Chapter 2: Object Orientation: The new software paradigm

2.1 Object Technology

The ***object-oriented (OO) paradigm/standard*** is a development strategy based on the concept that systems should be built from a ***collection of reusable parts called objects***. The original motivation of the object oriented paradigm was that objects were meant to be abstractions of real-world concepts, such as students in a university, seminars that students attend, and transcripts that they receive. This was absolutely true of business objects; you also need user interface objects to enable your users to work with your system, process objects that implement logic that works with several business concepts, system objects that provide technical features such as security and messaging.

2.2The potential benefits of object orientation

Here are some of the benefits of the object-oriented approach:

Reduced Maintenance: The primary goal of object-oriented development is the assurance that the system will enjoy a longer life while having far smaller maintenance costs. Because most of the processes within the system are encapsulated, the behaviors may be reused and incorporated into new behaviors.

Real-World Modeling: Object-oriented system tends to model the real world in a more complete fashion than do traditional methods. Objects are organized into classes of objects, and objects are associated with behaviors. The model is based on objects, rather than on data and processing.

Improved Reliability and Flexibility: Object-oriented system promise to be far more reliable than traditional systems, primarily because new behaviors can be "built" from existing objects. Because objects can be dynamically called and accessed, new objects may be created at any time. The new objects may inherit data attributes from one, or many other objects. Behaviors may be inherited from super-classes, and novel behaviors may be added without effecting existing systems functions.

High Code Reusability: When a new object is created, it will automatically inherit the data attributes and characteristics of the class from which it was spawned. The new object will also inherit the data and behaviors from all super classes in which it participates.

In general object orientation has the following advantages:

- simplicity: software objects model real world objects, so the complexity is reduced and the program structure is very clear;
- modularity: each object forms a separate entity whose internal workings are decoupled from other parts of the system;
- Modifiability: it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods;
- extensibility: adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones;
- maintainability: objects can be maintained separately, making locating and fixing problems easier;
- Re-usability: objects can be reused in different programs.

2.3 The potential drawbacks of Object orientation

When new software development technologies are adopted, it is often the case that, during the early stages of adoption, they are misused, abused or create totally unrealistic expectations. Lack of understanding and unrealistic expectations of new technologies seem to be the common denominators for the delay of their proper use. These types of problems can be avoided or minimized by a clear view of the pitfalls/drawbacks that developers face when adopting new technologies.

Many of the pitfalls/drawbacks described below are common to any software development as they are significant in the context of Object Oriented System Analysis and Design (OOSAD). They may threaten to disturb OOSAD projects in spite of the benefits of OOSAD.

The following are potential pitfalls/drawbacks from the viewpoint of the various entities involved in the OOSAD process.

Conceptual pitfall

Most conceptual pitfalls have two things in common: confusion about what OOSAD is and what it entails. There are several reasons for this to occur:

First, project managers and developers may confuse form with substance. When structured development became popular, developers and managers thought that structured development simply meant to eliminate GOTO statements and increase the use of subroutines. These steps helped, but they were just two surface features of deeper and more significant principles. Similarly, some developers and managers today think that OOSAD simply means defining classes, objects, and methods.

Second, managers and developers *may not recognize all the implications of OOSAD*. Managers and developers often assume that using OOSAD will eliminate various development bottlenecks since promises are made that OOSAD will reduce the management of complexity and will provide architectural modularity. *This different approach to architecture, design and coding **may require,** or at least may work best with, different management and scheduling techniques.*

Third, managers may be tempted/attracted to abandon and neglect traditional design and software engineering processes. OOSAD is often adopted because of its promise of increased productivity and a shortened schedule, *so sufficient time is frequently not allowed to make sure that procedures are followed correctly*. This may result in missed deadlines, schedule slippage, and project failures.

Fourth, developers may get some education about OOSAD, but not enough. Reading articles on OOSAD, or even a book or two or taking a class at a local university may create a false sense of confidence and understanding.

OOSAD requires more discipline, management and training than classic software development does. In all four of the above areas, time spent at the beginning will save time later. This approach, however, has always been difficult to sell to upper management because they would like to have the product ready in the least amount of time and cost.

Education and experience are keys for the success of any OOSAD project. A company in which upper management, technical management, and developers all work together - using realistic schedules, with continuous education, and solid engineering techniques.

