

Softstar Research, Inc.

Methodologies and Practices – White Paper

Uses of Use Cases

By David M. Rubin

Revision: July 1998



Table of Contents

TABLE OF CONTENTS.....	2
ABSTRACT.....	3
WHY OO.....	4
HISTORY.....	7
DEFINITION OF USE CASES.....	10
USE CASE SEMANTICS.....	12
PURPOSE.....	12
CONTENTS.....	12
PLURALITY.....	13
STRUCTURE.....	13
USE CASE TEMPLATE.....	14
USE CASE DIAGRAMS.....	17
IDENTIFICATION OF USE CASES.....	18
USE CASE USES.....	20
TRAINING.....	20
TESTING.....	21
CLASS MODELS.....	21
CLASS DISCOVERY.....	21
Noun Extraction.....	22
Verb Extraction.....	22
Adverb Extraction.....	22
Adjective Extraction.....	22
OID's [OSD's] / SEQUENCE DIAGRAMS.....	22
Primary course.....	22
Alternate course.....	23
USE CASE TOP RULES.....	24
METHODOLOGY, MY \$0.02.....	27
A TYPICAL PROJECT APPROACH.....	28
A TYPICAL ITERATION APPROACH.....	29
A TYPICAL HIGH-LEVEL OO METHODOLOGY.....	30
AUTHOR.....	31
INDEX.....	32
REFERENCES.....	33





Abstract

Use Cases have become the starting point of many current Object Oriented (OO) development methodologies. They generally serve as both a foundation and entry point for the rest of the analysis and development process.

It's been my experience while consulting with new clients that three areas of confusion are prevalent regarding Use Cases being applied for the first time. The first area is the structure of Use Cases themselves, including the format and a definition of what a Use Case is. Second is the content of Use Cases; specifically the content of the different sections that make up the Use Case. Third is the context of Use Cases. In context I refer to how Use Cases fit within the (an) OO process or development effort.



To phrase it another way; **why** are we doing Use Cases, **what** do they look like, and **how** might they be used when there completed.

During these consultations, I recommend many articles, references, and books to my clients as they embark on their first (and beyond) OO projects. Included in these recommendations are numerous compositions on OO methodology and process [RUP], OO notation [UML], and OO transition management. I particularly like articles describing real world experience and best practices from those in the industry that live it.

This paper is an overview of some of this documentation, including some techniques that have helped introduce teams to the OO world and the use of Use Cases.

It begins with a perspective on why OO has become so powerful and effective. Then continues with a brief history of Use Cases adapted from an article by Edward V. Berard of 'The Object Agency, Inc'. It continues with some ideas on Use Case semantics, which were first introduced by Alistair Cockburn in his article 'Structuring Use Cases with Goals'.

Finally I try to introduce the three areas identified above, what, why, and how regarding the Use of Use Cases.





Why OO

While first starting out in software development, structured technologies were all the rage. My years in the Air Force taught me the best of these, including my personal favorite 'TDSP', a.k.a. 'Top down Structured Programming' and/or 'Top Down Structured Design'.

At the time, these methodologies advanced both the industry as a whole and the software development process in general. Some of the more common methods for achieving a 'good design' were:

- ◆ Top-down design and construction,
- ◆ Divide and Conquer approach's,
- ◆ Limited control structures,
- ◆ Limited scope of data structures,
- ◆ Isolation of procedures, and
- ◆ Building around a function.

To begin to understand the differences between a structured and an object approach, consider the way different programmers might describe a car:

<i>Programmer</i>	<i>Car Description</i>
Structured Analysis	A little gas from the fuel tank goes into the fuel line, which feeds the gas into the engine. Then the engine converts the fuel into energy, which causes the engine to turn a small amount. Then the transmission attached to the engine turns a little bit. Then the driveshaft attached to the transmission turns a little bit. Then the differential attached to the driveshaft turns a little bit. Then the wheels attached to the differential turns a little bit. Then the car moves a little bit. These actions are repeated, over and over in a loop until the driver turns off the ignition.
Object Oriented	A Driver presses the gas pedal, which makes the car move. The engine stops when the driver turns off the ignition.

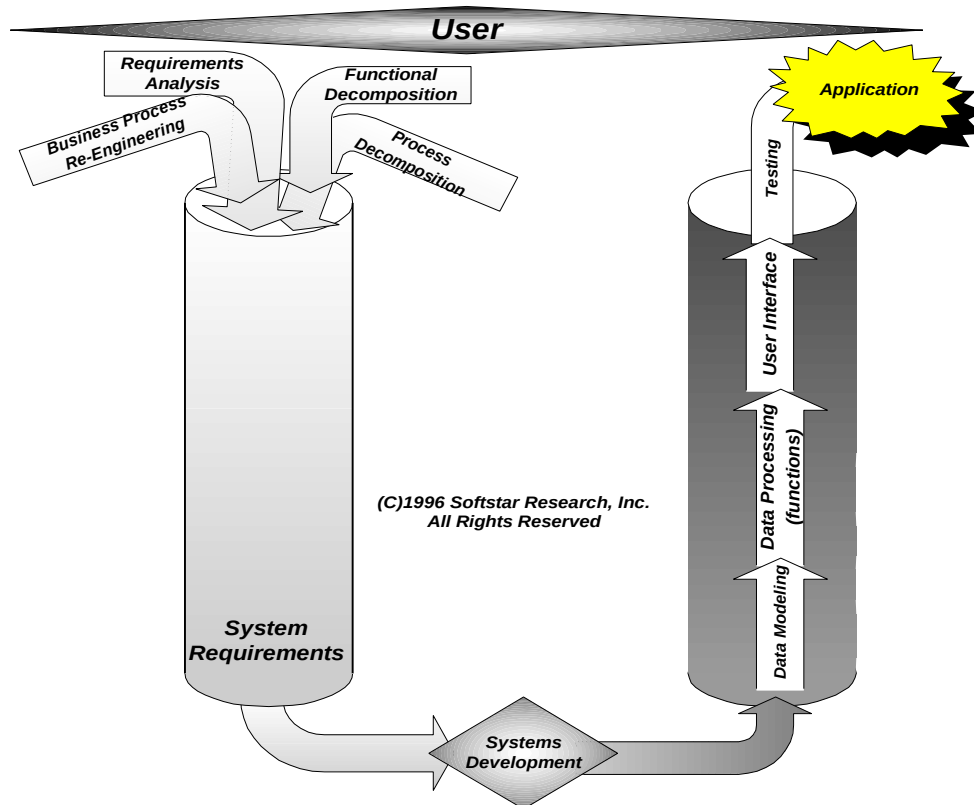
A couple of interesting things are reflected above. First, structured systems were always defined using notation and representation from the IS perspective. Requirements of a system by their very nature were technical in definitions. Second, the styles of the systems were reflected back to their users from this same technical perspective. Even back then (era 1970-1980), the concept of Use Cases could have



been applied effectively. Edward V. Berard¹ points out that Use Cases aren't necessarily an OO discovery, they were just adopted by the OO industry because they work.

Use Cases go to the heart of what I consider one of the fundamental problems with developing software, that of delivering a solution to a customer (or user), that the customer (or user) wants. In the past (as well as now), one way of looking at systems development is by using the '**Two Stovepipes metaphor**' to model the Software development process. (Note: Someone once said there are only two types of individuals in the world, those that can split and categorize everything into two, and those that can't).

The first stovepipe we'll call the 'process analysis' and 'business case' column, while the second stovepipe we'll call the 'delivery of systems' column.



In the first stovepipe, we do our '*identification of a problem*', which turns out to be the starting point for doing the business case. From there, we move into very *analytical* and '*requirements gathering*' type activities, which aren't necessarily the same as the requirements gathering for developing a system. These activities include things like process decomposition, functional decomposition, business flow diagrams, business event diagrams, workflow swim lanes, BPR activities, etc.

