Application Scenario
A Pattern Language for Business Process Control

by S. Ramesh

Table of Contents

1. Abstract

Application Scenario© is a pattern language for those systems in which the processes, rules, and interactions between business objects are the most complex part of the system. It describes a collection of patterns which make business processes explicit, and which model and implement these processes in a traceable way from requirements and analysis through design, and into first class objects in the implementation.

2. Introduction

A. Key to the Notation

The following notation conventions are used in this document:

Bold - Indicates the names of classes defined using this pattern

Bold Italic **-** Indicates instances or examples

Underlined - Indicates references to patterns (which may or may not be defined within this pattern language)

B. Motivation for the Pattern Language

In complex systems the most difficult things to model and implement successfully are the dynamic processes that the objects in the business undergo. The (static) business model is usually easier because there is more experience in implementing it. There is either an existing relational database with a data model to start from, or at least a paper based system that provides strong hints about the information that needs to be recorded, and how that information is partitioned.

The processes, on the other hand, are poorly documented as a series of business rules that the objects can follow. These rules change more quickly than the underlying static objects and they are also the least well understood part of the business, since they include many undocumented procedures and rules of thumb. This part of a business is often poorly understood both by management and by most business analysts. In addition, processes are typically not implemented in the software as cohesive units (as they are described in use cases) but rather are distributed as a web of interacting methods on business objects that must implicitly fulfill the analyzed use cases.

Because few people have a thorough understanding of the business processes, the requirements for this part of an application are often missing important steps, rules and constraints. This leads to problems of change management and traceability - it is difficult for a development team to prove that it has completed an application when the real users, brought in at user acceptance testing, find that the processes are not satisfactory. Also, since the interacting methods on business objects have little resemblance to the analyzed use cases, it is hard for a development team to be sure that they have implemented the process correctly, even assuming the original requirements analysis was perfect.

The process model, thus, does not benefit from one of the greatest benefits of object technology - the enhanced traceability from requirements through analysis, design and implementation of the object model. This improvement in traceability occurs because the OO software development process is a process of detailing and enhancing an existing model from one phase to the next. There are no major paradigm shifts, and a user that helped to create the original requirements analysis should still be able to speak knowledgeably, using the same language, with the implementers of the final product.

Things are different when we come to model the active processes that make up the business, however. The original use cases may be partially decomposed into scenarios during the analysis phase, but these continue to be at a high level. During the design phase object communication diagrams and event trace diagrams are used. These are usually at a very low level, making them hard to relate to the use cases. Because they are at such a low level, they take a long time to complete well, and are often abandoned except at the public interfaces of subsystems.

Event traces and object communication diagrams also do not provide the same amount of bang for the buck as the object model, because they are at a low enough level that academic design issues no longer dominate. This is because the complex web of interaction between objects needs to take into account the idiosyncrasies of the language and platform it is built on, the technical requirements of performance and memory utilization, and the aesthetics of minimally coupled but very cohesive code. These interactions are not amenable to top down design, but rather are built from the bottom up using design patterns, heuristics and artistry.

Unlike the static business model, which is iteratively and progressively enhanced during the software development process, the active process model is analyzed using one paradigm, designed using another, and finally implemented with a third. A user has no common language to relate their understanding of the business processes to the web of methods in the code.

Besides poor traceability and the complexity of implementation, there are other problems with using implicit processes to model a business.

The static business objects often get polluted with flags, indicators, and states that are part of the process, not an intrinsic part of the business objects themselves, and also with multiple presentations of themselves or their data for different business processes. This greatly reduces the cohesiveness of the objects, and leads to method bloat where the objects have very large number of methods that are difficult to maintain.

Also, where the processes are complex and interrelated, it is very difficult to implement the processes on one object without having intimate knowledge of other objects. This violates encapsulation and leads to a brittle, tightly coupled system.

Finally, since processes tend to change much faster than the underlying static objects, there is a general problem with maintaining the business objects. Although only one part of the business objects change, the entire system must be tested, and the system either becomes unstable or late (or often both).

There is a rule of thumb in object design that states - when one part of an object changes faster than the rest, create a new object to encapsulate the change. This suggests one possible solution to this family of problems - the business process as an object in its own right, with a traceable life cycle starting as use cases during the analysis phase and being enhanced to detailed scenarios during design and application scenarios during implementation. Implementations using this type of pattern are common throughout delivered systems.

