
Software Development Life Cycle Training

Use Case Driven Design

Use Cases represent a very effective mechanism for the solicitation, capture, and communication of software system requirements. Typical business tasks are performed in a specific business context, such that there are a linked set of actions that must be performed together to fulfill the business objectives. Structured requirements tend to obscure these dependencies by capturing requirements as a set of independent statements. While this approach makes managing requirements easier, and sometimes eases the job of validation by test, it forces the developers to infer these connections. Since the dependencies between, say, billing and inventory control for leased equipment is not explicit, there exists the possibility that these connections will be incorrectly designed.

Use Cases, by contrast, describe the system behavior from the perspective of the system user. This approach highlights the interdependency between different flows, including the dependency of one or more areas on fundamental processing (e.g. security). By placing the system requirements into a form that explicitly notes task interactions, the designer is better able to determine the effect of one action on other related flows.

Designs can often be directly derived from the Use Case flows by determining the system Actors, interfaces, and primary processing flows, as was discussed in Set B, Part 3. Architectural Analysis.

Use Cases vs. Structured Requirements

The primary difference between structured requirements and use cases is the “story” aspect of use cases. Use cases are text-based descriptions (or diagram-based in the case of Activity Diagrams) of the interaction between an external Actor¹ and the software system. As such, they are built from a series of “flows” representing the basic, alternate, and exceptional paths that can be followed during any given session. By capturing system behavior in response to Actor directives, the requirements engineer can place the system specifications into the business context, such that each use case results in something of value to the Actor. This technique also better supports the design concept of encapsulation, since each system function is grouped with related functionality – facilitating cohesive design.

Structured Requirements (i.e. “the system shall perform ...”) by contrast are based on the ability to reduce the system requirements to minimal atomic units. These units are then captured as statements that can be directly coded as functions. This approach often leads to difficult to connect functional directives that may be missing critical elements such as error handling or data structures. Structured requirements are, however, very good at capturing non-functional requirements for performance, scalability, usability, and reliability.

¹ Recall that an Actor may be a human user or some other external system (e.g. hardware/software)

Another critical area where Use Cases and Structured Requirements tend to differ is in the mechanisms used to trace between implementation and specification. This is an important feature of system testing, where the system is verified that all of the intended functionality is present and conforms to the original specifications. Traceability between requirements and system design for Structured Requirements is often accomplished via some form of trace matrix (Figure 1).

	Design: AccountManager	Design: SecurityController	Design: ProductCatalogManager	Design: ShoppingCartController	Design: CreditAgencyAdapter
Requirement 1: Create Customer Account	X				
Requirement 1.1: Modify Customer Profile	X				
Requirement 2: Authenticate User (Login)		X			
Requirement 3: Display Product List			X		
Requirement 4: Check Out (Purchase)				X	
Requirement 4.1: Validate Credit Card					X
Requirement 5: Manage Product Selection				X	

Figure 1. Structured Requirements Trace Matrix

Use Cases utilize a slightly different traceability in the UML using the idea of a “Use Case Realization” where the software components implementing the use case functional flows are noted (i.e. subsystems). Rather than trace the fine level of each requirement statement, the use case realization strategy permits a higher level abstraction that is typically easier to maintain. If the testing team needs the additional detail, it is possible to perform the Use Case traceability at the Scenario Level, with the additional overhead of maintaining the linkages. Using a tool to support this level of traceability is a good idea to reduce the level of effort required to keep the trace matrix accurate.