As we work down this stovepipe towards the end, we come out (end up) with a good understanding of the business itself. We also discover why there might be an





opportunity to develop a system. All through this first stovepipe, we constantly kept the **user** involved. In fact, not only were they involved, most of the activities accomplished here were driven by the users and the business.

Next come the second stovepipe, that of systems development. A common technique in the past was to start at the bottom of this stovepipe and work our way to the top. By starting at the bottom, this usually involved significant modeling from an attribute/data perspective. Using the output of the analysis stovepipe and feeding that into data flow diagrams, entity relation diagrams, IE modeling constructs, etc. Eventually we identified the specific data in the system, and the relationships between that data, which helped create our first cut-data, entity and relational models.

On top of these 'data and entity model' diagrams, we used other diagrams to figure out how we were going to create, modify, move, and generally manipulate the data. These diagrams helped describe what type of processing we need to do on that data (also represented as data flow diagrams, entity-relations diagrams, function point analysis, state diagrams, etc).

From there we continued to work our way up the stovepipe, until we're finally left with the need to throw a user interface on top of the work already done. The user interface usually drove the processing already defined, which drove the coding already done, which manipulated the data already defined. At the end of all this, we threw the entire package over the wall, back to the original users.

Here's where things really broke down (actually, they broke before this, we just didn't know till now). The users ultimately resisted the system. It turns out not to be very close to what they wanted, and how they wanted it. Lot's of finger pointing usually occurs right about this stage. Interestingly, there are a couple of 'immediately identifiable problems' that I believe contributed to the users dissatisfaction.

The first identifiable problem is the way in which the system was defined. The IS departments inherently interprets [filters] the user requirements through their own perceptions and experiences. These perceptions are for the most part very technical in nature, which usually drives the development of systems from the IS perspective, and not from the user perspective.

Second, the business is continually moving forward while the actual system is being developed. This inevitably changes the business requirements that the system must satisfy. If problem one is not the cause (misinterpreted requirements), then the fact that the actual system was built to those requirements can be a problem. It's a difficult no-win situation.

What's needed is a way to develop a system that constantly keeps the users involved in the definition, while at the same time being flexible enough to accommodate changes during the development of the actual system itself.

It is this need that has helped popularize the OO process, which I believe starts with Use Cases and continues with an incremental/iterative development model.

In this updated 'stovepipe' model, the system development phase again starts with the users, with the use of 'Use Cases', and continually keeps the user in the loop throughout the entire development process.



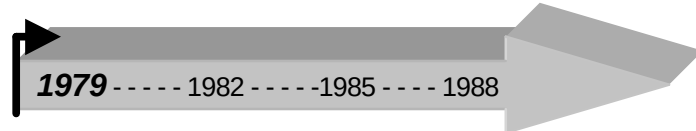


History

The following history is a paraphrased section of an article written by Edward V. Berard of The Object Agency, Inc.² Mr. Berard's introduction is among the best historical overview's of Use Cases that I've read.

In June of 1979, the Ada programming language became a reality³. The U.S. Department of

Defense set up Ada training classes at West Point, the Naval Post Graduate School, Georgia Institute of Technology, the National Physical Laboratory (U.K.), and the U.S. Air Force Academy in Colorado Springs, Colorado.



At the Air Force Academy, the task of designing a weeklong Ada training course fell to Major Dick Bolz and Captain Grady Booch. It was impressed upon Booch and Bolz that any Ada training course must emphasize software engineering, not merely the syntax and semantics of the language.

This point was further reinforced at an ACM symposium on Ada in late 1980⁴. Booch realized that it would be difficult merely to teach an Ada "syntax course" in a week's time. Therefore, he sought out some simple mechanism for incorporating software engineering into the training course.

Booch found the work of Russell J. Abbott⁵ to be particularly appealing. (Abbott's work was later revised and published⁶.) Abbott's approach was first to write a paragraph describing a solution to the given problem. Once the paragraph was written, it could then be examined and parsed into individual elements. The nouns, noun phrases, and pronouns would suggest candidate Ada packages, and the verbs in the paragraph would suggest candidate functions and procedures to be contained within those Ada packages.

Booch liked Abbott's approach, and combined it, along with influences from Smalltalk⁷, object-oriented computer hardware^{8 9 10}, rigor in the software development process¹¹, and a reverence for software engineering in general¹², into a process he called "**object-oriented design**." The first versions of Booch's object-oriented design process and notations can be found in Bolz and Booch, 1981¹³, Booch 1981¹⁴, Booch, 1982a¹⁵, and Booch, 1982b¹⁶.

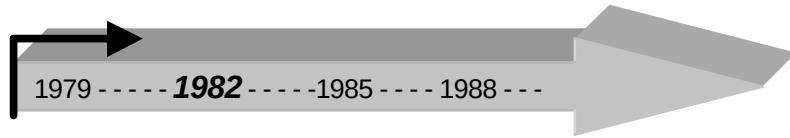
These early efforts by Booch were important because of his attempt to view object-orientation not merely in terms of a relatively informal coding practice, but rather as a (at least partial) 'life-cycle' process. Object-oriented programming, ordinarily, was viewed as similar to structured programming¹⁷, i.e., a process that focused primarily on coding discipline. To be sure, object-oriented programming required an informal, almost entirely intuitive, means of conceptualizing the problem, but there was little in the way of formal guidance provided.





Through his descriptions of the object-oriented design process and his diagramming techniques, Booch was striving for something that was closer in form to structured design¹⁸ than to a largely informal coding strategy.

Booch's 1980-1981 concept of object-oriented design involved the beginnings of his now well known "Booch Diagrams," and a simply defined process¹⁹, i.e.:



1. Define the problem.
2. Develop an informal strategy for the abstract world.
3. Formalize the strategy.
 - a. Define the objects and their attributes.
 - b. Define the operations on the objects.
 - c. Define the interfaces.
 - d. Implement the operations."

It is the second step in Booch's process (i.e., the development of an informal strategy) that involves what some today would call a "Use Case". The "informal strategy," as Booch originally described it, was "in the form of a paragraph" describing a solution to the defined problem.

Of course, compared with what Jacobson was to describe years later, there was little in the way of a well-defined process for creating a set of well-defined Use Cases.

The Object-Oriented Design Handbook for Ada Software²⁰ was one attempt to define a systematic and repeatable process for the creation of both individual and sets of "informal strategies."

The purpose of the "informal strategy" was twofold:

1. The informal strategy provided a means of identifying candidate objects and their attributes; e.g., nouns, pronouns, and noun phrases were used to suggest candidate objects.
2. Just as importantly, the informal strategy described the interactions and interrelationships among the objects that would effect a solution, i.e., the informal strategy had to solve the defined problem.

In the early 1980s, there was much resistance to Booch's object-oriented design approach. Software practitioners — never known for their writing and documentation skills {even to this day} — complained about the informality of the paragraph format. Science (Computer Science) after all was very much disciplined and formal by its very nature.

Many suggested more rigor could be accomplished with graphics (e.g., data flow diagrams and state transition diagrams a la [Ward and Mellor, 1985]²¹), or formal (mathematical) techniques like VDM^{22 23}.

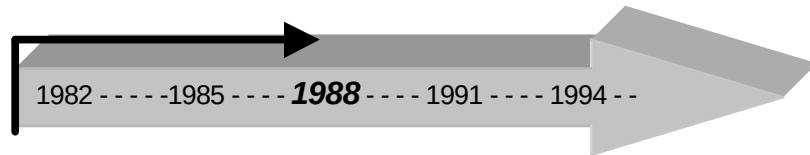


Booch himself, more than once, considered ways of integrating graphical techniques into the strategy defining process, e.g., the use of data-flow diagrams²⁴. Today, there are increasing numbers of formal techniques for expressing object-oriented designs (e.g., "object-oriented Z"²⁵). However, there is also a decided shift towards relatively simple ways of expressing object-oriented analysis and design. It is the (seeming) informality of Use Cases, which makes them attractive to many object-oriented technology users.