Analysis and Design Pitfalls

Developers often carry into OOSAD a bias toward the traditional system development life cycle (SDLC). Analysis and design pitfalls come from the struggle to manage complexity, while gaining a sense of how object technology works. Two possible situations are related to this pitfall. First, developers may take a SDLC approach to analysis and design and then attempt to implement it using object-oriented techniques. Second, developers may attempt to use an OOSAD approach, but then create classes with their hierarchies and connections which is more along the lines of traditional programs. In both cases, there may be confusion about the relationship between object classes that would result in poor solutions to the problems, loss of benefits of OOSAD, and disillusionment / lack of expectation with OOSAD.

The skill of object-oriented programming, much less object-oriented analysis and design, cannot be acquired overnight. Thus, before developers start a project, they must decide whether to use OOSAD. They have to ensure that OOSAD is being adopted for the right reasons and have a good understanding of the risks involved.

Environment, Language and Tool Pitfalls

Environments, languages, and tools constitute the most controversial area of OOSAD and are most subject to change as new technologies are developed. The environment comprises the operating systems and application environments in which a given project will run: Windows, OS/2, Unix and others. Languages include the various object-oriented languages used to implement object design: Smalltalk, C++, Java, and others. Tools are what developers use to create and test the application: editors, compilers, browsers, source code management systems, computer-aided software engineering (CASE) packages and others.

Selection of the wrong environment, language, or tool cans plague/outbreak the OOSAD project with constant problems or bugs in the development of applications. There can also be significant problems in development and deployment because the new system may not be compatible with the current environment. Selection of the target environment and language for OOSAD is perhaps the single greatest pitfall.

To prevent this pitfall, it is important that developers consider technology, compatibility and economic issues before selection of environment, languages and tools. Further, developers should validate their selection of tools and languages early. They should try out the complete set of tools that they intend to use for production development as early in the system development as possible.

Implementation Pitfalls

Like many other of the pitfalls mentioned previously, this is not unique to OOSAD. Many faults and failings of software development come from neglecting other software engineering activities such as comprehensive analysis, design, development scheduling and deliverables, and proper planning for system testing and installation/conversion.

To prevent intensification/growth of this problem in OOSAD, it is important that analysis and design be sufficiently completed. It is important to define design, implementation, documentation and coding standards for each subsystem and class before coding and implementation take place. This would seem obvious, but it is easily overlooked in the OOSAD environment.

Class and Object Pitfalls

There are many pitfalls in this category. Each one of the following can lead to poor hierarchy and class design, unpredictable behavior of objects, unnecessary complexity in the project, loss of OOD benefits, low rate of code reuse and product instability.

- Confusion of is-a, has-a, and is-implemented-using relationships;
- Confusion of interface and inheritance during implementation ;

- Use of inheritance to violate encapsulation;
- Use of multiple inheritance (MI) to invert the is-a relationship; and
- Use of Multiple inheritances in any circumstances.

These and similar misuses of inheritance seem to have short-term values, but they usually come back to haunt developers in the end.

To avoid these pitfalls, developers should set guidelines to create class hierarchies. For each class to be defined, use the descriptions to determine the relationships with existing classes. Further, set up the interface and implementation inheritance according to the class design. Also, set guidelines for how methods should be exported and inherited or overridden.

Reuse Pitfalls

OOSAD is more difficult and took longer, unless developers could use class libraries and other tools that provide for code reuse. So, it seems that reuse is a benefit of OOSAD, but that the payoff will be in the long run, not the short term.

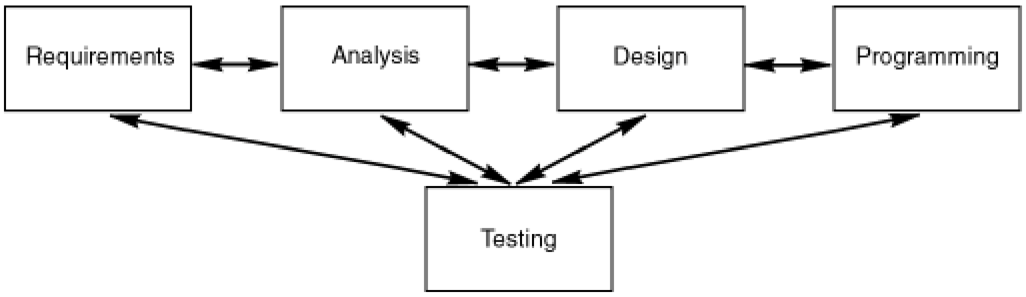
One of the primary goals of OOSAD has been reuse of code. *However, the explicit goal of reuse is harder and takes longer because reuse takes time and effort up front*, whereas OOSAD is promoted and adopted as a means to get software developed more quickly.

