Software Architecture

Introduction

"Java" and "UML" are two of the current buzzwords in the software industry. And there is another one, more technically: "Software Architecture". For most software designers, "Software Architect" is one of the top job titles on the business card. So, what is "Software Architecture"? If you have a look to [1] or [2], you'll find quite a lot of different definitions. One of the first definitions I've seen was given in [3]. There, the software architecture "provides generic mechanisms and structures for managing data and control for the system as a whole". I think this is still a very good general definition of software architecture, but it doesn't explain the details of software architecture. So, I will take a different approach here.
For me, there are two main aspects of software architecture: (1) (the structure aspect) dividing the whole system into several distinct parts and defining the interactions between those parts, and (2) (the uniformity aspect) creating uniformity throughout the whole system. This second aspect is sometimes referred to as the "habitability" of a system: how easy it is to develop a sense of where to look for some specific detail in the system. While I believe that both (1) and (2) are equally important, most current discussion, papers and textbooks are about (1) only, and quite a lot of the definitions in [1] reflect only the structural aspect of software architecture. One of the reasons for this might be that it is much easier to provide guidelines and techniques for the much more "real" structure than for a somewhat "fuzzy" uniformity, and this is also the reason why I will only discuss the structure aspect in this article (but I will return to uniformity in another article).

A Sample System

As a running example, I will use a software system for a distributor. The main business processes for this system are to buy and to sell products. But if you have a closer look to the system, you will find quite a lot of different use cases for different users. The system should provide two different user interfaces for selling the products: a web interface for online ordering and a usual GUI client for the sales person taking orders at the phone. Both of these UIs use a powerful search machine to search for products based on various different specifications. The customer should be able to track the way of his/her order on the web, so you need an additional web interface for this. The stock clerks need a GUI to print packing lists and commit the shipping. And the product manager must be supported with sales figures and some lists that show those products that are currently low on stock. The product manager also has a GUI to enter new products to the system, alter product descriptions, etc. And finally, all orders must be reported to the accounting system for billing, general accounting statistics, etc. A possible use case diagram (in UML) is shown in Fig. 1.

Subsystems

Starting from the use case diagram, you can decide on the major subsystems. But what is a subsystem? One main aspect of a subsystem is separate development: in general, the different subsystems can be developed independently. To achieve this, you need clearly defined interfaces: the functional services a subsystem provides to or requires from other subsystems.
As it is always wise to separate the view (user interfaces) from the (business) model, you might decide to have one user interface subsystem for each of the different users and one subsystem for the model. As you also might use some existing library for the database access and something to connect to the company's accounting system, you might have the subsystems shown in Fig. 2.

Components and Distribution

Not only the functional side of the subsystem interfaces must be defined, but also the technicalities. So, you must decide as what type of physical entities you want to provide your subsystems: they could be normal (static or dynamic) class libraries. But you could also provide them as ActiveX controls or JavaBeans. Also, a subsystem could be a separate process, in which case you must decide on the type of communications between the processes.

Given the subsystems from Fig. 2, the central question regarding the physical packaging is about the "Business Objects" subsystem. In our example, the two web interfaces will probably run on a web server. As the web server will run outside of a firewall, but the critical business processes should only run inside of the firewall, the "Business Objects" subsystem will be a separate process on a different machine anyway. So, you probably want to keep the internal GUI clients separate as well. But if you have the processes on different machines, you need some mechanisms to communicate between them. While you could develop your own communication functions, you probably will use some library based on low-level protocols or better some object-oriented distributed framework, like CORBA, DCOM, or JAVA RMI.

Concurrency

As we have a multi-user system, you must think of the concurrency requirements of your system. Of course, it should not happen that two different customers are promised the last item in stock of a specific product. Also, as you have decided to use a central application server (the "Business Objects" subsystem), you must think again of the physical nature of this server. How do you handle different clients connecting simultaneously to the server? Is there a different server process spawned for each client, or is there a separate thread inside the same process for each client, or does the server processes one request after the other? For the chosen alternative, you must carefully select the (server-) internal communication mechanisms, identify the shared resources and decide on a policy to manage these.

Summary

Perhaps, this small example has shown you some of the tasks that are related to software architecture. One of the main tasks is to define the overall structure of a software system, i.e. the partitioning into several subsystems and the interactions between them. Part of this are the definitions about the distribution, communications and concurrency between the parts.
But another aspect is the uniformity among the parts, and I will come back to this aspect in another article. There, I will also come back to the very beginnings of my writings for Overload: the exception specifications and their consequences on the flexibility of a system.

References

[1] Software Engineering Institute, http://www.sei.cmu.edu/architecture/definitions.html
[2] World Wide Institute for Software Architecture, http://www.wwisa.org/
[3] Sally Shlaer, Stephen J. Mellor: Object Lifecycles, Yourdon Press, 1992, ISBN 0-13-629940-7

