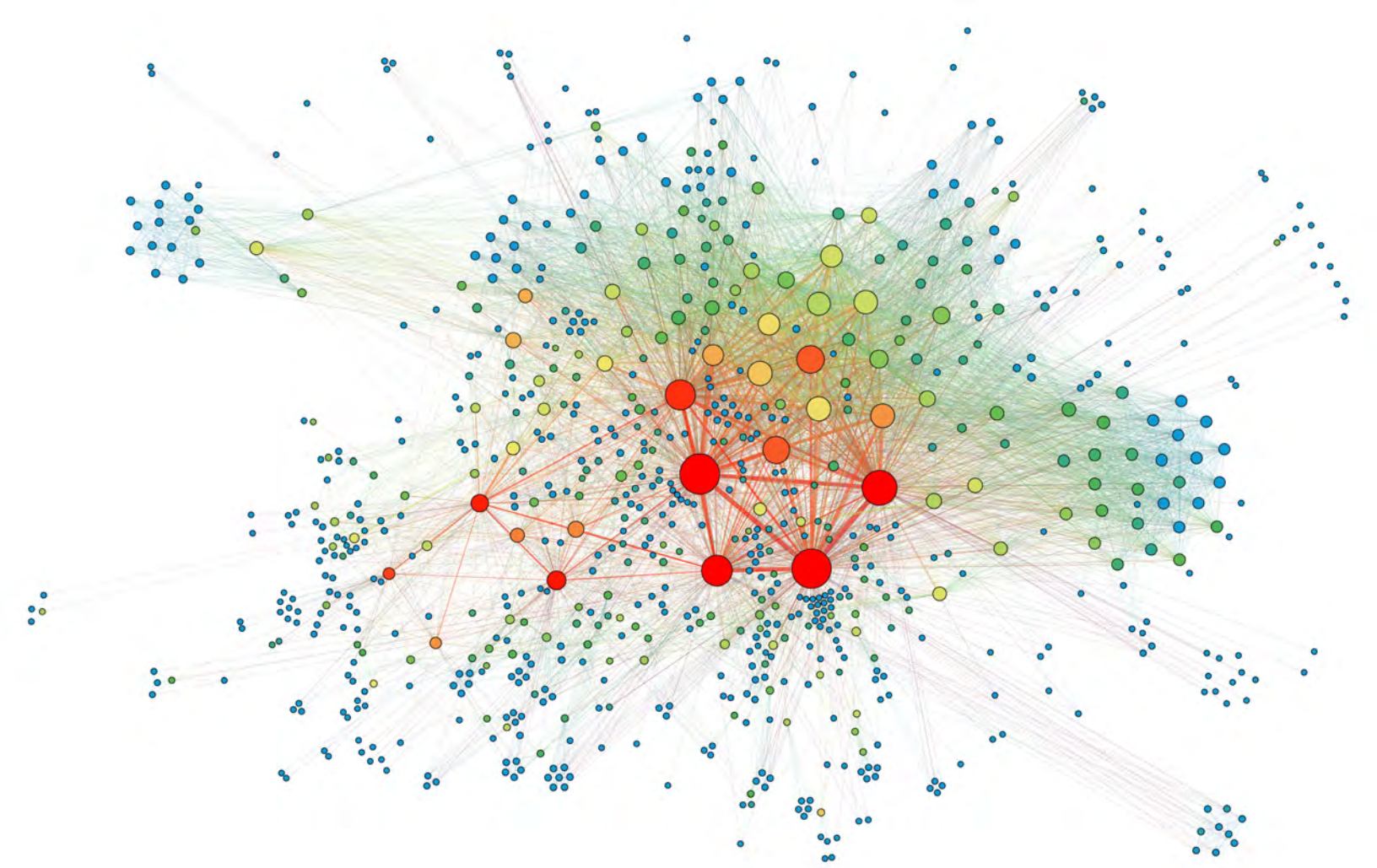


Object-Oriented Analysis & Design

Visual Modeling with UML

SM21-IRS16983-01
2021-01-25

exgnosis

© Copyright 2021 by Rod Davison All Rights Reserved

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA. In the absence of explicit permission from the author to do otherwise, this means:

1. Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
2. Non-commercial — You may not use the material for commercial purposes.
3. No derivatives — If you remix, transform, or build upon the material, you may not distribute the modified material.
4. No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

You may make printed copies of this electronic document; however the author requests that creating unnecessary hard copies be avoided whenever possible.

Attributions and Credits

This work references, quotes and cites published standards, best practices and material created by individuals and organizations that comprise the software development community. All copyright claims by Rod Davison apply only to the material uniquely created by the author for this work and does not apply to any content from other sources or authors that is cited, quoted or referenced.

Original material in this document may be quoted provided that the quotations are both in accordance with the principles of fair use, properly cited and attributed to this source. Quotes of content that is the work of others should be properly attributed to the original sources and authors.

Every effort has been made to ensure all quotations and images from published copyrighted sources have been cited in accordance with accepted academic practices and fair use or are used in compliance with their stated copyright licence. Any unattributed quotes or misused images are oversights that should be reported to the author so that the appropriate corrections can be made.

Any products described in this document may be protected by one or more US. patents, foreign patents or pending patents or copyrights. Trademarked names may appear in this material but rather than use a trademark symbol for every occurrence of a trademarked name, the name is used only to the benefit of the trademark owner with no intention of infringement of the trademark.

Liability

The information in this publication is distributed on an “as is” basis. While every precaution has been taken in the preparation of this material, the author assumed no responsibilities for errors or omissions or for damages resulting from the use of the information contained herein.

Document Information

Document Title:	Introduction to Object Orientation: OOAD and Visual Modeling with UML
Document Type:	Student Manual
Document ID:	SM21-IRS16983-01
Release Date:	January 25, 2021
Author:	Rod Davison
ProTech ID:	PT25573
IRS ELM:	16983
Delivery:	Virtual – January 25-28, 2021

"The primary objective of copyright is not 'to reward the labour of authors, but to promote the Progress of Science and useful Arts.' To this end, copyright assures authors the right to their original expression, but encourages others to build freely upon the ideas and information conveyed by a work. This result is neither unfair nor unfortunate. It is the means by which copyright advances the progress of science and art."

Justice Sandra Day O'Connor

Table of Contents

Module Three: Models Views and Architectures

Module Four: Software Development Processes

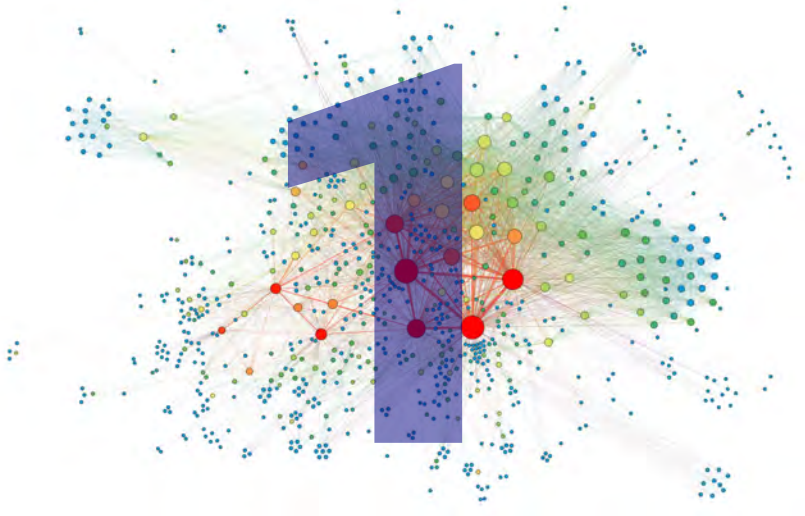
Module Five: Unified Software Development Process and UML

Module Six: High Quality Requirements

Module Seven: High Quality Requirements

Module Eight: Use Cases and User Stories

Module Nine: Solutions Analysis



Object-Oriented Analysis and Design: Visual Modeling with UML

Introduction to Object Orientation

I have yet to see any problem, however complicated, which, when you looked at it in the right way, did not become still more complicated.

Poul Anderson

Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.

Alan Kay

It is not enough to do your best: you must know what to do, and THEN do your best.

W. Edwards Deming

We are stuck with technology when all we want is stuff that works.

Douglas Adams

exgnosis

1. Introduction

Most of the courses and instructional materials I see on OOAD and UML are defective. Most never do more than provide a cursory, incomplete and often incorrect coverage of object-orientation. To be fair, the audience this material targets programmers so the basic approach is to rush through the OO “stuff” to get to the “how to write code” part as quickly as possible.

This approach does get people writing Java or C++ code quickly, but they are often totally lost when it comes to writing code for large or complex applications. A comment I hear a lot from many OO programmers is “I can write Java code that works, but how do I know what Java code I should be writing?” These programmers could write syntactically correct code in an OO language: it compiled and executed without errors. But when faced with the challenge of designing “good” OO code, they were at a loss because they had no idea what “good” OO code should look like.

As one colleague of mine noted “Just because your writing passed the spell checker and grammar checker without errors doesn’t mean you are a good writer.”

Programming languages are all designed to be used in a specific way, just like any other tool. One of the assumptions made by the developers of the OO languages (or any other programming language) is that the programmer has used some sort of design process to develop any sort of architecture for their application. Over many decades I have seen woefully few development teams actually following a design process. Many organizations do have a defined development process, but these so-called processes are often ineffective and so poorly thought out that they should be called “anti-development” processes. As one team lead said to me, “If we tried to follow our process, we would never build a damn thing that worked.”

The situation for UML is similar. Unlike how it’s presented in much of the instructional material, it is not just a collection of diagrams. UML was originally designed as part of the Unified Software Development Process where each diagram had a well-defined role at a specific point in that process. Trying to use UML diagrams without an understanding of what they are intended to model and how they are designed to be used robs the user of much of the value of visual modeling.

The goal of this course more than just to teach you how to do OOAD and visual modeling but how to do it as it was intended to be done so that you get the maximum return on your efforts when you use OOAD.

2. Paradigms and Processes

For many decades, programmers and others involved in software development resisted the idea that software development shared any common features with other kinds of development. Software development was “unique” – a special sort of creative activity. The common belief was that nothing anyone else did or had ever done could be compared to programming because software was unlike anything else anyone had ever made or produced. Software development was thought of as more like composing a symphony or writing poetry as opposed to more mundane processes like building a house or making a meal.

In the early 1970s, I asked a friend of mine who worked as a programmer at IBM to teach me how to program. His response was, *“You can't teach programming any more than you can teach someone to have perfect pitch. You either ‘get’ programming instinctively, or you don't.”*

By the early 2000s, the work of the Software Engineering Institute and other groups demolished this idea that software development was exempt from standard engineering principles. However, once the realization dawned on the software development community that development processes are critical to software production, the pendulum swung too far in the opposite direction and the industry became obsessed with the erroneous idea that if we could just find the right production process (Agile, RUP or whatever), then the software would literally write itself.

In this section, we want to clearly delineate the difference between a production process, which is how we organize our activities to get the work done, and a paradigm, which refers to the actual techniques we use to build things. The same paradigm, object-oriented for example, can be used with different processes, and multiple paradigms can be used with a given type of process, Agile for example. Experience shows that confusing these two, for example thinking an Agile technique is part of the object-oriented paradigm, makes it much more difficult to learn, understand and master either.

2.1 Paradigms

First, we need to define exactly what a paradigm is. The modern idea of a paradigm was popularized by Thomas Kuhn in his famous book *The Structure of Scientific Revolutions* where he defined a paradigm as:

> *The entire constellation of beliefs, values, techniques and so on shared by the members of a given community. It also denotes one sort of element in that constellation, the concrete puzzle-solutions which, employed as models or examples, can replace explicit rules as a basis for the solution of the remaining puzzles of normal science. [Kuhn70, pg. 175]*

Kuhn points out that, at least in science, shared paradigms allow us to collectively do more than could if we all did everything in our own idiosyncratic manner. Paradigms reflect the power of people working together to solve problems or accomplish goals. As Kuhn points out:

> *Paradigms gain their status because they are more successful than their competitors in solving a few problems that the group of practitioners has come to recognize as acute. [Kuhn70, pg. 23]*

In other words, paradigms exist because they work. But it is also important to note in that last quote that paradigms focus on the specific problems that are important to the paradigm practitioners.

Paradigms, in a more generic sense, are everywhere because humans tend to approach producing a wide range of things in similar ways. For example, we can think of different kinds of cuisine as “food preparation paradigms”: French cooking, Indian cooking, Tex Mex, Barbecue, Mediterranean, etc. Not a perfect example since there are many sub-paradigms in each of these[1].

So what sort of things might be found in each of these cooking paradigms?

We find specific preferred cooking techniques (stir-fry versus roasting), choices of seasoning (curry versus ketchup), different culinary tools (double boilers versus woks), different ideas on what sort of foods should be used (haggis versus pufferfish), different ways of deciding if something is “well-prepared”, different ways of creating and assembling dishes for presentation and so on.

This variation in cuisines does not imply in any way that one cuisine is better or more correct than any other, just that they are all different. We can account for these differences by considering the factors that influence the development of a paradigm.

1. The nature of the problem to be solved.
2. The resources available to solve the problem[2].
3. Any constraints that prevent certain kinds of solutions.
4. How well the solutions generated by the paradigm methods solve the problems.

2.2 The Engineering Production Process

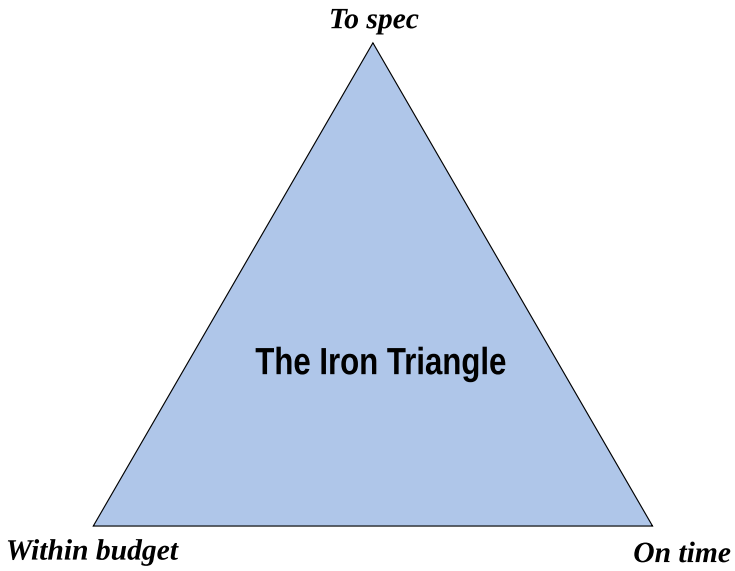
Many people cook intuitively – they consistently create incredible meals on the fly based on experience and skill. But being a fantastic cook doesn't mean you can run a restaurant. Many talented cooks fail catastrophically when they open a restaurant because running a restaurant means cooking under real-world production constraints; specifically:

1. **Specification:** You can no longer just cook what you want, you have to cook to customer specification – you have to cook what the customer orders. If they want roast chicken, you can't decide to cook the grilled salmon instead, even if you really like making grilled salmon.
2. **Time Constraints:** You have to deliver the meal to the customer within a reasonable time – customers won't come back if they find themselves waiting longer than they expect to.
3. **Budget Limits:** You have to make sure that it doesn't cost more to deliver the meal than you are charging the customer for it.

These three constraints are known in engineering as the iron triangle. These are the triad of production factors that rule every single engineering project or other activity where we build or make something. An old saying in engineering is "You can have it good, fast and cheap, but you only get to pick two," which describes the reality of working under these constraints. If you want it fast and cheap, it's not going to be good.

When you run a restaurant, the iron triangle doesn't apply for just one meal but for every meal you create, day after day, customer after customer. At the minimum, this means that the kitchen has to be organized to support meal production and there has to be an established process for the kitchen staff to follow consistently so that you can meet the iron triangle constraints every time, whether you are preparing one meal or a hundred meals at the same time.

Running a restaurant is not the same as cooking. Many amazing cooks open restaurants and fail because they don't know how to run a restaurant. On the other hand, many fast-food restaurants have become successful because they have mastered a restaurant production



process specialized for fast food, but no one would claim that the minimum wage hamburger assembler in the back of a fast-food restaurant is a good cook or a chef.

In the book *The E-Myth Revisited*, Michael Gerber points out that the main reason why people who are very good technically at what they do fail when running a business because they fall prey to:

> ***That Fatal Assumption****: If you understand the technical work of a business, you understand the business that does the technical work. The technical work of the business and a business that does the technical work are two different things. [Gerbr01; pg 13]*

In other words, people who have become experts in a paradigm, whether it's cooking or programming, fail when they assume that they don't need a production process.

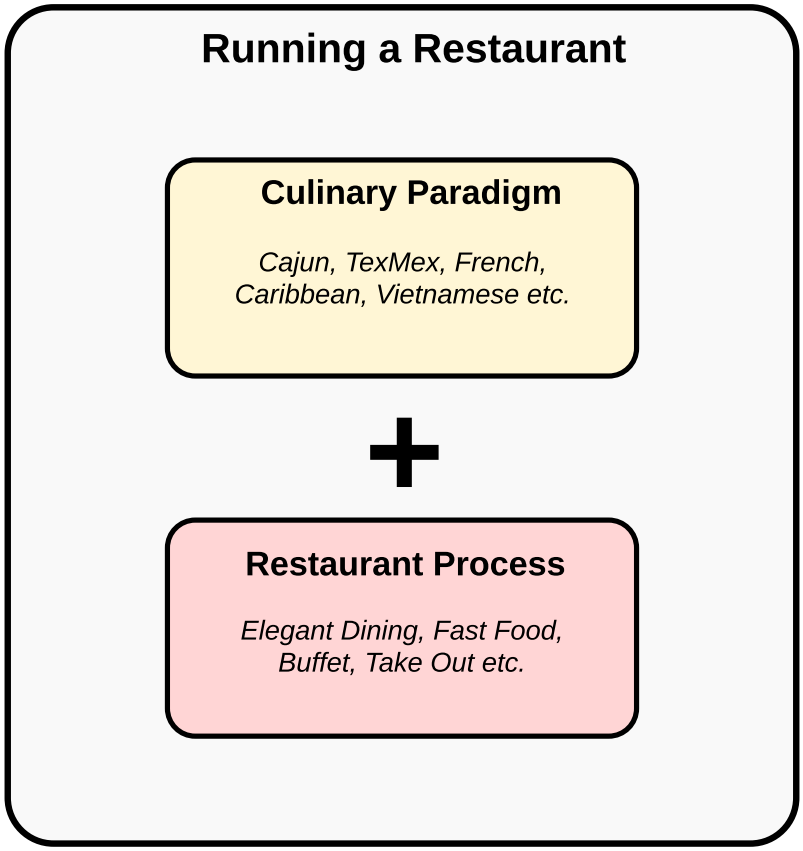
2.2 The Independence of the Production Process

No matter what type of cuisine a restaurant specializes in, in order to meet the production constraints of the iron triangle they have to adopt some kind of restaurant process. Some typical restaurant processes are:

1. **The Formal Dining Process.** Reservations are made, you are greeted by the maître d', seated and waited on by various staff.
2. **The Fast Food Process.** You place your order at a counter, pay, then carry your food to a table to eat.

3. **The Buffet Process.** You take a plate and pile it with food from dishes on a buffet table.
4. **The Take-out Process.** You place your order, then pick up your food and take it home.
5. **The DoorDash model.** You order through a meal delivery service that picks up your food and delivers it to you.

It is important to note that these restaurant processes are totally independent of the style of cuisine or the culinary paradigm used by the restaurant. There can be TexMex formal dining rooms, French cuisine fast-food restaurants, Vietnamese buffets and so on. We should note that the success of a restaurant depends on producing high-quality meals (mastering the culinary paradigm) and running and restaurant efficiently (mastering the production process).



Kuhn's idea of a paradigm represented a significant insight but, unfortunately, was soon co-opted by faddish business consultants and gurus with claims that they could bring success to anyone by helping them "shift their paradigm." This claim is hype and nonsense since most of these people had never even read Kuhn or understood what a paradigm is.

To understand why object-orientation emerged as such a dominant paradigm in software development, we have to understand what a paradigm shift actually is, not what the faddish gurus later claimed it to be.

Kuhn stressed that paradigms are powerful social structures that enable their practitioners to work in a coordinated manner to accomplish much more than they normally would be able to achieve by working independently.

So why do paradigms shift? Unlike what many so-called gurus claim, it's not because the existing paradigm is broken or "wrong." This is worth mentioning because many OO proponents make the claim that object-orientation "fixed" the structured paradigm that came before it because the structured paradigm was irreparably defective. This is just totally incorrect.

Paradigms shift because the subject they deal with or the environment they are used in changes. The paradigm starts to lose power when it is unable to adapt to these changes. Kuhn illustrates this with the example of the shift in paradigms in physics from Newtonian classical mechanics to Einstein's Relativity and Quantum Mechanics.

A paradigm is dominant as long as it is a highly effective tool for solving the set of problems that its practitioners have decided are important. However, there are always data or anomalies that the paradigm can't explain. Initially, these anomalies are treated as errors or are ignored by the paradigm's practitioners.

However, there are usually smaller paradigms that emerge specifically to study these anomalous events, often with a high degree of rigour. As the anomalies continue to accrue, eventually they can't be ignored any longer forcing the dominant paradigm to evolve to explain both what has previously been explained and these anomalies. If the dominant paradigm can't evolve, then it is either replaced by or has to co-exist with one or more of the new paradigm that can explain the anomalies.

In physics, classical mechanics worked fine for everyday mechanics problem but failed for things at cosmological scales and atomic scales. Physics had to extend the classical paradigm to include Relativity to explain things at the cosmological model but could not be extended to account for atomic level anomalies. Quantum Mechanics had to be adopted as a co-paradigm to manage atomic level physics. To this day, the holy grail of physics is to somehow unify these two paradigms into one TOE (theory of everything).

3 Paradigm Shifts in Software

There exist a number of different software development paradigms.

1. **Structured Programming.** Also called procedural programming, this was the dominant paradigm in software engineering until the late 1990s when OO exploded in popularity. Structure programming languages include COBOL, FORTRAN, ALGOL, C and most scripting languages like PERL and Python.
2. **Object-Oriented Programming.** This paradigm is characterized by designing programs that follow an OO model. Typical OO languages are SmallTalk, Java and Eiffel.
3. **Functional Programming.** This paradigm is modeled after the mathematical ideas of functions and functional analysis. This paradigm is exploding in popularity with the rise of AI and data analysis. Typical functional languages are Haskell, Erlang, Scala and ML.
4. **Logical Programming.** This paradigm is characterized by writing programs in the form of first and higher order logic. The classic example of a logic programming language is Prolog.
5. **Symbolic Programming.** This paradigm is built around the idea that programs are just data and should be able to re-write themselves in various ways. The classic example of a symbolic language is LISP.

There are many languages, like C++, that are designed to be used to write code according to multiple paradigms. For example, in C++ a programmer can write pure structured code or pure OO code depending on the language features they choose to employ.

3.1 Cross Paradigm Languages

Over the last decade, there is an increasing trend to not think of a programming language as belonging to a specific paradigm. What we are seeing instead is programming languages adding features of different paradigms to their capabilities. For example, COBOL has introduced objects to produce object-oriented COBOL. Java, Python and JavaScript now have incorporated a number of functional programming features.

The major languages introduced in the last decade – Go, Python, Julia and Rust – don't fit into any paradigm but freely take ideas from a number of different paradigms.

3.2 Dominant Paradigms

Only two of the paradigms, maybe three soon depending on what happens with functional programming, were the only ones that achieved what we might call widespread industrial use: structured programming and OO programming. The reason for their dominance is that they

solved the problems that the software development community were faced at the time.

The factors that have initiated the change in paradigms have been:

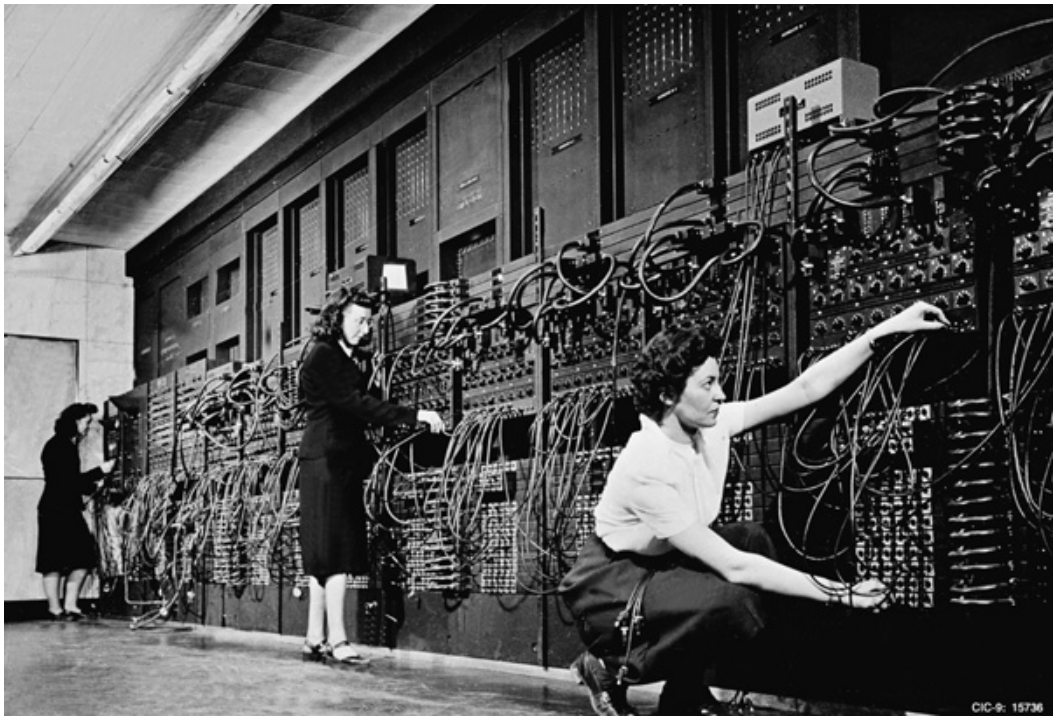
1. The increasing complexity of software and the rise of complex systems.
2. Changes in hardware technology.
3. Changes in who uses the software.