2.3 The object orientation software process

If you've read anything about the object-oriented paradigm, then you quickly discovered that it is a ***software development strategy*** based on the idea of building systems from reusable components called objects. The primary concept behind this paradigm is that instead of defining systems as two separate parts (data and functionality), you now define systems as a collection of interacting objects. Objects do things (that is, they have functionality) and they know things (that is, they have data). Developing software is hard, and because you typically apply object technology to solve complex problems.

In this part we will see a minimal process, depicted in Figure 1, ***for developing software using object-oriented techniques***. It is minimal because it focuses on the core activities of software development: requirements, analysis, design, programming, and testing. It does not cover other important topics such as project management, metrics, architecture, and system deployment. Nor does it include topics that would make it a true software process, such as the operation and support of your system once it is put into production. As you will see, the basics are complicated enough; there's no need to get ahead of ourselves yet.

Figure 1. A minimal software process



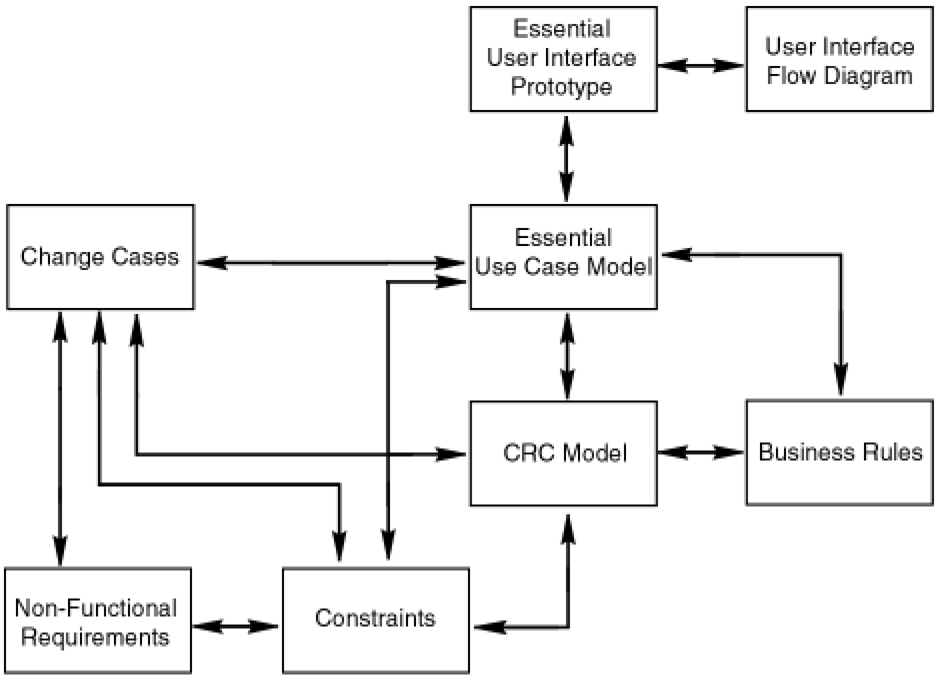
We can say software development is ***serial*** *on the large scale and* ***iterative*** *on the small scale, delivering incremental releases over time.* Taking this to heart, we will see the ***major object-oriented software development activities -- requirements gathering, analysis, design, programming, and testing*** -- in a serial manner, although you will discover almost immediately that each of these activities is actually quite iterative in practice.

Object-oriented requirements gathering

First, there is no such thing as "object-oriented requirements." ***Requirements should be technology independent***; therefore you should really be concerned only about requirements at this point in your development life cycle. Regardless of the hype surrounding "iterative development," the first step of any software development effort is to gather requirements -- you may not gather all of your requirements at once, but you should at least start with a few. You cannot successfully build a system if you're not clear on what it should do. The greatest problem during this stage is that many people do not want to invest the time to elicit requirements; instead, they want to jump right into programming. Moreover, your developers want to get into the "real work" of coding, and senior management wants to see some progress on the project, which usually means they want to see some code written. ***You need to communicate with all project stakeholders that this preliminary work is critical to the success of the project and that their efforts will pay off in the long run.***

Figure 2 depicts the relationships between the artifacts that you will potentially develop as part of your requirements engineering efforts. The boxes represent the artifacts and the arrows represent "drives" relationships.

