Enterprise Component: A Compound Pattern for Building Component Architectures

Ali Arsanjani, IBM, MUM

Pattern Name

Enterprise Component

Also Known As:

Technology-neutral Component

Context

The paradigm of building software from reusable parts and creating a capital –intensive production life-cycle which also has the flexibility of modifiability by replacing a component and plugging in a new one seems to be the future of software development. Third-party vendors can contribute to large-scale development efforts by building components and having them certified and documented well enough to be used in a repository of browsable components. We need software assets (from modeling down to code and test) that are reusable and are able to be (re-) composed in several ways; can be extended by plugging in other cohesive, runnable pieces without having to have the source code for each functional unit of software?

We have been building small-grained components for quite some time. Small-grained objects tend to build object graphs with multiple inter-dependencies. Often, small-grained components do not scale to build larger components of value across a business line or product line; at an enterprise scale. In order to match the demands of business requirements with functionality that not only works correctly but also satisfies non-functional requirements of scalability, performance and reliability, we need to map the business architecture to a suitable component architecture that can be rapidly re-configured to satisfy a dynamic collaboration.

Many times, we would like to satisfy the open-closed principle (software is open to extension but closed to modification) but find that the software built before we came on board the project will not conform to any given standard or pattern and thus changing it will be even more difficult. Thus, a pattern is needed to serve as a development standard against which teams can base their software development efforts with the hope that by conforming to the standards, their software is more reliable, extensible and will be easier to extend without making too many intrusive changes (open-closed principle). Thus, for example, we would like to be able to incorporate business rule changes by having code that makes it easy to find, modify and re-deploy business rules in a component context with minimal effort and wasted time.

Problem

How do you design and implement an enterprise-scale coarse-grained business component?

What are some best-practices for modeling and implementing medium- to large-grained components that have special provisions for easily finding, changing and re-deploying business logic based on changes to business requirements – all in an assembly-oriented paradigm of software development?

How do you model, design, implement and deploy software that is founded on an assembly-based paradigm? How do we build enterprise-scale, reusable software assets for business systems?

Forces

- **Standards or "one-shot":** Any software can be written in a "one-shot", "ad hoc" manner that would provide the necessary functionality; but can we scale this to large numbers of people on multiple teams? Standards can be taught to ensure uniform and extensible development of code across teams. One-shot solutions are a bad recipe for lack of extensibility. Need to teach large group of developers how to design and implement a technology neutral business component at an enterprise scale.
- **Build or Buy**: Should we rebuild the system from scratch or buy a new system? Should we/can we buy parts?
- **An independent unit of functionality vs. small-grained instances**: Everyone defines a component differently. What is a component? How can we build something on which there is no consensus?
- Assembly vs. Construction: Should we build the system from scratch or should we integrate ready-made parts into the cohesive structure of a running application after some modification and customization?
- **Large-grained vs. small-grained units of functionality**: should a component be a conglomeration of objects or a single object?
- **Should it encapsulate a business function or a smaller technical micro-architecture**? Should we start by mapping the business needs based on a domain architecture or Blueprint? The business architecture should be mapped to the software architecture to guarantee relevance of the software system to business priorities and drivers.
- **Maintainability vs. extensibility**: Software tends to have an architecture that gets more brittle with changes; entropy creeps in and hardens the systems arteries. Thus, legacy system are developed over time that are monolithic and difficult/costly to maintain. New software is easy to maintain; just change it. Pretty soon you will end up with a "legacy system" which constrains your way of development, functionality you can offer the business you are supporting. So writing software that can be changed easily to accommodate new requirements that were not envisioned is hard.
- **"Changing the guts vs. changing the skin"; Intrusive vs. Non-intrusive changes to business functionality: Open/Closed principle**: "Software should be open to extension but closed to modification" [Meyer86]. If we are writing components, what should their hotspots be? Should they have extension points and variation points? How can we implement these variation-points in a uniform and consistent manner?

Solution

Therefore, starting at a business level, identify key business domains and partition them using a Blueprint. Identify and encapsulate each Subsystem that will later be implemented as a component. To have teams of developers design and implement a standard architecture of a technology-agnostic component, you must specify a template design that can be reused. Due care must be taken to encapsulate what they already have without making developers needlessly recreate code they already have. Thus, put a façade around their framework or subsystem. Add a Mediator for every major business grouping of use-cases and make the rules governing the component Pluggable and configurable by putting the business logic inside of Rule Objects (aka Rules).