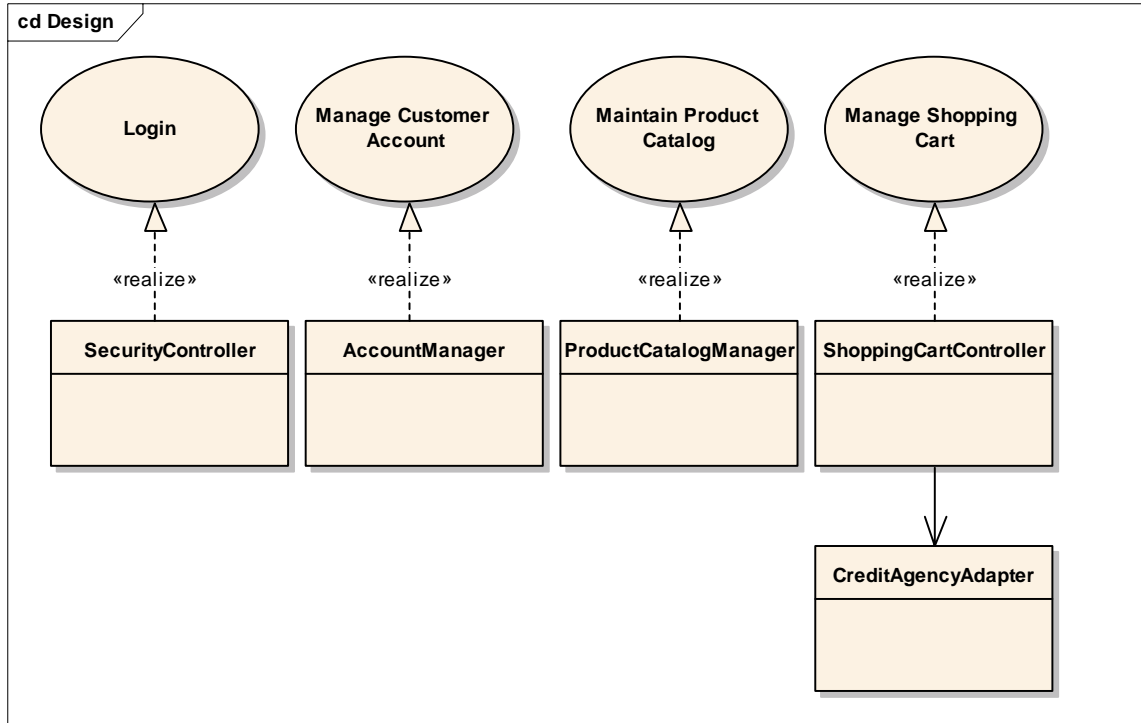


Figure 2. Use Case Realization (UML Diagram)

Finally, Use Cases can be more readily shown to handle the low probability (but important) marginal paths. Activity diagrams are particularly useful for indicating this situation (Figure 3):