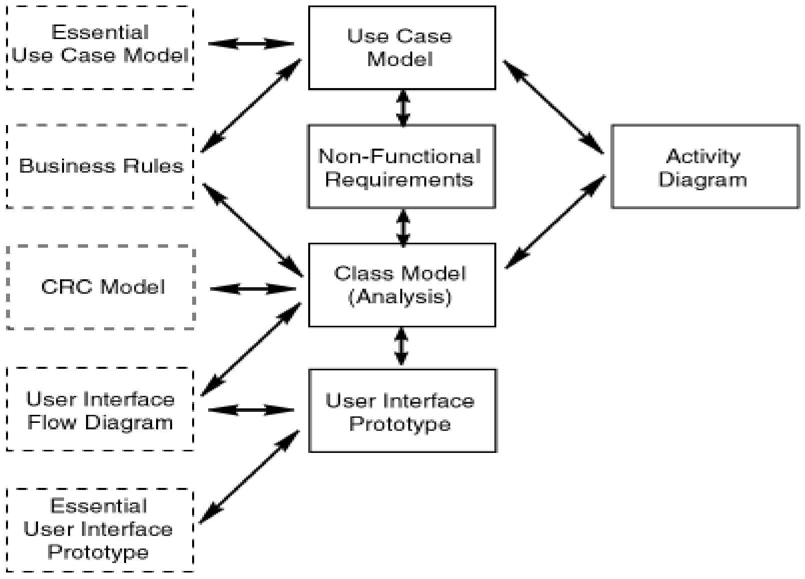
Figure 2. Overview of requirements artifacts and their relationships


Essential User Interface Prototype
User Interface Flow Diagram
Change Cases
Essential Use Case Model
CRC Model
Business Rules
Non-Functional Requirements
Constraints

Object-oriented analysis

The purpose of analysis is to understand what will be built. This is similar to requirements gathering, the purpose of which is to determine what your users would like to have built. The main difference is that the focus of requirements gathering is on understanding your users and their potential use of the system, whereas the focus of analysis shifts to understanding the system itself. *Figure 3 depicts the main artifacts of your analysis efforts and the relationships between them.* The solid boxes indicate major analysis artifacts, whereas the dashed boxes represent your major requirements artifacts. As before, the arrows represent "drives" relationships -- for example, you see that information contained in your CRC model drives or effects information in your class model, and vice versa. There are three important implications of Figure 3: first, analysis, too, is an iterative process. Second, taken together, requirements gathering and analysis are highly interrelated and iterative. As you will see, analysis and design are similarly interrelated and iterative. Third, the "essential" models, your essential use case model and your essential user interface prototype, evolve into corresponding analysis artifacts; respectively, your use case model and user interface prototype. Similarly, your Class Responsibility Collaborator (CRC) model evolves into your analysis class model.

Figure 3. Overview of analysis artifacts and their relationships

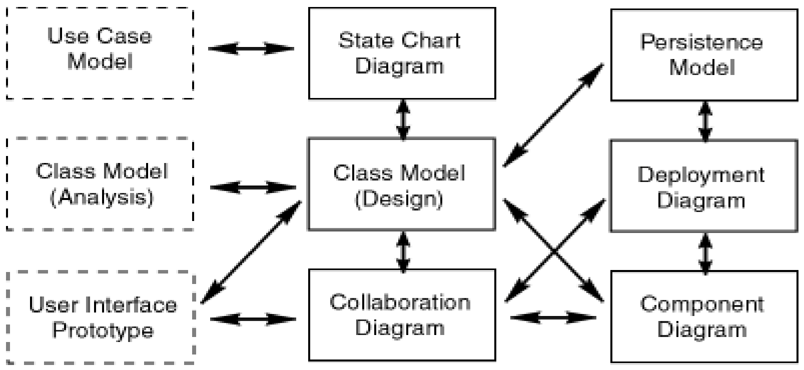


An important concept that needs to be clarified regarding Figure 3 and similar figures throughout this article is that every possible "drives" relationship is not shown. For example, it is very likely that as you are developing your use case model you will realize that you are missing a feature in your user interface, yet there is not a relationship between these two artifacts. From a purely academic point of view, when you realize that your use case model conflicts with your user interface model, you should first consider what the problem is, update your use case model appropriately, propagate the change to your essential use case model, then to your essential user interface model, then, finally, into your user interface model. Yes, you may in fact take this route. Just as likely, and probably more so, is that you will instead update both your use case model and user interface model together and then propagate the changes to the corresponding requirements artifacts. *This is an important aspect of iterative development* -- you do not necessarily work in a defined order; instead, your work reflects the relationships between the artifacts that you evolve over time.