In the section that follows, we can see the transition from no paradigm to structured analysis and design to OOAD.

3.3 Evolution of Computing

This section provides a bit of background to a very brief history of software engineering. While this is not going to be critical to your ability to work OOAD, it does provide some information and context that you may find useful in understanding the relationship between the structured world and the OO world. Especially in an organization like the IRS where both paradigms are alive and well.

3.3.1 The early years (1940-50s) – hardware rules



When the first computers were developed in the 1940s, they were strictly pieces of hardware; there was no need for software because you programmed the computer by physically altering the wiring. Interesting note, in old science fiction movies when the scientist goes off to pro-

gram the robot or computer, he often grabs a pair of pliers and a screwdriver. The accompanying photo shows how the ENIAC was programmed by interconnecting the electron tube registers with cables inserted in plug board.1

Computers were very expensive, very unreliable and primarily used as automated calculation engines. Operating systems were in their infancy or non-existent so the primary role of software was to manipulate hardware, at least where software existed.

3.3.2 The code cowboys – 1950s-1970s

Into the 1950s, computers continued to be very expensive and unreliable, but primitive operating systems became the norm to speed up the use of the hardware even though the OS had to often be loaded from punch cards or paper tape prior to every program run.

The quality of code was determined by how efficiently it used the hardware, even if the code was unreadable by humans. As a result, programs were very small in size, written in assembler for efficiency and usually automated one particular calculation. Serious programming was done as close to actual machine code as possible. There was no way to automate “systems” and no thought of anything that looked like a paradigm.

The primary role of computers was to automate repetitive, difficult and error-prone engineering and statistical calculations like producing numerical solutions for sets of differential equations. IBM’s prediction in the 1950s that there was a worldwide market for no more than six computers was probably quite accurate given their cost and limited usefulness.

Software development or programming does not exist in any substantial form early in this stage but starts to slowly evolve with the development of early high-level languages like COBOL, FORTRAN and the OO language SIMULA. Quality of code is measured not only in terms of producing correct results, but is intertwined with the efficient use of hardware. On the correctness side, the simple problem domains mean that correctness checks can generally be done quickly and informally, usually by running a couple of known test values through the program.

3.3.3 The mainframe era – 1970s-1990s

By the early 1970s computers, like the IBM 360-370 family, had become more powerful and reliable. Operating systems created a virtual layer between the programmer and the hardware that allowed programmers to use high-level languages and focus on the problem domain rather than manipulating the hardware. For example:

1. The operating system created a layer of abstraction between the user and machine that allowed programs to be portable and no longer tied to a specific machine or device.
2. Better data storage, random-access capabilities and the introduction of databases allowed programs to “pick up where they left off last time.”

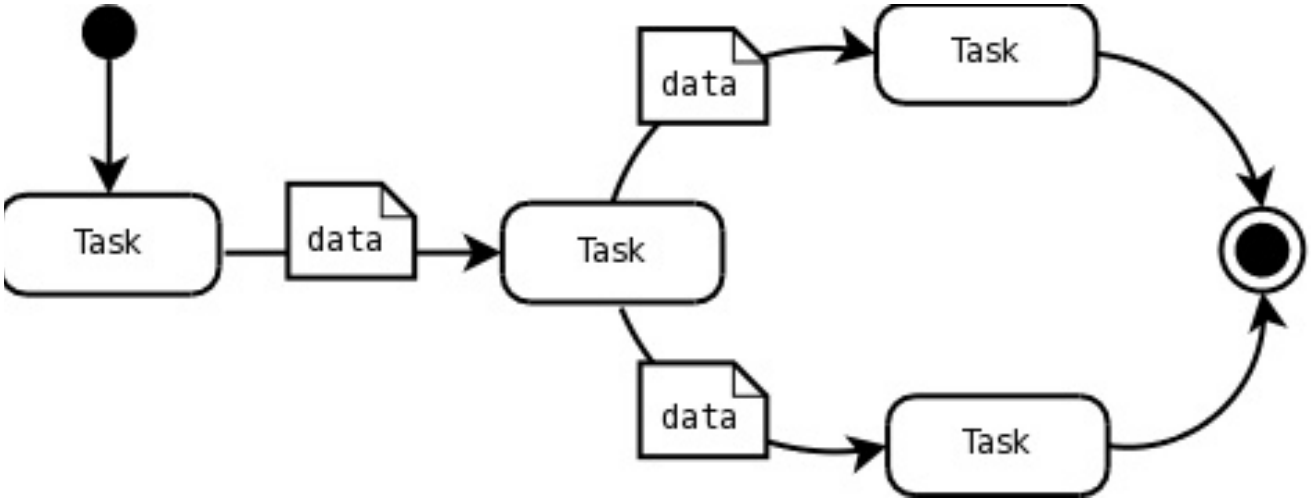
3. High-level languages allowed a wider variety of problems, especially ones involving non-numeric data, to be automated.
4. Multitasking, multiprocessing and real-time access via systems like TSO allowed the computer to become a shared resource.
5. Previously developed code could be reused through code libraries.

Keep in mind that by today's standards these were still small machines. Many mainframes came with 360K of main core memory, but software developers now had the capability to undertake automating whole systems rather than just individual tasks.

> *Users of the370 Model 165 will have a choice of five main core storage sizes, ranging from 512,000 to over 3-million bytes. Seven main memory sizes are available for the Model 155, ranging from 256,000 to over 2-million bytes. [IBM Marketing Material]*

Given the above specification for the system 370, it must also be remembered that at the time, core memory was about $10 per byte. However, the capabilities of the mainframe allowed programmers to go beyond automating a single task but could start to automate a sequence of tasks, or a workflow. The image above is of an IBM 370 CPU being unpacked.

In a workflow, we have data that is moved from station to station (in the real work, usually a specific worker) that performs some operations on the data which can be thought of as a sequence of tasks. Programs of the time automated this model in what we call batch processing where data is read into a system, and undergoes various transformations at the hands of program modules that perform specific tasks.



3.3.4 The development crisis

With all these new capabilities, the old cowboy way of developing software, a process that might be called “channeling code”, no longer worked. Projects now required much more code to be written and maintained, and projects required teams of developers instead of being individual efforts.

Suddenly, developers also had to deal with users. Since existing business systems were being automated, like payroll systems, users had expectations about how those systems were supposed to work and demanded that they work correctly.

Projects were now starting to consume significant resources based on the promise that the completed system would deliver a return on investment. This meant that software developers were now accountable to the bottom line and had to deliver on time and on budget and to spec. This was a new a terrifying prospect for many software developers – their code had to produce correct results based on someone else's idea of what the program should do.

> *Many systems were being developed during the 1960s. They were huge conglomerations of software written by people who really didn't understand that software, as well as hardware, had to be engineered to be reliable, understandable and maintainable. Endless hours and countless dollars were spent tracking bugs that should never have entered the system in the first place. Errors in the earliest phases of the project were not located until long after the products were delivered to the customers. These errors were enormously expensive to correct.*
>
> *People turnover on the projects often resulted in large numbers of software modules being scrapped and then being rewritten by new people because the existing modules couldn't be understood. So much attention was given to these problems that eventually computer scientists and industry people began devoting considerable resources to the problem of constructing computer systems. This spawned the development of the field of software engineering*[4].

3.3.5 Software engineering (SE) practices

The solution was to think of building software as engineering problem and adopt engineering type practices. Eventually, this resulted in the development of:

1. **Structured programming** – standardized programming practices supported by rigorous mathematical models.
2. **Software testing** – the application of quality control concepts and techniques to software.
3. **SDLC (System Development Life Cycle)** – engineering project life cycle management concepts applied to software development projects.
4. **Structured Analysis and Design** – the incorporation of engineering modeling methods and design concepts into software development. Diagramming techniques – ER diagrams, dataflow diagrams, and flowcharts – were derived from engineering diagrams.
5. **Business Analysis** – a result of realizing that the systems had to fulfill business requirements and these had to be identified and understood (engineering economics).

Did it work?

YES!

The phenomenal growth of computerization and the success of many of the systems, which are still working today, is a direct result of adopting software engineering practices.

But it didn't happen overnight. There were bitter battles that went on for years between the old style programmers that still thought in terms of "writing to the bare metal" and the new software engineers. There was also a lot of confusion about what actually constituted software engineering, and how to translate the powerful mathematical techniques that were being developed into practical tools.

3.3.6 The maturing of the structured approach

Eventually the structured approach won over most of the code cowboys, at least when it came to building large systems. The reason was simple, in the large mainframe world, the structured approach delivered the goods.

In fact, in the early years OO, many of the early OO theorists tried to emulate the rigour that had become characteristic of the structured approach. Jim Rumbaugh, in his OO modeling methodoglogy OMT still relied on data flow diagrams as a critical part of his model.

In retrospect, we can see that there were three factors that led to the widespread acceptance of the software engineering approach.

1. The increasing complexity of the software.
2. The increasing reliance on the software by organizations to accomplish their business functions, i.e. there was increasing economic pressure to get the job done right. (Software Quality).
3. The improvements in the power and capability of hardware.

The complexity issue is one that we will be seeing over and over again in software testing. At this point, the whole complexity issue is worth a little detour in order to appreciate fully what its implications for us as developers actually are.

4 Complex Systems

So what exactly are complex systems and how did they come to be? In many respects, complex systems are the outcome of both technological innovation over the last several decades and the social changes that have occurred as a result of the increasing acceptance of and reliance on software.

Hardware continued to get more powerful and cheaper. In the 1980s, desktop computing started a trend towards distributed and embedded computing. As processors started to appear everywhere, networks started to develop to tie them all together because the use of protocols allowed different and incompatible networks to connect seamlessly.

This gave rise to a new kind of system that grew instead of being built. These new complex systems tend to resemble biological systems more than traditional engineering-type systems.

Most complex systems arose as the result of connecting preexisting systems into larger systems. This often meant that these new systems were not built but tended to keep growing as more and more layers of complexity were added.

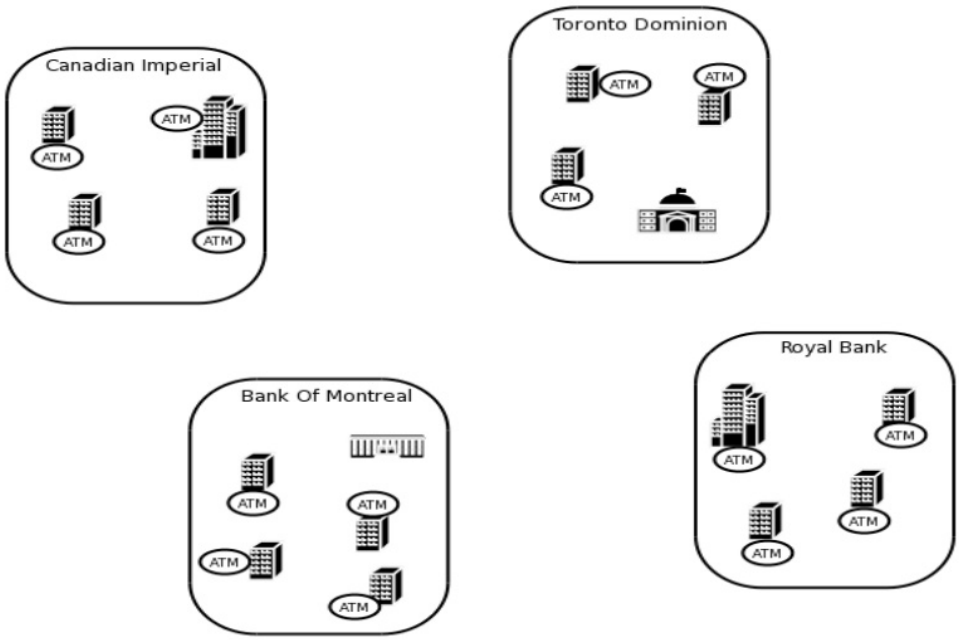
4.1 How Complex Systems Arise

We will take a look how a typical complex system arises. The example that we will use is that of the Canadian Banking System. The reasons for picking this example are:

1. The Canadian banking system is similar to the US system in terms of services, products and technologies.
2. The Canadian system is simpler with only six major banks, each of which has a national presence. The Canadian system is not regionalized as the US system, and there are not as many players.
3. Because the Canadian system is simpler, technological trends are easier to spot and tend to occur before they occur in the US.

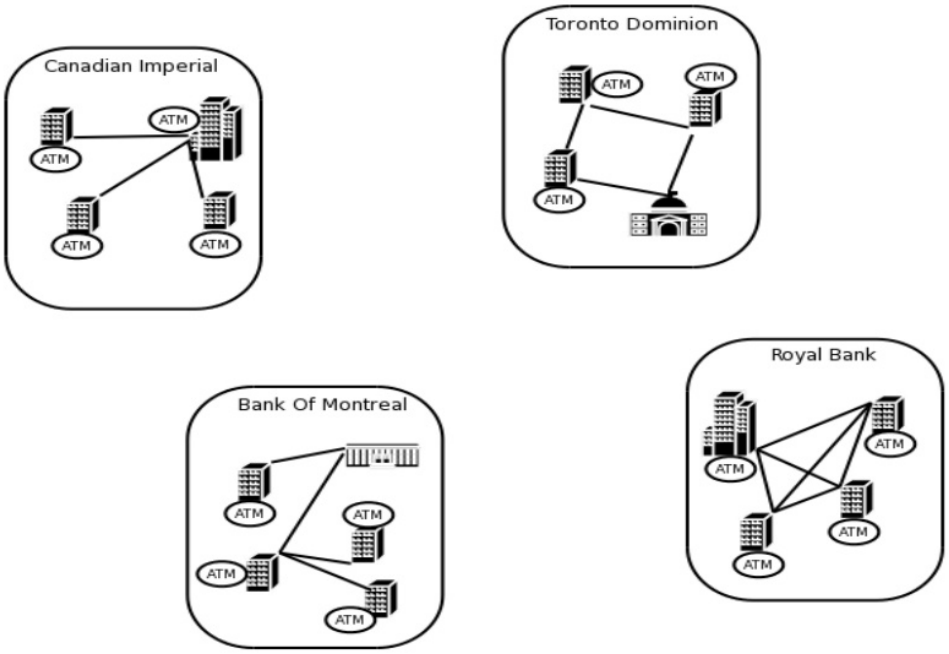
While we are looking at the Canadian banking system, the same fundamental process also applies to other complex systems like the phone system, power utilities and the Internet.

4.1.1 Phase one – going distributed



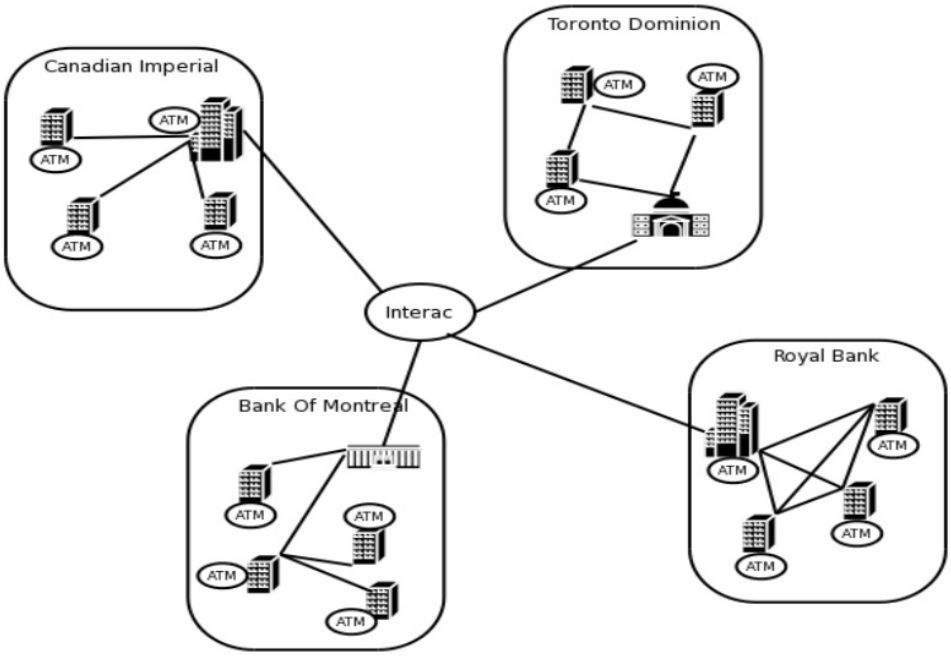
In the early 1980s Canadian banks introduced “after hours banking machines.” These were ATMs located at bank branches that allowed users to access their accounts outside of banking hours, but only at their own branch. The processing was distributed but not networked.

4.1.2 Phase two – networking



After initial acceptance by customers, one bank networked its ATMs, which allowed its customers to access their accounts from any of that bank's branches across the country. Other banks quickly followed. Banks could now also introduce standalone ATMs connected to the bank's network at non-branch locations.

4.1.3 Phase three – protocols



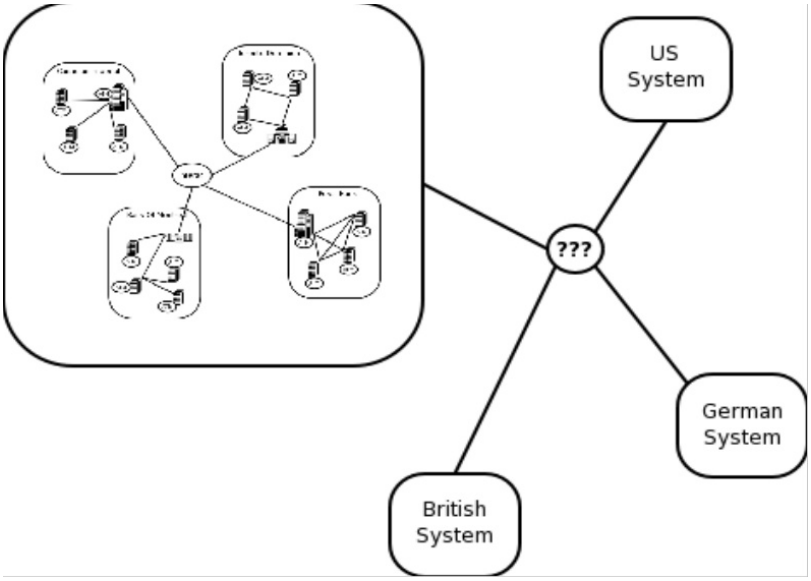
Since there were now multiple incompatible networks, the banks agreed on a protocol to allow them to exchange information. This protocol (and its supporting infrastructure) is called "Interac." Users can access accounts from anyone's ATM and it makes it possible for users to use their bankcards to make direct payments to merchants connected to the network.

4.1.4 Phase four – more protocols

Once the Canadian network is in place, the next step is to connect Interac with other similar networks. Through the use of protocols, Interac can now connect with and exchange information with various US networks and other national networks. This does not require any knowledge of how the other networks actually work or are structured internally; each national network can be treated as a black box at this level.

This process of layering or building up complex systems from preexisting simpler systems has occurred in many of our mission critical systems: the Internet, the phone system, the electrical power grid and many others.

The layering also place in organizations where the layers are added to over time until a similar kind of hierarchy emerges.



4.2 Characteristics of Complex Systems

The Canadian banking example is representative of how a number of systems developed over the 1980s and 1990s; other examples include the phone system and the Internet. As these systems become more accessible to users, the more they are used; the more they are used, the more they are made accessible to users. And the more mission critical they become. This is called the Law of Increasing Returns[5].

> *Like it or not, the marketplace isn't stable. The world isn't stable. It's full of evolution, upheaval and surprise. Economics had to take that ferment into account. And now he [Brian Arthur] believed he'd found the way to do that using a principle known as "increasing returns" -- or in the King James translation, "To them that hath shall be given." Why had high-tech companies scrambled to locate in the Silicon valley area around Stanford instead of in Ann Arbor or Berkeley? Because a lot of older high-tech companies were already there. Them that has gets. Why did the VHS video system run away with the market even though Beta was technically a little bit better? Because more people happened to buy VHS systems early on, which led to more VHS movies in the movie stores, which led to still more people buying VHS players, and so on. Them that has gets.*

In other words, the law on increasing returns explains, in part, what drives the formation of complex systems and provides a mechanism that explains the rise of mission critical software.

Now that we know what complex systems are, we can answer the question that probably is uppermost on your mind right now: what does this have to do with OOAD? Remember our first assertion in this course – how we design our solutions depends on the nature of models we use. These complex systems have some significant differences from the older structured

systems that will have a direct impact on how we approach their development and design.

4.2.1 Mission Critical Software

These complex systems often had reached the stage that Grady Booch calls Mission Critical Software or Industrial Strength Software, which essentially are systems we cannot afford to have fail.

> *Mission Critical systems tend to have long life spans and, over time, many users come to depend on their proper functioning. Often this dependency arises gradually and unnoticed over time until the organization discovers that it can no longer function in the absence of the software.[Booch94: pg 4]*

To make the situation even worse, the more we rely on a system, the more useful it becomes, and the more useful it becomes, we rely on it even more. This means these systems grow and become very, very large, which leads to the other part of the mission critical software dilemma.

> *The distinguishing characteristic of mission critical software is that it is intensely difficult, if not impossible, for the individual developer to comprehend all the subtleties of the design. Stated in blunt terms, the complexity of such systems exceeds the human intellectual capacity. as, this complexity we speak of seems to be an essential property of all large software systems. [Booch94: pg 4]*

So we have a dilemma: the more we need it to work right, the more complex it is to make it work right. We now have a crisis on our hands because the structured approach does not scale well to handle large and complex systems, but we cannot afford to have these systems fail.

4.2.2 Hierarchical in Structure

Older systems were divided into functional subsystems, complex systems tended to be hierarchies of subsystems and sub-subsystems. Looking back at the Canadian bank example, it can be seen that the hierarchical structure of the banking system was an artifact of the complex system being “grown” from simpler systems by layering levels of connection via protocols.

> *Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached[6].*

> *All systems have subsystems and all systems are parts of larger systems. The value added by a system must come from the relationship between the parts, not from the parts per se[7].*

These last comment by Rechtin has some interesting implications for modeling our programs. Our Canadian banking example is very obviously a hierarchical system (How?)

4.4.3 Distributed and Modular

Older systems usually had a single CPU and well-defined flows of control, complex systems have multiple CPUs and exhibit concurrent distributed processing and simultaneous threads of execution. Older systems tended to be monolithic while complex systems tend to have autonomous subsystems working in a coordinated but independent manner.

If a complex system is built up of connections between subsystems, then this means that connections between subsystems would be easier to break than connection within a subsystem. Or to think of it another way, a complex system decomposes naturally in the reverse order from how it was built up.

> *Intra-component linkages are generally stronger than inter-component linkages. This fact has the effect of separating the high-frequency dynamics of the components – involving the internal structure of the components – from the low-frequency dynamics – involving interaction among components*[8].
>
> *The choice of what components in a system are primitive is relatively arbitrary and up to the discretion of the observer of the system. As systems evolve, objects that were once considered complex become the primitive objects upon which more complex systems are built. [Booch94: pg 12]*

This characteristic has produced some interesting systems problems where complex systems experience partial failures – some parts fail but the rest of the system continues to function.

For example, the Credit Union used by the New York City employees suffered the loss of the subsystem that validates ATM withdrawals against accounts in the 911 attack. This meant that anytime someone tried to take money out of their credit union account through an ATM, the transaction was approved regardless of the amount of funds actually available.

The credit union, in a goodwill gesture, allowed members to make withdrawals on the honour system. After it was discovered that several members had withdrawn hundreds of thousands of dollars they did not have in their accounts, they reluctantly filed fraud charges against them. Suddenly hundreds of other members offered to make restitution for their own banking “oversights.”

This means that we may have local failure but not global failure in our systems. One might say that complex systems might not fail, but will often do the best they can, even if it is not what they are supposed to do.

4.4.4 Event Driven

Older systems were usually batch or job oriented, while complex systems tend to be event driven and oriented to responding to external stimuli from the environment. There is no flow of control but there are usually multiple threads of execution (often running on different processors) which are responding to these external events.

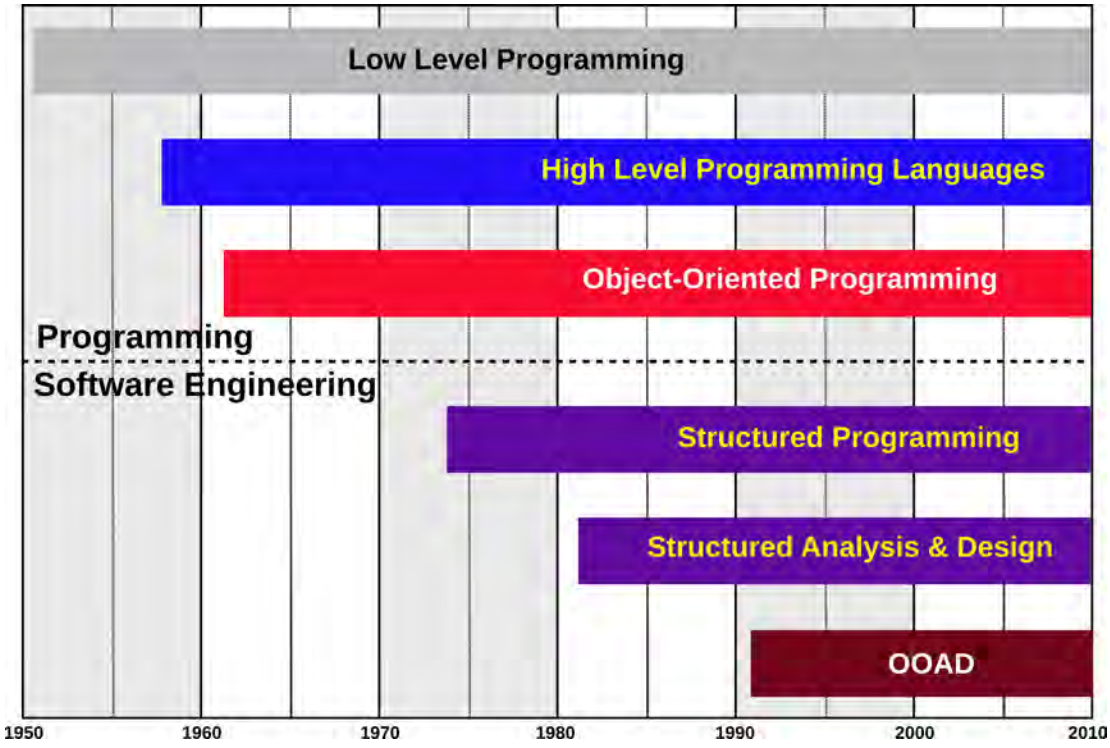
From a design point of view, this means that the strategies for programming system functionality that are based of the old input-compute-output model may not be adequate to design these event driven systems. Combined with the previous point about modular and distributed and we can see that the input event space is now multidimensional and incredibly complex.