C. The Patterns

This pattern language is made up of six alternative patterns, any one of which can be used to encapsulate business processes, and of another nine patterns that can be used effectively in combination with one of the Scenario patterns.

The six basic scenario patterns are:

1. Class Based Application Scenarios
2. Instance Based Application Scenarios
3. Externally Defined Application Scenarios
4. Class Based Typed Application Scenarios
5. Instance Based Typed Application Scenarios
6. Externally Defined Typed Application Scenarios

The additional nine patterns that can be used effectively with one of the Scenario patterns above are:

1. Application Context
2. Object Representation Framework
3. Application Support Layer
4. Application Scenario State Machine
5. Optional Application Scenario
6. Externally Defined Application Scenario State Machine
7. Business Rule
8. Externally Defined Business Rule
9. Persistent Application Scenario

Application Context, Optional Application Scenario and Persistent Application Scenario are incremental additions to the Application Scenario pattern that add additional behaviour and flexibility at low cost.

Application Scenario State Machine and Externally Defined Application Scenario State Machine add a very disciplined approach that is appropriate for complex systems, or systems that put a very high premium on stability. Business Rule and Externally Defined Business Rule work well with all Application Scenario patterns, but especially with the Application Scenario State Machine patterns.

Object Representation Framework is a pattern that further decouples the GUI, business processes, and business models, so works very well with any Application Scenario pattern. Application Support Layer is an approach in which Application Scenarios are used to create the entire logical view of the process model of the application. This pattern works well to simplify a complex architecture for application developers, and in effect allows Application Scenarios to act as a Facade for all the architecture subsystems, reducing the learning curve for new developers.

3. Class Based Application Scenarios

A. Forces

1. The business processes that the business objects must implement are poorly understood and change more quickly than the business objects themselves.
2. The processes are difficult to trace from requirements to implementation and testing.
3. The processes are complex and interrelated, and are difficult to implement within individual business objects without explicit knowledge of other business objects.
4. The processes require the maintenance of state which does not cleanly belong to the individual business objects.

5. Services provided by architectural components need to be repeatedly invoked in a standard way at the same points by multiple business processes (e.g. authentication and authorization services, persistence services, gui display services, printing services)

B. Resolution

Decompose use cases into scenarios in the design phase and into first class objects, **ApplicationScenario's**, in the implementation phase. Note that scenarios may include steps that are not explicitly part of the original use case, such as user authentication and authorization. Example application scenarios may be ***Open Account***, ***Configure PVC***, ***Add Customer, Book Policy*** or ***Distribute Fund Interest***.

ApplicationScenario classes are peer classes, each of which implements one scenario or use case. **ApplicationScenario's** use the Composite pattern, so that a use case from the analysis, which is decomposed into scenarios during the design phase, exists as an identically named **ApplicationScenario** in the implementation, composed of child **ApplicationScenarios** that each represent one of the use case's scenarios. **ApplicationScenario's** may be further decomposed during the implementation phase, both to handle complexity, and to promote reuse.

The actual methods in an **ApplicationScenario** translate the steps of a use case into methods implemented on the business objects or on architecture objects. In this pattern, all of the actual work is still done by methods implemented on business objects and architecture objects. **ApplicationScenario's** act as a high level abstraction that relates the behaviour of implemented objects to the required business processes described by use cases.

Those **ApplicationScenario's** that work on business objects are business scenarios, while those that work on architecture objects are service scenarios.

Service scenarios (e.g. authorization and authentication, printing, gui display, printing, user action/popup menu) are a great source of reuse, since a single scenario may be sufficient for each type of service in the system, and these may be reused in most business use cases in the system. Service scenarios also reduce the learning curve of new developers working on a mature architecture, since they can encapsulate the code needed to use the architecture (see Application Support Layer).

Business scenarios, however, are the real meat of the Application Scenario pattern. Business objects should not know about any scenarios that they take part in, though **ApplicationScenario's** may have an intimate knowledge of the business objects that take part in their process. Business object classes may also know about the **ApplicationScenario** classes they can take part in (building a meta information graph that relates the static objects to their active processes).