Object-oriented design

The purpose of design is to determine how you are going to build your system -- information needed to drive the actual implementation of your system. ***This is different from analysis, which focuses on understanding what will be built.*** As you can see in Figure 4, your analysis artifacts, depicted as dashed boxes, drive the development of your design artifacts. As before, the arrows represent "drives" relationships -- information in your analysis (conceptual) class model drives information in your design class model, and vice versa. There are three important implications of Figure 4: first, like requirements and analysis, design too is an iterative process. Second, taken together, analysis and design are highly interrelated and iterative. As you will see in the next section, *design and programming are similarly interrelated and iterative*. Third, your analysis class model evolves into your design class model to reflect features of your implementation environment, design concepts such as layering, and the application of design patterns.

Figure 4. Overview of design artifacts and their relationships



There are several high-level issues that you must decide on at the beginning of the design process. First, do you intend to take a pure, object-oriented approach to design or a component-based approach? With a pure OO approach your software is built from a collection of classes, whereas with a component-based approach your software is built from a collection of components. Components, in turn, are built using other components or classes (it is possible to build components from non-object technologies).

A second major design decision is whether you will follow all or just a portion of a common business architecture. This architecture may be defined by your organization-specific business or domain architecture model, sometimes called an enterprise business model, or by a common business architecture promoted within your business community. For example, standard business models exist within the manufacturing, insurance, and banking industries. If you choose to follow common business architecture, your design models will need to reflect this decision, showing how you will apply your common business architecture in the implementation of your business classes.

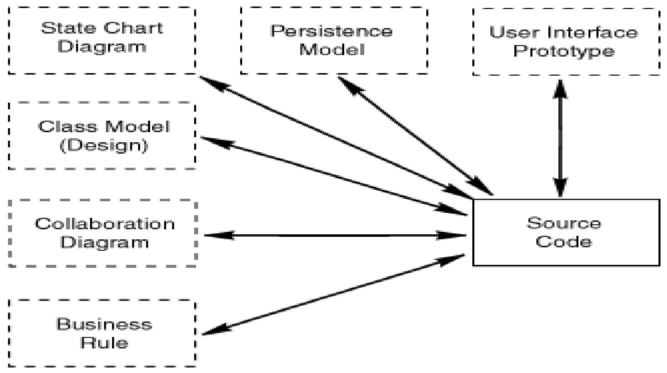
Third, you must decide whether you will take advantage of all or a portion of a common technical infrastructure. Will your system be built using your enterprise's technical infrastructure, perhaps comprised of a collection of components or frameworks? Enterprise JavaBeans (EJB) technology, CORBA, and the San Francisco Component Framework are examples of technical infrastructures that you may decide to base your system on. Perhaps one of the goals of your project is to produce reusable artifacts for future projects. If this is the case, then you want to seriously consider technical architectural modeling.

Fourth, you must decide which non-functional requirements and constraints your system will support. You refined these requirements during analysis and, hopefully, resolved any contradictions, but it's during design that you truly begin to take them into account in your models. These requirements will typically pertain to technical services; for example, it is common to have *non-functional requirements* describing security access rights as well as data sharing approaches. As you try to fulfill these requirements you may find that you are unable to implement them completely. Perhaps it will be too expensive to build your system to support sub-second response time, whereas a response time of several seconds proves to be affordable. Every system has design trade-offs.

Object-oriented programming

The purpose of object-oriented programming is to build your actual system -- to develop the code that fulfills your system's design. As you can see in Figure 5, your design artifacts, depicted as dashed boxes, drive the development of your source code. The most important implication is that design and programming are highly interrelated and iterative. Your programming efforts will immediately reveal weaknesses in your design that will have to be addressed. Perhaps the designers were unaware of specific features in the programming environment and, therefore, did not take advantage of them.

Figure 5. Design artifacts drive development of source code



What isn't so obvious is that you will focus on two types of source code: object-oriented code such as Java code or C++, and persistence mechanism code such as data definition language (DDL), data manipulation language (DML), stored procedures, and triggers. Your class models, state chart diagrams, user interface prototypes, business rules, and collaboration diagrams drive the development of your object-oriented code, whereas your persistence model drives the development of your persistence code.

Object-oriented testing

A fundamental rule of software engineering is that you should test as early as possible. Most mistakes are made early in the life of a project and the cost of fixing defects increases exponentially the later they are found.

Technical people are very good at things like design and coding -- that's what makes them technical people. On the other hand, technical people are often not as good at non-technical tasks such as gathering requirements and performing analysis -- perhaps another trait that makes them technical people. The end result is that developers have a tendency to make more errors during requirements definition and analysis than during design and coding. We will discuss this topic in detail in chapter 7.