Users interact with the new systems directly instead of through an intermediary like an MIS department, as was often the case in older systems. Notice in particular the move towards highly interactive system in most areas in our daily lives, many of which used to be performed through a human intermediary. The trend has been going on for decades, starting with the elimination of elevator and telephone operators.

There is often a fuzzy or ambiguous boundary between the complex system and its environment. In terms of system boundaries, we have used the term fuzzy to emphasize that the coupling between the environment and system is so tight and fine grained, that it may be impossible to tell with precision where the boundary between the system and environment lies.

The increased interactivity of these new systems has also led to the problem that the system, especially in terms of its boundaries, is defined by the users or those who interact with it.

5 Paradigms to Software Development Processes



The general evolution from a programming paradigm into a software engineering approach seems to go something like this:

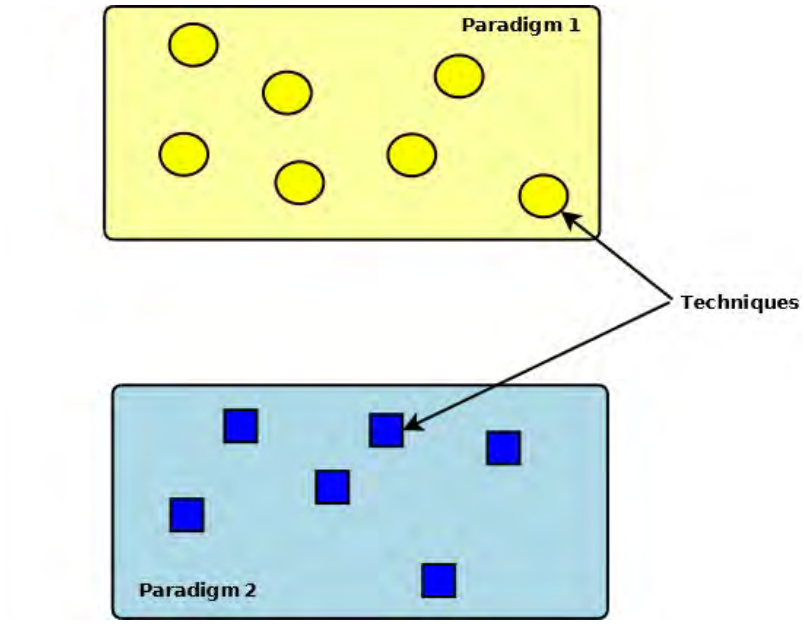
1. A programming model is proposed. As the programming model is used, the community of programmers who work with that model develop characteristic programming techniques, consistent programming styles and identify and catalogue best practices for writing programs using the model. The model becomes a paradigm.
2. As the programming paradigm matures, it starts to be used to solve specific problems that are often not addressed by the established software engineering approaches of the day. It's very common to find these new paradigms used in research or to develop prototypes of new classes or kinds of applications.
3. Social, business or technological change, often a combination of all three, result in the existing software engineering approaches becoming out-dated and no longer effective in solving the new kinds of problems that the changing environment poses.

4. However, many of these changes in the emerging environment come from the very areas of research that our new programming paradigm has been used in for the last while, so this new programming paradigm is seen as a good fit to solve the kinds of problems the existing software engineering approaches are having difficulty with. The new programming paradigm is “mainstreamed.”
5. Once in mainstream use, the programming paradigm is used in projects that are subject to the Iron Triangle.
6. Existing software engineering practices are modified to support the new programming paradigm or new practices are created from scratch, both of which become the basis for a new software engineering approach based on the programming paradigm.

The diagram shows exactly what we would expect to find since it follows the general emergence of approaches in other branches of engineering. Aside from more highly specialized and niche type industries, we have now seen this process played out with procedural programming, which evolved into the structured software engineering approach and now with the object-oriented paradigm which led to the development of OOAD.

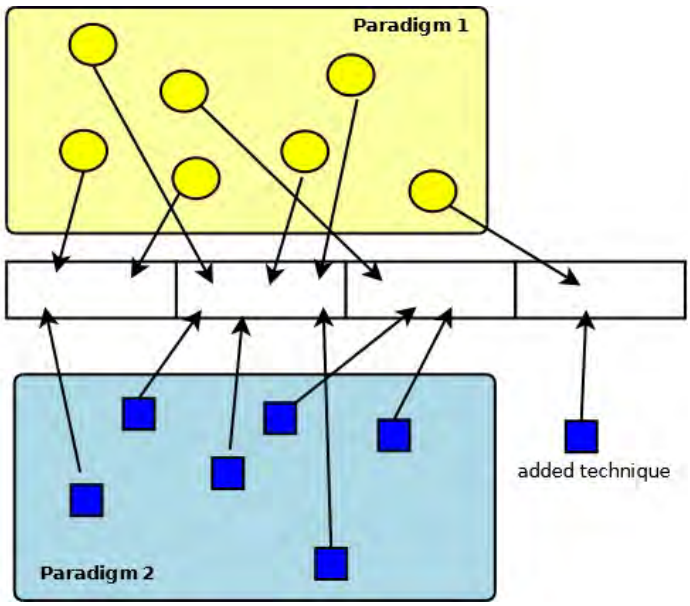
5.1 The Paradigm and The Engineering Process

Generally, in the development of a paradigm, a number of techniques are established that are characteristic of the paradigm. For example, use cases are a technique we tend to find the OO paradigm while structured decomposition is a technique we tend to find in the structured paradigm. This is illustrated generically below.



The techniques are generally not highly organized with respect to each other except within the context of the paradigm. When we add an engineering process, it has the effect of organizing our techniques. In other words, the process gives us a series of steps to follow and the paradigm tells us what to do within each of those steps.

The engineering process also has the effect of identifying techniques that may be missing from the paradigm but are needed in the engineering process. For example, a common type of missing techniques are those that involve requirements gathering and documenting or quality assurance.



Here the process has been used to embed the activities of both paradigms into the process structure. It also showed that paradigm two didn't have any techniques for handling the last part of the process. When this happens, and it often does, we go back into the paradigm and develop techniques to make our paradigm “fit” into the process.

In summary, we have to think of the two parts of a software development process being: the paradigm that tells us how we do things; and the engineering or production process that tells us when to do it in order to meet our production constraints. Of course, this is a general sort of analysis just for the purposes of helping you get the idea of how this all works.

Notes and References

1. In my discussions of paradigms, I have taken some liberties with using culinary styles as paradigms only to provide an analogy that I hope everyone can understand. Most of what we see in the technical world is an application to what we do in other aspects of life. Software development processes, for example, are just applications of general engineering production processes. I find students get the technical material quicker when they can relate it to other things they already know and understand.

2. An anthropology colleague of mine spent several years living with a tribe in the highlands of New Guinea who were essentially hunter-gatherers. Their whole diet consisted almost exclusively of the wild boar they hunted and the wild yams they gathered, both of which had to be cooked over an open fire. Their cooking paradigms were quite elaborate since, as he put it, “Pork and yams, three time a day every day. When that’s all you’ve got to eat, you better come up with some pretty creative ways to cook them or else you just go mad. I was rather impressed the first time I had a pork pancake.”

   This particular tribe had a variety of ways to cook yams and boar using just an open fire, but they had no idea how to cook any other kind of food, even when they had access to it, except by preparing it the same way they prepared yams and boar.

3. The photo is part of the history of supercomputing display at the Oak Ridge National Laboratory web site (http://www.csm.ornl.gov/ssi-expo/P2.html)

4. An Introduction to Operating Systems; Harvey M. Dietel; 1984; Addison-Wesley, Boston MA ISBN 0-201-14501-4; page 15

5. Complexity: The emerging science at the edge of order and chaos; M. Mitchell Waldrop; 1992; Touchstone Books, Simon and Schuster; New York NY; ISBN:0-671-87234-6; page 17

6. On Time and Space Decomposition of Complex Systems; P. Courtois; June 1985; Communications of the ACM Vol(28)6; page 596

7. The Art of Systems Architecting; Eberhardt Rechtin,October 1992; IEEE Spectrum Vol(29)10; page 66;

8. The Science of the Artificial; H, Simon; MIT Press; Cambridge MA; 1983; page 218

Object-Oriented Concepts

Object-oriented' is the `new and improved' of computer-software salespersons (the ones with the slicked-back hair, polyester suits, and pointy teeth). The literature presents object-oriented programming as a panacea that will save computer programmers from virtually every conceivable problem. It's supposed to improve performance and make your code both more maintainable and easier to understand. It's supposed to isolate inexperienced programmers from the complexity of a low-level language like C. It's supposed to change the baby and take out the wash. Unfortunately, most of the people who use the term `object-oriented programming' would be hard pressed to define it. Although OOP does have many benefits, you don't get them just by using an object-oriented language.

Alan Holub

Object-oriented programming is an exceptionally bad idea which could only have originated in California.

Edsger Dijkstra

1 Object-Oriented Origins

Some of the basic concepts that comprise the foundation of the object-oriented paradigm were formulated in the late 1950s. However, it took several decades for the paradigm to evolve into the form it has today. Object-orientation has progressed through a series of stages which refined the paradigm further by adding new tools and concepts that built on the previous stages.

Each of these stages was characterized by the formulation of a fundamental idea or principle, which in turn led to the development of further techniques, concepts and practices that are now integral parts of the current form of the paradigm.

In this module, we will look at three fundamental principles that been the motivation for many of the ideas that are the basis of today's object-oriented techniques. These principles are:

1. **The Principle of Iconicity:** This was formulated in the early 1960s to explore the idea that users should be able to use computer applications without training. This led to analyzing systems and problems in terms of interfaces and implementations. Currently, the idea of decoupling interfaces and implementation forms the basis for many of the analysis and design techniques used in OOAD and OO programming.
2. **The Principle of Recursive Design:** Until the 1980s, almost all computing happened in monolithic mainframe environments. The dominant structured approach was ideally suited to software development in the hardware and computing environments that existed through the 1960s to the early 1990s. However, as the cost of hardware plummeted and distributed computing became increasingly common and software increased in complexity, the structured paradigm proved to be very difficult to apply in this new distributed, networked environment. The principle of recursive design became an alternative approach for thinking about how to build software in these environments.
3. **The Object Model:** The object model is a set of basic ideas and operational definitions about what these “object” things are and provides a framework that supports the other two main principles.

In the remainder of this module, we will look at each of these principles in turn and explore the other ideas and techniques that have been generated from them. Once you have a good understanding of these ideas, you will find the rest of the material to be a lot simpler to learn, understand and use in practice.

2 The Principle of Iconicity

Professors Ole-Johan Dahl and Kristen Nygaard (pictured together in the 1960s) did much of their innovative work at the University of Oslo. Recipients of the Turing Prize and John von Neumann Medal, both men passed away in 2002. The *Association Internationale pour les Technologies Objets* established the Dahl-Hygaard prize in 2004 awarded annually to

> *two individuals that have made significant technical contributions to the field of Object-Orientation. The work should be in the spirit of the pioneer conceptual and/or implementation work of Dahl and Nygaard which shaped our present view of programming and modeling, now known as Object-Orientation.*

The origin of the object-oriented paradigm can be traced back to the development of their Simula programming language in the 1960s. The name Simula, an acronym for **sim**ple **u**niversal **la**nguage, described the original motivation for developing Simula, which was to provide a means of simulating systems. In many respects, this was one of the first attempts to think about what had to be done with technology to make it usable by the general public.

The fundamental problem that motivated Nygaard and Dahl was

“Why should people have to learn how to interact with a computer? Why can we not design a computer program that resembles what it automates so that people can interact with it based only on what they already know?”

2.1 The Principle of Iconicity

Trying to simulate the world in a recognizable way can be expressed in terms of iconicity:

Systems should be iconic – they should look like what they automate.

This means that if we are automating an accounting system, for example, when an accountant uses our automated system they should see things they know from the accounting domain – ledgers, journals, accounts, statements – and can do everything with them that they would do in their paper-based system such as post transactions, generate statements and other accounting activities.

Any accountant should not need to learn anything new to use the program, their knowledge of the problem domain, in this case the field of accounting, should be sufficient for them to use the program without having to be trained on it.

2.2 Domain Modeling

One of the consequences of following the principle of iconicity is that if we are going to show the user an iconic representation of their domain, then we have to know what is in that domain from the user's perspective. This was not part of the structured paradigm and led to the challenge of figuring out how these domain things ought to be modeled so that the models were both iconic and usable by developers.

The original ideas on how to do this modeling eventually evolved over a number of years into the object model which, interestingly enough, was in direct conflict with existing models of cognition. However, as we will see later when we talk about the object model, there was a significant revolution brewing in cognitive psychology which brought cognitive psychology closer to the object-oriented way of thinking. As George Lakoff noted:

> *Cognitive science is now in transition. The traditional view is hanging on, although the new view is beginning to take hold. Categorization is a central issue. The traditional view is tied to the classical theory that categories are defined in terms of common properties of their members. But a wealth of new data on categorization appears to contradict the traditional view of categories. In its place there is a new view of categories, what Eleanor Rosch has termed "the theory of prototypes and basic level categories." [Lakof87: pg xii]*

To update the quote, the transition is long over and the "traditional" view is now just an interesting historical artifact. The other fact we should note is that this work on domain modeling predated the idea of data modeling, which didn't really come into existence until the late 1970s by about 15 years.

2.3 The Simula Programming Model

One of the remarkable things about Simula is how the concepts and ideas that were used to create a programming model look like what we see in many object-oriented languages today. However, to understand the challenge that Professors Nygaard and Dahl faced, a quick description of what they had to work with is illuminating.

2.3.1 The Technology of 1963

Simula was originally developed to run on the Univac 1107 which in 1963 was one of the most advanced computers available. The computer was offered at a considerable discount to the Norwegian Computing Center in August of 1963 by UNIVAC, along with a contract to implement Simula on the 1107, which was done by January 1965.

Because we are used to 2021 era technologies, to understand the challenges that Dahl and Nygaard had in implementing Simula, consider some of the characteristics of the “seven” as it was known to its admirers:

1. Cost was about $1.5 million or about $13 million in today's dollars.
2. It had about 256 Kbytes of memory and used tape storage with additional magnetic drum storage that provided up to 6 Megabytes of random-access storage.
3. Input was done with decks of punched cards which could be read at 300 cards a minute. Output was via a line printer.
4. It could process the payroll of a company in four hours.

2.3.2 The Simula Approach

Simula was developed as a superset or extension of ALGOL, which was probably the most advanced of the three main structured languages of the time, the other two being COBOL and FORTRAN.

This design isolated the code that the user interacted with into an interface layer which allowed them to define classes and work with program constructs that looked like domain objects without breaking the ALGOL code that would actually execute.

2.3.3 A Simula Example: The Drafting Package

This package defines the geometrical concepts of points, lines, figures, etc. and enables the description of geometrical constructions to be made in a fairly neutral notation. The way we picture the concepts of the problem area is very closely mirrored by the Simula descriptions1.

A geometrical point is completely specified by two Cartesian coordinates (x,y). But it may have other attributes associated with it too – for example, its distance:”r” from the origin, the angle, “theta” it makes with the positive x-axis. In Simula, we describe such a concept in a “class declaration. The class declaration for points could be:

```
CLASS POINT(X,Y); REAL X,Y;
      BEGIN REAL R, REAL THETA;
            REF(POINT) PROCEDURE ROTATED;
            BOOLEAN PROCEDURE EQUALS;
            REAL PROCEDURE DISTANCE;
            R := SQRT(X ↑ 2 + Y ↑ 2)
            THETA := ARCTAN2(X,Y)
      END
```

The declaration has a name (POINT) and parameters; within the body of the class, further parameters are defined.

ROTATED(P,N) will generate a new point by rotating this one about the point P by N degrees,

EQUALS(P) will check if this point and P represent the same geometrical point, DISTANCE(P) return the distance between this point and the point P.

Each time a point is created, R and THETA are calculated from the actual parameter values of X and Y.

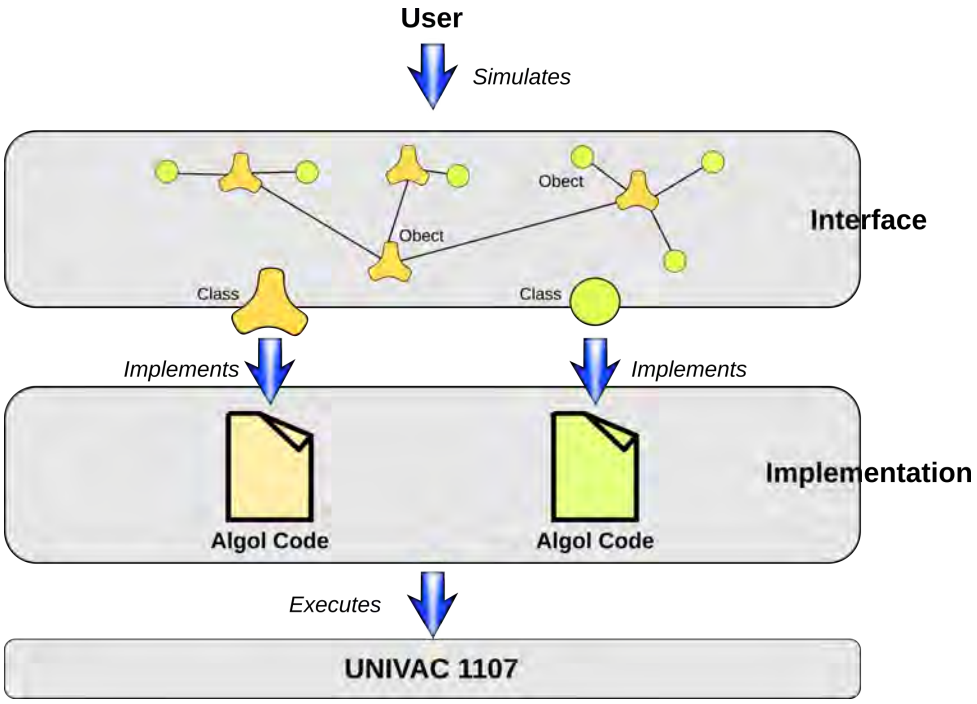
A second declaration we can give describes the concept of a chord:

```
CLASS CHORD (P,Q); REF(POINT) P,Q ;
      BEGIN REAL LENGTH;
            IF P== NONE OR Q == NONE OR P.EQUALS(Q)
                  THEN ERROR("ILLEGAL CHORD DEFINITION")
                  ELSE LENGTH := P.DISTANCE(Q) ;
      END
```

Once these classes have been defined, it is now possible to write simulations involving points and chords, and to create definitions of other geometrical objects as well. If we understand geometry, then we understand what is going on in this code; for example, chords are defined by their end-points, and points have coordinates. This is domain knowledge.

What this means is that we can write a simulation of a geometrical problem using the language of geometry – points and chords for example – without having to worry about what the underlying ALGOL code actually is by creating two layers called the interface and the implementation. This idea of decomposing a system down into these two layers has become a fundamental concept of the object-oriented paradigm, usually referred to as decoupling the interface and implementation.

We can depict these two layers in the following diagram.



The concept of the decoupling of the interface and implementation is a re-statement of modular design techniques in other areas of engineering, including structured programming. Because it is applied as a basic principle of how we think about system design, it recurs throughout the OO paradigm in many places and forms.

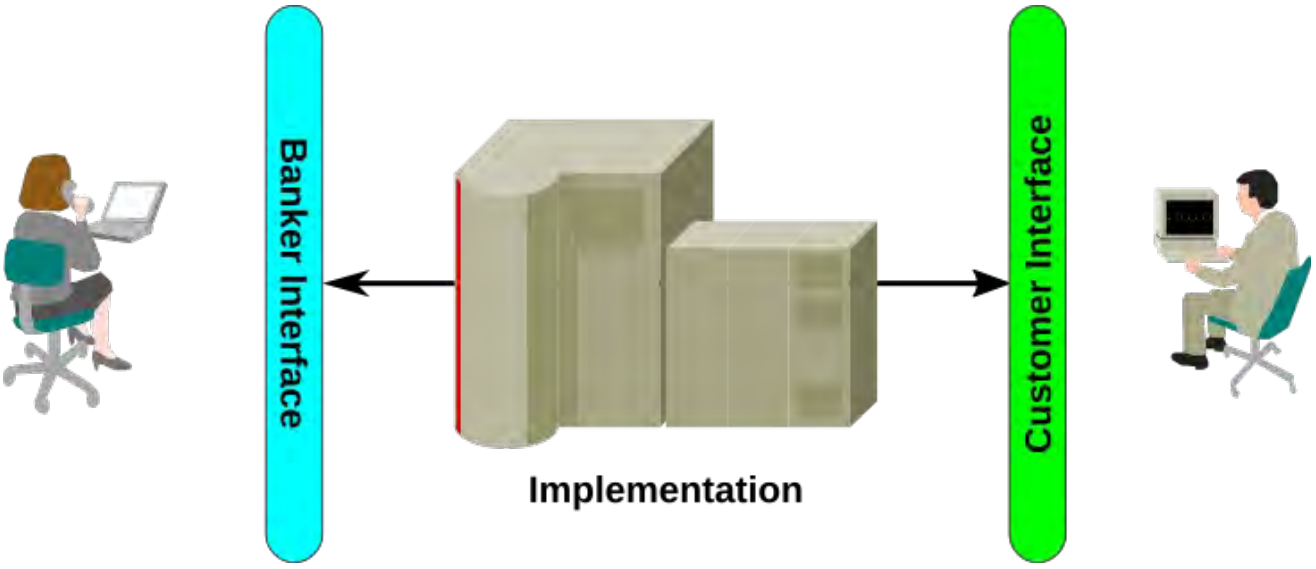
The Simula programming language introduced many of the object-oriented programming techniques to accomplish this decoupling of interface and implementation such as inheritance, encapsulation and virtual classes. Because of this, many OO developers mistakenly believe that object-oriented is just using these techniques when, in reality, these programming techniques are tools developed to implement object-oriented designs.

2.4 Interface and Implementation

A fundamental architectural pattern of OO is the interface-implementation decoupling.

Users interact with the system only through interfaces that are iconic. However, the system implementation – the program that executes and does the work – is not iconic. The implementation is designed however the developer chooses in order to meet the requirements for the system and given the resources developers have to work with.

One of the main mistakes that OO developers and designers make is assuming the whole system should be iconic when, in fact, it is only the interfaces that need to be iconic. The implementation is a black box to the users of the system and does not need to be iconic in any way whatsoever. This tends to occur when developers do not clearly differentiate between interface and implementation and have users interact directly with the implementation.



This separation of interface and implementation has two significant advantages.

1. The implementation can be modified or changed completely without any impact on the interfaces being used. This frees the developer to use any engineering or software approach or tools they want in the development or modification of the implementation.
2. When there are multiple types of users who come from different domains, rather than changing the implementation, we can now provide multiple interfaces to accommodate the different types of users.

To illustrate the last point, consider a banking application with two main types of users: bankers and customers. The banker interface is going to model a very accounting type domain with credits, debits and transactions. However, the customer domain doesn't have debits and credits but has deposits and withdrawals instead. Most customers might find the banker interface too confusing to use, but they can easily use the customer interface that shows them the "things" they already know about as bank customers.

2.5 Implications for OOAD

The iconicity principle has two important corollaries about what we need to do in order to be able to build iconic systems.

2.5.1 Domain Modeling

In order to have an iconic application, we need to understand what the users know about the application domain and how that knowledge should be expressed in the system we are about to design. We need to model the application domain in a way that extracts the important features of the domain so that they both recognizable and usable by the end-users. However, we also have to ensure we don't get bogged down in the details that are not needed for our interface.