The fact that this informality also serves to bridge the systems development gap with the users is an obvious benefit {perhaps one of the *most important* benefits}. Use Cases are being incorporated several object-oriented methodologies, e.g., Fusion²⁶, Wisdom [IBM, 1996], Visual Modeling Technique (Walter Fang of IBM), etc. In addition, as Use Cases become part of the UML Unified Modeling Language [Booch, Rumbaugh, Jacobson, 1996], in a truly real sense, Booch has come full circle.

By 1986, there was already quite a few object-oriented design approaches from which to choose. At the 1986 Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) conference²⁷, Ivar Jacobson presented his ideas on the design of large, real-time systems²⁸. In this presentation, Jacobson discussed such things as "blocks," "services," "objects," "signals," and "messages," along with references to a formal specification language, i.e., FDL.

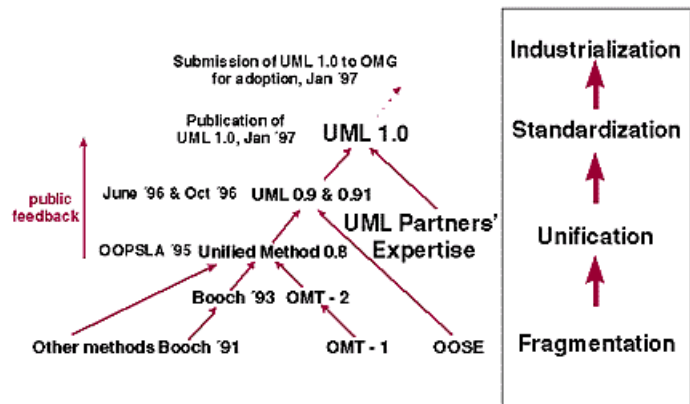
It was not until the 1987 OOPSLA conference²⁹, however,



that many in the object-oriented community were introduced to both "ObjectOry" and "Use Cases." In his 1987 presentation³⁰, Jacobson described ObjectOry as a technique, and Use Cases as a tool used within that technique. The "use" in Use Cases was clearly intended to be from the view of a "user of the system".

"The system is described in ObjectOry as a black box by describing a number of aspects of the system. These different aspects, each corresponding to a behaviorally related sequence are called Use Cases. "The basis of ObjectOry is that it shall be designed for its users. We want to build the system for them. In order to safeguard that the users really get the system they want and need, we want to structure the system's total behavior in aspects, where each aspect corresponds to what we can call a Use Case."

In 1996 the collaboration between Jim Rumbaugh and Grady Booch on what was then the Unified Method; transforms into the Unified Modeling Language (UML) and officially includes Ivar Jacobson 'Use Cases'.





Definition of Use Cases

The classic definition of Use Cases comes from Ivar Jacobson's '*Object-Oriented Software Engineering*' (OOSE)³¹ which states: **"A Use Case is a sequence of transactions in a system, whose task is to yield a measurable value to an individual actor of the system."**



To define Use Cases from a different perspective, I think back to a technique I learned many years ago regarding the design of software systems. The technique was to *"write the users manual"*, before designing the actual program. With that technique in mind, Use Cases can be thought of as a technique for writing the users manual of a system, before actually designing the system itself.

To further understand Use Cases, we will attempt to decompose the classical definition defined above in a somewhat formal notation:

- ◆ A **Use Case** is "a sequence of transactions through the system, that is, an instance"³². Using the concept of a class as the set of all items that share a collection of similar characteristics, it is suggested that many similar courses of events be grouped into a "use-case class." (Note that this definition, i.e., a class is a *set* of instances, is not the same definition of class that is used in a Smalltalk, C++, or Eiffel context.)
- ◆ An **actor** is "a role that someone or something in the environment can play in relation to the business". Alternatively, Jacobson³³ defines an actor as representing "everything that needs to exchange information with the system," and Christerson and Jonsson³⁴ define actors as "everything that interacts with the system."

An "individual actor" (sometimes referred to as a "user") is defined to be an instance of class actor. Further, the same person referred to as a 'user' can assume more than one role.

- ◆ "**Transactions** in a system" implies that the system will make available to its actors a set of capabilities that will both allow the actors to communicate with the system and to accomplish some meaningful work (i.e., work that results in some meaningful value).

Transactions (plural) also imply that this may be in the form of one, or more than one step.

- ◆ "A **measurable value**" implies that the performance of the task (or transactions) has some visible, quantifiable, and/or qualifiable impact on those things, which lie outside of the system, and in particular, the actor who initiated the task.
- ◆ A **transaction** is defined as "an atomic set of activities that are performed either fully or not at all. It is invoked by a stimulus from an actor to the system or by a point in time being reached in the system. A transaction consists of actions, decisions and transmission of stimuli to the invoking actor or to some other actor(s).





Another important aspect of Use Cases is that: “**The set of all Use Case descriptions specifies the complete functionality of the system.**” .³⁵ This implies that all of the Use Cases taken together 'is' the system.

Jacobson goes on to say that, any Use Case may be augmented with (stated) alternative courses of events. Further, any Use Case may include other Use Cases, which implied the Use Case being defined '**extends**' the included Use Case.





Use Case Semantics

Use Case semantics in this reference refers to the assemblance of the Use Case itself. On smaller projects, consistent semantics is important because of the diversity of the Use Case audience. This diversity often leads to misinterpretation. On larger projects, Use Case semantics becomes a necessity, especially with multiple facilitation's extracting multiple Use Cases.

Alistair Cockburn in his article 'Structuring Use Cases with Goals' identified he encountered 18 different definitions of Use Cases. He further classified these definitions along four dimensions; **purpose**, **contents**, **plurality**, and **structure**. Within each dimension, he has identified the specific values the dimension could take on. Mixing just the values of each dimension alone yields over 24 different combinations of Use Case semantics.

I found Mr. Cockburn's work important enough that understanding the complexities of Use Case semantics would be difficult without it. He does not recommend any one particular combination over another, just that the combination be identified and articulated to the entire project team.

Purpose

The purpose dimension is divided along two vectors, stories and requirements. The first vector 'stories' means that Use Cases could be structured to gather stories of how the system will be used. I've found that these stories often are used (and useful) as a precursor to a process decomposition activity (or functional decomposition activity). The other vector of purpose is requirements. By requirements, we mean fleshing out the details of the system being designed from the 'actors' perspective.

Values

- ◆ Stories
- ◆ Requirements

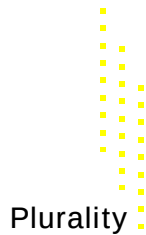
Contents

The contents dimension is divided among three vectors, contradicting, consistent prose, and formal content. The contradicting dimension defines whether the Use Case is consistent, or whether the Use Case is self-contradicting. If the Use Case is consistent, then is it formatted in plain prose or is it structured using a formal notation.

Values

- ◆ Contradicting
- ◆ Consistent Prose
- ◆ Formal Content





The plurality dimension addresses the multiplicity of content of the Use Case. Does the Use Case have a one-to-one relationship with a scenario, or does a Use Case contain more than one Use Case (or scenario).

Values

- ◆ One
- ◆ Multiple

Structure

The structure dimension addresses multiple Use Cases taken together (i.e. a collection of Use Cases). When Use Cases are associated with other Use Cases, do they have a formal structure, an informal structure, or do they form an unstructured collection.

Values

- ◆ Unstructured
- ◆ Semi-formal
- ◆ Formal Notation





Use Case Template

The following is a recommendation for a starting point in defining a Use Case template. This template follows the semantics of;

- ◆ Requirements,
- ◆ Consistent prose,
- ◆ Multiple, and
- ◆ Semi-formal.