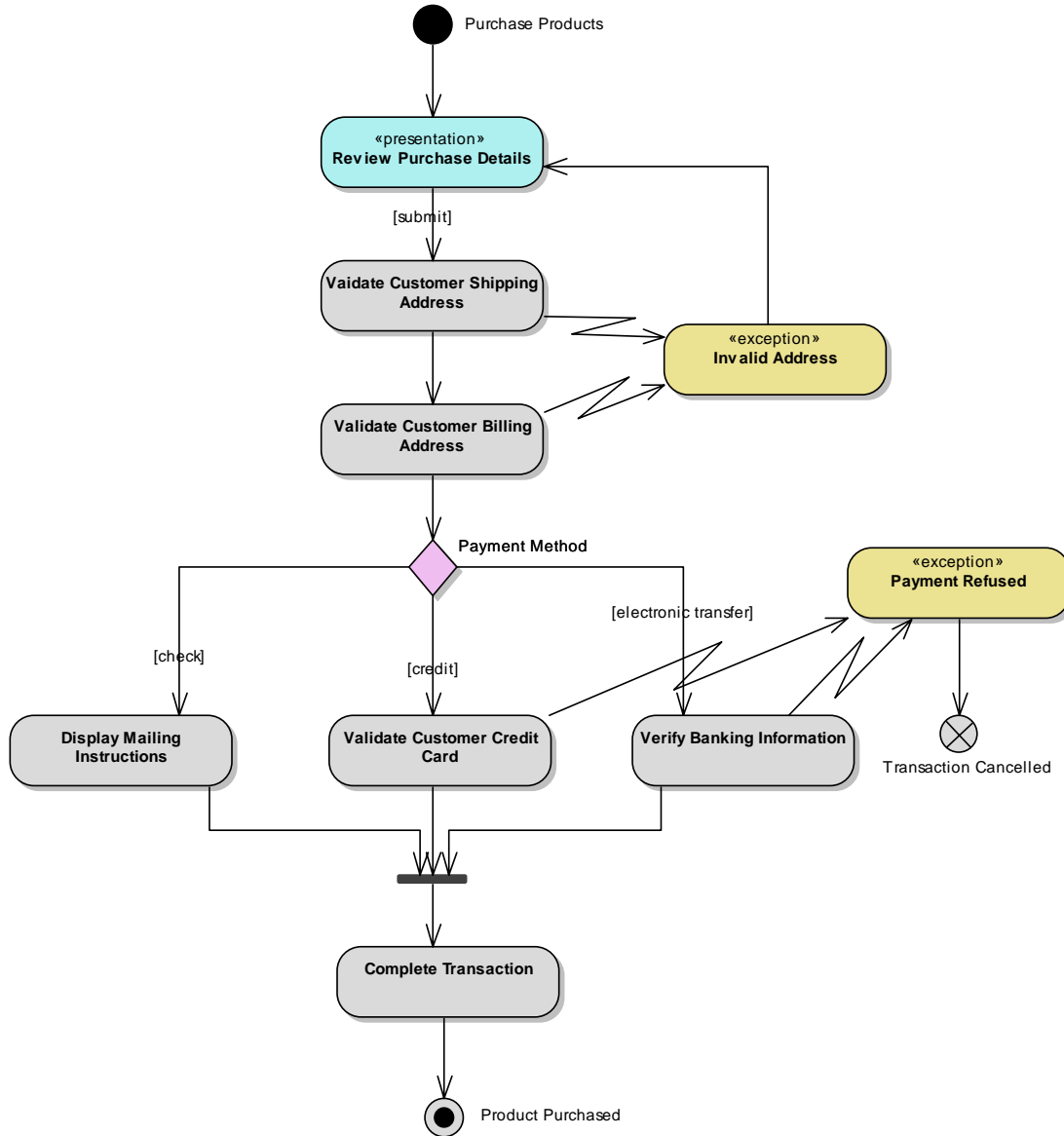


Figure 3. Use Case Activity Diagram

As a rule of thumb for user interactive systems the use cases capture 85-90% of system requirements. The remaining requirements (e.g. non-functional URPS – Usability/Reliability/Performance/Scalability) are captured in the more traditional Structured Requirements document, e.g. Supplemental Specifications.

Interpreting the Analysis Model

If an analysis model has been created, the identified interfaces, controllers, and entities can be fairly readily mapped to implementation classes and interfaces (Figure 4, and see **Set B: Part 5. Use Case Analysis**). The intent of an analysis model is to bridge the gap between requirements and system design, and can be created from either use cases or

structured requirements. In either case the three main component types for a system will be outlined, which is a good starting point for design specifications.