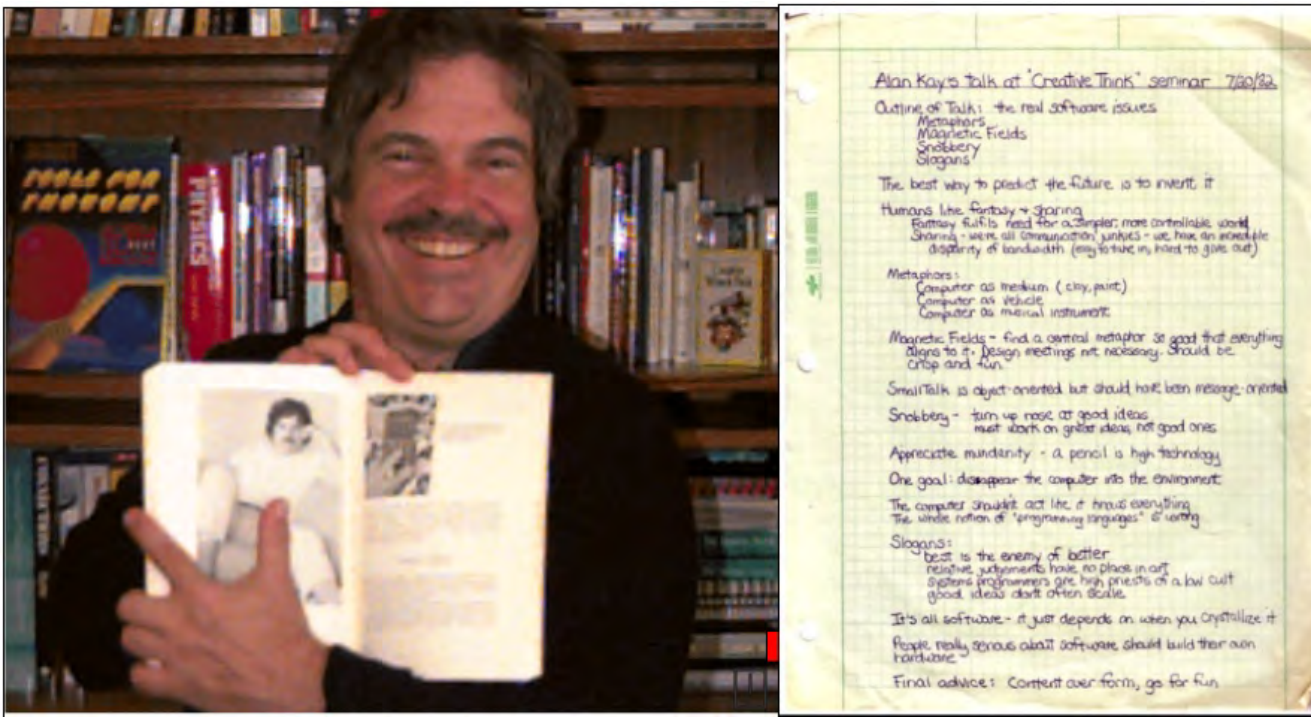
2.5.2 Clear Separation Between Interface and Implementation

Iconicity does not mean that the implementation of the code is iconic, it means that iconicity is expressed in how the user interacts with the system. In order to allow iconic applications to run on standard hardware and operating system, we need to clearly differentiate the iconic part of the application, which is the interface through which the user interacts with the system, and the non-iconic part of the system, which is the implementation that is never seen by the user.

An interesting conclusion that we can draw from the second corollary is that we don't need a special object-oriented language to have iconicity.

3 The Principle of Recursive Design

One of the theorists influenced by Simula was Alan Curtis Kay. Although Simula was the seminal object-oriented language, it wasn't until Kay invented SmallTalk that the term “object-oriented” was first used. In fact, SmallTalk is still considered to be the only true “pure” OO language in use.

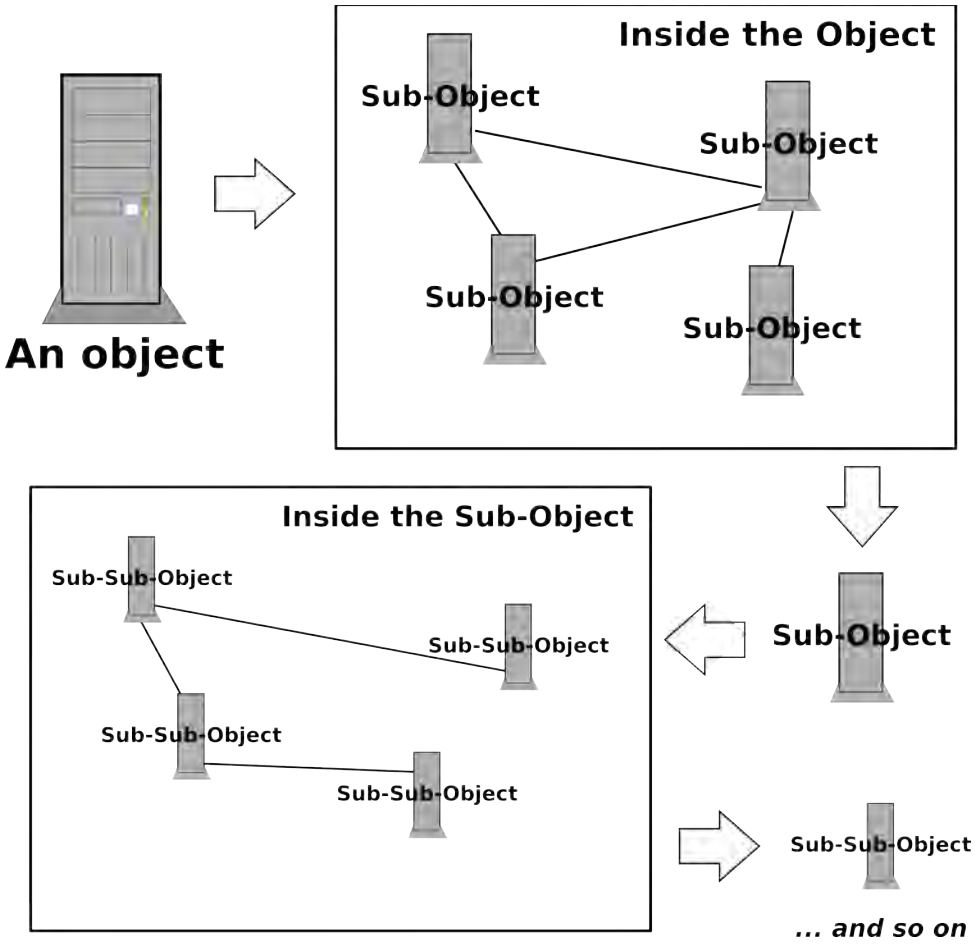


The design of Simula assumed that the implementation of an OO application would be on a mainframe since that’s all there was available at the time. Kay was interested in developing an alternative way of building an implementation by emulating the way complex systems naturally emerged. This implementation would be in terms of a network of computing objects, some of which might be mainframes, but all working in a coordinated manner.

Kay provided a set of ideas about how to actually design an object-oriented system by putting together a set of principles for object-oriented development. Most of these ideas were developed using naturally occurring systems as examples of how one could design software systems.

> *I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning – it took a while to see how to do messaging in a programming language efficiently enough to be useful)*

This approach led to what Kay termed The Principle of Recursive Design.



Instead of dividing a system (or computer) up into functional subsystems, we think of it as a being made up of a collection of little processing engines, called objects. Each object will have similar computational power to the whole and processing happens when the objects work together. However, and this is the recursive part, each object can then be thought of in turn as a collection of sub-objects, each with similar computational power to the object, and so on.

To elaborate a bit more on this idea:

1. A system is built up in layers.
2. Each layer is itself an object.
3. Each layer is made up of a collection of peer objects which provide the functionality of that layer.
4. There is no restriction to the types of objects that exist within a particular layer.

This way of looking at systems allows us to develop some basic principles of software design that are highly effective when we use an object-oriented programming language. It is important to note that even with this approach, there is not really a definition yet of what an object is. We will return to that in the discussion on the object model, but it turns out that we don't really need a rigorous definition of what objects are to use this idea of recursive design.

3.1 Agented Systems

Many of the models Kay used to develop his ideas was a class of problem domains called agented systems.

An agented system is one that is represented as a collection of autonomous and self-contained processing agents that collaborate to accomplish the system's objectives. Agented systems have several basic properties:

1. **They are distributed.** There is no central CPU or processing node. Each agent has its own processing capability.
2. **They are concurrent.** The processing done by each agent proceeds independently of the processing in any other agent.
3. **They collaborate.** Agents work together by collaborating to accomplish tasks. This collaboration takes place by the sending of messages back and forth.
4. **They are heterogeneous.** The agents that make up a system do not all have to be of the same kind. As long as they can send and receive the appropriate messages, they can be of any type.

The dominant structured software engineering paradigm worked very well when automating workflows through a centralized CPU, but it did not work as well when dealing with distributed and concurrent systems. With the rise of distributed computing in the real world through the 1980s and 1990s, this weakness of the structured approach became a liability, especially given the mission-critical nature that software took on during that era.

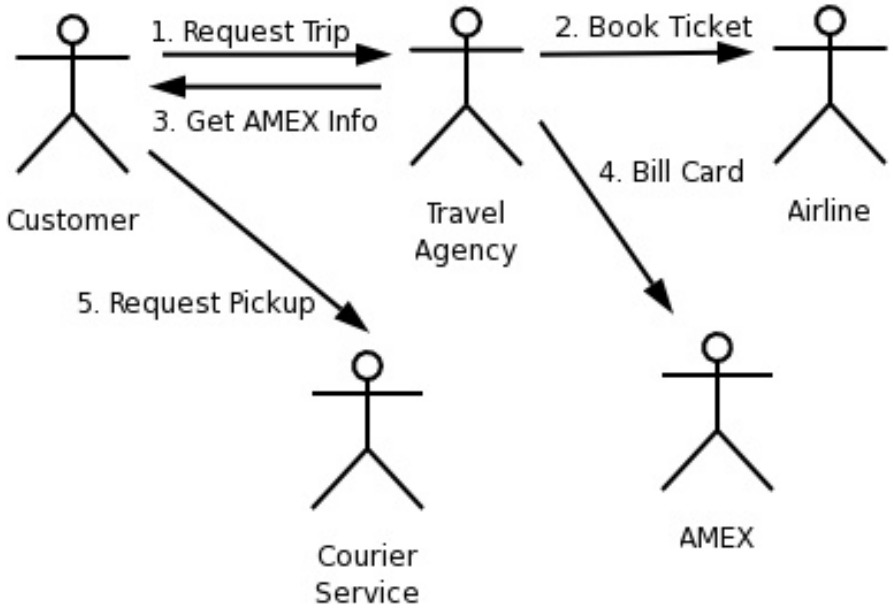
3.1.1 Agented System Example

Consider the scenario of buying a plane ticket from a travel agent. This is an example of an agented system where the agents are: the customer, a travel agency, the credit card company, the airline and the courier service.

The agents collaborate in the following way:

1. The customer calls the travel agency and asks for a ticket to a London.
2. The travel agency calls the airline and books the ticket.
3. The travel agency calls the customer back and gets their credit card number.

4. The travel agency calls American Express and gets the transaction approved.
5. The customer calls a courier to pick up the ticket.



This example illustrates a couple of the ideas already presented.

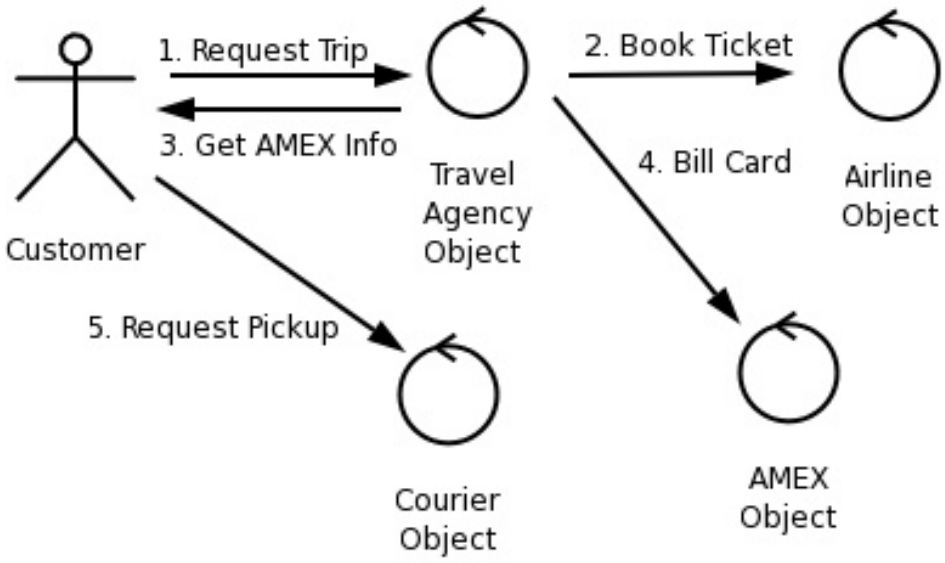
First, we can see the principle of recursive design because the credit card company, airline and courier company are “objects,” their functionality is actually delivered by the contributions that sub-objects make (the courier driver, dispatcher at the courier company for example, credit card authorization centre, the individual travel agent and the airline booking representative).

We can also see the separation of interface and implementation. For example, the customer can call the courier service and request a delivery without ever having to know anything about the courier service and how it works. All that the customer has to know is the interface or how to speak with the person at the other end of the phone.

3.1.2 Designing an Agented System

Now we see iconicity in play. In the object-oriented approach, we write a program that iconically automates the agented activity that we just described. We do this by creating software objects, each object corresponds to an agent and each object is possibly running on its own processor (i.e. the objects are distributed over a variety hosts).

If we automate the system as depicted in the above diagram, our software system is a collection of objects coordinating their activities by sending messages back and forth to accomplish the goal of getting the customer a ticket to London. We have iconicity since we can recognize which software objects correspond to which agents because the system is designed to look like the agented system in original example.



This system is iconic because it resembles what it automates. Non-computer literate domain experts should be able to understand how to interact with the system because it resembles something they already know how to do or use. But even more importantly, if we automated only part of the system, the airline object for example, iconicity means that the other agents would be able to interact with it analogously to how they interacted with the original airline agent.

Recursive design also comes into play as we try to figure out how to design the agent objects.

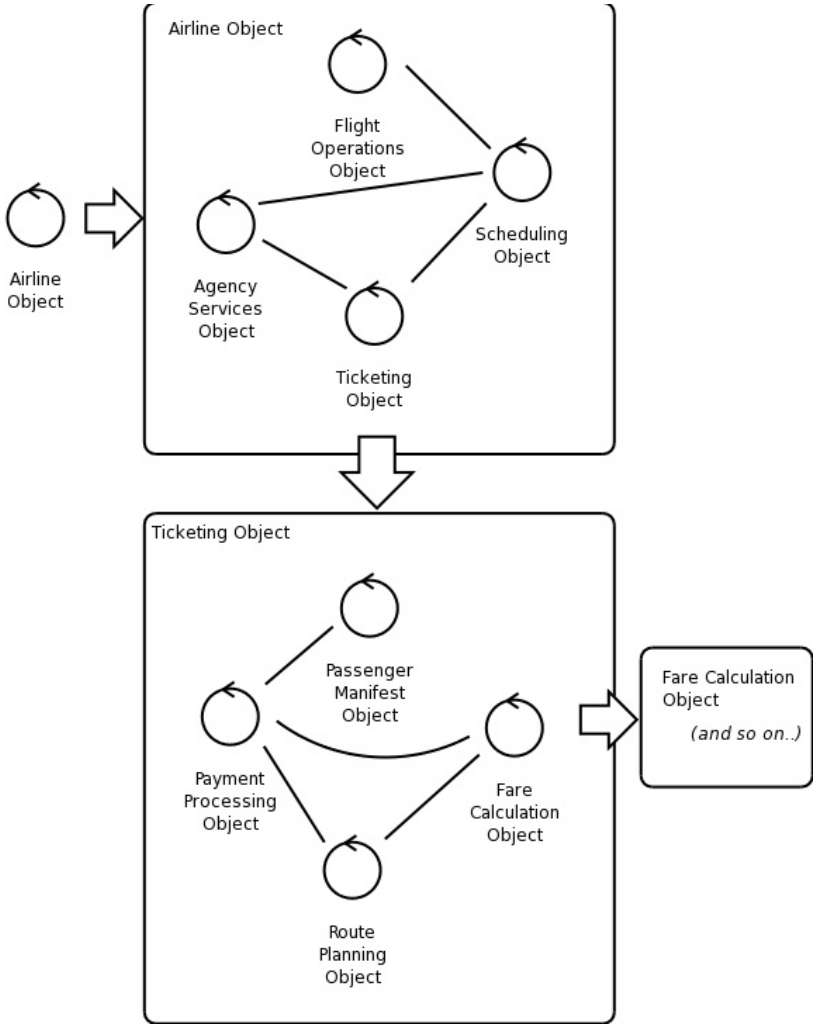
In the diagram, we see the principle of recursive design at work in the ticket purchase agent object called an “Airline”. When we open up an Airline object, we find it is made up of a collection of sub-objects, each of which is responsible for delivering some part of the overall functionality of the Airline object.

For example, the Ticketing object is responsible for doing ticketing, but when we open it up, we find it is made up of sub-objects, each of which specializes in some aspect of the Ticketing object’s functionality.

You may have noticed that at each stage, the collection of sub-objects that makeup an object is not arbitrary. In well-designed object-oriented systems, the specific sub-objects comprise a functional decomposition of the original object.

3.2 The Principle of Recursive Design

The Airline object in our agented system example is more like a layered system with a complex internal structure. Since this problem of “layering” within agents is a common one given the kinds of systems that OO models, it is addressed by a significant reinterpretation of the principle of functional decomposition, the basic axiom of the structured approach.



This does not mean that the structured approach cannot deal with hierarchies, just that it takes a lot more work on the part of the programmer since it is not fundamental to the way the structured approach works.

The key principle of OO, which was originally formulated by Alan Kay, the inventor of Smalltalk, is The Principle of Recursive Design

> *The structure of the part mirrors the structure of the whole.*

Instead of dividing a system (or computer) up into functional subsystems, we think of it as a being made up a collection of little processing engines, called objects. Each object will have similar computational power as the whole, and processing happens when the objects work together. However, and this is the recursive part, each object can then be thought of in turn as of a collection of sub-objects, each with similar computational power to the object, and so on.

Just like a little change in the axioms of geometry (“let parallel lines meet”) can have far-reaching repercussions, so does changing the basic definition of what a system is. The more traditional model is called a functional decomposition where we identify the major processes and divide up our system along strictly functional lines. The functional decomposition approach works well in the structured world where our main building blocks of applications are subroutines – little packages of functionality expressed as algorithms. Instead, OO introduces a design principle that allows us to simulate the hierarchies out in the real world, as we need to do if we follow the principle of iconicity.

However, we haven’t completely thrown away the idea of a functional decomposition OOP, we just use it recursively, as we will see in a moment. In other words the sub-objects are not arbitrary, the divide up the functionality of the original object into cohesive packages.

4 The Axioms of Object-Oriented Development

In this section, we are going to look at the six axioms of object-oriented development. These were derived by the author from Alan Kay's six principles of object-oriented programming. Kay's original principles were designed for SmallTalk programmers, and I have taken the liberty of modifying them slightly to make them more generic.

The reason for looking at these axioms early on is that they are very helpful in doing object-oriented development modeling

1. They provide heuristics for understanding what it is that needs to be modeled in terms of classes and functionality.
2. They provide guidelines for organizing the data and functionality into coherent packages and classes.
3. They provide guidelines to help you decide how what is needed in terms of the interfaces and interrelationships, both delegation and inheritance, between classes.
4. They provide a basis for developing a program or application architecture.
5. They are the motivation for specific language features in many object-oriented programming languages.

We will go through this quite quickly because, if it has not been apparent up until now, object-oriented is in many respects just a formalization about how people tend to think about the world. You have been doing object-oriented all your life and never knew it.

4.1 First Axiom

> *Everything can be modeled as an object.*

That's pretty much it. The important thing to note here is that this axiom has two corollaries that are very useful in doing object-oriented development.

1. People tend to represent important things, events, actions or concepts in an application domain as objects. If we model the application domain iconically, we should be able to express everything as an object of one kind or another.
2. If everything is an object, then the basic programming language construct we need to do iconic system development using recursive design is an “object.”

I've restated this axiom a bit differently that Kay's original stronger form “Everything is an object” since it was clear that he didn't mean, as some have erroneously claimed, that everything is only an object, but rather he meant that we can represent anything as an object.

4.1.1 Objects in the Application Domain

Humans are “wired” at the neurological and cognitive levels to perceive the world around them in terms of objects, a fact that has been verified by cognitive science, linguistics and neuroscience. Why should that be so?

Objects are what we talk and think about; we talk or think about what is important to us. When we want to talk about something, we make it into an object.

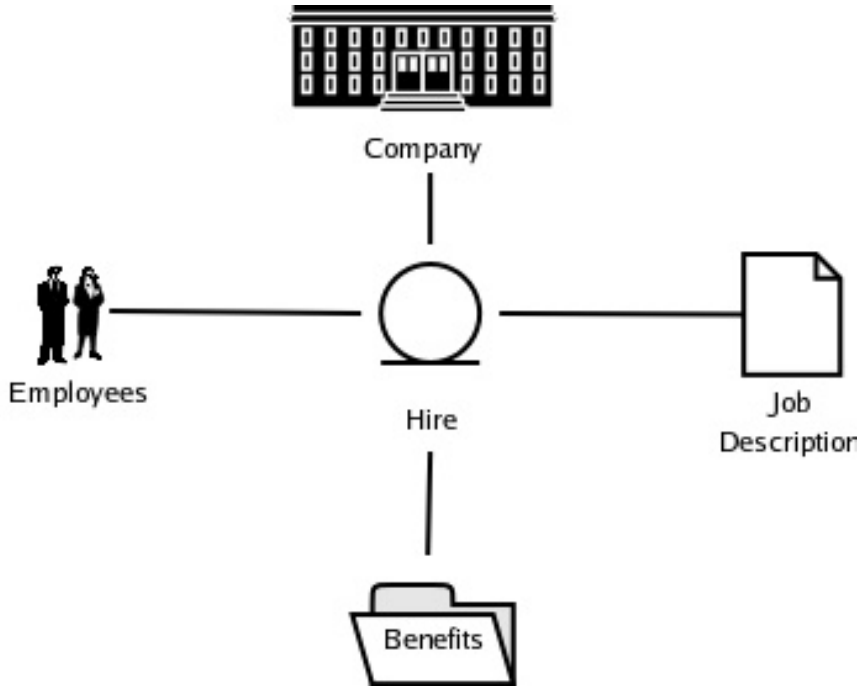
The principle does not state that objects exist independently of the observer or that objects have any kind of objective reality, it says that we organize the world around us into objects. The principle doesn't say that there are only objects or that something is an object and nothing else, but rather that anything can be depicted, described or thought of as an object.

For example: a person is born, which is an action, but we can speak of “a birth.” There is nothing unusual is the sentence, “The blue of his eyes was striking.” Yet, we are talking about blue as if it were an object.

This process of making things into objects is so common that every language on earth has a way of creating nouns from anything else, a process called nominalization.

4.1.2 Transaction Objects

A commonly occurring type of object we find in object-oriented domain modeling is a transaction object. This is some event, usually a transaction of some kind, that is central or critical to a system or to a business and which, because it is important, is thought about and talked about as if it were a physical object.



For example, if two people get married, then we have a transaction object called a marriage. When a marriage ends, we have another transaction object called a divorce.

4.2 Second Axiom

> *Systems are collections of objects collaborating for a purpose and coordinating their activities by sending messages to each other.*

Objects work together by making requests of each other for services, and these requests are made by passing messages. Systems can be viewed as communities of agents. An arbitrary collection of objects is not necessarily a system since it lacks organization.

Systems tend to be gestalts – the system is more than the sum of the objects that make it up. The "more" is the layer of organization that coordinates the objects. We often see a phenomenon called emergent behaviour in systems when the system displays behaviours that are not present in the individual objects that make up the system.

Systems, like objects or colours, don't actually exist "out there," but are an artifact of how our brains process and organize information. This is similar to the fact that there exist different wavelengths of light, but what we call colours only exists as the result of our neural processing of those wavelengths. Similarly, relations between objects exist, but the perception, classification and interpretation of those relationships – that which makes a collection of objects a system – is an artifact of our cognitive organization.

Generally, you will find that a collection of domain experts will agree fairly well on what the objects are that make up a system. Where they tend to disagree it will usually be on the nature of the relationships between the objects that the system is made up of, or the relationships between the system and its environment.

There are two perceived qualities of a collection of objects that tend to lead observers to describe the collection as a system:

1. **Organization:** There is a coordination of activity or a perceived structure to the relationships between the objects.
2. **Goal Directed Activity:** The activities of the individual objects are directed towards meeting the goals the system has as a whole rather than those the individual object might have. This goal-directed activity is also described in terms of the object acting as part of a coordinated whole rather than individually.

4.2.1 Some Design Implications

When we consider the first and second axiom together, we can consider a system itself to be an object. This means that we have a sort of "zoom out" aspect to an object-oriented application. We can take any system of objects and refer to them as a whole single object.

The technical term for this is encapsulation where we package up a collection of objects into a single unit that we treat itself as an object.

4.3 Third Axiom

> *An object can have an internal structure composed of hierarchies of sub-objects.*

In other words, an object can be thought of as a system (axiom 2) where the functionality of the object is “parceled out” or delegated to sub-systems, which themselves can be seen as objects (axiom 1).

This axiom is critical to managing complexity as a design principle. This axiom is very applicable to design situations where an object design is starting to become very complex. We can manage the complexity by identifying cohesive chunks of behaviour within the original object, then creating new sub-objects which “package” up or encapsulate those behaviours. Then the original object becomes a system made up of these newly defined sub-objects.

An object can have an internal structure made up of a system of objects. Conversely, given a system of objects, we can view the system as being the internal structure of a new “system” object that we define.

Notice that the first three axioms are all interconnected – and all directly derived from the principle of recursive design.

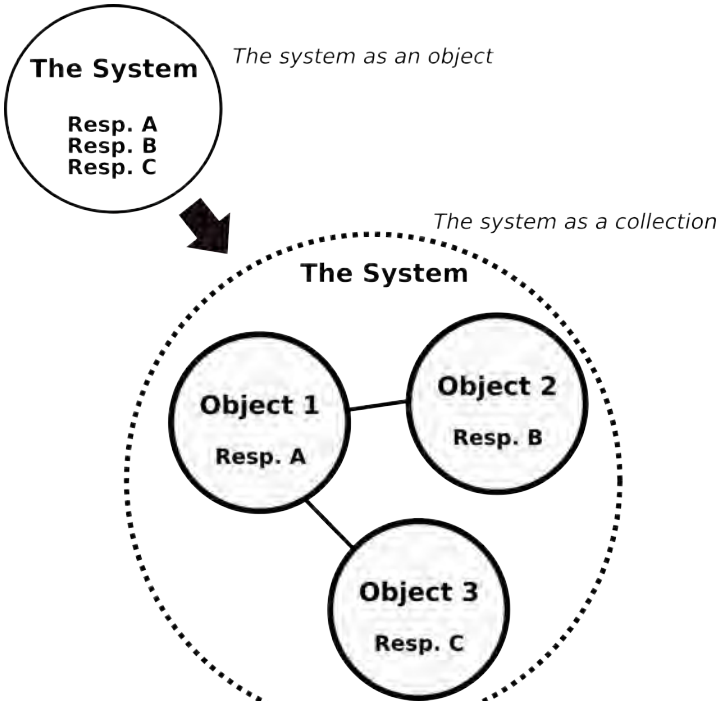
4.4 Fourth Axiom

> *Objects within a system have individual responsibilities, the responsibility of a system as a whole is distributed across the objects that make up the system by delegating specific responsibilities to individual objects.*

This is the first axiom that tells us what defines objects in a sense – an object has a responsibility that contributes a well-defined set of functionality to the system that it is part of. Conversely, the functionality of a system is parceled out to the objects that make it up. Objects can be thought of as bundles of responsibility.