Remember that the template only serves as a style guide to promote consistency among multiple people and multiple Use Cases. It is in **no way** the all-encompassing, all defining structure of the Use Case. Like everything else in any OO 'life-cycle', the template itself is iterative in nature and will change as time goes on based on an iterative feedback loop.

The template example below is structured into two columns. Column 1 contains the title of the template section, column 2 contains the template section description. Remember when describing Use Cases to always try and keep in mind the distinction between the '**user's**' problem and context, and the systems implementation domain.



In other words, **keep implementation details out of the Use Case**. This doesn't mean ignore implementation details that are discovered while doing Use Cases, it just mean that these details should be **documented separately**. I often suggest keeping a separate '*implementation notes*' document that has references back to a Use Case where an implementation detail was discovered.

Title	A one-sentence title of the Use Case being described, usually no more than a few words.
Summary	The summary is a brief synopsis of the Use Case, usually a shorter version of the description described below. The summary is often divided into three points, and can be documented as three separate bullet items. The three points are the trigger, the action, and the result. These points can be stated using the following template for sentence structure. 1. This Use Case begins when ... 2. This Use Case does ... 3. This Use Case concludes (or ends) when ...
Description	A short, well-written paragraph that defines the nature of the 'Use Case' in prose form. Sentences within the description generally contain 4 parts and can be of the following form: 1. Time Sequence 2. Actor 3. Action





	4. Constraint(s) For example, "At the end of the month, the accounts receivable clerk will mail advises to all customers whose balance is over a certain amount". The previous sentence contains the following elements as mapped to the list above: 1. Time Sequence: 'end of the month' 2. Actor: 'accounts receivable clerk' 3. Action: will mail advises 4. Constraint(s): 'balance is over a certain period'.
Use Case Goal	Statement of the actors business goal or objective met by this Use Case. If the actor is an actual business user, the goal is based on his or her own words. A good format is, "I want to be able to...". In the example above, the goal might be "I want to be able to send accounts receivable notices". Often times the 'Use Case Goal' is duplicated in the part of the Use Case description, and a separate section is not desired.
Actor(s)	A list of one or more actors participating in this Use Case.
Assumptions / constraints	A statement or statements that affect or modify the systems behavior or operation of the Use Case.
Basic course of action	An ordered list of the steps taken by the actor or actors that constitutes the Use Case. This is the primary sequence of steps usually taken to satisfy the Use Case. Actions are generally described from the actors perspective, not the analysts perspective. Try to avoid any details that are implicit implementation considerations. Each step in the basic course should be numbered, so that the steps can be referenced easily elsewhere.
Alternate course of actions	Any alternates set of steps that are exceptions to the basic course. The first sequential step in an alternative course should reference back to a step (identified by number) in the basic course. Alternate course(s) can also reference how a basic course is reentered after corrective action is taken, or they can end without reentering the basic course. Alternate courses of action can also be a mechanism to reduce the total number of Use Cases defining a system.
Questions	Any unresolved points needing clarifications that are uncovered (surface) while writing of the Use Case.





<i>Business Owner</i>	Administration data. The business owner (user) who is responsible for the Use Case.
<i>Author</i>	Administration data. Use Case author's name; usually the domain or business analyst.
<i>Last Updated</i>	Administration data. Date the Use Case was last updated and by whom.

Again, the template outlined above is not all encompassing. It is merely an example of a template that can be used as a foundation. Depending on the system being developed, numerous other sections that can be added.

Example may include Security, Deliverable Project Phase, Priority, Complexity, Etc.

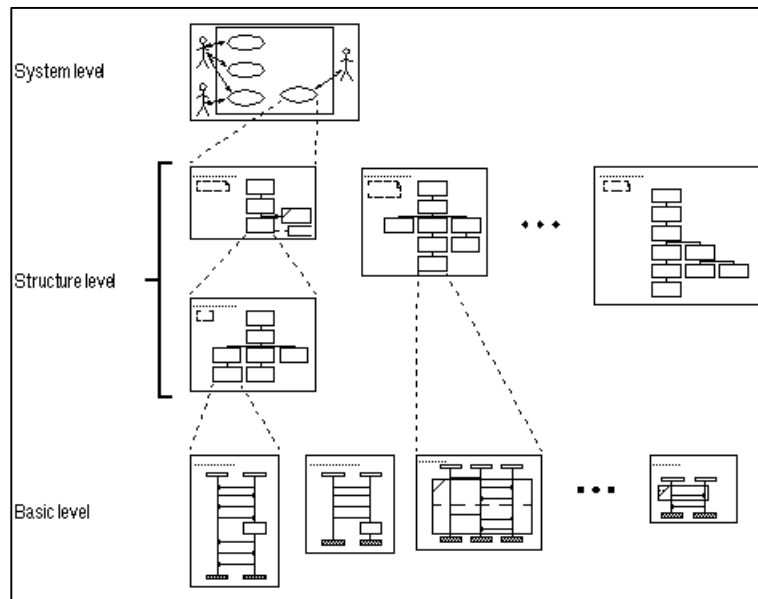


Use Case Diagrams

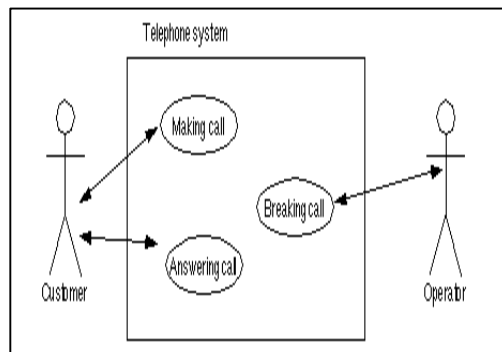
Use Case diagrams are used to frame the scope of the system that the individual Use Cases describe. They also visually describe one or more Use Cases, which with a particular actor interacts.

Use Case Diagrams usually comprise two different levels

of information, the Top Level Use Case Diagram (or System Use Case), and the Low Level Use Case Diagram (or Actor Use Cases diagram).



The Top-Level Use Case diagram can be thought of as a “*level 0-context diagram*”. Its primary purpose is to understand the context of the entire system with regard to the systems external interfaces.



The low-level Use Case diagram is used to depict which Use Cases within the system the external actors actually interact with.

These diagrams also have value when designing (or laying out) the user interface of the system. Understanding the groupings of functionality that individual actor(s) interact with will make for a much more cohesive and organized system.





Identification of Use Cases

The first rule of Use Cases, **don't** do them without the users of the system. With that said, here are some techniques I've used when trying to brainstorm (identify) candidate Use Cases. (For further techniques, see the section of 'Top Ten Rules of Use Cases'.)

Generally, it's a good idea to start with the Actors. Actors are generally easily identified by asking questions like;

- What user groups will be using the system?
- Who within those user groups will be using the system?
- Are there any existing systems today that must be considered?
- Why are we even discussing a system in the first place?

Once some of the Actors have been identified (of course using an Actor template form), we begin to identify the Use Cases themselves. Always remember that the initial list of Actors and Use Cases are just that, *'the initial list'*.



It may be useful to think of the entry point for a Use Case as a trigger or an event. That is to say, *something* must happen to invoke the system to do something else; that should conclude in some measurable result. Invocation of that *something* is generally triggered by an actor. This actor then becomes the actor of the Use Case.

To further discuss triggers, traditional 'Process Decomposition' usually identifies three types of triggers or events that force a process. These three triggers are Internal, External, and Time.