ApplicationScenarios keep their own state. They also implement those methods that don't belong on individual business objects, and also implement conversion methods to translate information from a business object into the format required for this particular process.

Some **ApplicationScenario's** have attached gui's (e.g. ***Create Account***) while others may not (***Commit All Changes***). Those application scenarios that have attached gui's open the appropriate window, and act as mediators of the communication between the gui and the business object. Every leaf **ApplicationScenario** should open zero or one gui windows - if a single leaf scenario is opening two or more windows, the business processes are becoming implicit again. It is very common during implementation to create a single ApplicationScenario that uses meta information from a passed business object class to open the appropriate gui for its parent scenario.

Although the gui may observe the business object, in the Application Scenario pattern the gui should never request an action or service directly from its business object or the system. All business process related calls must be directed through the **ApplicationScenario**. The gui may also observe the **ApplicationScenario** for those attributes (whether physical or calculated) that are attributes of the business process and not of business object itself. The ultimate form of this is that the gui only observes the **ApplicationScenario**, which acts as a [1]Proxy for its business object(s).

C. Consequences

Business process is encapsulated in the **ApplicationScenarios**. Also since the scenarios are composites, it is easier to represent complex processes. Traceability is enhanced because there is a one to one correspondence between the use cases that represent business processes in the analysis phase, and implemented **ApplicationScenario's** in the delivered application (typically, the top level **ApplicationScenario** representing the use case would be composed of implementation level service and business scenarios).

One concern about this pattern is that the number of scenario classes may become too large. Instance Based Application Scenarios and Externally Defined Application Scenarios reduce this problem.

Note that one decision that is very hard to make when using this pattern, is whether to implement a hierarchy of **ApplicationScenario's** based on similar actions performed, or on similar objects performed on. For one resolution to this problem see Typed Application Scenarios.

D. Known Uses

This pattern has been used to implement the branch sales platforms of Canadian Imperial Bank of Commerce, and the Standard Chartered Bank of Hong Kong (including: sales tools; work management; deposit, chequing and savings accounts; loans; and mortgages). It is a part of the application architecture developed at client sites to support the base frameworks of IBM's Visual Banker frameworks.

4. Instance Based Application Scenarios

A. Forces

Similar to the problems for Class Based Application Scenarios. In addition there are a very large number of business processes that need to be implemented, requiring too many **ApplicationScenario** classes. Also, this is a very convenient intermediate step to take when moving toward an Externally Defined Application Scenario framework.

B. Resolution

Decompose use cases into scenarios in the design phase and into first class **ApplicationScenario** objects in the implementation phase, as in Class Based Application Scenario. Only three or a few **ApplicationScenario** classes need to be implemented.

The **ApplicationScenario** superclass keeps track of the name of the application scenario it is implementing, and the attributes which maintains its state. A **CompositeApplicationScenario** subclass implements a list of its child **ApplicationScenario's** and the methods required to invoke child scenarios. A **LeafApplicationScenario** class implements an ordered collection that maps each step of its implementation to a method called on one of the attributes in its dictionary.

Meta information is used to create the appropriate subclass required by an **ApplicationScenario**, to populate a **CompositeApplicationScenario** with its sub-scenarios, to populate a **LeafApplicationScenario** with its process - method mappings, and to populate any attributes that are peculiar to that type of **ApplicationScenario**. This responsibility is normally given to an **ApplicationScenarioFactory**. Note that methods that are particular to a business process, in the case of instance based application scenarios, either require subclassing for that process, or must be implemented in the business objects.

C. Consequences

Business process is encapsulated in the **ApplicationScenarios**. Also since the scenarios are composites, it is easier to represent complex processes. Traceability is enhanced because there is a one to one correspondence between the use cases that represent business processes in the analysis phase, and implemented **ApplicationScenario's** in the delivered application (typically, the top level **ApplicationScenario** representing the use case would be composed of implementation level service and business scenarios).

Note that one decision that is very hard to make when using this pattern, is whether to implement a hierarchy of **ApplicationScenario's** based on similar actions performed, or on similar objects performed on. For one resolution to this problem see Typed Application Scenarios.

Note that, since each instance of an **ApplicationScenario** contains information both about the definition of that type of scenario, and about the state of the current instance, it is important to partition this information in a way that will make it easy for a new developer to understand.