As an analogy, consider a project team made up of ten people. We can think of the team as an object in the sense of “The team is starting a new project” or “The team has been very productive,” but we know that the “team” object is really a collection of objects (people) and not a physical object itself. However, we also know that the team is made up team members (axiom three) and, like most well-organized teams, the various team members are allocated different roles and responsibilities, each of which contributes to the overall functionality of the team.

By a functional decomposition, we mean that each object specializes in some aspect of the overall of the system's functionality – the objects do not all have the same responsibilities. Looking again at a team, it is not effective to have all the team members be given exactly the



same responsibilities but rather the various team members specialize. The team functionally decomposed into types of team objects like business analysts, programmers, testers, hardware technicians, secretaries and others.

4.5 Fifth Axiom

> *An object presents an interface that specifies which requests can be made of it and what the expected results of those requests are. This interface is independent of the actual internal workings of the object – remember the separation of interface and implementation from before.*

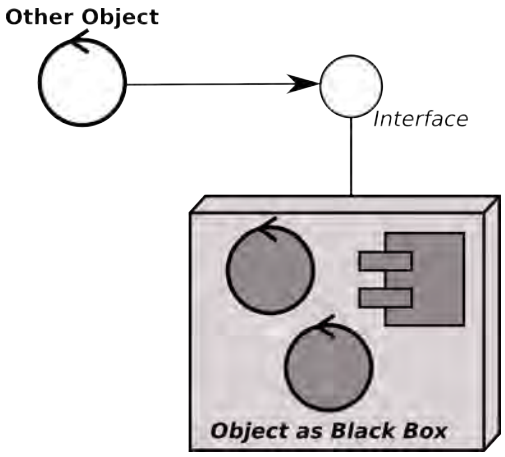
The interface presented by the object is often referred to as the object's "contract." This principle is one that plays a very important role in the design of the classes and objects.

An interface is what is presented to the other objects. In its simplest form, it can be thought of as a list of the messages that the object understands, and a description of how the object will respond as a result of receiving a particular message.

In the 1970s, a linguist named J.P Grise wrote a paper in which he talked about the "speech contract" or an elaborate set of rules that govern conversation. He pointed out that while we all can follow the rules, it is difficult for us to list what they are. Two of the rules he cites are: "Speak a language we both understand" and "Answer questions or state your intention not to answer." Violating either of these rules results in chaos in a conversation. Try it and see for yourself.

The speech contract is an example of an interface. In fact, one of the terms that we often find used in OO when we define interfaces for objects is “programming by contract.” That is, the interfaces are contracts.

What mean by the interface being a contract is if another object sends a message that is in the interface, the receiving object will respond in the manner specified in the interface. Once



the interface is defined, other objects really don’t need to know anything about how an object implements the interface in order to interact with it..

For example, if someone asks you how old you are, the interface specifies a response to be provided in years since birth, rounded down the nearest whole year. How you get that figure – remembering it, calculating it from your birth year, or looking it up on your driver’s license or what else we do internally – in what we call the implementation is of no interest to the person asking.

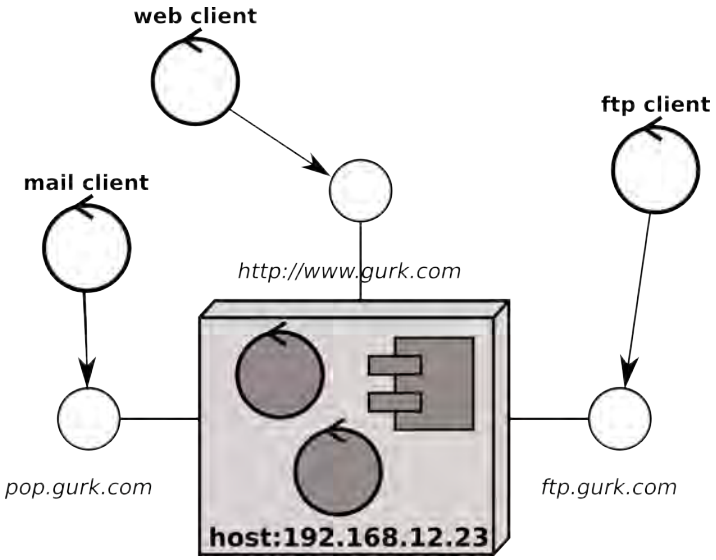
An object can be thought of as a black box, where only the interface is visible to other objects.

4.6 Sixth Axiom

> *An object is of one or more types. A type includes a role the object plays, a set of responsibilities and an interface. Each type may be derived from a hierarchy of types.*

This is one of the most important concepts in object-oriented since it ties together the ideas of roles, interfaces and responsibilities under the concept of “type. This axiom is different than the preceding axioms because it talks about types while the other axioms are concerned with structure.

This axiom allows for what we call multiple inheritance, which means that a given object may be of more than one type. For example, a person can be of both type parent and type employee. However, there is an ongoing debate in the object-oriented community whether multiple inheritance is not a good design practice and if single inheritance should be used exclusively. We will explore this debate a bit later on. For now, we will use multiple inheritance since it makes OOAD much more natural and iconic.



In the diagram we see an object of type “host” which is actually three “types” of host; a web server, and ftp host and a pop mail host. Each one of these types is associated with a set of responsibilities – render web page, list directory, route mail to user for example – and an interface that clients talk to while the object is acting as (or the role of) a web server or an ftp host or a pop mail host.

5 Objects

The concept of “object” is central to the object-oriented paradigm but the concept can’t be defined clearly because it is too fundamental to the way we think – our brains are wired to interpret the world in terms of objects. This is a bit awkward because object-oriented is about modeling systems in terms of objects – designing and building objects – and ensuring that we have built our systems and objects correctly.

The best practical definition of an object belongs to Grady Booch’s where he defines an object as

> *Something you can do things to.*

A good test to see if something is an object is to see if it can be measured or evaluated in some way.

The issue of whether objects exist or not independently apart from the observer is a centuriesold ongoing philosophical debate that is entirely beyond this course. However, from a practical and experimental point of view, we know that the objects users think of having an external, objective existence are in fact subjective and may vary from observer to observer depending on their degree of “solidity.” Generally, there is pretty good agreement about the “things that are out there” among a group of stakeholders.

5.1 What is an Object?

There is no formal object model like we have for Euclidean Geometry. Instead, we have contributions from a number of sources that have a common conceptual similarity. Because of the fundamental nature of the concept of an object, trying to come up with a definition for an object is about as useful as trying to come up with definitions for other intuitive concepts that refer to wired-in portions of our cognitive systems (like emotion and colour).

Here are some samples:

> *An abstraction of something in a problem domain, reflecting the capabilities of a system to keep information about it, interact with it, or both; an encapsulation of Attribute values and their exclusive Services. (synonym: an Instance) [Coad and Yourdon]*

> *An object is a visible or tangible thing of relatively stable form; a thing that may be apprehended intellectually; a thing to which though or action is directed. It is a unit of structural and behavioural modularity which has properties. Each object can be characterized by the set of operations that can be performed on it and by the set of operations that it can perform on other objects (either of these two sets may be empty), and by the set of states which it goes through during its lifetime. Each object encloses components, i.e., objects of which it has been constructed, and the operations that can be performed on it, which it makes*

> *available for other objects to perform. When these characterizations are accurate, we recognize an object: a self-contained thing which exhibits behaviour. [Colbert]*
>
> *An object is a person, place, or thing. An object may be physical or conceptual... The idea is that an object is a single entity or notion. Each object is a unique individual. An object may be related to or made up of other objects, but each object is unique. [Embley]*
>
> *The object is an Instance of a Class. It is the computer representation of a real world object. The object has Properties as defined in the class of which it is an instance. [IBM]*
>
> *Anything to which a concept, or object type, applies – an instance of a concept or object type. In OOPLs, it is any instance of a class. [Martin and Odell]*

5.2 Describing Objects

We can describe what all objects have in common even without having a clear definition of "object." All objects have:

1. **Properties.** These are the properties of an object, often describes as "what an object knows."
2. **Behaviours.** The functional capabilities of the object, often referred to "what an object does."
3. **Identity.** Something that uniquely distinguishes a specific object from all other objects of the same type.
4. **Type.** We will discuss types in the next section.

Objects that make up a system are independent collections of data and functionality. It is possible that each object has its own internal processing capability and therefore its own version of the class functionality and its own data which may not be the same as the data held by other objects of the same type.

6 Classes, Abstraction and Objects

The next step up in our discussion is the idea of a class, a concept we model with class diagrams. Every object is of some type, and in object-oriented, we model types as classes. However, we do have to be careful because there are two different concepts of “types” that we model in OOAD.

1. **Domain Classes.** These correspond to the types that have conceptual reality in the real-world application domain. The structure of domain classes uses prototypes that we have to discover by investigation and analysis. These domain definitions tend to be somewhat fluid and fuzzy. We often find disagreement among stakeholders as to what these classes are.
2. **Design Classes.** These are the classes we define to build our system. These are the classes that will exist in the software and do not necessarily correspond to the domain classes in our problem domain.

Domain types are defined by a process of abstraction in which common features of a collection of objects are used to create a prototype or typical example of that type. Once a prototype is agreed upon, objects that are encountered “out there” are compared to the prototype to see if they resemble it enough to be considered to be an object of that type. This is not always a yes/no answer but may be a matter of degree.

For design classes, we write a class definition to be the template that is used to create objects of that type. Unlike the real world, in the design world, there is no “fuzziness” to what type a class or type is.

In either case, real world or design world, when we have an object of a given type, we say that it is an instance of that type. In the design world, to instantiate an object is to create an instance of an object from a class definition.

6.1 Abstraction, Prototypes and Domain Classes

In the past, there was a discontinuity between the way that object-oriented dealt with classes and the way the rest of the sciences thought of types or categories. The classical theory or objectivist model stated that classes existed independently of the observer and membership in a class was a matter of checking off all the properties that an object of a specific class should have. An object either belonged to a class or it did not.

However, this changed dramatically over the last 30 years, as described by George Lakoff in *Women, Fire and Dangerous Things: What Categories Reveal About the Mind*

> *It was Eleanor Rosch who developed what has come to be called “the theory of prototypes and basic level categories”. Before her work, the classical theory was taken for granted, not just in psychology but in other disciplines. n a series of*

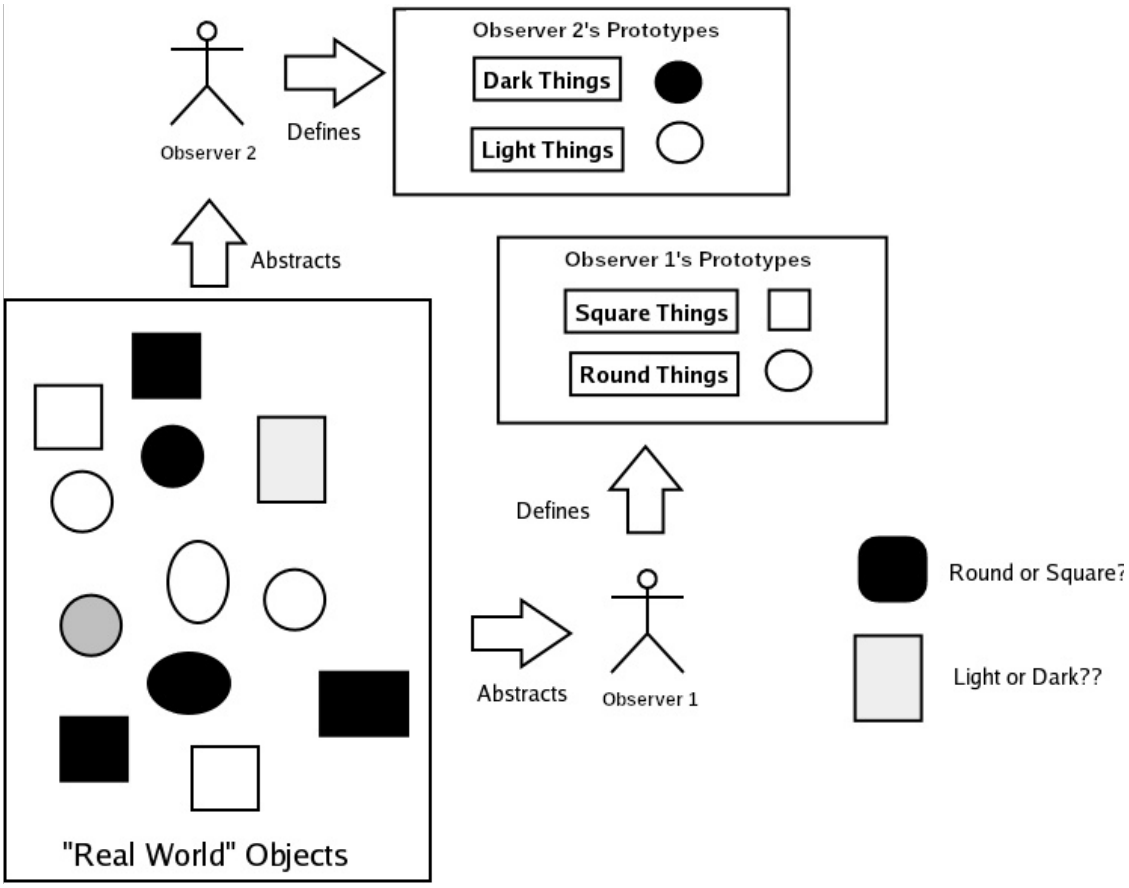
> *electrifying papers, Rosch and her associates presented an overwhelming array of empirical studies that challenged the classical view. These experimental contributions are generally recognized by cognitive psychologists as having revolutionized the study of categorization within psychology.*

Rosch's prototype theory is consistent with the object-oriented way of thinking in terms of classes and abstraction. Her work validates the object-oriented model.

6.2 Abstraction

We perform abstraction, also called generalization, based on three kinds of similarity among objects.

1. **Appearance:** Objects that are perceptually similar. e.g. trees, rocks, circles, bangs, stinks.
2. **Interaction:** Objects that we interact with in similar ways. e.g. tools, vehicles, food.
3. **Relationship:** Objects that exist in similar relationships with other objects. e.g. employees, parents, furniture.



The implication of this process is that there are no objective types out there. Each person creates their own types guided by feedback from the other people they interact with. A given type may be the result of any combination of these three abstraction processes taking place at once. This also implies there is no “right” answer as to what makes up the types in an application domain.

6.2.1 The Fuzziness of Domain Types

It is important to remember that domain types exist only because they are defined by the observers. We cannot speak about types without reference to the particular point of view that created that type. This is important to remember when we are drawing our class diagrams when doing domain modeling – there is not a “right answer in the back of the book.”

People create types. The process of abstraction is also hardwired into our cognitive makeup. It is a capability displayed only by humans, and to an unknown degree by gorillas, orangutans, chimps and dolphins. However, in order to communicate meaningfully with each other, groups of people derive a community consensus about the various types are that they want to exchange information about.

Abstraction is a mechanism that allows us to process information in real time and to compensate for the relative slowness of our brain’s processing capabilities. You can voluntarily disable the abstraction process, but it takes training and effort to do so. There are Zen exercises that involve “seeing things as they are,” for example. For more information, a good book is *Drawing on the Right Side of the Brain* by Betty Edwards who describes this process in the context of art.

6.2.2 Design Classes

We will be looking at design classes a bit later on in this course, however there are a couple of points that we need to emphasize about them.

We define design classes. They are not “out there” like domain classes, we invent whatever classes we want to or need to in the design world.

Iconicity does not mean that our design classes have to resemble in any way the domain classes that are out there. Iconicity means that the users should see things that look like the analysis classes they know in the interface. This does not mean there is iconicity at the design level.

For example, in a real-world card game, there is a role played by a person called the “dealer” who distributes the cards. There is no reason to have a dealer in a virtual card game since virtual cards, unlike those in the real world, can deal themselves out. There is no need for a virtual dealer” class.

7 Modeling Structure with Classes

A standard way of modeling classes is with UML Class Diagram. The amount of material that exists on UML is staggering, and a lot of it free on the Internet, which means that including generic “how to draw UML diagrams” material in this manual is redundant.

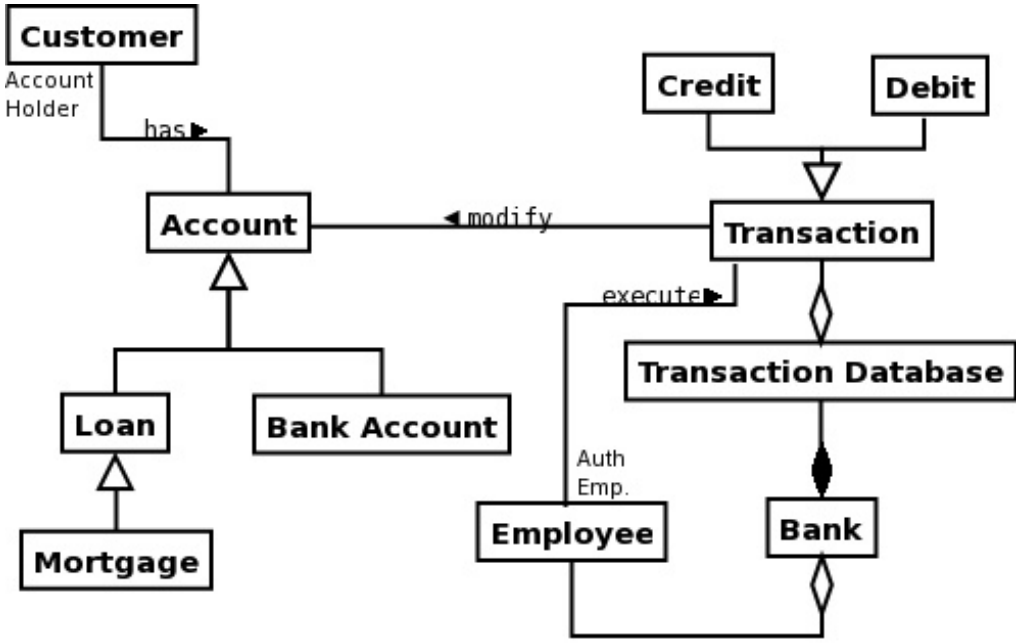
A class diagram describes classes and the static relationships that exist among them. Class diagrams can also show the the operations and attributes displayed by the objects of that class’s interface.

As we move through a development project, the kinds of classes that we are modeling goes through three distinct phases. At each phase, the definition of the class becomes more fixed, clearly defined and more detailed.

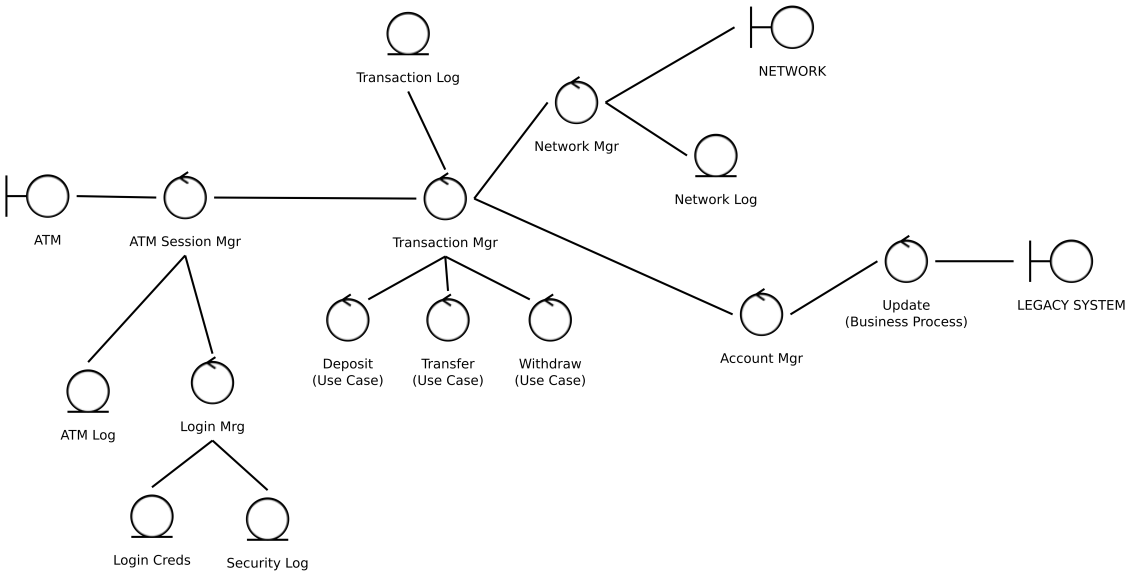
1. In the early phases of a project, we are interested in identifying the domain classes that represent the objects that make up our system from the point of view of the domain experts – the people that know the problem domain. We are interested in the logical relationships those classes have with each other. This is a coarse-grained, high-level view of the types of things in the domain. At this level, classes tend to be fluid and resemble the types that have psychological reality to the users and stakeholders of the system.
2. As we move into analysis (developing the solution) phase, we define a set of high-level analysis classes that will form the foundation for our design based on the functionality required to satisfy the functional requirements. We are not interested in this point about the details of the class interfaces. Our focus is on the responsibilities of the class and its general behaviour and characteristics (often called “what a class knows and does”). We are also interested in the way in which groups of objects (as instances of the classes) collaborate to get things done.
3. As we move into the design phase, we are interested in how a high-level design class is implemented in a particular architecture. We are now focused on the details of how classes are related, the details of the interfaces the classes expose including the specific syntax of the messages they exchange. We are also interested in the “choreography” or sequencing of messages between instances of various classes.
4. As we move into the implementation phase, we tweak our class designs in order to make them into good class definitions according to what our programming language requires in its class definitions. These classes will be turned into code so are concerned with Java or C# constructs, like which constructors need to be defined in a class.

Since our needs when modeling classes change through a project, there tend to be four types of class diagrams that we employ. Each type of class diagram shows more detail about the objects.

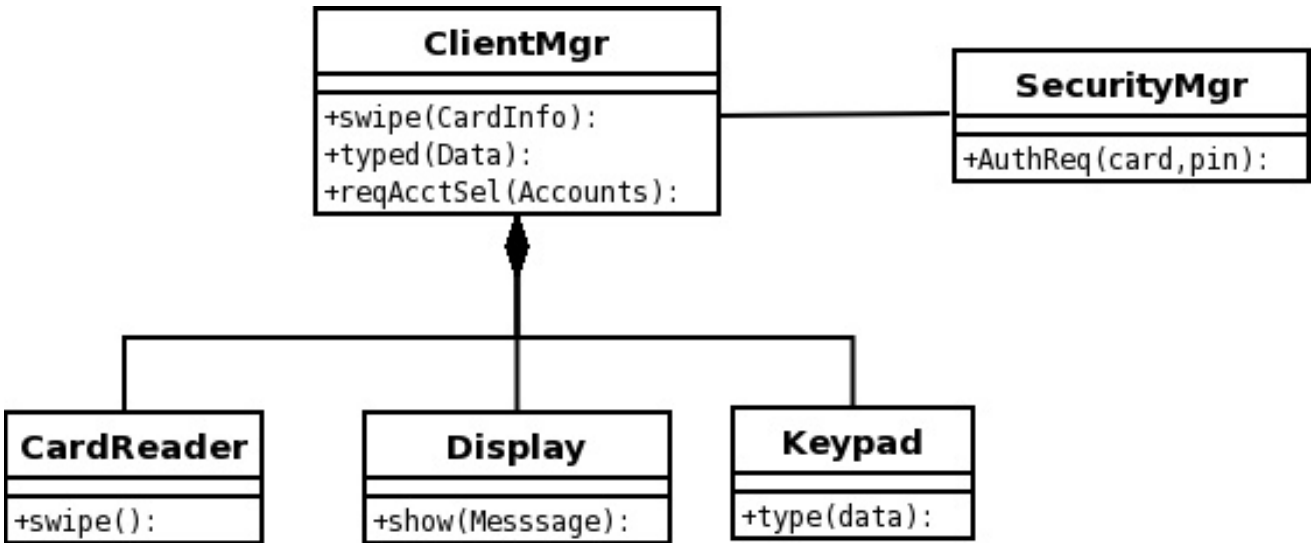
1. **The Sketchy Class Diagram:** Also called the domain class diagram. This diagram is the simplest because it just shows the domain analysis classes and their relationships. This is the one we are going to look at in this module.



2. **The Analysis Class Diagram:** This shows our architecturally dependent classes that will be used in the high-level design. These class diagrams do not show detail but are usually drawn with standard boundary, entity and control stereotypes. The emphasis is primarily on defining the roles and responsibilities of each class.



3. **The Design Class Diagram**: This is the class diagram that most UML courses start with. It defines the interfaces of a class and enough detailed information about the relationships between classes that a programmer has all the design information they need to start writing code. We could think of this as a blueprint from which one can construct objects.



4. **The Implementation Class Diagram**: This diagram documents the actual code design of the class as implemented by the programmer. It is the design class diagram with programming language-specific information added and a record of what changes were made from the design class in order to implement that class in code. For example, if the design class diagram specifies multiple inheritance, then a programmer has to change the class structure to implement those classes in Java since Java does not support multiple inheritance.