- ◆ **Internal Trigger.** An internal trigger is something that happens internally within an organization. Usually this implies things caused by employees or internal systems. For example, an internal trigger could be a legacy computer system requesting data from the OO system. Alternatively, employees submitting an expense report to the accounting department.
- ◆ **External Trigger.** An external trigger is something that happens externally to an organization. This often implies things that the internal organization has no direct control over when it happens. For example, a customer calls a customer service representative.
- ◆ **Time Trigger.** A time trigger is something that happens at a pre-determined time. Usually a time trigger has some sort of regular interval for triggering. For example, every night at 11:00 a backup process starts. On the other hand, every other Wednesday the payroll module of the accounting system activates to ensure payroll is met that Friday.





Identification of these triggers (from the Process Decomposition world) can be equally useful to formulate the beginning of Use Cases.

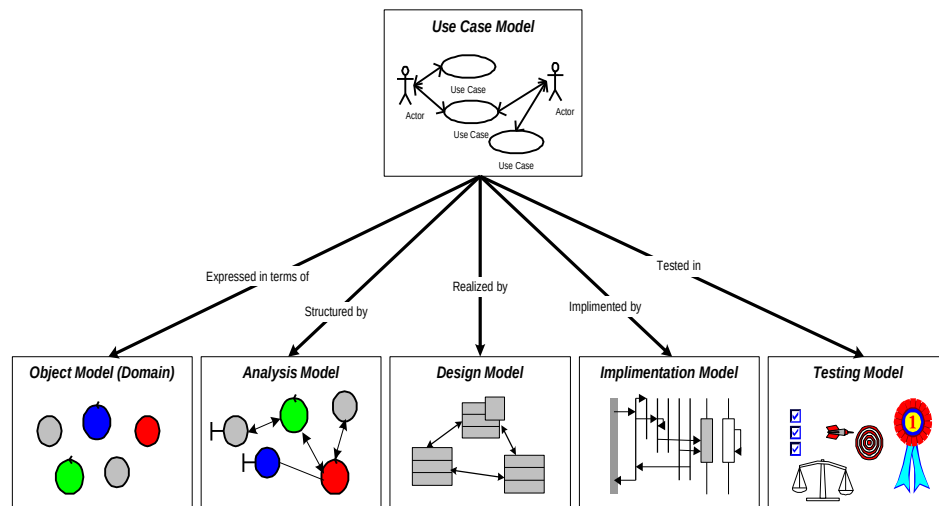
Other techniques to identify Use Cases can come from the legacy 'Structured Methodology' worlds. Often I use the divide and conquer method to start identifying pieces of functionality from my users when first documenting Use Cases.





Use Case Uses

The third area of confusion I've often run into is not knowing how Use Cases will [and can] be used throughout the development 'life-cycle'. While there are many more numerous places and examples, I've tried to outline some of the more common that I see time and time again on multiple projects.



By having an understanding of how something will be used, it inherently leads to the better creation of such a thing [this is the whole point of Use Cases them selves in system development].

Think of this next section as a kind of 'Use Case for Use Cases'.

Training

By combining the Use Cases defined for a system with the user interface(s) [defined] for the system (or user interface prototype), the initial user training documentation can begin.

Common sense says that if we know the following:

1. How the actors [users] want to interact with the system (primary course, alternate course), and
2. What results they [actors] get from the system [Use Case goal], and
3. How the system actually looks [user-interface],

Then we ought to be able to start the training live-cycle.



Decomposing these test scenarios into individual test scripts is often done by looking for exception processing conditions [alternate course(s) through the Use Case]. At a minimum, the Use Cases can directly drive the high level-testing plan.

S

Use Cases can be used as the starting point for CRC sessions, which in-turn drives the class models and further class discovery. Even without using techniques such as CRC sessions, the descriptions in the Use Cases themselves can be a good starting point for identifying class relationships.

[illegible]

very

Class discovery is the starting point of all class and object diagrams. While there are many techniques for doing class discovery, using the descriptions in Use Cases is often one of the easiest.



Noun Extraction

Noun extraction is one of the easiest techniques for class discovery. At its simplest form, it becomes a matter of decomposing the Use Case description and identifying the actual nouns. These nouns then directly become Candidate Classes'

Verb Extraction

Verb extraction is a common technique for exploring candidate methods on classes. By simply searching through the textual description of a Use Case for verbs, one can easily identify candidate methods. The methods discovered are generally considered part of the class that identified by the noun the verb is referencing in the description.

Adverb Extraction

Adverb extraction, like noun and verb extraction, is used to identify candidate attributes within a Use Case description. Adverb's are usually words that indicate a particular quality of a noun, thereby translating into a candidate attribute for the class.

Adjective Extraction

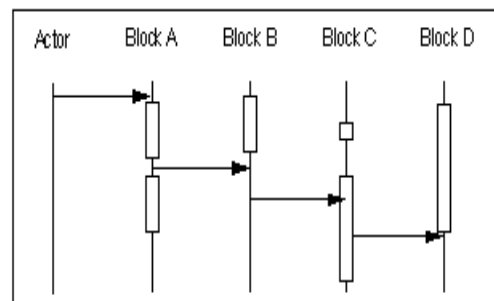
Adjective extraction is a special case of adverb extraction. These words generally identify the particular state of a noun, thereby translating into candidate attributes. Depending on the complexity of the adjective, they can also identify candidate class that may require associated state diagrams.

OID's [OSD's] / Sequence Diagrams

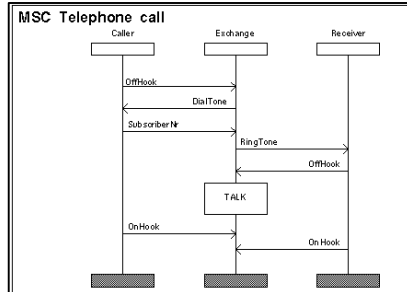
Primary course

The primary course through a Use Case can usually be translated into one or more OID's (Object Interaction Diagrams) or sequence diagrams (OSD's).

The steps in the primary course usually indicate sequential acts that must be performed to satisfy the Use Case, thereby translating into interaction between different object to facilitate the course itself.



Alternate course



Alternate courses, like primary courses, generally identify exception steps that must be performed outside of the primary course to satisfy the Use Case.

Generally, these are also translated into interactions between objects, that are documented as OID's and/or sequence diagrams (OSD's).





Use Case Top Rules

The Use Case rules presented here are an expansion of an article by Ari Jaaksi of Nokia Telecommunications³⁶. They help document real-world experiences and 'best-practices' from multiple OO projects.

These rules can be used as guiding principals on projects that incorporate Use Cases.

1. ***The most important requirements must be specified in Use Cases.*** In the first phases of a system development project, functional and other requirements for the system are collected. Use Cases are generally thought of as a tool used to analyze and document these requirements.

Thus, if we have a requirement for a system being developed, that requirement should be defined and documented in one or more Use Cases.

It should be noted that the requirements being described here are 'functional requirements', and not necessarily 'technology requirements'. These types of requirements are generally defined in the context of architecture. If during the discovery of functional requirements, technical requirements are identified, they should be captured in a related document.

2. ***A Use Case describes something the designer can be proud of, and the customer is willing to pay for.*** I'm especially fond of this rule. It helps define the proper scope and detail of a Use Case, thereby providing consistency among multiple Use Cases.

Use Cases that are too broad in their definition are generally too complex to understand, or too vague to be useful. While Use Cases that are too narrow in their definition are generally too detailed. This rule helps provide a balance.

Therefore, an entity that a customer is willing to pay for while at the same time the designer is proud of, is a good candidate for a Use Case. This rule helps further promote the idea that a Use Case must describe something that is beneficial to the end user.

3. ***A Use Case depicts a typical way of using a system, but nothing more.*** One single Use Case should depict just one possible way of using the system. A Use Case should not try to cover issues outside of its area, and should not try to define all the possible way of performing a task. Instead, each Use Case should define the recommended way of performing a task that the customer (i.e. actor) needs to carry out.