5. Externally Defined Application Scenarios

A. Forces

Similar to the problems for Instance Based Application Scenario. In addition:

1. The requirements for the business processes are changing very rapidly or are not yet well defined, or,
2. There is a need for non-developers to be able to change business processes on the fly, or,
3. Changes in the business processes need to be distributed without necessitating a change in the code base.
4. The business process is itself the sellable commodity (e.g. derivative contracts, commercial banking products) and there is a significant competitive advantage to bringing products to market quickly.

B. Resolution

Similar to the solution for Instance Based Application Scenario.

Keep the meta information that defines an individual **ApplicationScenario** external to the code base, however, in a database or in a configuration file.

C. Consequences

Business process is encapsulated in the **ApplicationScenarios**. Also since the scenarios are composites, it is easier to represent complex processes. Traceability is enhanced because there is a one to one correspondence between the use cases that represent business processes in the analysis phase, and implemented **ApplicationScenario's** in the delivered application (typically, the top level **ApplicationScenario** representing the use case would be composed of implementation level service and business scenarios).

Since the definition of the **ApplicationScenario's** exists in data, changes to the business processes can be implemented without touching the code. Note that the definition of the **ApplicationScenario's** has to be very well partitioned between the code based engine, the code based definition of the meta data, and the data based definition of the instances, in order to be able to flexibly create new business processes in unexpected ways. This is quite difficult to get right on the first try, and Externally Defined Typed Application Sessions may be a better choice if this is a major requirement.

Since only the data defining a particular set of business processes needs to change for the behaviour of the system to change, the changes are likely to be extremely localized, and unlikely to affect disparate parts of the system (as code changes occasionally do!). This greatly reduces the testing and distribution requirements associated with changing the business processes, and makes these changes faster and less costly.

The information may be kept on a server for quick distribution to all clients. In the case where business analysts or non-developers may change the process on the fly, a data administration application can be built to make it simple to make the necessary changes while maintaining the integrity of the system.

Note that one decision that is very hard to make when using this pattern, is whether to implement a hierarchy of **ApplicationScenario's** based on similar actions performed, or on similar objects performed on. For one resolution to this problem see Typed Application Scenarios.

D. Known Uses

This pattern forms a part of the architecture of the trading and account management applications of Financial Technologies Inc. which have been implemented at many brokerages and investment banks in the U.S. and Europe, including Cedel Bank of Luxembourg.

6. Class Based Typed Application Scenarios

A. Forces

Similar to Class Based Application Sceranios. In addition:

1. There are a large number of **ApplicationScenarios** in the system.
2. It is possible to divide the business processes into a few types of processes that occur on different business objects or collections of business objects.
3. There is a lot of similarity in the **ApplicationScenario's** that perform a similar operation on different business objects, that cries for a way to increase reuse.
4. There is a lot of similarity in the **ApplicationScenario's** that work on a particular class of objects, in the way that they manipulate those objects, that cries for a way to increase reuse.
5. It is very difficult to choose between implementing a hierarchy of **ApplicationScenario's** that perform similar processes, or a hierarchy of **ApplicationScenario's** that act on similar objects.

B. Resolution

Compose an **ApplicationScenario** with an **ApplicationScenarioType** and an **ApplicationScenarioSubject** (or a verb and a subject). Example application scenario types are: ***Create***, ***Modify***, ***Delete***, ***Display, Print, List, Authorize, Link With, Unlink, Notify***, etc. Example application scenario subjects are ***Customer***, ***Account***, ***Policy***, ***Fund***, ***Connection***, ***Switch***, etc.

The subject of an application scenario is a wrapper (or delegate) around one or more business objects, which specifies the type of object(s) that are involved in the application scenario. There is about one **ApplicationScenarioSubject** class for each unique noun in the list of application scenarios (not including nouns that are adequately described as subtypes of a superclass). State related to the type of business object, and conversion methods that are used to translate information in that business object into a format appropriate for business processes are maintained by the application scenario subject.

It its simplest form, the **ApplicationScenarioSubject** can be an analysis object that in implementation is simply the class (or reflection information) of the business objects that take part in an **ApplicationScenario**. This reverts to a variation of the Class Based Application Scenario in which a helper class provides the information about the process. In this case, however, the subject is not able to act as an Adapter for the business object, and also does not provide process based conversion methods.

