4. *Compute_t* component retrieves the trajectory from the shared repository and brings timing information to each point. It stores the result in the shared repository.
  5. *Display* component retrieves the complete trajectory and takes care of its presentation.

Problem

- How to design an application whose parts operate on, and coordinate their cooperation via, a large set of shared data.

Context

- Many data-driven systems operate on massive amounts of data provided by various external sources.
- Typically, core responsibilities (e.g. monitoring and control, alarming, reporting, etc.) are largely independent of one another.
- It is the state of the data that determines the control flow and collaboration of these tasks.
- Connecting the tasks directly would hard-code a specific business process into the application, which may be inappropriate if specific data is unavailable, not of the required quality, or in a specific state.
- However, we need a coherent computational state across the entire application.

Forces

- Reliability and robustness.
  - Data have to be completely and correctly transmitted to the targeted components.
- Performance.
  - A given level of performance, in term of transmission time for example, is often part of the basic requirements of a system.
- Understandability.
  - Understanding the relationships between components and the communication model is essential during the maintenance phase.
- Flexibility.
  - Relationships between components can evolve. The communication mechanisms must therefore be flexible enough to meet changing requirements.
  - It should be possible to swap in new components in order to infuse new technology or simply to replace obsolete or failing components.

Solution

- Maintain all data in a central repository shared by all functional components of the data-driven application and let the availability, quality, and state of that data trigger and coordinate the control flow of the application logic.
- The repository is known and accessible by all the software components of the system and represents the only means of communication between components. When a component produces some information that is of interest for other components, it stores it in the shared repository.
- A component has no knowledge of which components have produced the data it uses, and which components will use its outputs.

Shared Repository Benefits

- 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.
- 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.
- Adding, modifying, and replacing components is relatively easy since each component only interacts with the shared repository.

Shared Repository Liabilities

- Coordinating components via the state of shared data can introduce performance and scalability bottlenecks if many concurrent components need access to the same data exclusively and are thus serialized.
- 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.
- Because of the indirection, communication between two components does not appear clearly. This requires extra documentation to make clear the relationships between components.
- Changing the structure of the shared repository has a strong impact on a system: it implies changing the interfaces of some components.

Variation: Blackboard Pattern

- The Blackboard pattern tackles problems that do not have a feasible deterministic solution for the transformation of raw data into high-level data structures, such as diagrams, tables or English phrases.
- Examples include speech recognition, submarine detection based on sonar signals, and the inference of protein molecule structures from X-ray data.
- Such tasks must deal with several hard challenges: input data is often fuzzy or inaccurate, the path towards a solution must be explored, every processing step can generate alternative results, and often no optimal solution is known. Nevertheless, it is important to compute valuable solutions in a reasonable amount of time.

Blackboard Pattern (2)

- The *blackboard* is a repository that stores the current state of the solution.
- The components function *opportunistically*:
  - There is no predetermined order in which the components are activated.
  - A component's condition part examines whether the component can make a contribution to the computation's progress by inspecting the data written on the blackboard.
  - An action part reads one or more inputs from the blackboard, processes it, and writes one or more outputs back to the blackboard. Alternatively, the action part could erase data from the blackboard because it identifies the data as not contributing to the overall task's solution.