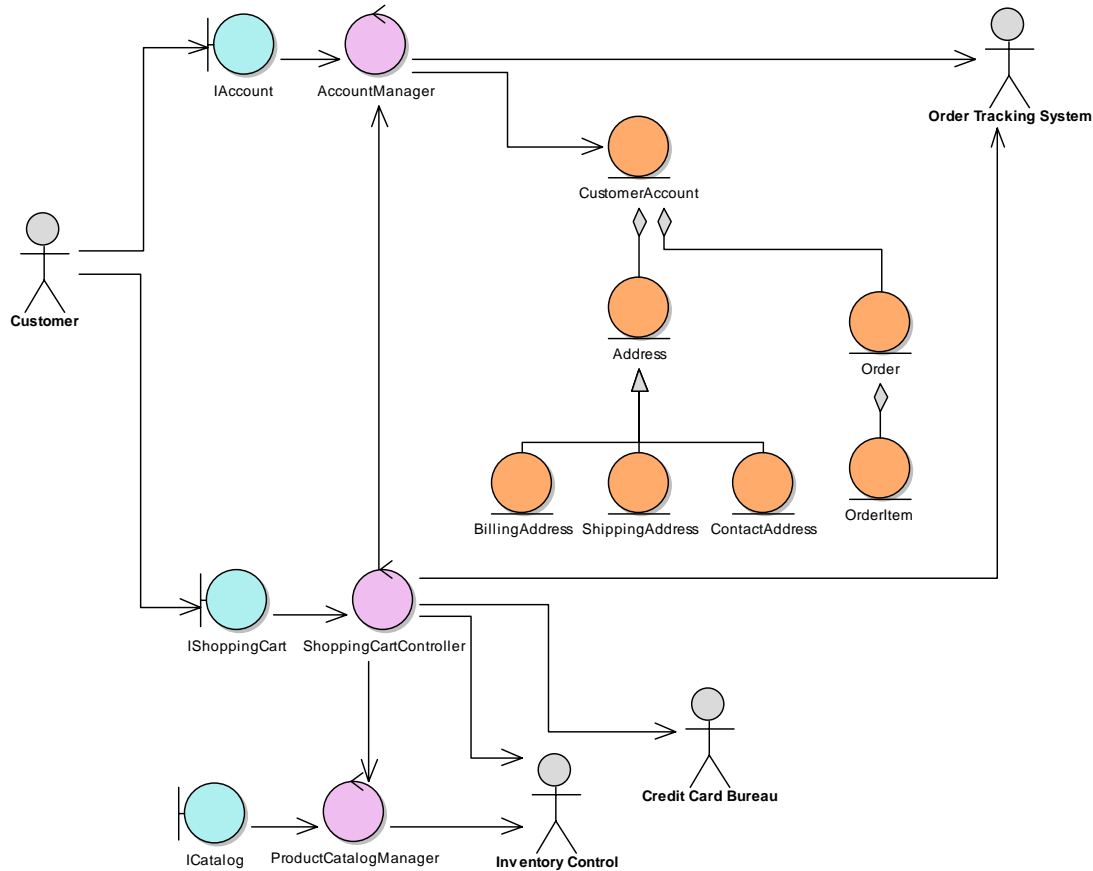


Figure 4. Partial Analysis Model (On-line Transaction Processing)

Interfaces in an analysis model are abstract “place-holders” that are then defined by some implementation mechanism. For example, the *IAccount* interface in Figure 4 can be implemented using a variety in input/output mechanisms, including a command line interface, web-application, client application, main-frame workstation, etc. The key feature is that there is a dependency between the *Customer* actor and the *AccountManager* component. The designer is then expected to take the non-functional requirements into account and select the appropriate technology for implementing the interface. In Figure 5, the designer (in this case, me) has chosen to implement the *IAccount* interface with a J2EE Model-View-Controller servlet/session-bean solution. The design classes highlight the critical dependencies of the implementation, including the connection between the view (*AccountServlet*) the controller (*AccountManager*) and the data model (Data Access Objects – DAO, and Value Objects, VO of the Data Access Pattern). Note that the design introduced an element that was not in the analysis model – the *OrderManager*. In this design, the circular dependency between the *AccountVO* and the *OrderVO* (shown with two dependency arrows) is managed by the *OrderManager*, which is acting in the Mediator Pattern).

