The Sponsor-Selector Pattern

Eugene Wallingford
University of Northern Iowa
wallingf@cs.uni.edu

Abstract. This pattern is intended as the first pattern in a pattern language for designing knowledge-based systems. It addresses the issue of dynamic control of problem solving.

SPONSOR-SELECTOR — Object Behavioral

Context

You are developing a knowledge-based system. You are finding that each problem-solving method is embedded throughout the system and that adding new methods requires extensive modification. Even after a new method has been added, the system is unable to select it dynamically at run-time due to hard-coded control statements.

Intent

Provide a mechanism for resource selection that enables an object to dynamically select from existing resources, and to fully integrate new resources, at run-time.

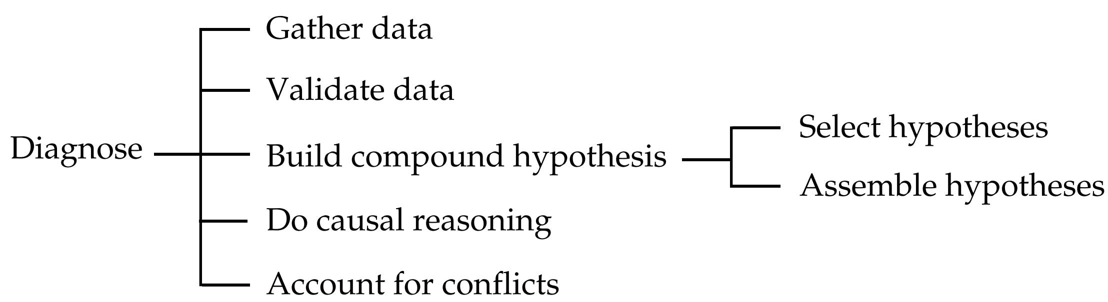
Also Known As

Proxy + Strategy

Motivation

Sometimes a class requires a collaborator but cannot know until run-time which resource it will need. Furthermore, the set of potential collaborators may change, either dynamically during system execution or statically through programmer modification. At run-time, the class would like to select the most appropriate collaborator based on the context information available at that point.

Consider, for example, the case of a system engaged in the task of medical diagnosis. The system has at its disposal a set of objects capable of performing particular diagnostic sub-tasks. The task/sub-task structure may look something like this:



At any point during a diagnostic session, which sub-task should be performed next? This is the question of problem-solving control.

The system has an object responsible for performing each task in this hierarchy. At run-time, the diagnostician object would like to select the most appropriate sub-task object to invoke, based on the context information available at that point. This involves determining which of the objects is likely to make the most progress toward the goal of arriving at a useful diagnosis.

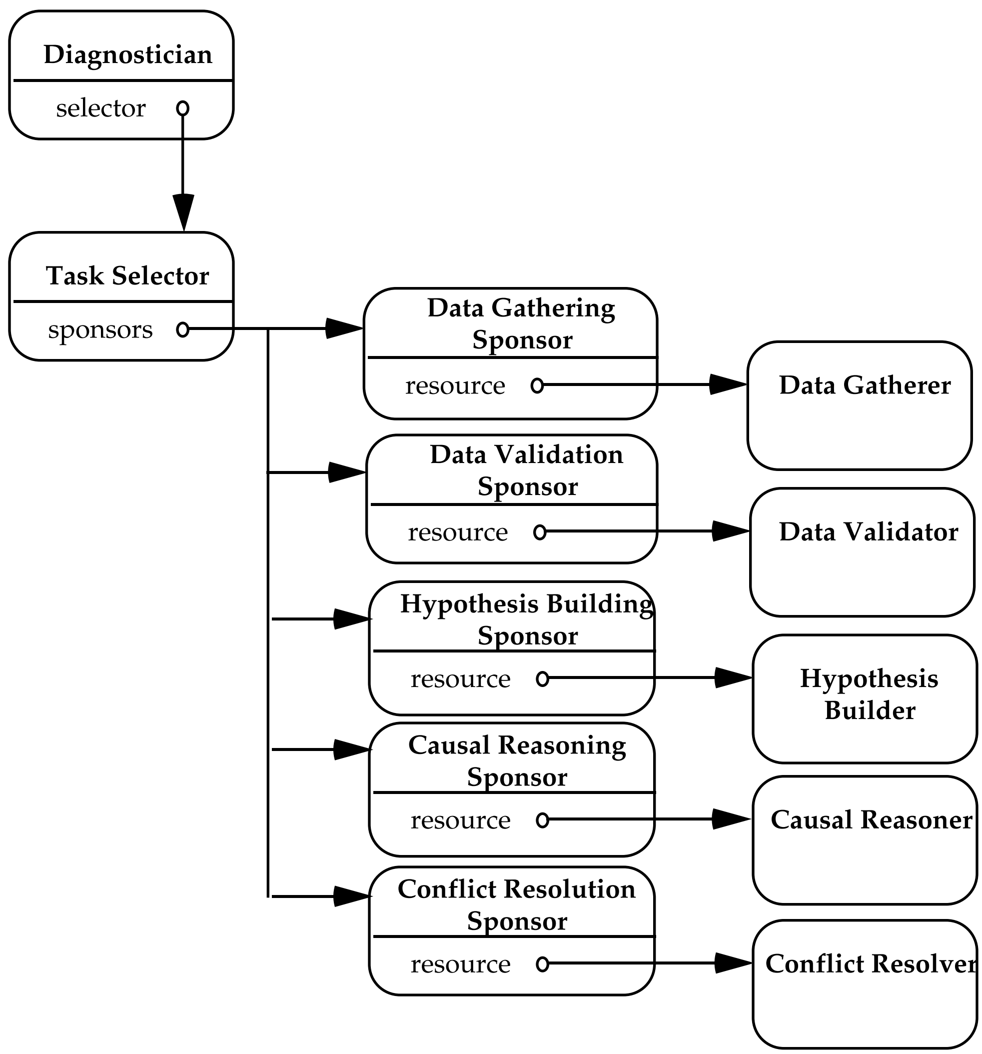
One way to solve this problem is to encode the selection mechanism directly into the diagnostician. This solution is much less than ideal, since it clutters the client code of the diagnostician with three undesirable features:

- knowledge of when a particular resource is applicable
- knowledge of which resources to prefer and in what contexts
- large conditional statements that will become even more complex as new resources are added to the system

This sort of hard-coded control structure leads to two kinds of difficulties. If a new resource is added to the system at run-time — say, through some form of machine learning — the system is unable to use the resource, because the system's control knowledge makes no reference to the new object. If the programmer adds a new resource to the system, the system's control knowledge must be modified to make it accessible.

A second way to solve this problem is to separate the selection code into a separate class that specializes in the selection of a particular resource, in this case, the selection of the next problem solver to invoke. This solution frees the client code of irrelevant knowledge and of modification in the face of new resources. But it still confounds resource applicability knowledge with resource preference knowledge, making the new "selection" object as prone to the same difficulties when new resources are added to the system.

A third way to solve this problem involves going one step further and separating these two responsibilities into different classes:



Each sub-task problem solver has a sponsor object whose responsibility is to recommend when it can be used. The selector object takes these recommendations as input and decides which problem solver should be invoked next. In the course of a diagnostic session, whenever the diagnostician is ready to move on to another sub-task, it sends a request to task selector, which broadcasts the request to all of the sponsors in collection. Each sponsor evaluates the applicability of its resource and sends a rating back as its answer. The selector then uses these ratings, along with other problem-solving knowledge, to select the "best" resource for the client in the current context.

A new problem solver can be added to the system by constructing a sponsor for it and informing the selector of the new sponsor. Some modification to the selector's decision structure may be necessary, depending on the complexity of the selection process, but at least this knowledge is not confounded with the sponsor's resource-specific knowledge.

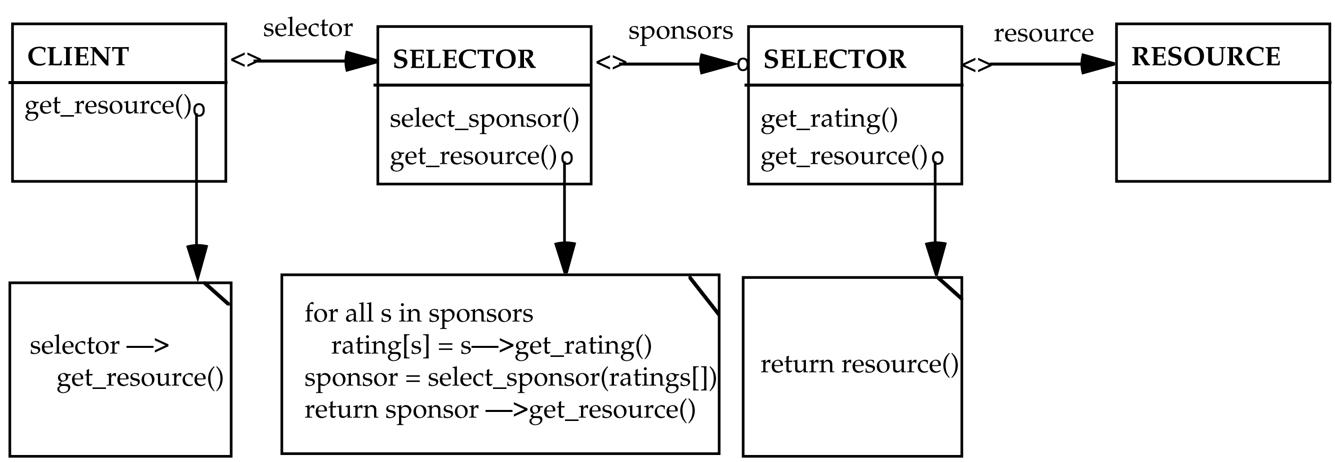
This structure is an example of the Sponsor-Selector pattern. The key to the pattern is the separation of three behaviors: recommending a resource, selecting among resources, and using a resource.