7.1 Class Relationships

We recognize three kinds of relationships between classes that are derived from our basic axioms of object-orientation. These relationships are known by several different names but are standard across object-oriented and have support in all object-oriented languages.

1. **Inheritance.** This is also called a class decomposition, is-a relationship or an is-a-type-of relationship. We differentiate between two kinds of inheritance called generalization and specialization.
2. **Aggregation.** This is also called object decomposition or an is-a-part-of relationship. We also differentiate between two kinds of aggregation: containment and composition.

3. **Association.** This is considered to be a logical or semantic relationship between two classes.

Historically, the semantics and notation of the class diagram is derived from the entity-relationship diagram; however, the first two relationships above have special notation because of their importance in object-oriented modeling.

7.2 Inheritance

Inheritance is the converse of abstraction in a sense. We won't try to give some formal definition of inheritance since it is really a formalization of what you do all the time. Instead, we will just describe how it works.

7.2.1 Specialization

If we have an existing class“telephone,” some telephones might form a subset of telephones because they are telephones with something “extra.” For example, historically we had telephones, then“cellular telephones” were introduced as a special kind of telephone.

We also have “watches” and we have a special kind of watch called “digital watches.” In the domain, we usually find this specialized class has some sort of adjective to indicate it is a specialization of the original class. We can have an object of type watch or an object of type digital watch. Specialization says that since digital watch is a type of watch, it does everything a watch does.

The problem is that in the domain classes, different groups of stakeholders may have different ideas of which class is the specialization. Analogue watches may be "normal" watches for the over forty-year-old crowd, but the under thirty-year-old group have grown up with digital watches as the norm, so non-digital watches are, to them, the specialization of the normal digital watch type.

In specialization, we can instantiate objects of the type of the original class (watch) and objects of the type of the specialized class (digital watch or analogue watch).

7.2.2 Generalization

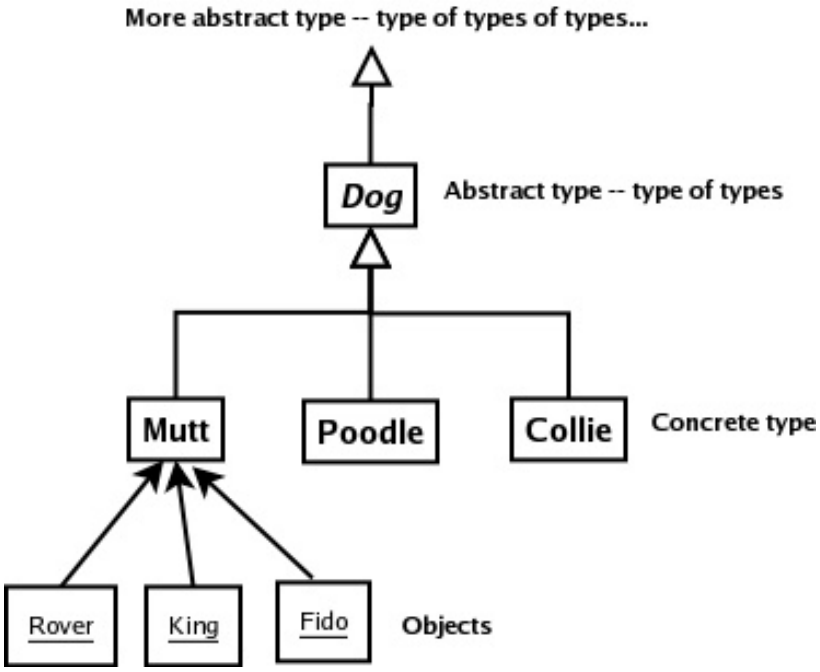
Generalization is the opposite of specialization. If we have digital watches and analog watches as two independent types, we can see that there is something common between them. To capture this commonality, we create a new class “watch” that is a superclass of the original two classes.

We generalized digital watches and analog watches to create a new type because it is impossible to create an object of type watch. Objects of type “watch” do not exist – a watch has to be either a digital watch or an analog watch. In this case, we call the class “watch” an abstract class since we cannot instantiate objects directly from it.

7.2.3 Abstract Types

If we think of a class itself as an object (our first axiom) then we can group classes into classes-of-classes. This is the start of our abstraction hierarchy.

How do we know something is an abstract class? Because the class description is generally



not complete enough for us to specify a well-formed object of that type. Or in plain English, objects of that class just don't exist.

Consider "bank account". It is an abstraction of the types "savings account" and "checking account." Users know this. Try going to a bank and opening a plain bank account – Insist it be neither a checking nor savings account, just a plain bank account. Security will quickly escort you out and advise you never to return. Everyone knows a bank account has to be either a checking or savings account because those are the concrete types we actually instantiate objects from.

"Bank account" is an abstract class. The same is true of the types "dog", "tool" or "food." When we want to create an object of that type, we have to ask "what kind of tool?" or "what kind of food?"

7.2.4 Why have Abstract Classes?

Abstract classes, and prototypes in general, are often used when recording business rules or system knowledge in a form like a script or user story. We tend to store and communicate information as scenarios – this is called "episodic memory".

> *To get money out of an account, a customer goes to an ATM, swipes their card, enters their PIN, selects withdraw, selects the account and amount, then takes their cash and receipt.*

The prototypical customer above is an arbitrary object of the abstract type customer, as are all the other objects mentioned. In other words, we use an abstract class as a kind of placeholder in articulating a business rule or a description of a transaction that may apply to a number of similar concrete classes.

When various classes overlap in the ways we looked at earlier for objects (appearance, interaction or relationships) we often capture that commonality by abstracting it out and creating an abstract class like customer or parent or employee.

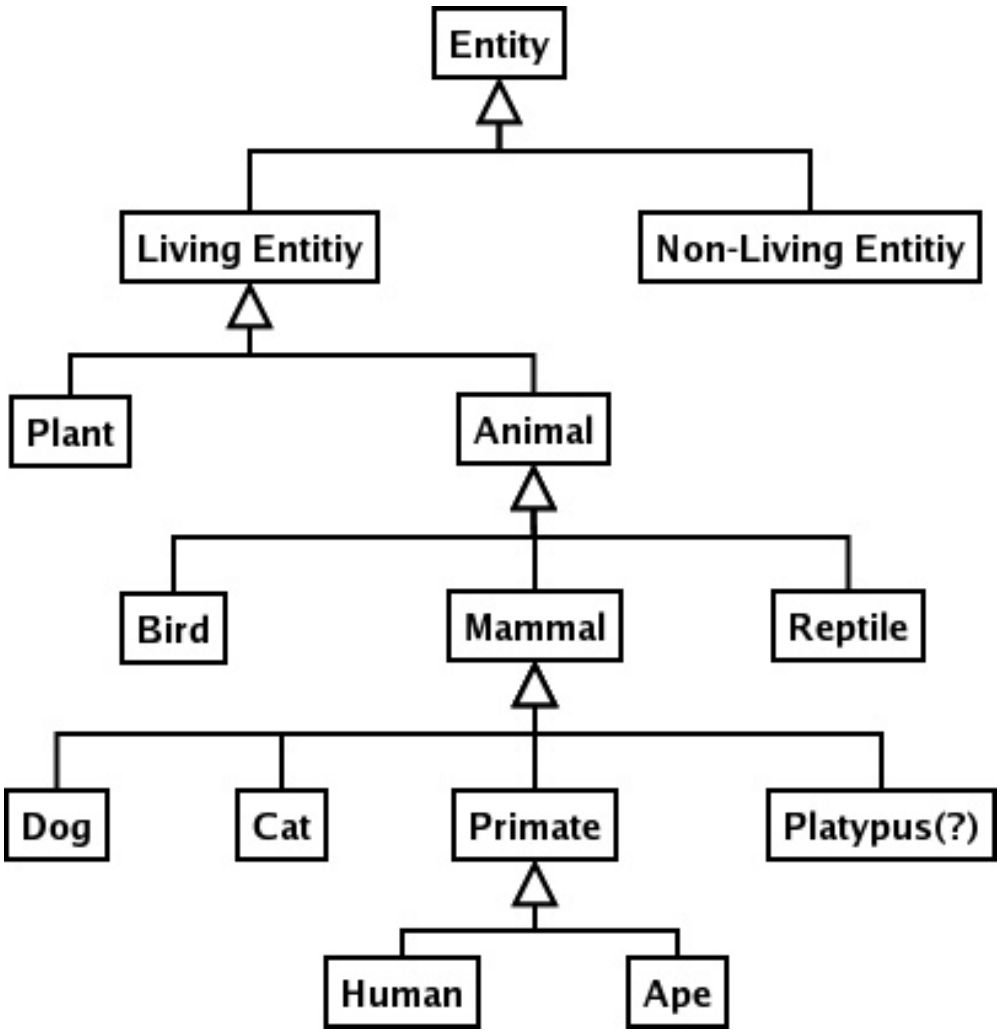
7.3 Types or Objects

It's sometimes hard to decide if we are dealing with classes, as in the customer and ATM example above, or actual objects. Remember that we can "do things" to objects. More specifically, an object has properties that have values that we can measure or report on.

In the preceding scenario, we cannot meaningfully answer questions like "What was the customer's name?" because there is no person object and properties, like name, have to be associated with an object. The customer in the example was not an object but a prototype that stood for the type "customer." This comes back to the idea that if you can measure it or evaluate its properties, then it is an object (this is a rule of thumb).

UML humor: "A woman gives birth every 30 seconds in the US. Someone has to find her and stop her." Can you explain the joke above in terms of mixing up classes and objects?

7.4 Inheritance Hierarchies



As we create more abstract classes, we eventually construct a class hierarchy which is called a class decomposition. Depending on our orientation, this is either an abstraction hierarchy or an inheritance hierarchy; it just depends on whether we are going up or down the hierarchy.

Each class in the class hierarchy inherits everything from all the classes above it. This is expressed as an “is-a” or “is-a-type-of” relationship.

In the hierarchy in the diagram, a human is a primate which is a mammal. Mammals have are warm-blooded and suckle their young therefore so do humans. If our hierarchy is wrong (is a platypus a mammal?), we can make inheritance errors; like assuming the platypus gives birth to live young.

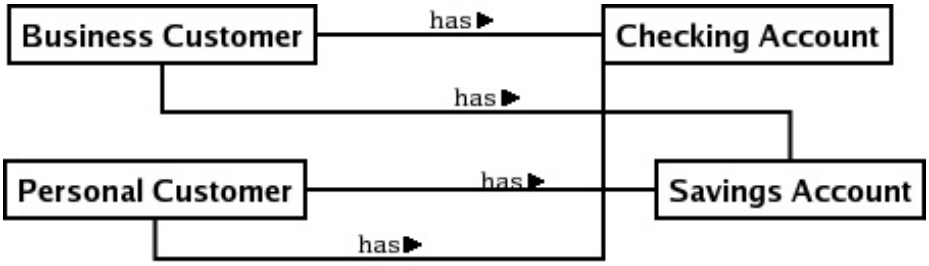
This is something you do all the time – this is just a formalization of your built-in way of organizing the world around you. In OO terminology, primate is a superclass or base class for human, and human is a subclass or derived class of primate.

In the diagram, we have used the standard UML notation to indicate generalization. Human and ape (concrete classes) are generalized (or abstracted) to create the abstract class primate. Similarly, we generalize the abstract classes dog, cat, primate and platypus to create the abstract class mammal. In any class diagram showing generalization, only the lowest level can possibly be concrete classes – i.e. classes we can create instances from.

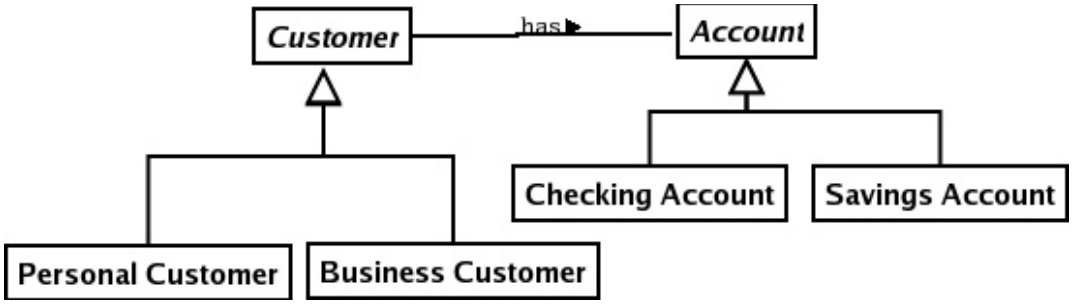
7.4.1 Why Use Generalization?

While many of the technical reasons for using generalization in object-oriented in general are beyond the scope of this module, we can state that generalization is used to make our models simpler by creating more general classes by abstracting based on the similarity of relationships.

Consider the first diagram below. We have two kinds of customer and two kinds of bank accounts. The diagram is confusing and the underlying relationships are difficult to see immediately.



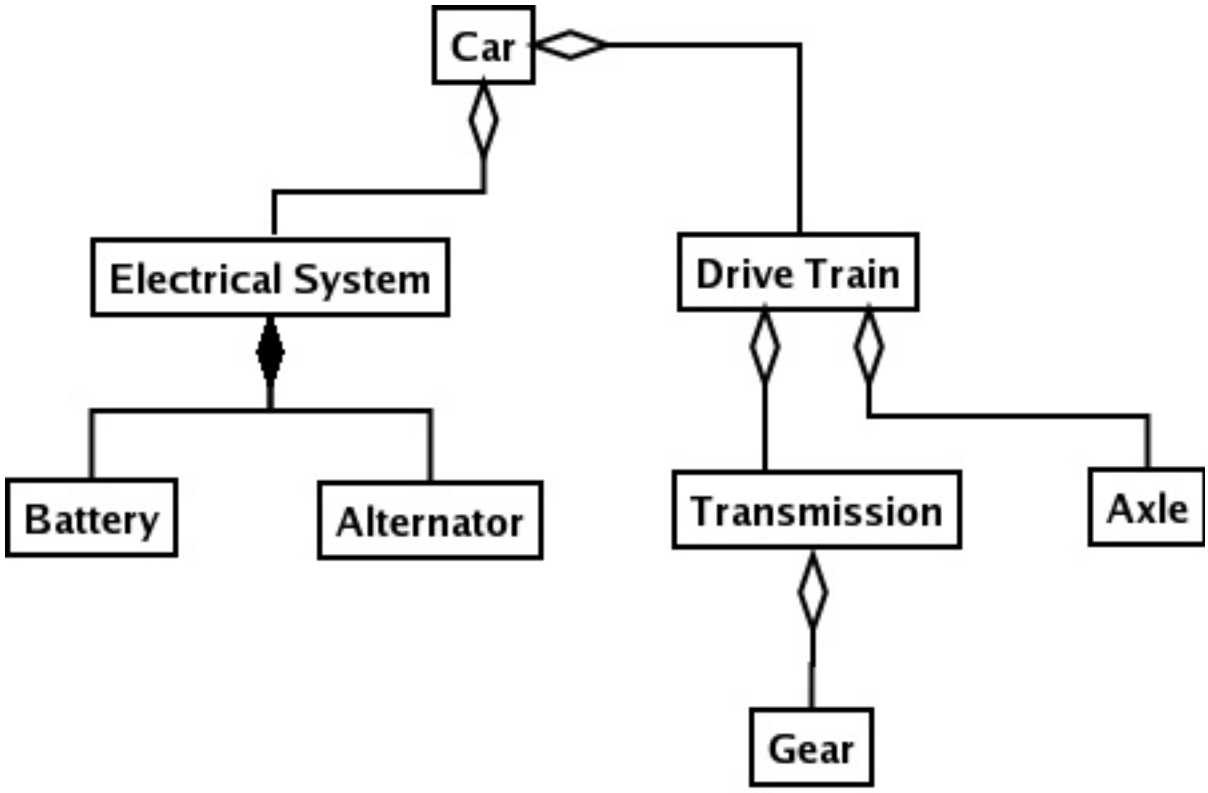
In the second diagram, we see the same model but with two abstract classes introduced to capture the commonality or relationships.



7.5 Aggregations and Compositions

The other kind of decomposition is called object decomposition or a part-of relationship. The UML term for this is aggregation. In the graphic below we show a portion of an object decomposition of a car. The diagram shows that a transmission is part-of a drive train which is part-of a car.

Here is a typical object decomposition. This is an aggregation tree, not an inheritance tree.



But “car” is a class (and an abstract one to boot – British humor). You had no difficulty understanding the object decomposition for a car even though it was an abstract class rather than an object. Now replace “car” with “motor vehicle”, we have a more abstract decomposition from which our object decomposition of “car” is inferable through inheritance. This is called orthogonality; the use of the class and object decomposition at the same time.

An aggregation can be homogeneous or heterogeneous. A homogeneous aggregation is when all the parts are of the same type – like a list of names. A heterogeneous aggregation is when the parts are of different types, like the car example above.

7.5.1 Composition versus Collection

Aggregations are classified on a scale of how tightly bound the parts are to the container.

At one pole is a collection or loose aggregation. For example, we might say that a company is an aggregation of employees, since a person can be an employee in more than one company at a time, and the employee has a continued existence even if the company disappears. This example illustrates a couple of the questions associated with determining the degree of binding between an object and its container.

Can an object belong to more than one collection at a time?

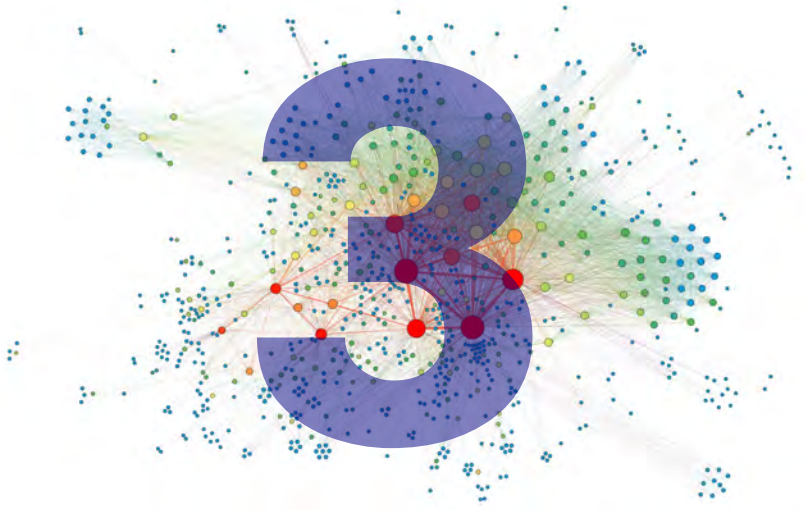
Can the object exist independent of the container?

At the other pole is a composition or whole-part relationship. For example, we might say that you are a composition of internal organs, because each organ can be contained in only one person and the organs generally do not have an independent existence after the person container dies. The binding between the object and its container is much tighter in a composition.

So how do we decide that something is a composition or aggregation? It's a matter of how the modeler perceives the relationship. For example, you would probably see yourself as existing in a composition relationship with your internal organs while an organ transplant clinic might see the relationship as a bit looser.

In the previous car example, we have modeled electrical system as a whole containing the parts battery and alternator. Our justification (remember as a modeler we make this choice) is that an electrical system is a name for a group of components working together – without the parts, there is no system. Perhaps you can come up with a reason to not use composition but to use aggregation instead.

Generally, we use aggregation (the open diamond). We tend to only use composition when we want to emphasize the whole-part nature of the relationship.



Object-Oriented Analysis and Design: Visual Modeling with UML

Models, Views and Architectures

We do not see the world as it is but rather as we are.
Anian Nin

Everything we hear is an opinion, not a fact.
Everything we see is a perspective, not the truth.
Marcus Aurelius

If we spoke a different language,
we would perceive a somewhat different world.
Ludwig Wittgenstein

exgnosis

1 Introduction to Models

Why should we care about models? Simply put, much of what we do in OOAD is the development of models: requirements models, specification models, domain models, process models, design models, architectural models and many other kinds of models. The entire development process relies extensively on models developed by the various members of the development team including business analysts, domain experts, developers, designers, testers and others.

We will use the terms model and modeling quite a bit in this module without any sort of rigorous definition; instead, it will be assumed that everyone understands what these terms mean and that our general common understanding of these terms will be adequate for this discussion.

It was once claimed that the primary activity that separates humans and animals is our ability to create abstract models. However, it now appears that model making is an essential part of any kind of cognition, whether human or not. Knowing that we are hard-wired to use models helps us to identify what makes models "good" or "effective" by looking our own intuitions and experiences as model users in everyday life.

Formal models are used in every aspect of software development. Whenever people engage in this sort of formal modeling activity, there are certain questions that always seem to come up, and should in the mind of every OOAD practitioner.

1. How do I know when my model is right? What does it mean for a model to be right?
2. How do I know what level of detail to put into my model? What is the right level of granularity? What can I leave out?
3. My model is different than someone else's model of the same thing, how can they both be right? Can one be more right than the other?
4. How do I know when my model is complete enough to stop modeling?

The answer to all these questions, and the answer that most people don't want to hear, is that most of the answers are that these are subjective judgments on the part of the modeler. However, we can develop a modeling “intuition” by examining what properties seem to make for better models. Consider the following two factors:

1. Formal modeling, like we are doing in this course, is just a special case of informal modeling, which is something everyone does all the time. It's not some new esoteric or abstruse technical area like multivariate calculus or quantum mechanics. You already are a modeler.

2. As a consequence of the previous fact, you have an intuition good enough to answer all of the questions posed above when you model, it is just a matter of hooking up that intuition you have from using informal models so that it works with these formal models.

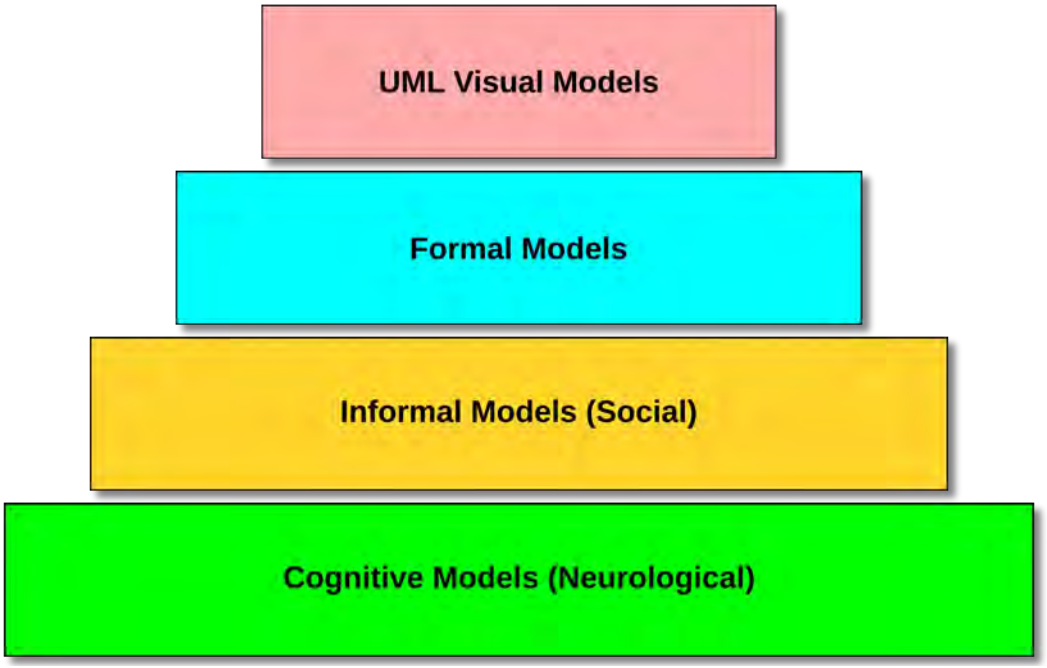
A well-designed model communicates information efficiently and effectively; and those two properties are often more the result of good modeling practices as opposed to the fact we are using UML. Diagrams are not models. Diagrams are used to describe models visually. In this module, we will examine the basic properties of models and what makes some models good and some bad, as well as how to work with a set of models in a unified way.

In this module we will look at:

1. The properties of effective models.
2. Understanding model perspectives through views.
3. Organizing models into collections using architectures.
4. How to test models.

1.1 The Modeling Hierarchy

This module is about modeling, which means that the first thing we need to do is identify what models are. Modeling is a fundamental human activity – we all create models all the time to help us process information about the world around us, to be able to make decisions and react to our environment.



These sorts of everyday models are intuitive and what we call informal – which is to say there are no explicit rules that govern how we make or use them, just an ill-defined set of common "ways" of modeling we all follow to some degree. This in itself is quite remarkable since it seems that no matter where or when someone lives, the mental models we create and use every day are similar the models that everyone else uses.

There are two basic reasons why our informal models of the world resemble the models others around us have. The first reason is common biology – our brains are structured to process information in a standard way which we all share. The second reason is that we learn a set of informal modeling rules through the use of language – a prototypical modeling tool – which is common across all cultures.

UML models are special types of formal models and therefore follow the best practices of formal models. Similarly, formal models are a special type of informal models, so understanding what makes for good informal models lets us understand how to evaluate more effective formal models. Informal models are similar in every human culture because the best informal models present and encode information in ways that are naturally based on how the brain encodes, stores and interprets information.

2 Biological Bases of Models: Neuro-cognition

Our brains are designed to be efficient in real-time as opposed to being exhaustive and thorough processors of data. To be fast thinkers, we take cognitive shortcuts by creating models. We extract salient features from our environment and generate prototypes of what we think is out there and then let our brains fill in the details. This allows us to react effectively based on a minimum of information and to take fast action in a possibly dangerous situation.

This approach has a high degree of survival value. If you were trying to survive on the Savannah 100,000 years ago, it would be useful to be able to identify a new animal that you had never met before as a potential predator before you became its dinner. Creating a mental model is an effective strategy since we do not have to analyze the whole situation in order to take action.