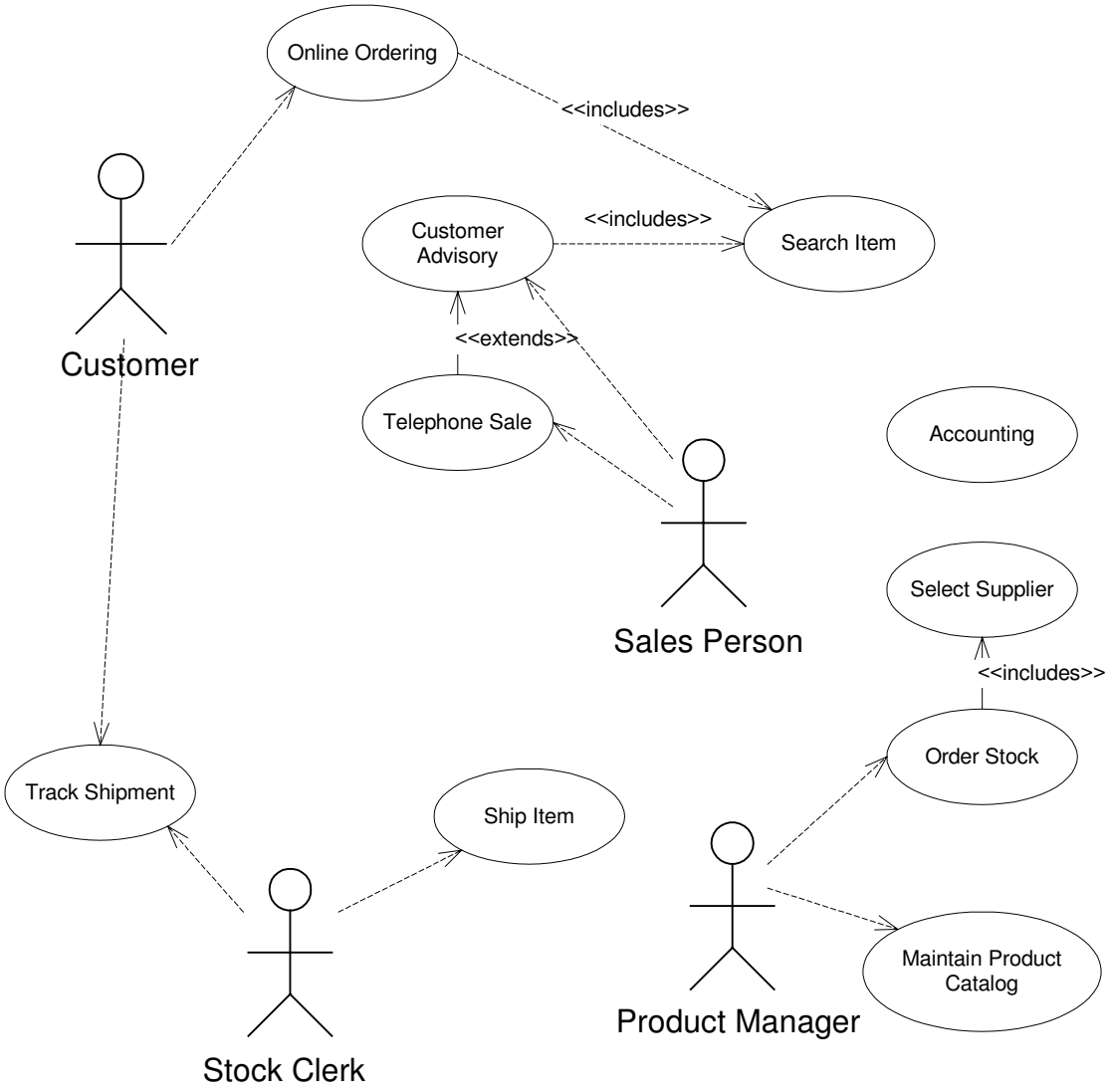

Figure 1: Use Cases

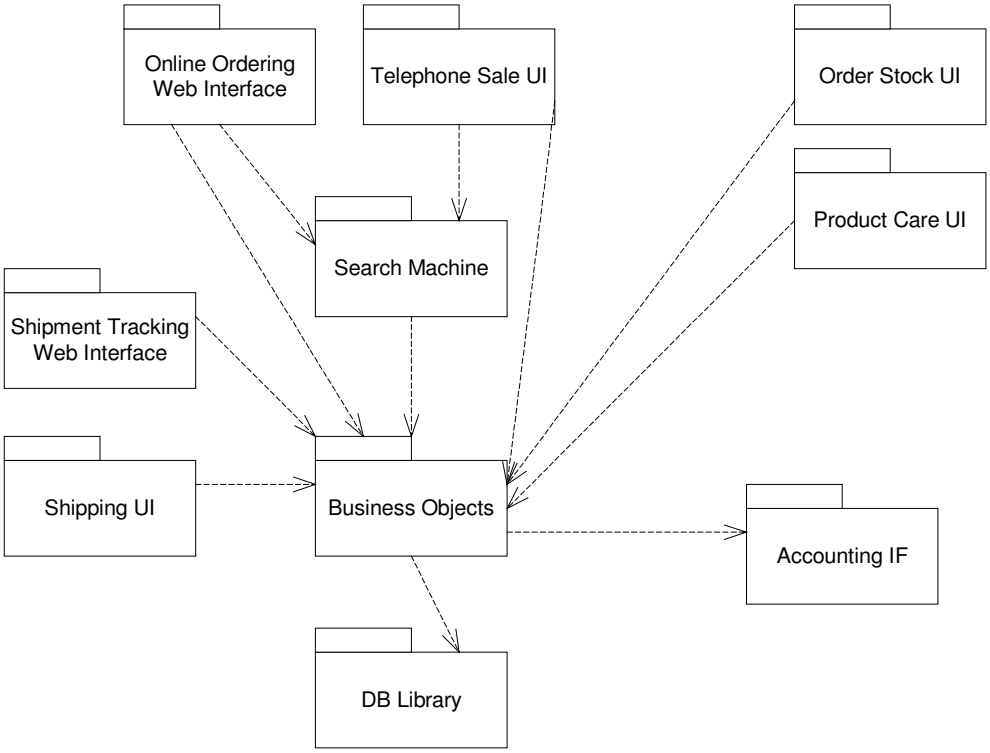

Figure 2: Subsystems