Other ways of using the system should be described in other Use Cases. Exceptions to the Use Case being defined should be described in the 'Alternative Courses' section of the Use Case. However, only where the alternate course adds value from the customers perspective. If the alternative course describes system exceptions





and/or error handling, it belongs in the architecture documentation and not the Use Case.

4. **A Use Case is a play.** One Use Case is like a classical play. Anybody who wants to take the role of the Actor must be capable of performing as intended in the play, just by reading the manuscript (in this case, the Use Case).

The system in this 'play' plays the role of another Actor. The Use Case should not provide the Actors with too much freedom, thereby allowing the play to end up in chaos. At the same time, the play should conclude in the same way each time the manuscript is executed.

5. **A Use Case has a beginning, a main body, and an ending.** Each Use Case should be a complete story, with a beginning, a main body, and an ending (see the Summary section of the Use Case Template, Page 14).

Each Use Case clearly defines where it starts (event or trigger) by explicitly defining the preconditions required. It then continues with the main body of the Use Case. The main body is the 'actual' thing the Use Case does; i.e. the functionality that describes how the actor interacts with the system.

Finally, a good Use Case has some kind of ending that closes the Use Case. The ending usually defines the tangible results that the Use Case has produced. The ending is generally the result that the customer is willing to pay for, and the reason for having the Use Case in the first place.

6. **A Use Case is like an essay written by an elementary school pupil.** Keep it simple. Enough said!

7. **A Use Case fits on two pages.** The entire text of the Use Case should fit on no more than two pages. Longer Use Cases are hard to understand and are generally contain much too much detail.

Either that or they contain multiple areas of functionality, and should be broken into multiple Use Cases.

8. **A Use Case is load and clear.** Each Use Case must make clear statements. The content should be so clear and explicit that the people who read them, such as customers and software designers, can form strong opinions.

A good Use Case describes the usage of the future system so precisely that they can only get better by stimulating conversation between customers and designers.

If nobody disagrees with the first version of the Use Cases, then there probably wrong.





9. **Customers, Users, and Software Designers can all sign the Use Case.**

10. **A Use Case is used as the foundation of system testing.** A Use Case forms the foundation of many things. One of the most important in the later stages of system development is that of system testing.

Referring to Ivar Jacobson's premise of '**The set of all Use Case descriptions specifies the complete functionality of the system.**³⁷', one can also conclude that if all of the Use Cases work, then the system is complete (and hopefully accepted).

The way to ensure if all of the Use Cases work is through a comprehensive test plan that includes detailed test scenarios. This test scenarios can (and should be) directly derived from the Use Cases.





Methodology, my \$0.02

Methodology means many things within the OO world, depending on the particular context the word is being used in at the time. First and *foremost*, it's important to understand the difference between a methodology and a notation. Too often, I encounter the student of OO that has just completed a class in the UML (Unified Modeling Language, language being 'Notation') and is ready to start coding.

At a high-level, a methodology is a process. It's a way of doing something. A notation is a language. It's a way of documenting something. The two often work hand in hand, but it's important to understand the differences.

With that said, one of the first things that I tell every client and every student is the following:



*Select any methodology. It can be OO, Structured, a combination of the two, or any other type of systems development methodology. The author can be any person, organization, and/or institution. Then follow that methodology exactly, and there's one thing I can surely guarantee: **You will fail!***

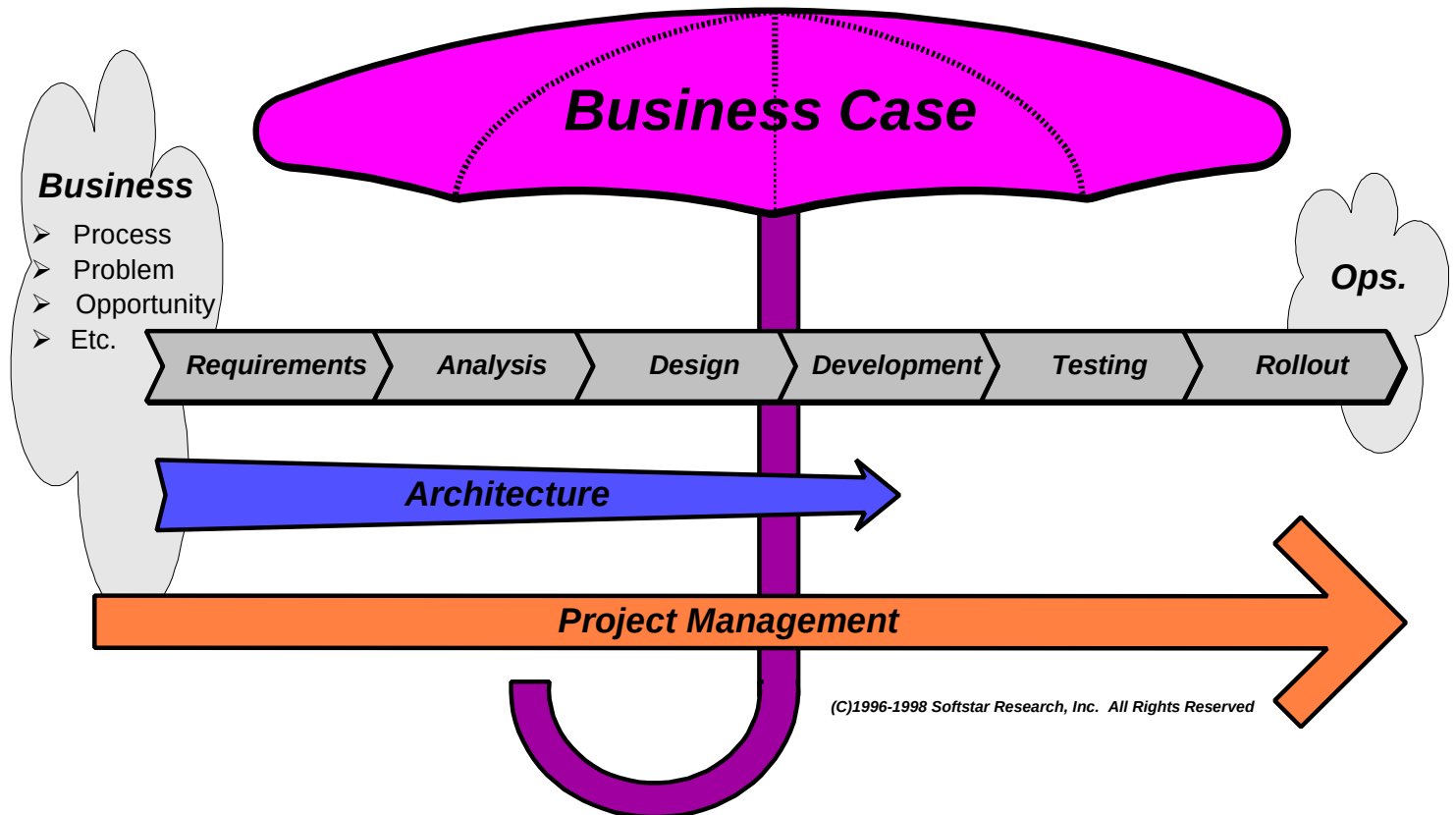
Systems development is still partly a science of trial and error. I have yet to see the all-encompassing cookbook of software development, and yet many organizations embark on the OO path as if it's some kind of 'silver bullet'.

Methodologies should be used as the baseline roadmap. A starting point defining proven best practices if you will. The methodology itself should also allow for the customization of itself, based on the current system, project, organization, and/or resources that are being applied.