Applicability

Use the Sponsor-Selector pattern when:

- a system has multiple resources available to it
- the set of resources may change frequently, either by programming or at run-time
- the applicability and utility of the resources cannot or should not be determined until run-time

Structure



Participants

- *Resource* (Hypothesis Builder, Causal Reasoner)
- *Sponsor* (Hypothesis Building Sponsor, Causal Reasoning Sponsor)
  - — contains knowledge of when its associated resource is appropriate for use
- *Selector* (Task Selector)
  - — contains knowledge for selecting among resources based on their individual appropriateness and, perhaps, global considerations
- *Client* (Diagnostician)
  - — the object that uses the selected resource

Collaborations

- The Client asks the Selector for an appropriate resource and receives one in return, if one is available and applicable.
- The Selector broadcasts a request to the resource Sponsors, each of which rates its resource and returns a rating for the resource. The Selector uses these ratings to select a Resource to return to the Client.
- The Client uses the selected Resource in the appropriate way. For example, if the resources are problem-solving methods that can perform the Client's current task, then the Client will send the Resource the appropriate problem-solving request and will receive in return the appropriate answer.

Consequences

Many of the consequences that follow from using Strategy also follow from using Sponsor-Selector. Sponsor-Selector has the following benefits and drawbacks:

1. *High degree of modularity*. Resources can be added, modified, or re-implemented with no effect on other resources or on client code. Only the resource's sponsor (in the case of addition) and the selector (in the case of addition and perhaps modification) need be modified. The trade-off is the high degree of modularity may make it difficult to share domain knowledge among the resources, leading to replicated implementations. Singletons and Flyweights may be used to address this problem.

2. *Decision control is made explicit and independent from other components of the system*. Sponsor-Selector further improves on the Strategy pattern's separation of method and client by further decoupling *selection* of the ConcreteStrategy from the Context. Strategy leaves the client with the job of selecting the appropriate ConcreteStrategy to instantiate in the client. This can clutter the client with extra data examination code, knowledge that is not part of the client's primary task (thus blurring the explicitness of its representation), and unwieldy conditional statements.

3. *Parallel resource selection*. Each sponsor is an independent agent charged with the rating of an individual resource's applicability and utility. They can be run in parallel, or in the background, while other processes are active. The selector can be implemented as an anytime algorithm, processing sponsor ratings whenever they become available and providing a resource selection on demand.

4. *Hierarchies of Sponsor-Selector modules*. Multiple Sponsor-Selector patterns can be grouped together to form a hierarchy. This provides a means for organizing complex resource sets, such as the many tasks and sub-tasks of a complex problem solver, in a way that promotes both system performance and system modifiability.

5. *Potentially tight coupling between Client and Selector*. Pulling all of the selection knowledge out of the client may be unwise. Selection knowledge may be so context-specific as to be (1) not reusable as a separate object and/or (2) easier to express directly in the client, since large amounts of information may need to be passed from client to selector.

5. *Potentially tight coupling between Selector and Sponsor*. Sponsor-Selector may be inappropriate if the sponsor needs to take into account knowledge about the relative quality of the method's solutions in a particular context and/or global constraints placed on the selected object's behavior in order to determine the applicability of its resource. For instance, if a there are global bounds placed on the time of computation of the resource, it may be difficult to build a local sponsor capable of taking such knowledge into account.

Implementation

Three main implementation decisions must be made:

1. *How does each sponsor rate the appropriateness of its resource?* Each sponsor should be coded with only *local* considerations in mind: Is my resource appropriate in the current context? It should not make reference

to any other resources or any other global questions, if at all possible. Otherwise, modifiability of the structure is impaired.

2. *What constitutes a rating?* In some domains, sponsors may be able to generate quantitative ratings on a continuous scale, which the selector then combines using some continuous function. Experience in constructing knowledge-based systems, however, indicates that in most domains the use of qualitative ratings of applicability are more appropriate. Such ratings are more amenable to the construction of local uncertainty calculi and more nearly correspond to the qualitative judgments made by expert problem solvers. One common approach is to use a multiple-point set of ratings that range from "highly applicable" through "neutral" to "highly inapplicable".