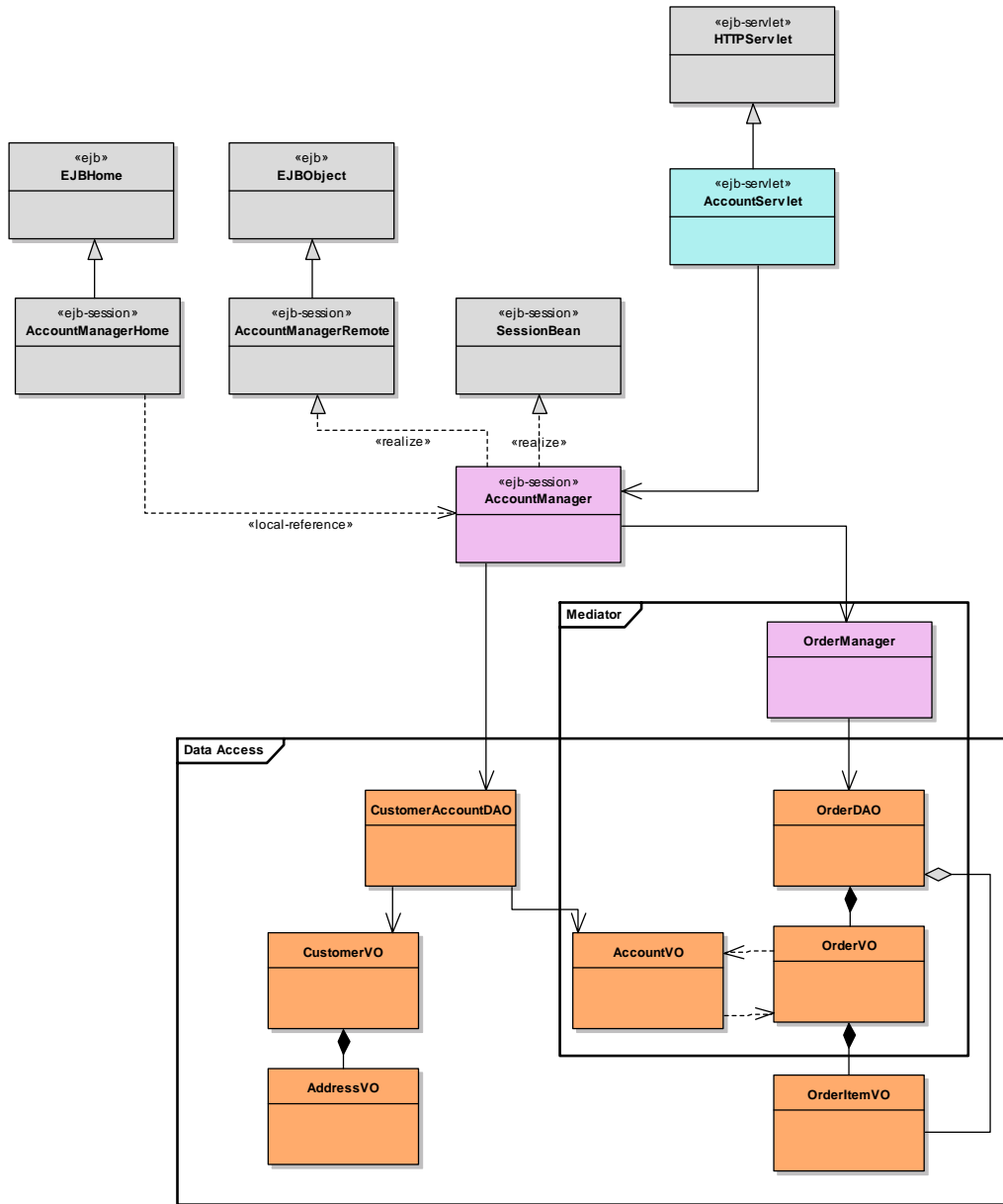


Figure 5. Simple Implementation Model for Account Management

This technique is based on a top-down approach to analysis and design. This is most useful for new systems where investigation of the problem domain is of greatest priority. The real value of the Analysis Model is to guide the initial Design Model, rather than to act as the sole design artifact. In other words, the Analysis Model is not intended to be directly coded, nor should it maintained after a candidate design is established. The Analysis Model is a powerful way to bridge the gap from the human-language requirements into the implementation language of software. Once a particular implementation is chosen, however, the Analysis Model loses its power and value. From that point forward, the Design Model is the primary development artifact.

Extending an existing system will often occur in a bottom-up approach since there is existing components that will need to be considered in the design. For this approach, a temporary Analysis Model may be constructed to show how the new requirements fit into the existing design, particularly where the design is complex. This permits the architect and designer to see the overall impact to the system before code development begins. Again, after the implementation is chosen, the Analysis Model should be discarded in favor of maintaining the Design Model.

Organize the System around Subsystems

During the transition from analysis to design highlighted above, it will become necessary to organize the design model to keep track of dependencies between different system functional areas. One of the best techniques for organizing code components is to group functional processing into a set of SubSystems, Components, and Packages. SubSystems are defined as a Package of Components that has a well defined interface, through which services are rendered. No dependencies exist directly on SubSystem components other than through the interface. This forces a “black-box” approach to the design, hiding details of implementation from other areas of the system.