For example, our predator prototype might be “If it has sharp teeth, big claws, makes growling noises and runs at me, then it's a predator.” If I see a new animal that has big teeth, long claws and runs at me, the model can be used to determine, “Three out of four is close enough, I'm getting out of here.”

Consider trying to listen to a conversation at a party; if you concentrate on the conversation, your brain abstracts out all of the other conversations so that you don't notice them and can focus on the person talking to you. This is one of the other aspects of cognitive models. We can abstract out and ignore what we have decided is not of immediate importance to us.

2.1 Autism: When Mental Models Break

One of the more remarkable insights we have about how our biologically based mental models work are reports of what happens when they don't work effectively, such as in the case of Carly Fleischmann[1]. Carly is an autistic girl who lives in Toronto and was diagnosed as having the mental capacity of a six-year-old. Or at least so everyone thought until she was eleven and sat at her dad's computer and typed “hurt teeth help.” This breakthrough moment revealed to her therapists and family that Carly was a normal intelligent girl who, as she put it, was trapped in a body that she could not control. Since then, Carly has become a leading figure in educating the public about living with autism through her blog and public appearances.

In Carly's case, her autism did not allow her to ignore irrelevant information like we can. When Carly was asked in a 20-20 ABC News story why autistic people cover their ears, flap their hands, hum and rock; she replied (this is transcribed from the video interview)

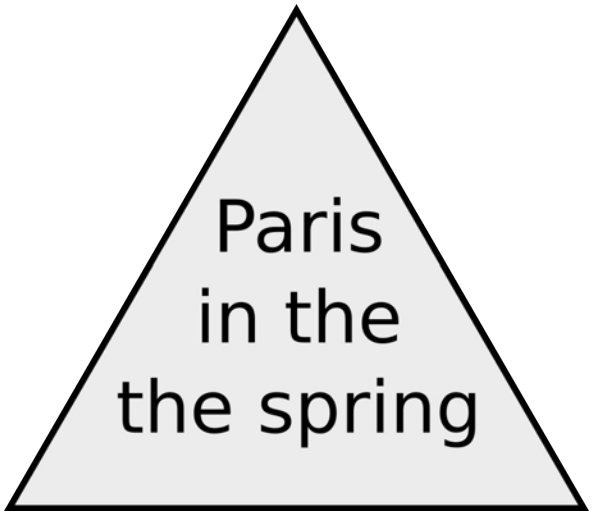
> *It's a way for us to drown out all sensory input that overloads us all at once. We create output to block out input. Doctors have said that we cannot process information. That's not really true. Our brains are wired differently. We take in many sounds and conversations at once. I take over a thousand pictures of a person's face when I look at them. That's why we have a hard time looking at people. I've learned how to filter through some of the mess.*

2.2 Evidence from Selective Perception

Selective perception is the result of our brains filling in the gaps in our perception. We perceive just enough to figure out what we are observing and then very quickly fill in the gaps from what we already know. Helen Meisenhelder in Selective perception: turning green into blue describes it this way:

For most of us, perception is reality. Yet, our perceptions do not always mirror reality, they are heavily influenced by our expectations, and prior experiences. We observe stimulus with a preconceived notion of how it should appear and find it difficult replacing that notion with the facts. Instead of reality, we "selectively perceive what we expect and hope to see". This biased phenomenon is defined as selective perception.

Selective perception makes it next to impossible for us to proofread our own writing because we know what we wrote and that is what we see every time we proofread the text. Interesting fact: to avoid selective perception problems, professional proofreaders proofread text backwards because if they see a sentence, they will just perceive enough of the words to fill in the rest. Like in this old optical illusion.



The brain edits out, or rather never processes, the second “the” in order to satisfy the expectation that we are reading a semantically and grammatically correct sentence.

2.3 The McGurk Effect

One of the most dramatic demonstrations of how the brain edits perception was discovered by Henry McGurk by accident[3] and occurs when the brain perceives different visual and auditory input from a speaker. The brain rewrites the auditory input so that it becomes consistent with the visual input.

For example, if a listener is watching a video where the person is mouthing the word “vase” but the audio has been dubbed so that they are hearing “base”, the following happens.

1. When the listener has his eyes open, he hears the word “vase”
2. When the listener has her eyes closed, she hears the word “base."

There are several excellent demonstrations of this on YouTube[2].

Even when you know the effect, you can't stop it from happening because it is taking place at a neurological level. For example, the following are all experimental discoveries about the McGurk effect[4].

1. The effect works on perceivers with all language backgrounds.
2. The effect works on young infants.
3. The effect works when the visual and auditory components are from speakers of different genders.
4. The effect works with highly reduced face images.
5. The effect works when observers are unaware that they are looking at a face .
6. The effect works when observers touch—rather than look—at the face .

2.4 Common Cross Species Neurological Architecture

As a final note on this fascinating side topic, ethologists (scientists specializing in animal behaviour) have been finding very convincing evidence that it is not just humans who have the ability to create these mental models. Various studies seem to indicate this ability is shared by the great apes and dolphins but also by species of birds, mammals like dogs and the octopus. This suggests that mental models may be the result of common ways of organizing brain structures and sensory processing.

3 Informal Models: Social Communications

When we communicate with each other we construct our messages in ways that are consistent with the way our brains process information. Messages that package information in ways that are consistent with our mental models have a sense of efficiency and effectiveness to them.

These effective communications methods become part of the language and culture of a community, and show remarkable consistency across different languages and cultures. The rules that we use in these informal models are the same ones that we use in formal models, which means if we ignore these rules creating a UML diagram, we wind up with a model that fails to be effective.

3.1 Social Basis of Models: The Speech Contract

The philosopher H.P. Grice[5] noted in the late 1970s that human conversations were governed by a social contract of great strength. In fact, this speech contract is so powerful that a deliberate violation of the contract by one party in a communication can cause a violent reaction from the other. In every culture in the world, breaking the speech contract is considered either inadvertent, which is taken as a sign of mental illness or stupidity, or deliberate on the part of the speaker, which is always taken as an insult. Intentionally breaking the contact is always a sign of contempt for the other person.

Social contracts are tricky things to identify and describe. A society controls the behaviours of its members through a system of norms and values. A norm is a "way we do things," although it is often more associated with the opposite "what we just don't do." A value is a reason or rationalization for a set of norms.

We learn the norms of a society through the socialization process – our parents and teachers show us how to be good little boys and girls with the goal of becoming productive members of our society by following its norms. The problem with norms is that in many cases we can't say exactly what they are because they are never explicitly stated. You learn norms by being sanctioned or violating them and by emulating the behaviour of others who serve as role models.

For example, if you were asked to describe polite conversation, you would be hard-pressed to define or describe it, but you generally would have no trouble knowing when the rules of polite conversation were being violated.

When these norms constrain or structure our interaction with others, they form a type of social contract. This social contract often exerts a compulsion on us to act a specific way but yet, because the norms underlying it are not overt, we are unaware of this influence.

For the purposes of this course, the part of Grice's analysis we are interested in are the rules of the speech contract which might be considered the rules by which we package information for efficient communication.

> *Our talk exchanges do not normally consist of a succession of disconnected remarks, and would not be rational if they did. They are characteristically, to some degree at least, cooperative efforts; and each participant recognizes in them, to some extent, a common purpose or set of purposes, or at least a mutually accepted direction.*
>
> *This purpose or direction may be fixed from the start (e.g., by an initial proposal of a question for discussion), or it may evolve during the exchange; it may be fairly definite, or it may be so indefinite as to leave very considerable latitude to the participants (as in a casual conversation).*
>
> *But at each stage, SOME possible conversational moves would be excluded as conversationally unsuitable. We might then formulate a rough general principle which participants will be expected to observe, namely: Make your conversational contribution such as is required, at the stage at which it occurs, by the accepted purpose or direction of the talk exchange in which you are engaged. One might label this the COOPERATIVE PRINCIPLE.*

The cooperative principle is about making communication efficient, which is of course what good models should do – be efficient in how they work with information. Grice elaborates on what this notion of cooperation means by positing a number of maxims that we tend to follow in our conversation.

QUANTITY

Relates to the quantity of information to be provided:

1. Make your contribution as informative as is required (for the current purposes of the exchange).
2. Do not make your contribution more informative than is required.

QUALITY

Try to make your contribution one that is true'

1. Do not say what you believe to be false.
2. Do not say that for which you lack adequate evidence.

RELATION

Be relevant.

MANNER

Be perspicuous (easy to understand).

1. Avoid obscurity of expression.
2. Avoid ambiguity.
3. Be brief (avoid unnecessary prolixity).
4. Be orderly.

Any detailed discussion of this topic is really beyond the focus of this course; however, it is useful to look at speech contract briefly to see that these maxims actually become important when we start to look at what makes “good” models.

3.2 Formal versus Informal Models

This course is about formal models. So far the biological example and the speech contract have been examples of what we call informal models. The primary differences between these and formal models are:

1. Informal models are part of our cognition which means we use them without us being aware of them. Formal models are defined independent of how we think and are constructed and interpreted based on a set of stated rules.
2. Informal models are general in nature and apply to many different kinds of data while formal models are defined to only apply to a narrow and well-defined type of data or experience.
3. Informal models tend to be qualitative and give us estimates while formal models are usually quantitative and produce precise results.
4. Informal models tend to have higher rates of error and are not as easily validated or verified as formal models.

However, the important point that needs to be made is that formal models are derived from how we do informal models – and this means that everyone has a “modeling intuition." For this course, what we want you to do is to start to fine-tune that intuition during the learning of the formal models so that you can start to develop your own sense of how to create and analyze formal models at a high level of competence.

4 Formal Models

From this point on in the course, we will only be discussing formal models so any reference to “model” without a qualifier should be understood to mean a formal model.

There are a number of reasons to use formal models but the main reason is to simplify. Remember that when we looked at the biologically based metal models, one of the reasons we created those models was to avoid being overwhelmed by details and instead be able to immediately identify and focus on the most important facts. As the UML User's Guide says:

> *We build models so that we can better understand the system we are developing. We build models of complex systems because we cannot comprehend such a system in its entirety. A model is a simplification of reality. [Booch99; pg 4]*

For example, suppose we are hired by the city of New York to provide information to individuals and groups about the city, whether they are tourists, businesses or whoever else.

If I wanted to get from Central Park to Penn Station, the aerial photo is a lot harder to work with than the map on the right. The simplification that the model (map) provides removes the clutter of extraneous details so that I can focus on the essential features that are necessary to plan out my route.

Simplification is the primary motivation for using formal models – we need something we can work with when the raw data of reality is too overwhelming to be processed and used effectively.

4.1 Analysis versus Design Models

Paradoxically enough, the first topic we are going to look at in this section is a pseudo-property of models, one that is often used to claim that there are two fundamentally different types of models:

1. **Analysis Models:** These are claimed to be models of what already exists or in some terminologies, models of the existing system as is.
2. **Design Models:** These are claimed to be models of what doesn't yet exist or in some terminologies, model of the system to be.

While this distinction is legitimate, being a design or analysis model is NOT a property of the model but rather describes how it is being used at a particular point in time.

In this course, I will avoid the terms analysis and design models and instead refer to using a model as a map or as a blueprint.

A model is a map when I use it to understand something that already exists, in the same way that I use a road map to figure out where various roads in a city lead to so that I can get where I want to go.

A model is a blueprint when I use it as a guide to create something that doesn't exist. A recipe for curried eggplant is a blueprint because it describes something that doesn't exist at that moment (to avoid a philosophical discussion we will just take that to mean that MY curried eggplant doesn't exist at this moment and I very much want it to, as opposed to curried eggplant in some generic or existential state).

To illustrate this, suppose that I want to build a GPS tracker. I search the Internet and find a circuit diagram that lays out exactly how to build a generic GPS tracker. A circuit diagram is a formal model but since the GPS unit doesn't exist yet, the diagram is a blueprint. After I build the GPS tracker the circuit diagram now becomes a map showing me what is in the existing GPS. Now suppose that a few months later, I decide to modify my GPS tracker by making it solar powered since I'm tired of buying batteries for it. Of course, I have totally forgotten how I built the GPS so I consult the map of the device (the same old circuit diagram) to figure out how to make the modifications to my GPS.

A model may be used as a map or blueprint depending on why it is being used – but being a map or a blueprint is not a property of the model itself.

4.2 Characteristics of Good Models

There have been a number of attempts in the IT community to define what makes a model good. Most of the attempts agree on some of the basic characteristics, although they state them in different ways, most of these lists are in many ways unintentional restatements of the maxims of the speech contract. The characteristics of good models are the following:

4.2.1 Models are Accurate

This is the maxim of quality (say things that you know are true). In formal terms we can express this as follows:

1. When a model is being used as a map, every element in the model represents something that actually exists in the real-world.
2. When a model is being used as a blueprint, every real world object that is critical or essential is depicted in the model – there is no model element that “should not be built".

The first point is easy to see. If we look back at our map of New York on the previous page, we can see that if fails this test. It shows the World Trade Center still existing in lower Manhattan. If I didn’t know anything about New York history, I might try navigating around Manhattan using the non-existent WTC as a reference point, which would of course fail. The map is unreliable since I can no longer be sure about the real-world existence of anything the map shows me.

We can imagine a similar situation when we have a design model. Suppose we get a children's swing set that we have to assemble. If the instructions (our model) show us how to add parts that don't come with the swing set to other parts that don't exist, then our model is unusable because it is inaccurate.

Consider this example:

NATO bombing continues; US blames bad maps for embassy strike

BELGRADE, Yugoslavia (CNN) – NATO followed its apology for the bombing of the Chinese Embassy in Belgrade with more air strikes on Yugoslav targets overnight Saturday and Sunday morning ... Contributing to the problem, CNN was told, were the maps for the area. All available maps for the area showed the Chinese Embassy in its former location and not its current location, where it has been for about four years.

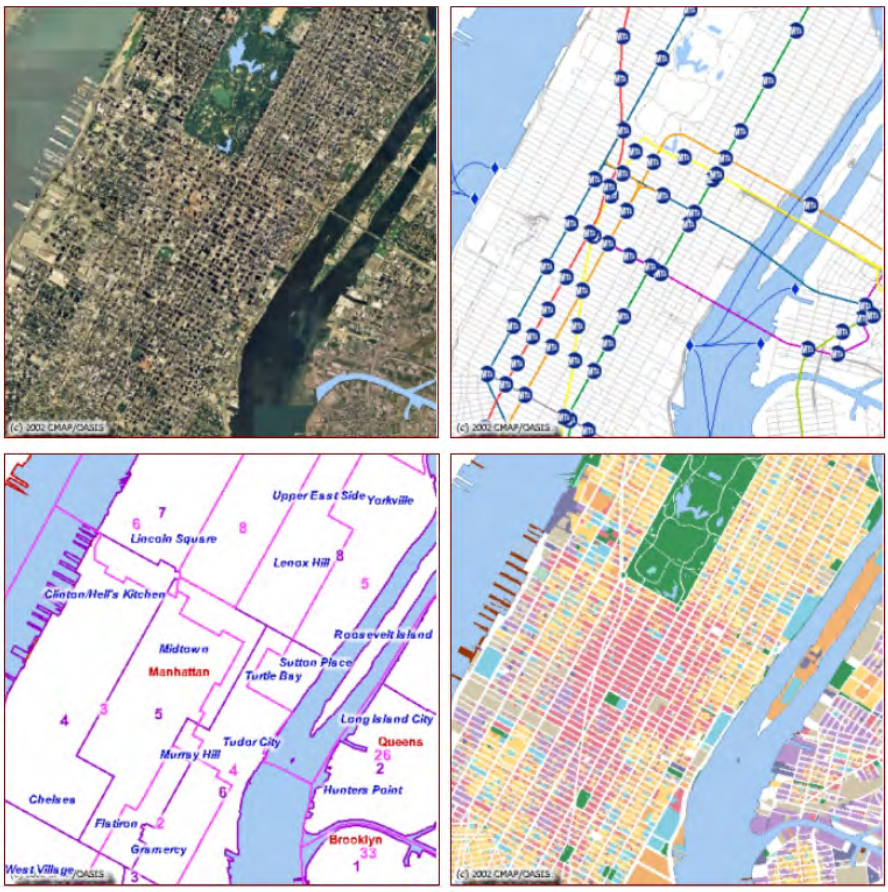
There are a number of different ways to express this concept of accuracy. Grady Booch and the other developers of UML expressed it this way:

> *Models have to be models of something – models are representations of some subject out in the real world.*

Robert Binder expressed it this way:

> *The best models are connected to reality.*

4.2.2 Models have a Frame of Reference



This is an expression of maxim of relevance. It means that good models do not show elements or details that do not contribute to the purpose of the model but only contain relevant information. Models simplify reality by emphasizing the important features and abstracting out the unimportant with respect to the purpose of the model.

However, it is the modeler who makes the decision about what is important. This means that all models are inherently subjective because they represent someone's decision about what should be depicted.

For example,the four figures above show a real-world image of New York and then several maps (models). Each map above uses a different frame of reference. The top right shows us the subway lines and MTA stations, the bottom left map shows us the borough and other boundaries but does not show any streets. The bottom right shows us how the land in Manhattan is used, but not boundaries or subways.

Each of these map/models has been created with an intent to depict some view or aspect of New York according to a frame of reference, which in these cases are transportation, politics and land usage respectively. These frames of reference are derived from the reason we created the model or, in other words, its purpose. If I want to find out where to catch the subway, the land use map is the wrong choice of model because it contains information that is irrelevant to my purpose for using the model and missing information I need. The information it contains may be accurate, it's just not useful.

4.2.3 Models are the Right Size

The idea of being the right size is a restatement of the maxim of quantity from the speech contract (make your contribution as informative as is required for the current purposes of the exchange but do not make your contribution more informative than is required). We can restate this in the following two rules:

1. Don't tell me stuff I already know.
2. Don't leave out things I need to know but don't know.

This is not the same as a frame of reference which says that we leave out information that is not related to the purpose of the model. This property says that we should leave out information that the person using the model already knows and should not leave out things that they don't know but need to know.

This means that two models may be of the same subject and have the same frame of reference but differ in the amount of detail based on the intended audience. The amount of detail is often chosen by anticipating the level of knowledge that the users of the model already possesses about the subject being modeled.

Recall our definition of a paradigm from a previous module. Up until the establishment of a paradigm, anyone writing about something in a field had to develop all their ideas from “first principles” or from scratch. Because they had no idea what anyone reading their work actually knew, they wrote massive works that explained every detail. Once a paradigm has been established, an author can assume that readers all have the basic paradigm knowledge. New results and concepts can be communicated without having to restate what everyone knows – massive treatises become short research papers.

There is a paradigm for Particle Physics. Published papers in the subject are incomprehensible to readers who don’t know the paradigm because authors assume that those reading the paper already share standard knowledge about the subject. For example, this is the abstract of the paper Calculation of the Cherenkov light yield from low energetic secondary particles accompanying high-energy muons in ice and water with Geant4 simulations by Leif Rädeland and Christopher Wiebusch.

> *In this work we investigate and parameterize the amount and angular distribution of Cherenkov photons, which are generated by low-energy secondary parti-*

> *cles (typically less that 500 MeV), which accompany a muon track in water or ice. These secondary particles originate from small energy loss processes. We investigate the contributions of the different energy loss processes as a function of the muon energy and the maximum transferred energy. For the calculation of the angular distribution we have developed a generic transformation method, which allows us to derive the angular distribution of Cherenkov photons for an arbitrary distribution of track directions and their velocities.*

Ummm yeah, ok.

4.2.4 Models Have a Well Defined Notation, Structure and Technique

This is the characteristic that most clearly differentiates formal models from informal models. Formal models have the following:

1. **Notation:** A symbolic way of representing the model. This can be diagrams, equations or some other symbolic notation for depicting the model.
2. **Structure:** A model often consists of a set of model elements and specific ways of arranging them so that some arrangements are meaningful and others are not. Arrangements that are not meaningful are usually disallowed by the rules of the model.
3. **Technique:** There may be specific ways that have to be employed to construct the models, usually described in some model construction process.

There is often a set of rules that defines what the legal model elements are, the legal structures that the model allows and the required techniques for constructing the model. This set of rules is called a meta-model: it is a model that describes how to make models of a particular type. The UML standard, for example, consists of a set of meta-models describing how to create UML models.

Techniques are often important because even though models simplify reality, the models themselves may be very complex and difficult to construct. For example, the following is a model of the quantum harmonic oscillator for one particle in 2D (whatever that means):

$$i\hbar\frac{\partial}{\partial t}\psi(x,t) = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\psi(x,t) + V(x)\psi(x,t)$$

$$V(x) = \frac{1}{2}m\omega^2 x^2$$

This is an example of a model that has a mathematical notation and set of techniques where the structure is described by the underlying physics model.

In contrast, a circuit diagram is a graphical notation, as is a state diagram or a flowchart. In this course, we will be distinguishing very clearly between the model itself and the notation used to describe the model. Any misuse of the model's notation runs the risk of an incorrect interpretation of the underlying model.

4.2.5 Models are Incomplete

Since every model abstracts out information, especially with respect to its frame of reference, no model can contain all the information about the subject being modeled. A single model from a single frame of reference cannot be sufficient for most of our purposes.

Modeling is most effective when we describe the subject from multiple points of view – the aggregation of the models provides a more complete description of the subject being modeled. An analogous situation is that of an investigator trying to figure out what happened during a hold-up. If they only ask one person what they saw, they will get only that person's perspective and what that person noticed. As they interview more witnesses, other details will emerge from different perspectives providing a more complete picture of the suspect.

This leads to a very important principle of modeling systems:

> *No single model is adequate to describe a system. Systems are best described a minimal, complete collection of orthogonal models.*

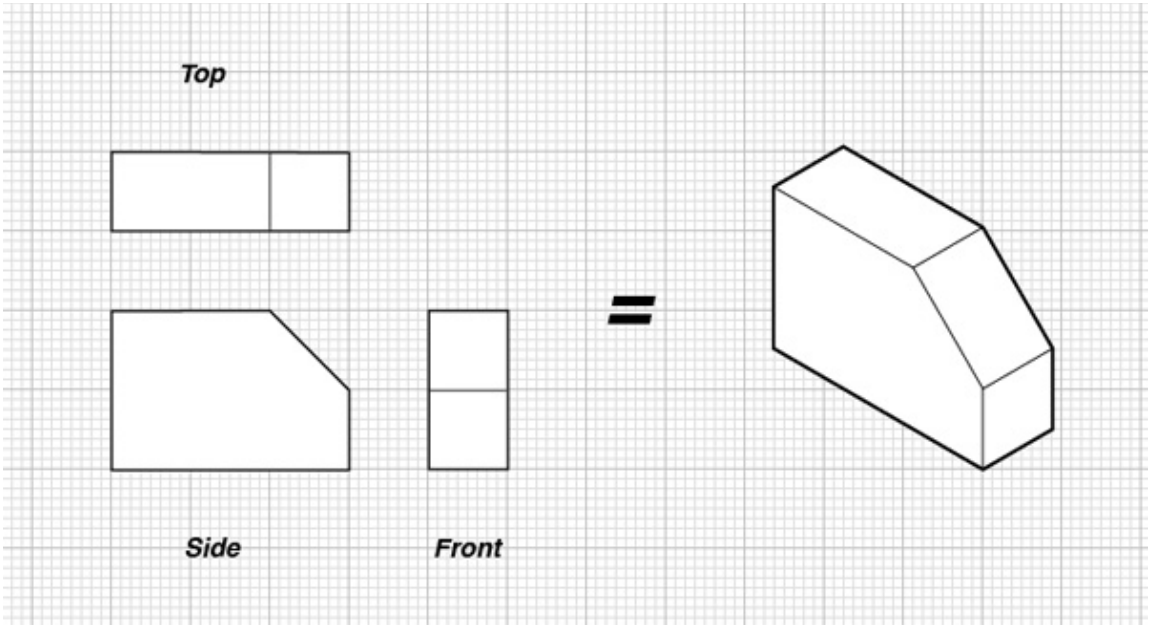
What does this mean?

Each model leaves something out or omits some sort of detail. A complete set of models does not leave out any important details. In other words, each important detail appears in at least one of the models in the collection.

Obviously for usefulness, we don't want this collection to become so large that it is unmanageable, so we look for the smallest collections of models that are complete. One of the reasons that UML became popular is that it uses a relatively small number of models compared to many of the OO modeling systems that preceded it

For now, we will consider orthogonal to mean that the models are all done from different perspectives, the overlap between models in what they show is minimal.

For example, a solid object can be described by three orthogonal views in engineering: the top, side and front views.


Top
Side
Front

5 Views and Architectures

One of the more confusing terms used in IT is “systems architecture” or “IT architecture” or “software architecture.” The confusion real since there are multiple competing techniques and methods, all of which use the terms but to mean different things. To compound the confusion is the existence of “business architecture” techniques which resemble many of the software architecture methods.

While we will not be getting into the whole subject of architectures, it does have a bearing on modeling since our choice of architecture influences our choice of models. Remember that we can best describe something in terms of a small set of orthogonal models, each of which (intuitively) is constructed from a specific view. We use the term orthogonal here to describe a set of models that describe the same subject from different and non-overlapping points of view.