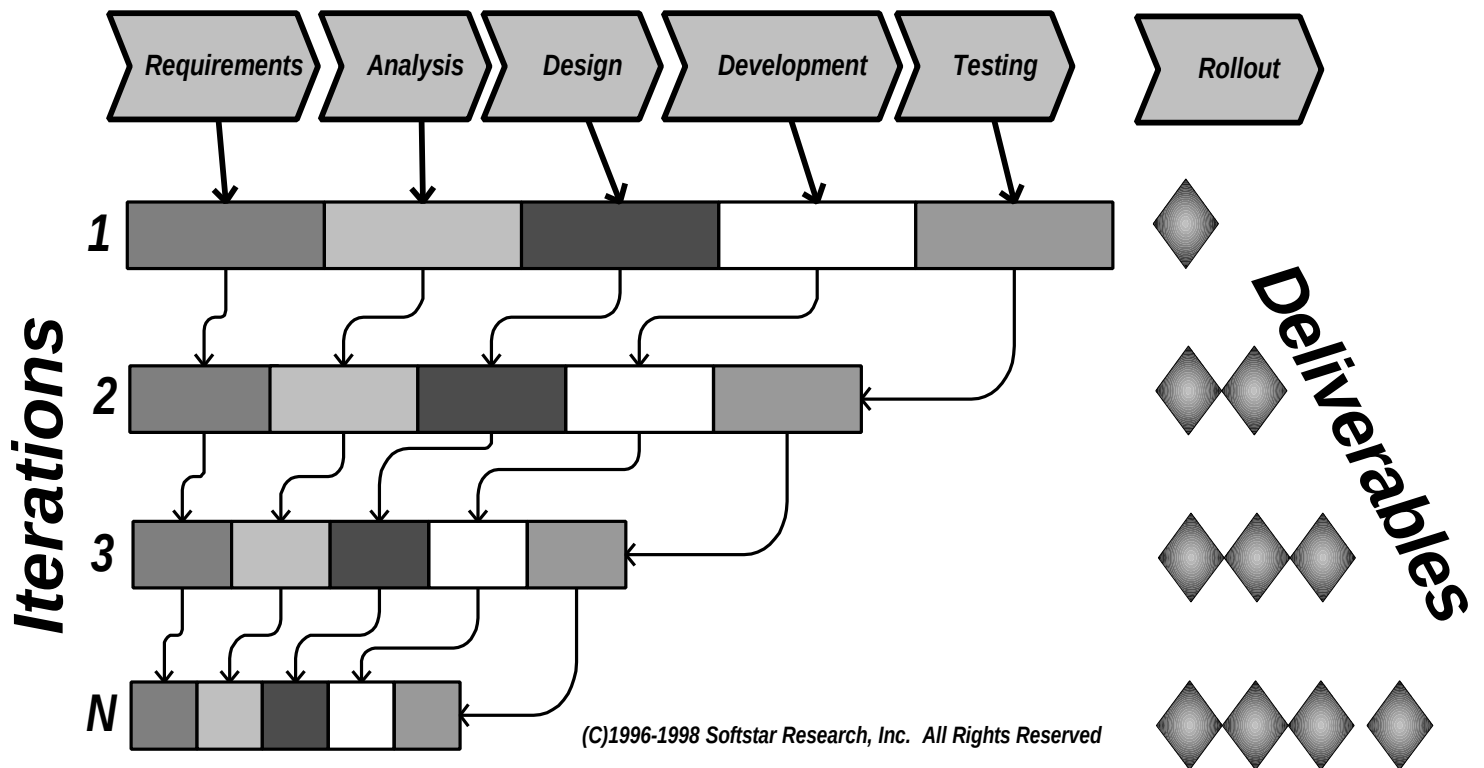
Use Cases provide an entry point into many typical OO methodologies used for systems development. While I've been successful employing Use Cases on a diverse variety of projects, they are not an OO Silver Bullet by any means. There's still a lot to be said for many of the 'legacy' structured methodologies.



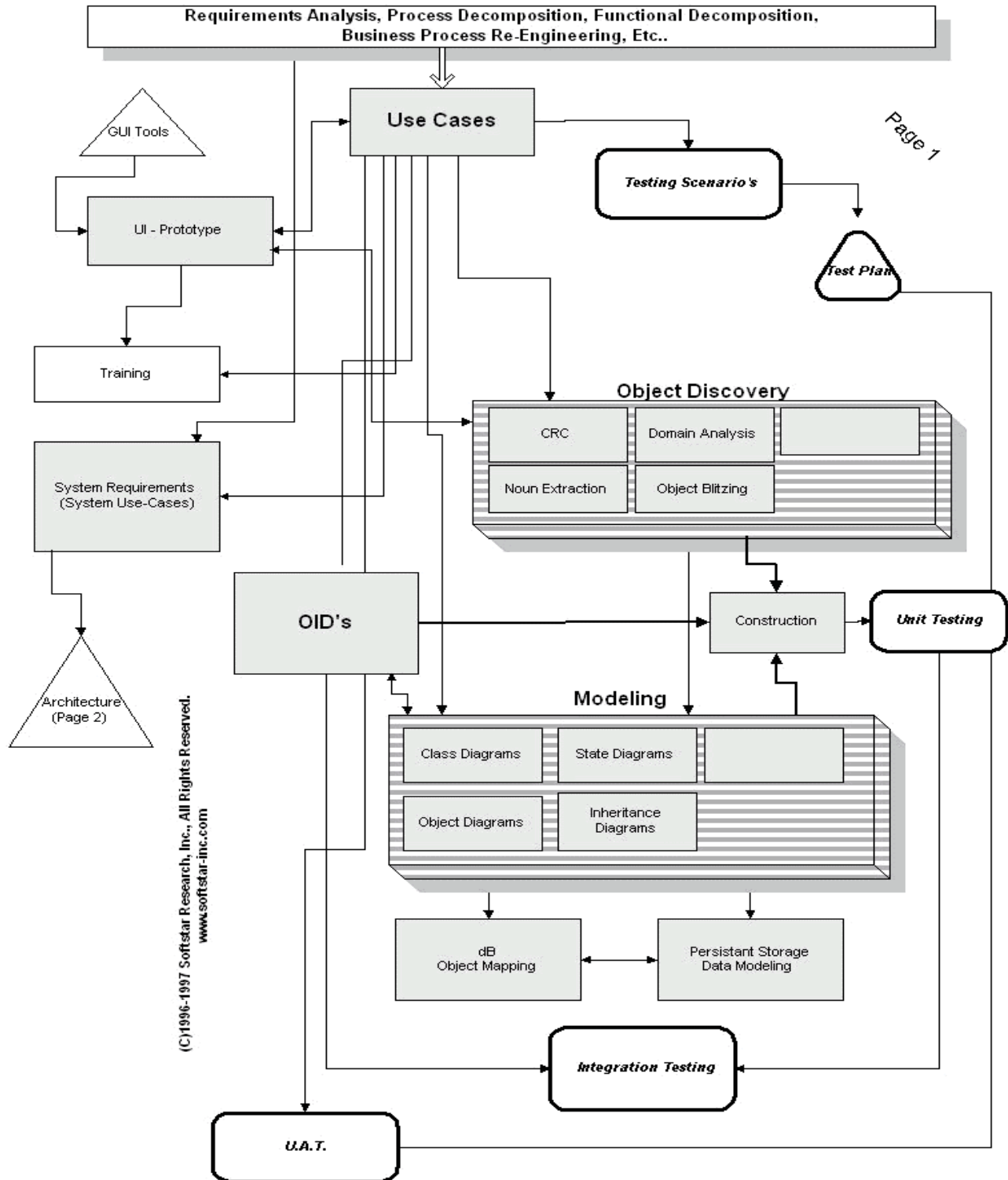
A Typical Project Approach



A Typical Iteration Approach



A Typical High-Level OO Methodology





Author

David Rubin is a Founder, Chief Technical Architect, and OO Methodologist at Softstar Research, Inc. He has been in the computer industry for over 20 years, starting his career with the United States Air Force in the 1970's, while at the same time teaching Computer Science courses at Oklahoma City College.