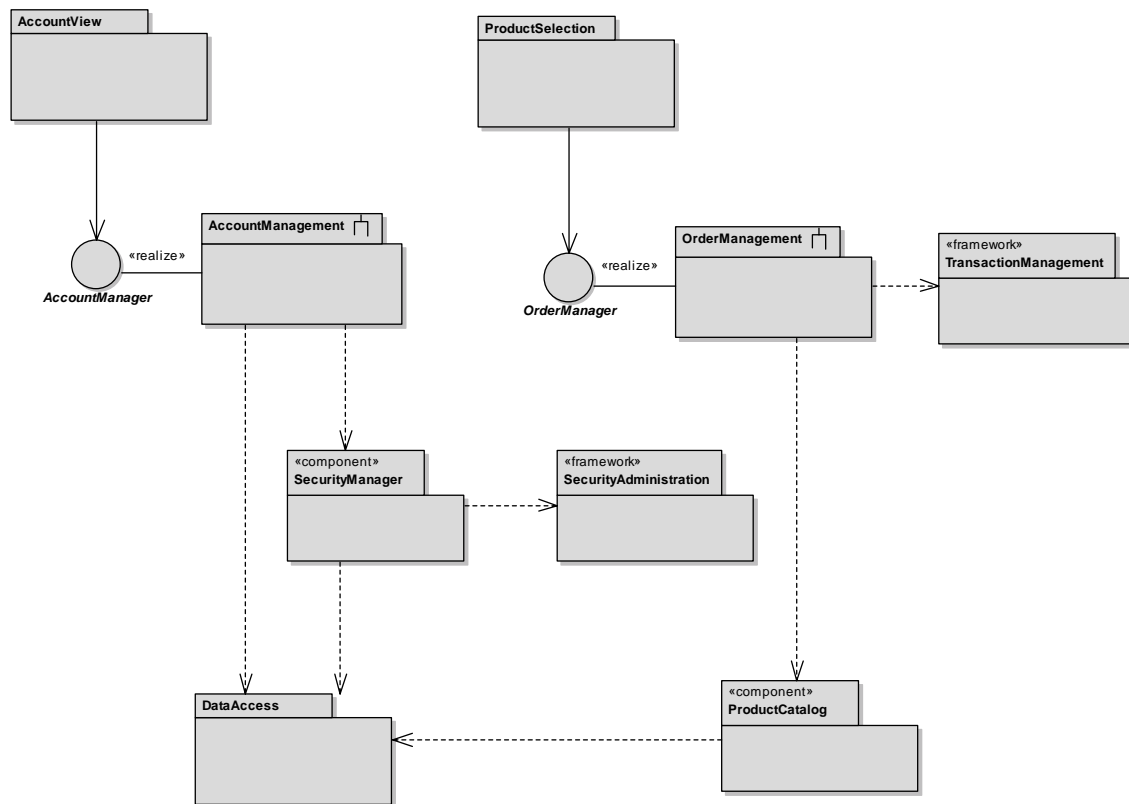


Figure 6. Structuring the System into Packages and SubSystems

Notice in Figure 6 that the difference between a subsystem and a package is the presence of an interface. A subsystem may have more than one interface; the only rule is that all communications with the subsystem occur via these interfaces. Packages are simply

organizational collections, albeit with an eye toward cohesive behavior. Components have characteristics of both subsystems and packages, in that they often have interfaces, but differ from packages in that they are usually a set of very tightly coupled classes that perform a well defined function. For example, in the figure above, the DataAccess package is used by several other subsystems and components, with no restriction on access to the contained classes. This package is a general collection of data classes and access objects. By contrast the SecurityManager and ProductCatalog are components that contain a well defined purpose, but may still be directly accessed. The AccountManager and OrderManager are subsystems and have a well defined interface for communication. The main difference between these three organizational elements is one of complexity management. Subsystems hide complex, changeable system elements, such as business rules around purchases. Components provide focused functionality, which limits the proliferation of dependencies. Packages allow logical grouping of related elements into hierarchies for better understanding of system interdependencies.

Interactions between Design Elements

System structural elements interact to provide system behavior. These interactions are what creates the need for dependencies, since a system that is totally disconnected is nothing more than a library of reusable processing routines. Once an initial structure is defined, the use case scenarios can then be used to derive the necessary communication and data transfer between system elements.