The recipe is thus: "Design a Component to consist of a Composite Mediating Façade [1] with Rule Objects." The Rule Objects provide a dynamically configurable and reflective behavior that can be plugged into the component in the spirit of the Meyer's Open-closed Principle using Rule Objects. The structure of the solution using this compound pattern is discussed below.

Here, the component contains a number of business objects that are potentially reusable across the enterprise. The application specific logic is encapsulated in the Mediator that the component contains (e.g., Mortgage Mediator). The Component's Façade is a set of interfaces that hide the complexity of the services that the business objects can potentially provide. They will frequently be the use-cases that this component must provide an implementation of; use-cases will become the interface messages on the façade. Business Objects can share Rule Objects and perform checks and validations based on the context passed to them/accessible to them through the Data Transfer Object, which is usually a "flat" "unintelligent" transport object.

Build Small Frameworks and encapsulate them to build components. Smaller frameworks are more reusable and easier to learn, use and maintain. Encapsulate them by a façade that exposes their primary business functionality as determined in a use-cases analysis. The framework execution flow will need to be modified to include a hot-spot for other mediators to be invoked if necessary (e.g., through a Template Method). This will allow us to circumvent a common framework integration problem (each framework have their own execution flows that do not anticipate integration with other execution flows).

At the design level, specify what the encapsulated light-weight framework should provide and what it requires. In this way, you define a Component Contract, which will be satisfied by other components within the Component Context.

Summary Solution

The following class diagram captures the geneal solution that the Enterprise Component compound pattern provides. The subsequent section entitled „steps to the Solution“ discuss the process of arriving at this solution structure using an Banking example.

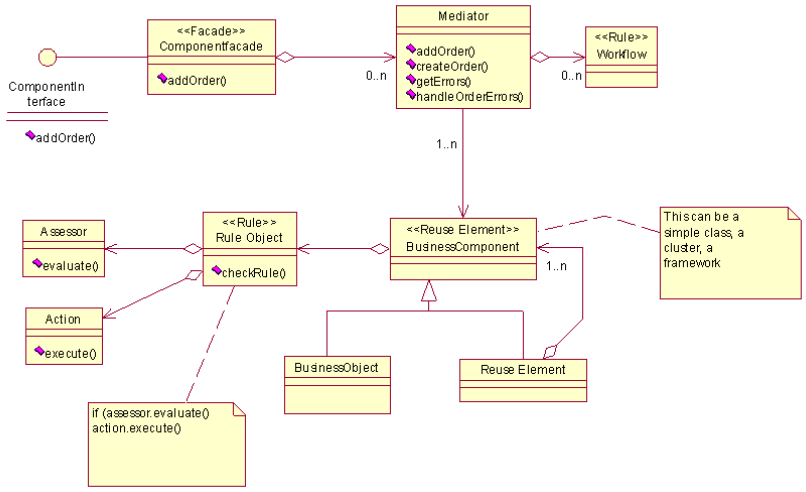

Figure 1: Solution Structure for Enterprise Component

Steps to the Solution

Step 1: The Façade

A façade is used to encapsulate the complexity of a system of objects or components and to present a uniform interface that hides the details of each individual object or component’s interface behind a common service-oriented interface that chunks the work in terms of larger granularity while its participating objects or components tend to be at a lower level of granularity.

Thus, the façade acts as a "gate keeper" and filters larger-grained, business process level requests for services.

There may be an existing set of classes that are collaborating in a given context. For example:

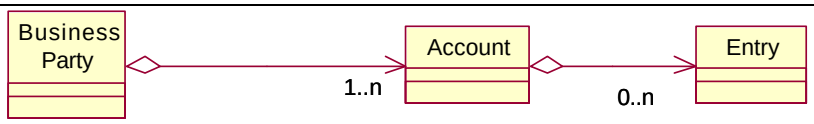

Figure 2 Business Object Model for Party-Account-Entry

A Business Party may have several Accounts with a Bank. Each of these Accounts will typically have number of Transactions or Entries in a Ledger associated with financial transactions that have been applied to it including debit, credit, transfer (e.g. debit and a credit) etc.