The type (or verb) of an **ApplicationScenario** represents the generic process that all business objects go through when performing that type of process.

There is at least one **ApplicationScenarioType** subclass for each unique verb in the list of **ApplicationScenario's** (not including synonyms). In the case of **CompositeApplicationScenario's**, the **ApplicationScenarioType** defines the child scenarios that need to be performed, and in practice it is often reused unchanged by all the appropriate **ApplicationScenario's**.

In the case of **LeafApplicationScenario's**, the type defines the process step to method mappings, and reuse depends on a consistent naming convention for similar methods on different objects, or on the **ApplicationScenarioSubject** acting as an Adapter. In either case some ApplicationScenario's that during the analysis phase seem to have the same type, will prove to have different types when the process they represent is analyzed in detail during implementation, and will require subclassing from a common parent.

C. Consequences

Generic definitions of business process are encapsulated in the **ApplicationScenarioTypes**. Individual variations are encapsulated in the **ApplicationScenarioSubjects**. This provides great flexibility in modeling business processes. Also, since the scenarios are composites, it is easier to represent complex processes. Traceability is enhanced because there is a one to one correspondence between the use cases that represent business processes in the analysis phase, and implemented **ApplicationScenario's** in the delivered application (typically, the top level **ApplicationScenario** representing the use case would be composed of implementation level service and business scenarios).

Although a Typed Application Scenario is a little more work to implement than a simple Application Scenario, it produces a great amount of flexibility and of reuse of analysis, design and implementation, and is usually appropriate for transactional systems, or other systems with many complex processes.

D. Known Uses

Class Based Typed Application Scenarios are a key component of the Strategic Stream Initiative frameworks at Goldman Sachs, and are used to implement all of the object oriented applications developed by the investment banking systems group, including the Principling Investment applications.

7. Instance Based Typed Application Scenarios

A. Forces

Similar to the problems for Class Based Typed Application Scenarios.

B. Resolution

Similar to the solution for Class Based Typed Application Scenario, and for Instance Based Application Scenario.

Compose an ApplicationScenario of a type and a subject (or a verb and a subject) as in Class Based Typed Application Scenario.

Only one or a few **ApplicationScenarioSubject** classes need to be implemented. Each instance keeps track of the attributes which maintains its state. The attributes may include method invocation signatures that are invoked for conversion methods.

Only two or a few **ApplicationScenarioType** classes need to be implemented. A **CompositeApplicationScenarioType** subclass implements a list of its child **ApplicationScenarioTypes**. A **LeafApplicationScenarioType** class implements an ordered collection that maps each step of its implementation to a method called on its **ApplicationScenarioSubject**.

Meta information is used to create instances of the appropriate subject and type subclasses required by an application scenario, and to populate the required information for that **ApplicationScenario**. This responsibility is normally given to an **ApplicationScenarioFactory**.

C. Consequences

Because typing is effective in reducing the number of ApplicationScenario classes in the system, this pattern is usually an intermediate step when moving toward an Externally Defined Typed Application Scenario framework.

Generic definitions of business process are encapsulated in the **ApplicationScenarioTypes**. Individual variations are encapsulated in the **ApplicationScenarioSubjects**. This provides great flexibility in modeling business processes. Also, since the scenarios are composites, it is easier to represent complex processes. Traceability is enhanced because there is a one to one correspondence between the use cases that represent business processes in the analysis phase, and implemented **ApplicationScenario's** in the delivered application (typically, the top level **ApplicationScenario** representing the use case would be composed of implementation level service and business scenarios).

Although a Typed Application Scenario is a little more work to implement than a simple Application Scenario, it produces a great amount of flexibility and of reuse of analysis, design and implementation, and is usually appropriate for transactional systems, or other systems with many complex processes.

8. Externally Defined Typed Application Scenario

A. Forces

Similar to the problems for Instance Based Typed Application Scenarios. In addition:

1. The requirements for the business processes are changing very rapidly or are not yet well defined, or,
2. There is a need for non-developers to be able to change business processes on the fly, or,
3. Changes in the business processes need to be distributed without necessitating a change in the code base.
4. The business process is itself the sellable commodity (e.g. derivative contracts, commercial banking products) and there is a large competitive advantage to bringing products to market quickly.

