Domain Model Pattern

Domain Model Context

- First and foremost, a software architecture must be a meaningful expression of the system's application domain.
- Specifically, the functionality and features provided by the system must support a concrete business, otherwise it has no practical value for its users.
- If the system's software architecture does not scope and portray the application domain appropriately, it will be hard, if not impossible, to provide user-level services and features that correctly address the functional requirements of the system.

Domain Model Context (2)

- Requirements in a working system must be addressed by concrete software entities. If these entities and their interactions are unrelated to the application's core business, however, it will be hard to understand and communicate what the system actually does.
- Similarly, it will be hard to meet system quality of service requirements, since they cannot be mapped clearly to the software elements where they are relevant.
- Without a clear vision of a system's application domain, therefore, software architects cannot determine if their designs are correct, complete, coherent, and sufficiently bounded to serve as the basis for development.

Forces: Variability

- A further concern when modeling the functional architecture of a system is variability.
- Variations can arise in regard to different feature sets, alternatives in business processes, choices for concrete business algorithms, and options for the system's appearance to the user.
- There are often complicated and ever-changing business rules involving validation, calculations, and derivations.
- Without a clear knowledge of what can vary in an application domain, and also of what variations must be supported, it is hard to provide the right level and degree of flexibility in a software system or product.

Problem

- When starting to build a large system we need an initial structure for the software being developed.

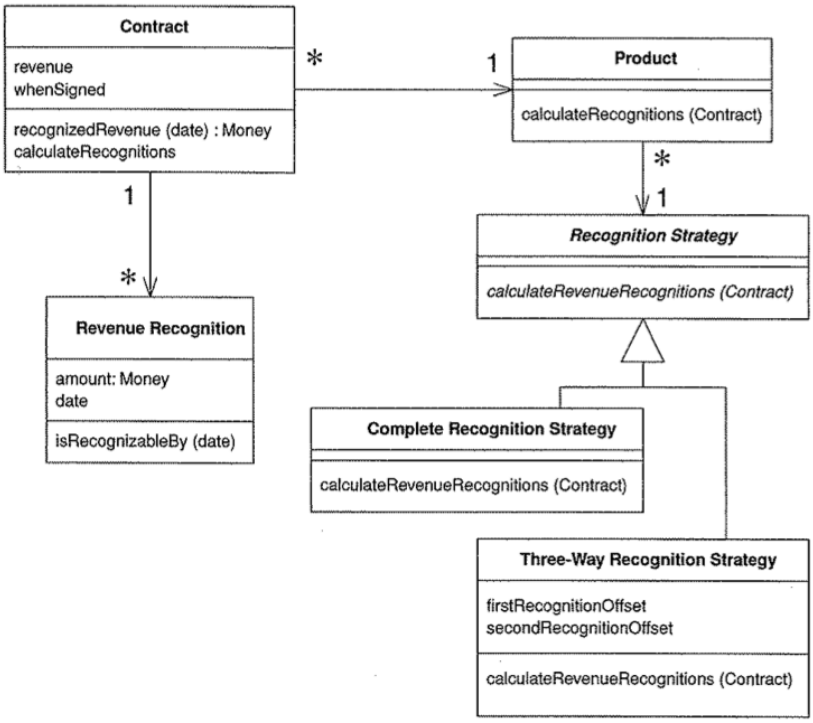
Solution

- Create a layer of objects that model a business area and its variability:
  - This layer defines a precise model for the structure and workflow of an application domain— including their variations.
  - Model elements are abstractions meaningful in the application domain; their roles and interactions reflect domain workflow and map to system requirements.

Example: Revenue Recognition

- Revenue recognition is all about when you can actually count the money you receive on your books. Some of the rules are set by regulation, some by professional standards, and some by company policy.
- Suppose a company sells three kinds of products: word processors, databases, and spreadsheets.
- According to the rules:
  - When they sign a contract for a word processor they can book all the revenue right away.
  - For spreadsheets, they can book one-third immediately, one-third in 60 days, and one-third in 90 days.
  - For databases, they can book one-third immediately, one-third in 30 days and one-third in 60 days.

Example: Revenue Recognition (2)



Domain Model Benefits

- Provides a starting point for developing a software architecture.
- Defines a precise model for the structure and workflow of an application domain, including variations.
- Helps to map requirements to concrete software entities and check whether requirements are complete and self-consistent.
- Isolates the domain model from the rest of the application (as in MVC).
- Accommodates variability in business rules.
  - For example, in revenue recognition, *recognition strategies* provide well defined plug points to extend the application.
  - Adding a new revenue recognition algorithm involves creating a new subclass and overriding the *calculateRevenueRecognitions* method.

Domain Model Limitations

- Dividing the core structure of a distributed software system purely along lines visible in the application domain will not always help to define a feasible baseline architecture.
- On one hand, a software system needs to include many components and exhibit many properties that are unrelated to its domain.
  - For example, quality-of-service requirements, such as performance and predictable resource utilization, are cross-cutting issues and therefore cannot be addressed through component decomposition alone.
  - Similarly, the need for responsive user interaction can conflict with the latency and partial-failure modes associated with networks.
- On other hand, developers want more than a system that simply meets the visible user requirements.
  - Developers are concerned with qualities such as portability, maintainability, comprehensibility, extensibility, testability, and so on.

Refining the Domain Model into a Software Architecture

- Finding a suitable application partitioning depends on framing answers to several key questions and challenges:
  - *How does the application interact with its environment?* Some systems interact with different types of human user, others with other systems as peers, and yet others are embedded within even more complex systems. Inevitably, there are also systems that have all of these interactions.
  - *How is application processing organized?* Some applications receive requests from clients to which they react and respond. Other applications process streams of data. Some applications perform self-contained tasks without receiving stimuli from their environment. Indeed, for some applications, it may not even be possible to identify any concrete workflow and explicit cooperation among its components.

Refining the Domain Model (2)

- *What variations must the application support?* Flexibility is a major concern in software development, especially when developing software products, or software product families, that are intended to serve a whole range of different customer needs. Some systems must support different feature sets, such as for small, medium, and large enterprises, to address different markets and customer groups. Other systems must support variations in business processes, so that each customer can model the workflow of its specific business appropriately. Yet other systems must support variations in algorithmic behavior and visual appearance to be attractive to a broad range of customers.
- *What is the life expectancy of the application?* Some systems are short-lived and thrown away when they are no longer used, such as an online trading program designed to exploit a transient market trend. Other systems will be in operation for thirty years or more and must respond to changing requirements, environments, and configurations, such as Telecommunication Management Network (TMN) system.

Refining the Domain Model (3)

- Other patterns help in the transformation of a Domain Model into a technical software architecture that can serve as the basis for further development:
  - Layers
  - Model-View-Controller
  - Presentation-Abstraction-Control
  - Microkernel
  - Pipes and Filters
  - Reflection
  - Shared Repository
  - Blackboard
  - Domain Object
- Each of these patterns provides its own answers to the questions raised above.