When implemented as a set of collaborating classes in detailed design, we may get something like this:

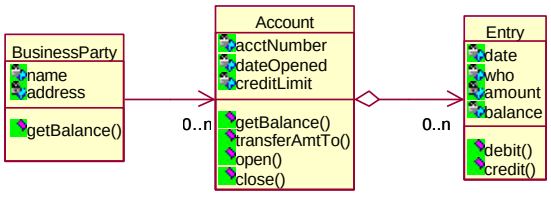

Figure 3 Detailed Design Object Model for Party-Account-Entry

Method Sub-pattern: *Subsystem Discovery*. When performing Object Discovery, objects may not be the right level of granularity. It is being increasing seen that higher Levels of Reuse need to be consciously adopted that are larger grained than a class: aggregation hierarchy, inheritance hierarchy, cluster, subsystem, framework and component. Of the latter Reuse Levels, the subsystem (a special type of cluster) seems most interesting since therein lies the opportunity to implement the subsystem as a component. ***Therefore***, we propose Subsystem Discovery or design by Cluster as an alternative to Object Discover within the context of large enterprise application development and especially those involving product line or business line architectures.

When we add the Façade, we get:

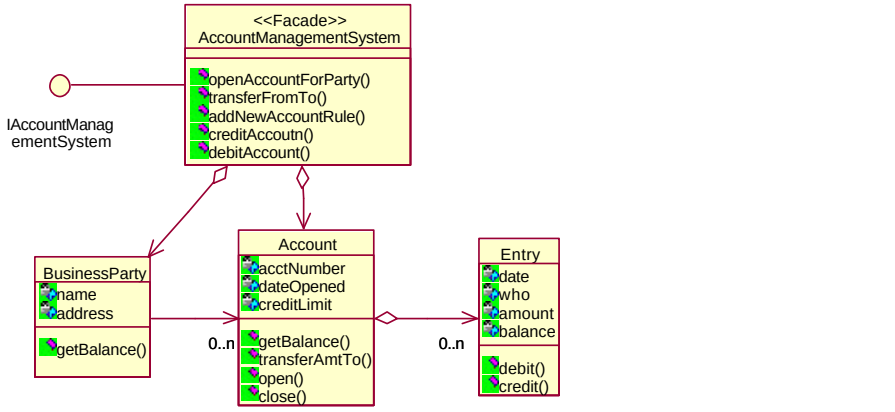

Figure 4: The PAT Aggregation Hierarchy as a Subsystem with a Façade

Often, the Façade who is an instance of the realization of the subsystem will implement an interface as in figure 3 above.

But what should go on the façade interface?

Method Sub-pattern: Use-cases on Façade Interface. When we perform use-case analysis, we tend to chunk up the use-cases into intents at the business level, and lower level use-cases; usually going down to three levels of use-case details (sub-use-cases and "sub-sub" use-cases). But what services should be provided by a large scale enterprise level component? We do not want to put fine-grained services on the interface, we do not want to have all helper methods and housekeeping methods so we are often challenged as to what methods to put on the Façade's interface. ***Therefore***, first choose the set of high level use-cases and put them on the Façade. This serves as a solid starting point that you can expand on and elaborate with helper methods later on. But the use-case on façade interface remains intact despite all other changes and perturbations.

Step Two: Adding the Mediator

A Mediator decouples the tangled web of collaborations and "uses" relationships between a set of collaborating classes ("colleagues") by having each refer to a central director, coordinator, controller or manager. This Mediator thus has a reference to all colleagues.

Having the Façade class bloated with implementing use-cases works only for small systems and quick and dirty implementations. Scalable solutions need separation of concerns and distribution of responsibility to effect better cohesion within the module.

Therefore, for every main grouping of use-cases that are now on the Façade interface, create a Mediator that will take on handling that type or class of use-cases. Use-case types stem from business process groupings and form a seamless mapping from the business architecture to the current software architecture.

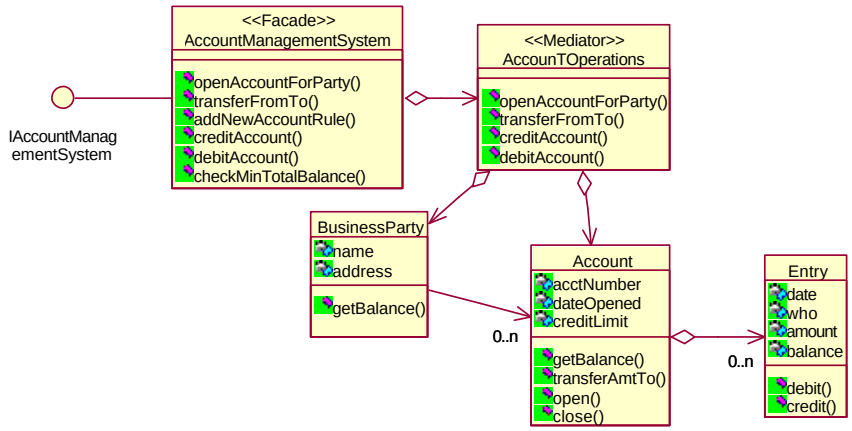

Figure 5: The PAT Subsystem with a Mediator

Notice that even in this small example, not all interface methods have made their way into this Mediator. We may decide to have one juts for handling business rules, both from a business rules handling perspective and from a rule management perspective; thus:

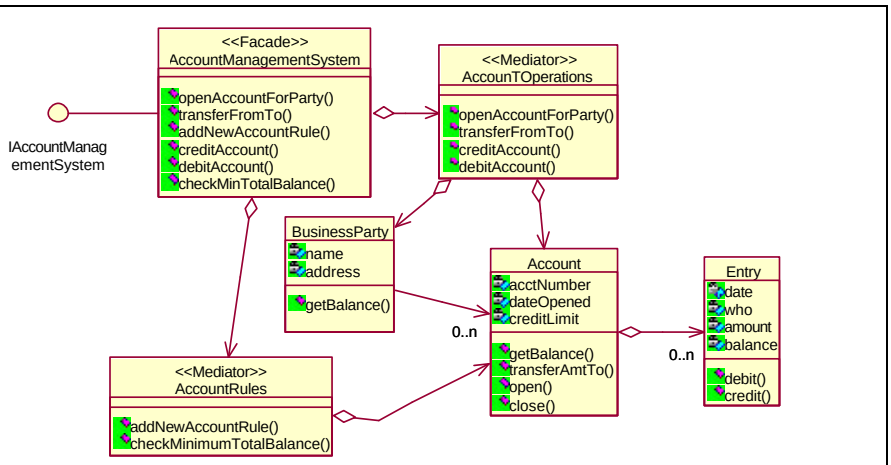

Figure 6: The PAT Subsystem with a Mediator

Sub-pattern: Mediators have Factories. Once you have a Mediator, you may want to delegate the creation of business object instances to a central Factory (or Home, in some cases). ***Therefore***, either the Façade has a Factory or the Mediator has Factory. The Factory will be responsible for loading Configurable Profiles, Configurable Workflows (reified, externalized collaborations that can be dynamically and adaptively manipulated and changed even at run-time, making a change in a collaboration or a workflow a configuration issue rather than one of code change and maintenance with the caveat that there are limits to this. See the related patterns Configurable Profile and Configurable Workflow.)

Implementation Note: Remember of course that in the trivial case, the Façade and Mediator roles can be "rolled" into the same implementation class.

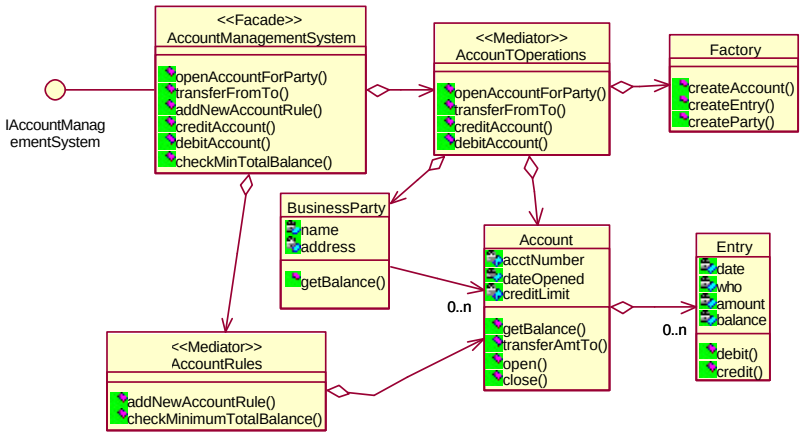

Figure 7: The PAT Subsystem with a Factory

Step Three: Pluggable Rules

We often must change the behavior of our Enterprise Component to reflect new changes in the business based on new requirements stemming all the way from competition, legislation to adding new product and services to capture new markets and retain clients.