B. Resolution

Similar to the solution for Instance Based Typed Application Scenarios. Keep the meta information that defines each individual **ApplicationScenarioSubject** and each **ApplicationScenarioType** external to the code base, however, in a database or in a configuration file. Since each instance contains information both about the definition of that **ApplicationScenario**, and about the state of the current instance, it is important to partition this information in a way that will make it easy for a new developer to understand.

C. Consequences

Generic definitions of business process are encapsulated in the **ApplicationScenarioTypes**. Individual variations are encapsulated in the **ApplicationScenarioSubjects**. This provides great flexibility in modeling business processes. Also, since the scenarios are composites, it is easier to represent complex processes. Traceability is enhanced because there is a one to one correspondence between the use cases that represent business processes in the analysis phase, and implemented **ApplicationScenario's** in the delivered application (typically, the top level **ApplicationScenario** representing the use case would be composed of implementation level service and business scenarios).

Since the definition of the **ApplicationScenario's** exists in data, changes to the business processes can be implemented without touching the code. Because of the great increased flexibility gained by separating the subject and type of the scenario, it is usually much easier to implement a very flexible data based process model using this approach than by using Externally Defined Application Session.

Since only the data defining a particular set of application scenarios has changed, the changes are likely to be extremely localized, and unlikely to affect disparate parts of the system (as code changes occasionally do!). This greatly reduces the testing and distribution requirements associated with changing the business processes, and makes these changes faster and less costly.

The information may be kept on a server for quick distribution to all clients. In the case where business analysts or non-developers may change the process on the fly, a data administration application can be built to make it simple to make the necessary changes while maintaining the integrity of the system.

Although a Typed Application Scenario is a little more work to implement than a simple Application Scenario, it produces a tremendous amount of flexibility and of reuse of analysis, design and implementation.

9. Application Context

A. Forces

1. The system needs to maintain information about the business processes that have occurred, and some running business processes may need to access this information.
2. You have **ApplicationScenario's** that are very similar except for slight differences in process based on what was done previously.

B. Resolution

In the **ApplicationScenario** superclass implement parent and children attributes to keep track of the trail of scenario invocation in the application. Implement a **UserSession** class as a [2]Singleton that acts as the parent of all of the **ApplicationScenario's** that were not started from within another **ApplicationScenario**.

C. Consequences

With this solution, **ApplicationScenario's** that are dependent on processes that occurred before them can query attributes or delegate responsibilities back through the invocation path using a Chain of Responsibility pattern. The system can start from the **UserSession** to determine any information that it needs about business processes that have been invoked on this session, or to learn any required information about this particular user. The implementers must take care, however, to make sure that **ApplicationScenario's** don't become coupled to others that occur close to them in time, by keeping the protocol for scenario to scenario communication minimal and very generic.

D. Known Uses

This pattern was used to implement the branch sales platforms of the Standard Chartered Bank of Hong Kong (including: sales tools; work management; deposit, chequing and savings accounts; loans; and mortgages). It is a part of the application architecture developed at client sites to support the base frameworks of IBM's Visual Banker frameworks.

10. Object Representation Framework

A. Forces

1. Static business objects may be represented in multiple ways, both in the gui, and as strings for display.
2. If **ApplicationScenario's** know about all of the ways each business object can represent itself, the system becomes brittle and tightly coupled.
3. The representations of business objects change more quickly than either the business objects themselves, or the business processes they undergo.
4. The different types of representations of business objects can be well described by descriptive keys that describe the context in which the object will be represented (e.g. default editor, quick editor, admin editor, default list display, simple display text, detailed display text).

B. Resolution

Create a meta framework for associating different types of representations with their business object class or application session subject via usage context keys (e.g. default editor, quick editor, admin editor, default list display, simple display text, detailed display text). **ApplicationScenario's** that require a representation from their business object(s) only ask their subject for the key appropriate to their use.

C. Consequences

This is a simple but valuable way to decouple **ApplicationScenarios** from the gui and text representation of business objects, while still allowing them to ask for the particular representation they need.

D. Known Uses

This pattern was implemented as part of the Strategic Stream Initiative frameworks at Goldman Sachs, and are used to implement all of the object oriented applications developed by the investment banking systems group, including the Principling Investment applications.