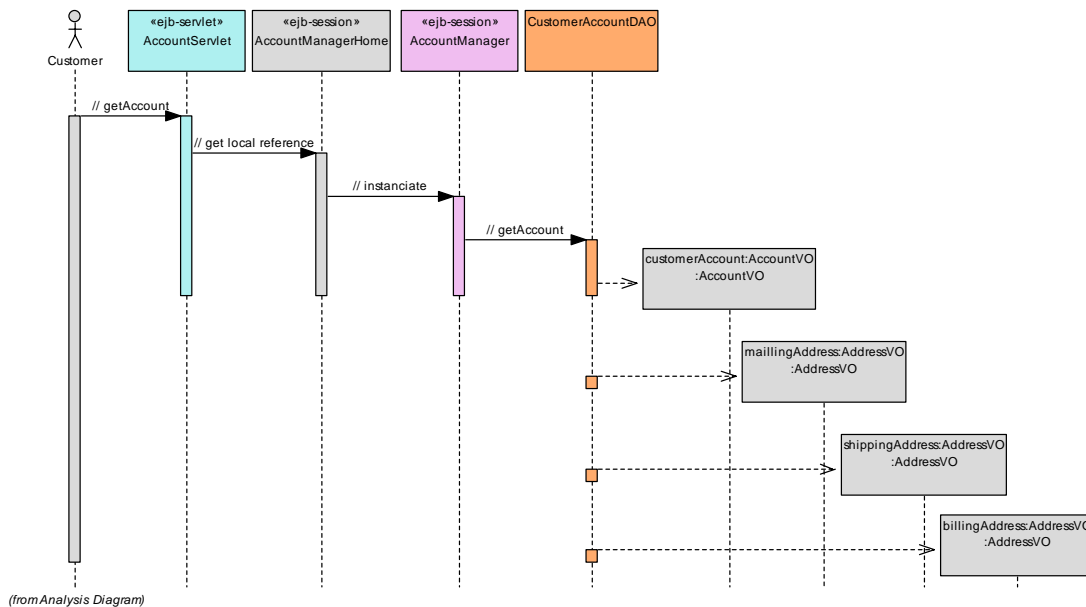


Figure 7. System Element Interactions

Figure 7 illustrates the first step for defining the interactions between system elements. In this example the use case specifies the ability for a Customer actor to access their account information, which includes three separate addresses. Since the designer has selected a J2EE servlet/session bean implementation, the first set of calls is between the

AccountServlet and the *AccountManagerHome* to get a local reference to the correct session bean from the J2EE container. The *AccountManager* then accesses the *CustomerAccountDAO* to retrieve the customer information (i.e. from a database). At this level of design the actual calls and parameters are not yet known, so the “//” is used to indicate where a specific object method call will be required. As more detail is added to the model (perhaps through prototyping or actual code), these calls will be fully defined.

Describe Data and System State(s)

The sequence diagram shown in Figure 7 illustrated the interaction between data access elements and the business processing logic of the system. However, it is often necessary to define the data elements in the design, especially when there is some state changes that are expected to occur. A well written use case should contain information on data and state transitions, but often it is up to the designer to infer this information. For example, consider a simple stock transaction (Figure 8). The stock will have descriptive detail, such as price, shares held, cost basis, etc. It will also have particular state information based on different business events. After the initial purchase the stock is Ordered, which transitions to Owned when the order is confirmed. The Owned state has several internal states of held, sell-order, buy-order, that describe how the stock may be manipulated by buy/sell events. After a sell order transaction is completed, the stock will be in the Sold state.

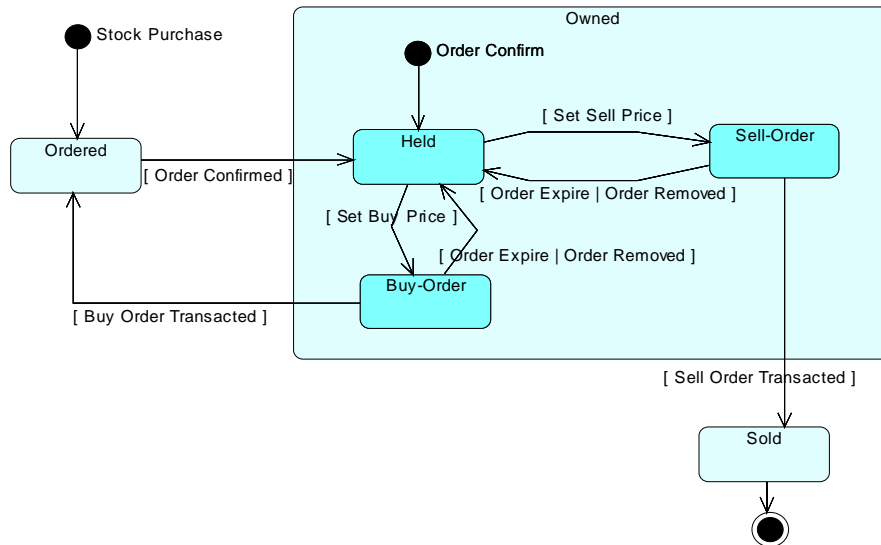


Figure 8. State Transition Diagram for Stock Investment

Detail Persistence Mechanisms

In addition to the definition of the data elements, attributes and state information, it is also the designer’s responsibility to define the data persistence mechanism. Most systems are created to use a relational database to store information, but recently the advance of service oriented architecture (SOA) has introduced network location access as

a source of information for system processing. Some common data access mechanisms are listed below:

- ◆ JDBC/ODBC (i.e. relational database)
- ◆ Network Socket (e.g. mainframe)
- ◆ Web Service
- ◆ File System
- ◆ File Transfer Protocol (FTP)
- ◆ Non-Volatile Memory (Flash Card)

While there are many ways to access and store information, it is usually beneficial to encapsulate these mechanisms into one or more “adapter” components that have well defined dependencies. This will reduce the likelihood that implementation details on data access, such as Structured Query Language (SQL) information, bleeding over from the data access elements to the data processing and display elements.