3. *How does the selector combine the ratings to select a particular resource?* The selector ordinarily uses the ratings returned by its sponsors as its primary criterion for selection. This may involve selecting the highest-rated resource, or breaking (virtual) ties using a default ordering of resources or a random choice from among the highly-rated set. But it may also need to apply "global" knowledge that would be beyond the scope of any single sponsor (e.g., comparisons among specific resources). Again, the extent to which the selector refers directly to its sponsors greatly affects the modifiability of the structure.

Two other implementation issues should be considered:

1. *Does the selector directly invoke the selected resource or return the resource to the client?* The former is more appropriate when the resources are behaviors to be invoked in the course of problem solving, since the client is willing to pass control to the selected behavior. One advantage of direct invocation is that resources no longer need to share a common interface. Sponsors can be constructed with a common invocation interface and can serve as adapters to the resources. Returning the resource to the client is more appropriate when the resources serve primarily as data stores, since the client may need prolonged interaction (and also a common interface) with the resource.

2. *Can the sponsor objects be assimilated directly into the resources?* If sponsorship decisions are trivial, the structure of Sponsor-Selector can be simplified by adding a rating behavior to the resource objects. This can be useful when the rating behavior can be implemented in a superclass and inherited by all of the resource classes.

Known Uses

Most of the uses to which Sponsor-Selector has been put in the field of knowledge-based systems have been in problem-solving control, that is,

- to decide which method will be used to perform a particular task, or
- within a method, to dynamically select which sub-task to address next.

DSPL (Brown and Chandrasekaran, 1986) introduced Sponsor-Selector as a mechanism for selecting plans in design problem solving. The mechanism was incorporated into a shell for building knowledge-based systems that do "routine design", in which much or all of the design task is accomplished through the use of pre-enumerated design and redesign plans.

TIPS (Punch and Chandrasekaran, 1993) generalized Sponsor-Selector into a mechanism for making dynamic control decisions about sub-tasks and methods. Punch then applied TIPS in the construction of a medical diagnostic expert system.

Sponsor-Selector was used as the basis for control plan execution in IPCA (Decker et al., 1994), a system that controls microwave composite processing in a soft real-time environment. Decker has argued that deadlines associated with hard real-time processing could pose problems to the architecture.

Goel and his colleagues (Ali and Goel 1996, Goel et al. 1994) have used Sponsor-Selector for dynamic method selection in navigational planning for mobile robots. Experiments in this domain show that Sponsor-Selector is a good design solution even in a situation where the client application is learning new control knowledge. As the planner acquires domain knowledge, Sponsor-Selector enables the agent to incorporate the new knowledge dynamically into its problem solving and to select the method that is indicated by that knowledge. The planner has been embodied in a physical robot that confirms the efficacy of its plans.

FAULTY-II (Benjamins 1993) uses a Sponsor-Selector pattern as the basis of an application framework for doing diagnostic problem solving. Tasks structures are explicitly represented in the framework, and a Sponsor-Selector mechanism is used to dynamically select which method will perform each sub-task.

Related Patterns

Strategy: Often, the Sponsor-Selector pattern is used to select one method from a family of methods, implemented using the Strategy pattern, that is capable of performing a particular task.

Proxy: Sponsors can be implemented using the Proxy pattern.

References

1. Ali, Khaled, and Ashok Goel. Combining Navigational Planning and Reactive Control. To appear in *Proceedings of the AAAI-96 Workshop on Reasoning About Actions, Planning and Control: Bridging the Gap*, Portland, August 1996.

2. Benjamins, Richard. Problem Solving Methods for Diagnosis. Ph.D. Thesis, University of Amsterdam, 1993.

3. Brown, David, and B. Chandrasekaran. Knowledge and Control for a Mechanical Design Expert System. *IEEE Computer*, 19:92-101, July 1986.

4. Decker, David, Bill Punch, Jim McDowell, and Jon Sticklen. An Intelligent Control Architecture for the Microwave Fabrication of Composite Materials Based on the Generic Task Approach to Knowledge-Based Systems. In *Proceedings of the AAAI Fall Symposium on Intelligent Systems*, New Orleans, November 1994.

5. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, New York, 1995.

6. Goel, Ashok, Khaled Ali, Michael Donnellan, Andres Gomez, and Todd Callantine. Multistrategy Adaptive Navigational Path Planning. *IEEE Expert*, 9(6):57-65, December 1994.

7. Punch, Bill, and B. Chandrasekaran. An Investigation of the Roles of Problem-Solving Methods in Diagnosis. In *Second Generation Expert Systems*, ed. by J.M. David, J.P. Krivine, and R. Simmons. Springer Verlag, Berlin, pages 673-688, 1993.