11. Application Support Layer

A. Forces

1. The software architecture in your system, though well designed, proves to have a steep learning curve for new developers.
2. You can partition your development team into architects/system developers who mostly implement the core services and frameworks, and application developers who implement the user apps.

B. Resolution

Implement an **ApplicationScenario** for every service available from the architecture. Make the **ApplicationScenario's** easy to plug in, using the eighty twenty rule. Use the **ApplicationScenario's** to shield application developers from having to interface with the software architecture.

Assume that a small percentage of complex or unexpected requirements will still need to directly interface with the architecture and make sure that there are back doors available for every subsystem. Also ensure that your architecture team's first priority is to implement (or help application developers implement) these requirements on a custom basis for the application development. This helps to make sure that the architecture evolves to simplify the job for application developers (when weaknesses in the architecture take up too much of the architecture group's time).

C. Consequences

An Application Support Layer like this seems to provide great benefits in simplifying an architecture, even for the architecture team. It is usually the case that even the architects and system developers know well only the subsystems they themselves have designed. A service layer makes it simple for all developers to learn different parts of the system, not only by simplifying the interface, but also by providing extremely well tested examples of how it is correctly used.

12. Application Scenario State Machine

A. Forces

1. You are modeling a complex transactional system in which the business processes have intricate interactive rules, and you want to make those rules an explicit part of the **ApplicationScenario's**.
2. Business processes need to be constrained and only allowed to occur when certain conditions are met.
3. Business processes can be chained together in paths that may be determinate or optional. (See Optional Application Scenario)

B. Resolution

In the design phase of the business process model development, partition **ApplicationScenario's** into the following parts: availability, preconditions, process, and post conditions.

For availability, determine what conditions would need to be met be met for the scenario to be available to its business object. For the preconditions, determine what conditions would have to be true for the scenario to process successfully. For the process, determine what changes need to occur in the system for the scenario to complete. For post conditions, determine what conditions need to be met to prove the scenario completed successfully.

In the implementation phase, availability and process must be implemented as methods on the **ApplicationScenario's**. Availability should be implemented on the **ApplicationScenario** definition (whether class based or instance based), so that a scenario does not need to be started to determine if it is available.

C. Consequences

In complex systems, this pattern helps to produce stable systems in a much shorter time than a less disciplined approach. It is also a great help to understanding the system during the analysis, high level design phases and testing phases , as this is another pattern that is useful during the entire software development life cycle.

Preconditions and post conditions can be considered contracts. Deciding whether or not to test the contracts in delivered code (i.e. implement them as methods on **ApplicationScenario's**) is an implementation decision based on the stability requirements of the system and the effects on performance and memory utilization.

This pattern allows the code to explicitly document the rules of the business process usage. It works well in combination with Business Rule.

D. Known Uses

This pattern was implemented as part of the Strategic Stream Initiative frameworks at Goldman Sachs, and are used to implement all of the object oriented applications developed by the investment banking systems group, including the Principling Investment applications.

13. Optional Application Scenario

A. Forces

1. You want to be able to specify all the control paths through the system using an Application Scenario State Machine (including those paths that are specified at runtime either programmatically, or from user input).
2. You want to be able to query any object at runtime and determine what business processes are available to it (in its current state).
3. You want to build a dynamic system in which the business processes available to any business object (in a particular state) can be queried and presented to the actor in an automatic way, without explicit application development.

B. Resolution

Implement an Application Scenario State Machine.

Then create an **OptionalScenario** subclass of **CompositeScenario**, and define instances in which the collection of potential children are not ordered, but rather queried for availability at runtime. Any available child may be chosen by the actor (whether a human user or a programmatic interface).

OptionalScenario's can then be used to control gui's and action menus throughout the system. They can also be used to query and define new process paths within the business model, creating a dynamic system in which new application paths can be created on the fly by changing the definition of **ApplicationScenario's**, without any explicit application development.

C. Consequences

This pattern adds a great deal of flexibility to a system with little cost (if you are already using an Application Scenario pattern).

D. Known Uses

This pattern was implemented as part of the Strategic Stream Initiative frameworks at Goldman Sachs, and are used to implement all of the object oriented applications developed by the investment banking systems group, including the Principling Investment applications.