Detail Architectural Structure and Mechanisms

During the definition of structure and the organization of system elements, it is often necessary to select an overall architectural form for the system. This provides the framework for all of the system elements, and permits a higher level of abstraction. One of the most common strategies is the Layered Architecture Pattern, shown in Figure 9. In this pattern, all dependencies are from one layer to the layer below, with no dependencies to other layers. This provides tight control on dependencies, since no dependency is longer than on layer deep; this is known as Strict Layering. If a layer is permitted to access system components from additional levels besides the one directly below, then the form is called Relaxed Layering. In both forms of the pattern all dependencies are “downward” between layers or “sideways” within a layer. Note that Services are the exception to this rule; these frameworks and components can be accessed by any layer.

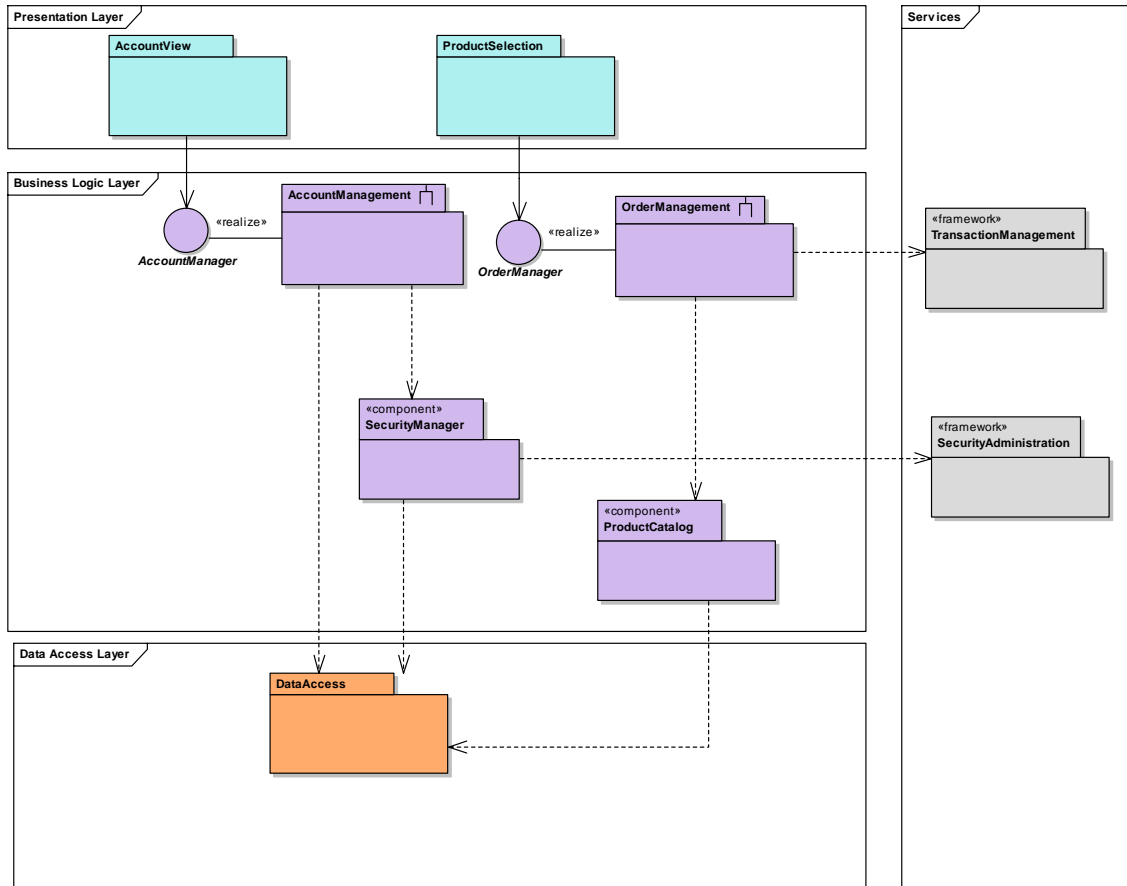


Figure 9. Layered Architecture Pattern

In addition to selecting an architectural form, this is also the point where common services and frameworks can be defined and added to the Design Model. Some common architectural mechanisms include:

- ◆ **Transactions**
- ◆ **Concurrency Management**
- ◆ **Logging**
- ◆ **Security**
- ◆ **Error Handling**
- ◆ **Communication Protocols**
- ◆ **Processing Distribution**