These business rules are at two levels, one that are more declarative and have to do with workflow or Reified Collaborations and those that relate to a particular class, type of group of business objects or components:

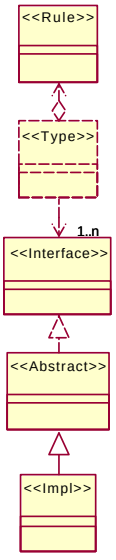

Figure 8: Meta-Model

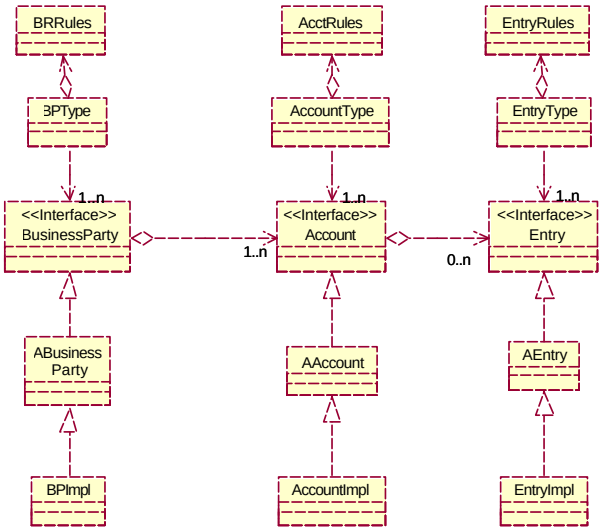

Figure 9: Adaptive Object Model for Party-Account-Entry

In general, we will use the Rule Object pattern that reifies a rule and makes it Pluggable:

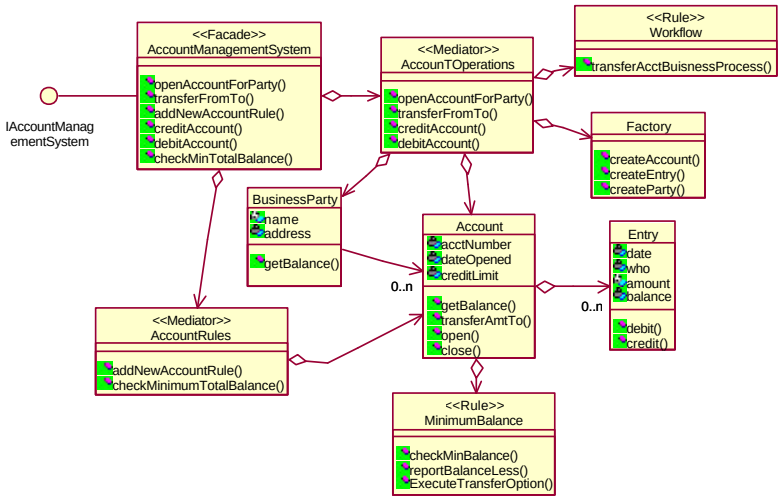

Figure 10: Adding a Workflow Rule and a Minimum Balance Rule

Step Four: Composite Elements

Although the ultimate programming language constructs to implement the business objects will be classes and their instances at run-time, we must emphasize that in the case of large enterprise systems, the Enterprise Component (EC) can be a smaller medium-grained component/subsystem such as a Rating Engine or a very large-grained application such as a Billing System. Thus, an Enterprise Component's Mediator may have references to more granular reuse elements such as other components that have a small subsystem's design under the covers and hide a set of classes under their own Enterprise Component. This Composite nature is the last essential pattern element of EC. (See figure one)

Consequences

You now have encapsulated a subsystem in a component. The services it has exposed may be invoked from any program that supplies the services the component expects – you have a component contract specified. You have a façade that not only exposes a chunk of functionality provided by the component, but also serves as a point where business rules can be added in adaptively to the component.

Now, using the `Mediator`, you can define and redefine workflow inside the component by changing the Rule Object that encapsulates its workflow.. You can thus have the property of dynamic configuration, which means that the mediator can use its `Factory` to create and configure a new set of business objects participating in a dynamic collaboration. A dynamic collaboration is possible through the use of a `Configurable Workflow` and `Configurable Profile`.

Known Uses

Used across multiple projects for various clients over the past 2.5 years. Many industry thinkers have mentioned one of the patterns in isolation but their combination has been seen to be extremely powerful. Currently EJB best-practices (Brown et al.) [2] and J2EE and Business Logic Implementations (Fowler et al.) [3] point to similar use of the façade.

References

[1] Brown, K., et al. *WebSphere Best Practices*, May 2001, Prentice-Hall
[2] Fowler, M., J2EE and Business Logic, Software Development West Proceedings, 2001.

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of reusable Object-oriented Software*, Addison-Wesley, 1994.