14. Externally Defined Application Scenario State Machine

A. Forces

Similar to Application Scenario State Machine. In addition:

1. The requirements for the business processes are changing very rapidly or are not yet well defined, or,
2. There is a need for non-developers to be able to change the processes on the fly, or,
3. Changes in the processes need to be distributed without necessitating a change in the code base.
4. The business process is itself the saleable commodity (e.g. derivative contracts, commercial banking products) and there is a significant competitive advantage to bringing products to market quickly.
5. You want to build a dynamic system in which the business processes available to any business object (in a particular state) can be queried and presented to the actor in an automatic way, without explicit application development. In addition, you want to be able to change the behaviour of the application and the processing paths without changes to the code base (do application development by having domain experts change data, rather than having application developers write code).

B. Resolution

In the design phase of the business process model development, partition **ApplicationScenario's** into the following parts: availability, preconditions, process, and post conditions, and implement. Implement these parts as in Application Scenario State Machine.

C. Consequences

If the definition of the **ApplicationScenario's** is kept external to the code base, it allows great flexibility in defining new processes in code. This is especially true if the Externally Defined Business Rule pattern is used to define the rules that govern availability (and optionally pre and post conditions) of **ApplicationScenario's**.

15. Business Rule

A. Forces

Business rules include availability, preconditions, and post conditions on **ApplicationScenario's**, as well as validation rules on attributes and objects in the system.

1. The business rules change faster than any other part of the system.
2. Explicit business rules within the system would improve traceability.
3. Explicit business rules acting as availability, pre and post conditions on **ApplicationScenario's** would greatly increase the flexibility of an Application Scenario State Machine

B. Resolution

Create a **BusinessRule** class, and a **BusinessRuleEngine** class.

A **BusinessRule** in its simplest case is a single statement that can be evaluated to true or false. More complex **CompositeBusinessRule's** allow boolean operators to chain simpler rules.

The **BusinessRuleEngine** evaluates **BusinessRule's** to determine the result. A **BusinessRuleEngine** may not be necessary in simple implementations, but is very useful in implementations that allow complex boolean composite **BusinessRules**.

C. Consequences

Explicit business rules greatly simplify and stabilize complex systems.

D. Known Uses

This is one of the most commonly used patterns, and there are many commercial products in OO languages, and entire systems built around this pattern

16. Externally Defined Business Rule

A. Forces

Similar to Business Rule. In addition:

1. The requirements for the business rules are changing very rapidly or are not yet well defined, or,
2. There is a need for non-developers to be able to change business rules on the fly, or,
3. Changes in the business rules need to be distributed without necessitating a change in the code base.
4. Explicit business rules acting as availability, pre and post conditions on **ApplicationScenario's** would greatly increase the flexibility of an Externally Defined Application Scenario State Machine

B. Resolution

Similar to Business Rule.

The definition of the **BusinessRule's** is kept external to the code base. A **BusinessRuleParser** may be necessary in systems that contain complex definitions of rules.

C. Consequences

Explicit business rules greatly simplify and stabilize complex systems.

D. Known Uses

This is one of the most commonly used patterns, and there are many commercial products in OO languages, and entire systems built around this pattern

17. Persistent Application Scenario

A. Forces

1. Some **ApplicationScenario's** last longer than a single user session.
2. There is a need to ensure crash recovery in certain critical **ApplicationScenario's**.

B. Resolution

Implement a **CompositeApplicationScenario** subclass called **PersistentApplicationScenario**. Add a decorator **ApplicationScenario** subclass called **PersistableState**. Whenever a **PersistentApplicationScenario** cycles through a child scenario that is persistable it commits itself to persistent medium. At any of those stages, a user can end the application and return to the same **ApplicationScenario** state on another day. If a user ends the application in a non-persistable state, when they return it will have reverted to the previous **PersistableState**.

C. Consequences

This is a very simple solution to both crash recovery and long **ApplicationScenario's**

D. Known Uses

This pattern was implemented as part of the Strategic Stream Initiative frameworks at Goldman Sachs, and are used to implement all of the object oriented applications developed by the investment banking systems group, including the Principling Investment applications.

---

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Design Patterns - Elements of Reusable Object-Oriented Software*, 1994

[2]