Throughout the 1980's and 1990's he furthered his career and education by holding numerous positions including Systems Engineer for Electronic Data Systems, Manager of Branch Automation for Total Travel Management, Associate Director for Cambridge Technology Partners, and Director for Claremont Technology Group.



He currently resides in the Midwest and travels the country (sometimes the world) consulting the merits of OO methodology and systems development.

He can be reached at drubin@softstar-inc.com.





Index

A	
Abbott.....	8
actor.....	11, 16
Actor.....	15, 16
Adjective	23
Adverb.....	23
Alternate course.....	16, 24
B	
Booch	8, 9, 10
C	
Class.....	22
CRC Cards	1
D	
design	4, 8, 9, 10
E	
event diagrams.....	5
G	
Goal.....	16
J	
Jacobson	9, 10, 11, 12
M	
Methodologies.....	1, 28
Methodology	28
N	
Noun.....	23
O	
object-oriented	4, 8, 9, 10
objects.....	9, 10, 24
OID's	23
OO	3, 4, 5, 7, 15, 28, 31, 32
OOPSLA.....	10
R	
requirements	5, 6, 13
Rumbaugh	10
S	
Science.....	9
Sequence Diagrams.....	23
Structured.....	4, 28
T	
Template.....	15
Testing	22
Top-down.....	4
Training	21
U	
UML	10
Use Case.....	3, 5, 9, 10, 11, 12, 13, 14, 15, 16, 17, 21, 22, 23, 24, 26, 27
V	
Values.....	13, 14
Verb.....	23





References

- ¹ Be Careful With “Use Cases” By Edward V. Berard The Object Agency, Inc.
- ² Be Careful With “Use Cases” By Edward V. Berard The Object Agency, Inc.
- ³ [SIGPLAN, 1979a]. Anonymous, “Preliminary Ada Reference Manual,”
- ⁴ Association for Computing Machinery, *Proceedings of the ACM SIGPLAN Symposium on the Ada Programming Language*, Boston, Massachusetts, December 9-11, 1980, Association for Computing Machinery, New York, New York, ISBN 0-89791-030-3, 242 pages, 1980.
- ⁵ R.J. Abbott, “Report on Teaching Ada,” *Technical Report SAI-81-313-WA*, Revised December 1980, Research Report for DARPA Order No. 3456, Contract No. MDA 903-80-C-0188, Science Applications, Inc. McClean, Virginia, 1980.
- ⁶ R. J. Abbott, “Program Design by Informal English Descriptions,” *Communications of the ACM*, Vol. 26, No. 11, November 1983, pp. 882 - 894.
- ⁷ A.C. Kay, “The Early History of Smalltalk,” *SIGPLAN Notices*, Vol. 28, No. 3, March 1993, pp. 69 - 95.
- ⁸ Intel Corporation, *iAPX 432 Object Primer*, Manual 171858-001, Revision B, Intel Corporation, Aloha, Oregon, 1980.
- ⁹ Intel Corporation, Engineering Specifications of the iAPX Extensions to Ada, Manual 171871-001, Intel Corporation, Aloha, Oregon, January 1981.
- ¹⁰ E. Organick, *A Programmer’s View of the Intel 432 System*, McGraw-Hill, New York, New York, 1983.
- ¹¹ L. Robinson and K. Leavitt, “Proof Techniques for Hierarchically Structured Programs,” in *Current Trends in Programming Methodologies, Volume 2*, Raymond T. Yeh, Editor, Prentice Hall, Englewood Cliffs, New Jersey, 1977.
- ¹² D.T. Ross, J.B. Goodenough, C.A. Irvine, “Software Engineering: Process, Principles, and Goals,” *IEEE Computer*, Vol. 8, No. 5, May 1975, pp. 17 - 27.
- ¹³ D. Bolz and G. Booch, *Software Engineering With Ada*, Department of Astronautics and Computer Science, United States Air Force Academy, Colorado Springs, Colorado, 1981.
- ¹⁴ G. Booch, “Describing Software Design in Ada,” *SIGPLAN Notices*, Vol. 16, No. 9, September 1981, pp. 42 - 47.
- ¹⁵ G. Booch, “Object Oriented Design,” *Ada Letters*, Vol. I, No. 3, March-April 1982, pp. 64 - 76.





-
- ¹⁶ G. Booch, "Solve Process-Control Problems With Ada's Special Capabilities," *EDN*, June 23, 1982, pp. 143 - 152.
- ¹⁷ E. Dijkstra, "Programming Considered as a Human Activity," Proceedings of the 1965 IFIP Congress, Amsterdam, The Netherlands, North Holland Publishing Company, 1965, pp. 213 - 217, Reprinted in *Classics in Software Engineering*, E.N. Yourdon, Editor, Yourdon Press, New York, New York, 1979, pp. 3 - 9.
- ¹⁸ E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1979.
- ¹⁹ G. Booch, "Describing Software Design in Ada," *SIGPLAN Notices*, Vol. 16, No. 9, September 1981, pp. 42 - 47.
- ²⁰ E.V. Berard, *An Object-Oriented Design Handbook for Ada Software*, EVB Software Engineering, Inc., Frederick, Maryland, 1985.
- ²¹ P.T. Ward and S.J. Mellor, *Structured Development for Real-Time Systems, Volumes 1-3*, Yourdon Press, New York, New York, 1985.
- ²² C.B. Jones, *Software Development: A Rigorous Approach*, Prentice Hall, Englewood Cliffs, New Jersey, 1980.
- ²³ C.B. Jones, *Systematic Software Development Using VDM*, Prentice Hall, Englewood Cliffs, New Jersey, 1986.
- ²⁴ G. Booch, "Object Oriented Development," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 211 - 221.
- ²⁵ S. Stepney, R. Barden, and D. Cooper, Editors, *Object-Orientation in Z*, Springer-Verlag, London, United Kingdom, 1992.
- ²⁶ D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.
- ²⁷ Association for Computing Machinery, *OOPSLA '86 Conference Proceedings*, special issue of *SIGPLAN Notices*, Vol. 21, No. 11, November 1986.
- ²⁸ I. Jacobson, "Language Support for Changeable Large Real Time Systems," *OOPSLA '86 Conference Proceedings*, special issue of *SIGPLAN Notices*, Vol. 21, No. 11, November 1986, pp. 377 - 384.
- ²⁹ Association for Computing Machinery, *OOPSLA '87 Conference Proceedings*, special issue of *SIGPLAN Notices*, Vol. 22, No. 12, December 1987.
- ³⁰ I. Jacobson, "Object-Oriented Development In an Industrial Environment," *OOPSLA '87 Conference Proceedings*, special issue of *SIGPLAN Notices*, Vol. 22, No. 12, December 1987, pp. 183 - 191.





-
- ³¹ I. Jacobson, M. Ericsson, and A. Jacobson, *The Object Advantage: Business Process Reengineering With Object Technology*, Addison-Wesley, Reading, Massachusetts, 1995.
- ³² I. Jacobson, M. Ericsson, and A. Jacobson, *The Object Advantage: Business Process Reengineering With Object Technology*, Addison-Wesley, Reading, Massachusetts, 1995.
- ³³ I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, Massachusetts, 1992.
- ³⁴ M. Christerson and P. Jonsson, *Tutorial 1: Object-Oriented Software Engineering*, Tutorial Notes, Presented at OOPSLA '95, October 15-19, 1995, Austin, Texas, Association for Computing Machinery, New York, New York, 1995.
- ³⁵ I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, Massachusetts, 1992.
- ³⁶ Ari Jaaksi. Our Cases with Use Cases. JOOP Journal of Object Oriented Programming, February 1998,
- ³⁷ I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, Massachusetts, 1992.