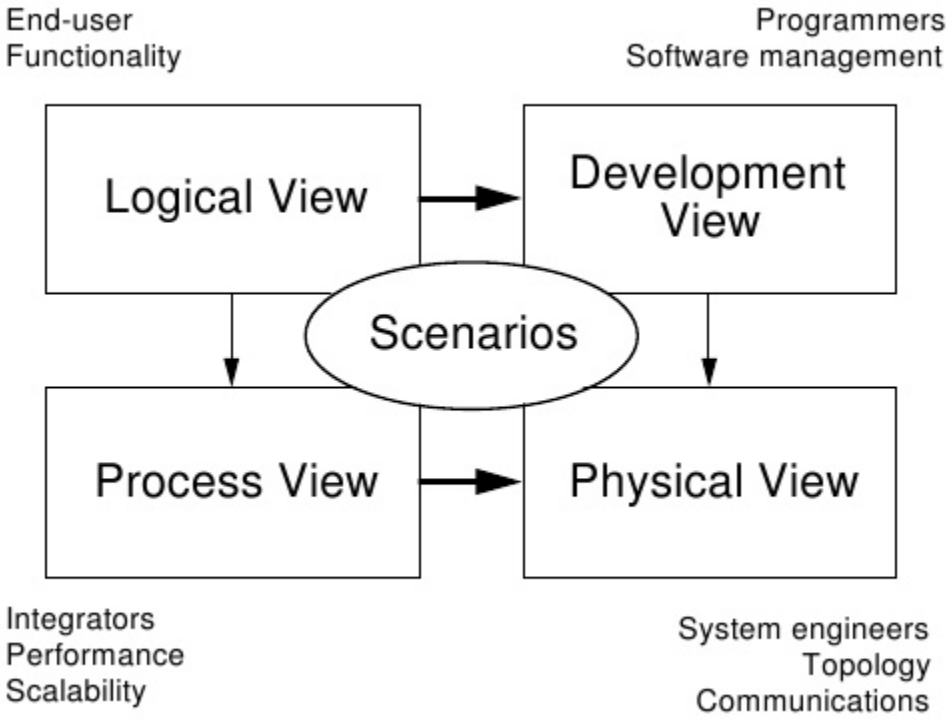
First we need to define what an architecture is, which is a bit problematic since there is no generally agreed-upon definition. However, we can get a flavor of what an architecture is by looking at a couple of industry definitions.

1. “The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.” (IEEE).
2. “A representation of a system in which there is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and human interaction with these components.” (SEI)
3. “An architecture is the most important, pervasive, top-level, strategic inventions, decisions, and their associated rationales about the overall structure (i.e., essential elements and their relationships) and associated characteristics and behaviour.” (OPEN Process Framework).

A core goal of any architectural system is to provide an overall complete description of the system and an identification of the various views that are used in the construction of models. A view in an architecture defines a modeling frame of reference. One of the reasons for defining an architecture is to ensure that no aspect of the system is overlooked so that the resulting set of models is complete and, hopefully, minimal.

5.1 The 4+1 Architecture

The 4+1 architecture was originally developed to be used with the Rational Unified Process (RUP) but has achieved wide acceptance outside RUP as well. As we will see later on, some of the UML diagrams, like the component and deployment diagram, are included specifically in the UML standard to enable specific views to be included in the modeling process.



The views of the architecture are described in the RUP documentation in the following way:

The Logical View

1. Addresses the functional requirements of the system.
2. What the system should do for its users.
3. It is an abstraction of the design model and identifies major design classes, packages and subsystems.

The Implementation View

1. Describes the organization of static software modules, source code, data files, scripts, components and executables for example and describes them in the development environment.
2. Identifies the issues of configuration management and development issues like change management.

The Process View

1. Addresses the concurrent aspects of the system at run time – tasks, threads and interactions.

2. Concurrency, system startup and shutdown, fault tolerance, response time, throughput and so forth.

The Physical View

1. Deployment is mapping software to processor nodes showing how the executables are mapped onto the underlying hardware topology.

The Scenarios

1. Contains key requirements that initially drive the discovery of architecture and later used to validate the architecture.

The scenarios can be thought of way of walking through the system as it performs a specific task and identifying how this task is reflected in the different models as seen through the various views.

For example, suppose that we have a scenario where a use is logging on to a banking website. The logical view will describe the experience from the user point of view but as we walk through the scenario we can also look at how the other views see that scenario.

From the logical view, the user sees a login screen, enters their credentials and then is taken to their home screen with a list of accounts and menus of options.

However from a physical view we have to ask where are the user credentials stored? How are they accessed to execute the login? Where is the client's account information located and how do we connect to it?

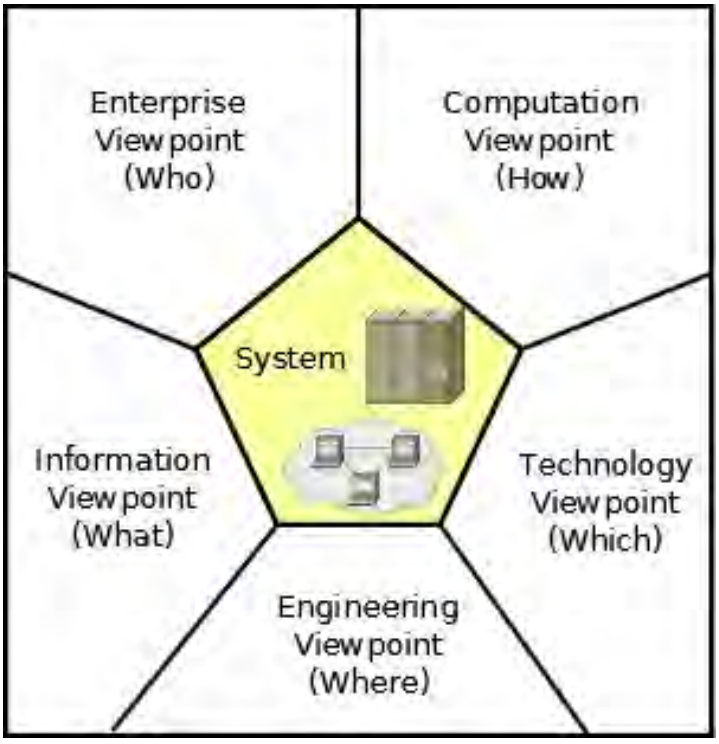
From a process view we may find ourselves asking how to support maybe ten thousand simultaneous users all trying to log in at the same time. What sort of latency is allowable? What sort of security is available on the user passwords? How do we keep all the data from each user isolated from other users?

From a development view we might ask where are the encryption modules located? Where is the login logic located and how do I ensure that the scenario is accessing the right versions?

5.2 RM-ODP

Another example of a competing architecture is the RM-ODP which is described as :

> *The Reference Model of Open Distributed Processing (RM-ODP, ITU-T Rec. X.901-X.904 | ISO/IEC 10746) is a joint effort by ISOIEC and ITU-T, which provides a coordinating framework for the standardization of open distributed processing (ODP) which supports distribution, inter-working, platform and technology independence, and portability, together with an enterprise architecture framework for the specification of ODP systems*[6].



This architecture was developed to respond to the problem of working with modern systems where the specifications for the system are so complex that it exceeds the ability of any one person to understand all aspects of the system.

The RM-OOP defines five viewpoints, each of which represents the concerns of a group of stakeholders who have a common interests or concerns about the system. Each viewpoint has it's own modeling conventions or techniques (called a viewpoint language) that is optimal for that viewpoint's audience.

The five viewpoints defined by RM-ODP have been chosen to be both simple and complete, covering all the domains of architectural design. These five viewpoints are:

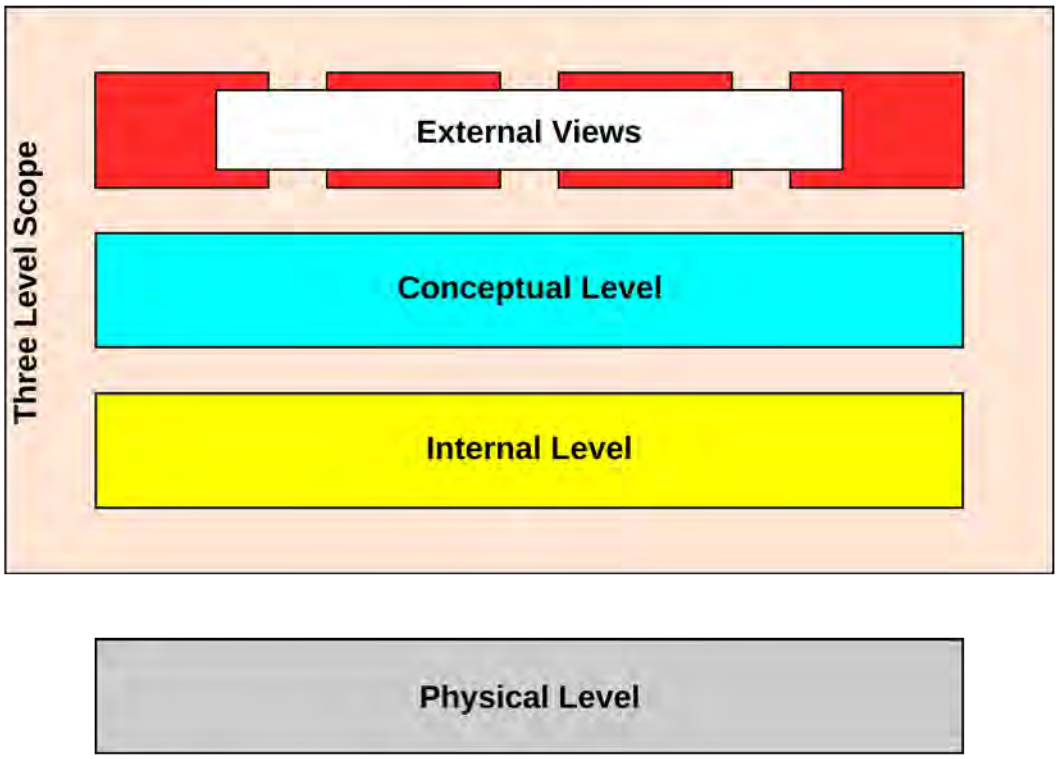
1. **The enterprise viewpoint**, which is concerned with the purpose, scope and policies governing the activities of the specified system within the organization of which it is a part;
2. **The information viewpoint**, which is concerned with the kinds of information handled by the system and constraints on the use and interpretation of that information;
3. **The computational viewpoint**, which is concerned with the functional decomposition of the system into a set of objects that interact at interfaces - enabling system distribution;
4. **The engineering viewpoint**, which is concerned with the infrastructure required to support system distribution;

5. **The technology viewpoint,** which is concerned with the choice of technology to support system distribution.

5.3 The ANSI-SPARC Three Level Architecture

The ANSI-SPARC Three Level Architecture was proposed in 1975 in an early effort to create a standard system database architecture. Although never a formal standard, it has been the foundation for the design of many database products even though it could not be fully implemented until 1998 when the technology finally became available to enable database vendors to provide updateable views.

We will find this architecture especially useful when exploring the relationships between different kinds of UML diagrams and what each diagrammodels.



The External Level

This is the level closest to the users and models how the data is viewed by the individual users. This implies that each user can have their own unique “model” or how they think the data is organized, which is often referred to as an “external schema”. As we will see in a later module, this is the level at which the business works with the data, usually in terms of objects or information that they are working with.

The Conceptual Level

The conceptual level can also be thought of as the community view or model of the data. This level models the entire information content of the database, but still at the level of abstraction that is above any implementation details. The conceptual level is defined by a “conceptual schema” but, unlike the external schema, there is only one conceptual schema. The view of the data does not correspond to any one stakeholder's view. Instead, the stakeholders' views are generated from this underlying logical model.

The main difference between the external level and this level is that we discover information at the external level while we define data at the logical level. The data models at this layer define exactly what entities we are recording data about, what attributes the entities have, the sorts of values those attributes can take, and the nature of relationships between those entities.

The Internal Level

The internal level is where we are concerned with the structure of the data so that it can be used for the purpose intended. There can be more than one “internal schema” which, in addition to describing how the data is stored, contains information about how to access the data and the layout of the data. Depending on the data requirements, a single logical model may be segmented with different segments implemented by different internal schemas.

This is still a level above the actual physical layer because we assume that we have available in infinite linear storage space. However, there may be more than one possible internal model, each corresponding to an alternative implementation of the data.

The models at this layer don’t define the data but are concerned with how we structure and organize the data defined by the conceptual modes so that it can be used for a specific purpose. For example, data can be implemented as a relational form when it will be used in transaction systems or in a dimensional model when it will be used for data analytics systems.

The Physical Layer

This is not part of the ANSI/SPARC architecture. The physical layer is concerned with a particular implementation of the internal schema on a particular system architecture. This is usually handled by various database products and applications.

Interestingly enough, this architecture closely mirrors Peter Chen's descriptions in his original paper on entity-relationship modeling[7].

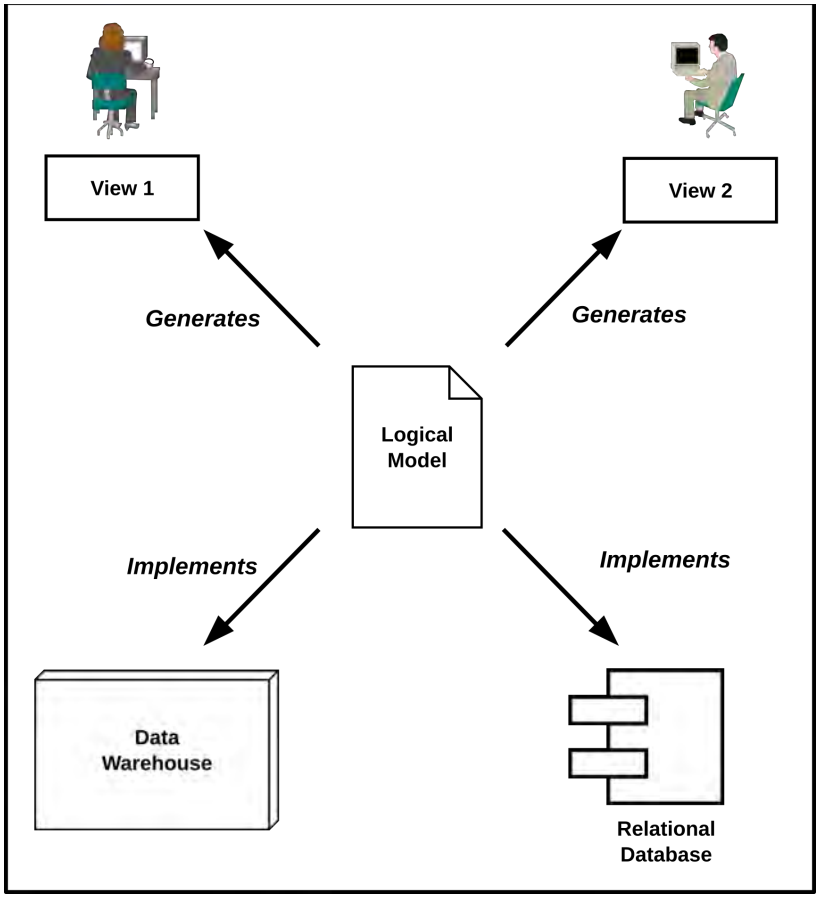
In the study of a data model, we should identify the levels of logical views of data with which the model is concerned, we can identify four levels of views of data:

1. Information concerning entities and relationships which exist in our minds.

2. Information structure – organization of information in which entities and relationships are represented by data.
3. Access-path-independent data structure – the data structures which are not involved with search schemes, indexing schemes, etc.
4. Access-path-dependent data structure.

The diagram below illustrates the three level architecture in practice. By exploring the views that the users have of the data – specifically their individual data models – we are able to define a logical data model that allows us to generate the specific external views. The logical model is where we define the data so we are free to design this model to whatever criteria we deem important, a topic we will return to later.

Since the logical model is not implementation specific, it can be used to generate relational models or dimensional models for data warehouses or any other kind of implementation model.



Notes

1. http://carlysvoice.com/home/aboutcarly/

2. “Try The McGurk Effect! - Horizon: Is Seeing Believing? - BBC Two” http://www.youtube.com/watch?v=G-lN8vWm3m0

3. McGurk, H & MacDonald, J (1976); "Hearing lips and seeing voices," Nature, Vol 264(5588), pp. 746–748

4. http://www.faculty.ucr.edu/~rosenblu/VSMcGurk.html

5. Grice, H.P. (1975). “Logic and Conversation,” Syntax and Semantics, vol.3 edited by P. Cole and J. Morgan, Academic Press. 22–40

6. http://www.rm-odp.net/

7. Peter Chen, Peter Pin-Shan (March 1976). "The Entity-Relationship Model - Toward a Unified View of Data". ACM Transactions on Database Systems 1 (1): 9–36. doi:10.1145/320434.320440.

Object-Oriented Analysis and Design: Visual Modeling with UML

Software Development Processes

Software is not released, it is allowed to escape.
Anonymous

If you can't describe what you are doing as a process,
you don't know what you're doing.
W. Edwards Demming

It is not a question of how well each process works,
he question is how well they all work together.
Lloyd Dobens and Clare Crawford-Mason

All human error is impatience, a premature renunciation of method.
Franz Kafka

exgnosis

1 Introduction

In module one, we introduced the two parts of a software development process:

1. **The Paradigm:** the set of techniques, concepts and best practices we use to build software; and
2. **The Production Process:** how we organize our paradigm activities to meet the iron triangle of production constraints. We also referred to this as the “engineering” process.

In this module, we take a closer look at the engineering process, including the Agile and Iterative variants used by the Unified Software Development Process, as well as taking a pragmatic look at the practical issues involved in process maturity.

The last part of the module is an overview of the Unified Software Development Process and the Rational Unified Process. Subsequent modules will provide deep dives into the OOAD and modeling activities that we perform during the various phases of the USDP and RUP.

1.1 The No-Process Option

I have spent a lot of time in organizations where the following quote is typical.

> *Software development process? You have to be kidding, we’re so busy here actually doing work that we don't have time to screw around with pretty little charts. Here, our people just get it done. We don't need a process here.*

There reflects a certain attitude in some organizations that processes are things that other people use to compensate for a lack of talent or work ethic. I have noticed is that many of those same organizations have a very similar way of doing things, almost like there is a default process that organizations wind up using when they don't actively define and use a process. I refer to this as the default no-process.

Every time we start a software development project, we use a process. This is not an option. We do not have the choice of whether or not to use a process. Our only choice is which process to use. This default no-process choice is what we call an anti-pattern, which is a consistent pattern of failure repeated over and over. Once you’ve seen this default no-process in action a few times, you start to see how it always unfolds in a very predictable series of steps to disaster.

Booch et al describe this state of affairs in the UML User Guide this way:

> *Curiously, a lot of software development organizations start out wanting to build high-rises but approach the problem as if they were knocking out a dog house.*

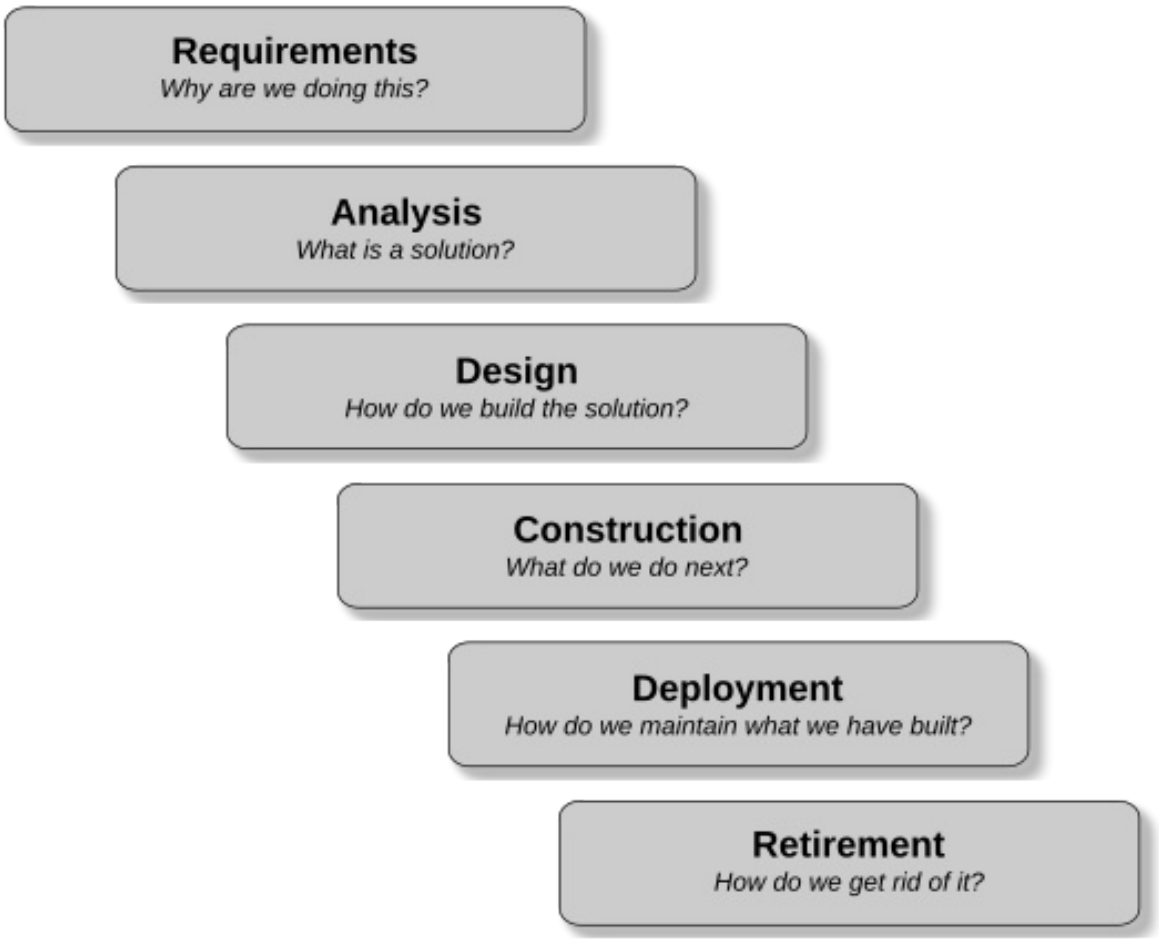
As the size and complexity of the development project increases, the chance of a successful outcome using the default no-process quickly approaches zero. There is no process free option in software development, only the option to chose your process. Like the old saying *“Failing to plan is planning to fail.”*

2 The Production Process

There exists a proliferation of software development processes and methodologies being promoted in the software development marketplace, each of which is usually claimed by its marketers to be the one that finally fixes the problems inherent in the other methodologies and will literally force your developers to write next to perfect code, even if they can't program.

More realistically, no matter what methodologies are actually in use, the ones that work share a common core of logical steps to be done in every project. This sequence of process steps are what we are calling the production process. More informally, it is a description of “How people build stuff.”

The production process should look familiar since parts of it are often referred to in the programming community as the waterfall system development life cycle (SDLC). However, the production process is just a description of how we need to solve a logical sequence of natural questions or problems as they occur in any kind of construction process. The production process is universal and can be observed any time we are successfully building or delivering something subject to the constraints of the iron triangle.

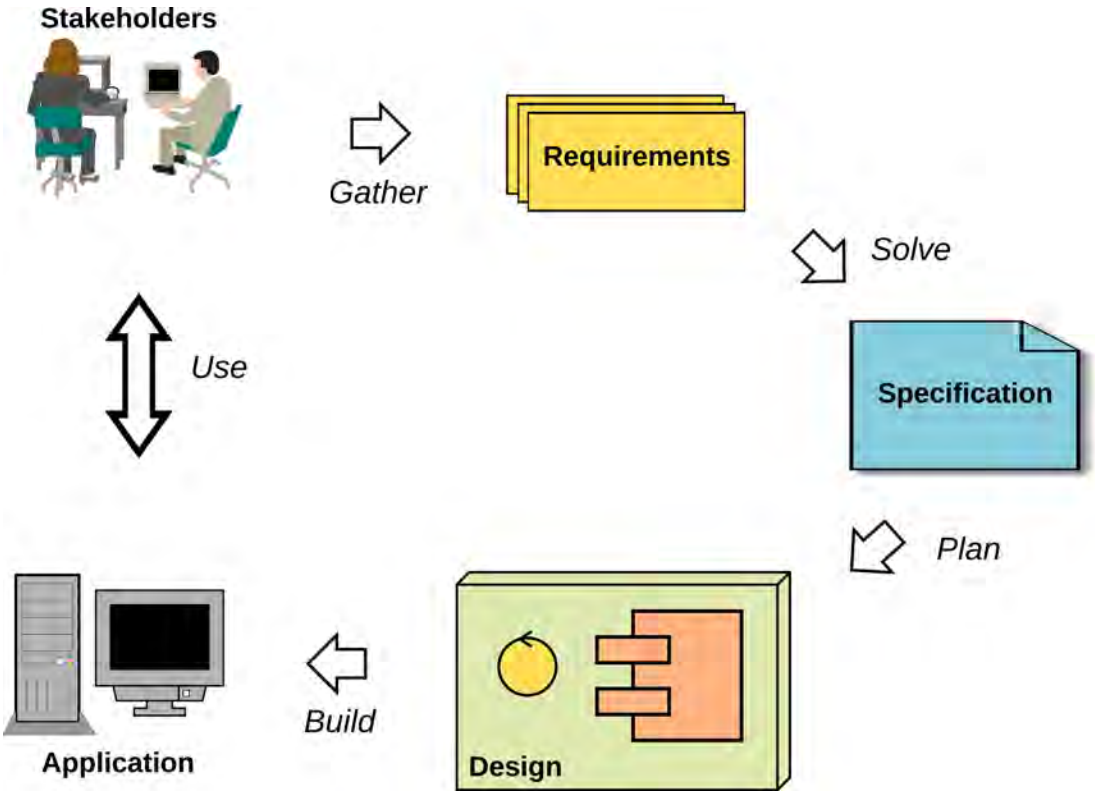


These steps are:

1. **Requirements:** The basic question that has to be answered during requirements is *"Why are we building this?"* This is generally followed by two other kinds of questions. *"What do the users need what we are building to do for them?"* At this stage, we need to know the functional requirements (what the stakeholders expect what we are building to do for them) and the non-functional requirements (what the performance criteria are that what we are building has to meet). The second question is *"What are the constraints – budget, time, architecture, etc. – that have to be considered when building the solution?"*
2. **Analysis:** In the analysis phase we find a solution to the problem. We often formulate the solution in a specification that describes what is to be built. The specification serves as a design target, which means that whatever design we propose has to produce a product that meets the description of the solution in the specification. We may come up with multiple ways to solve the problem which differ in terms of cost, quality and other factors, but eventually, we need to settle on a single solution.
3. **Design:** A specification describes a solution but does not tell us how that solution should be built. A design describes how the solution is to be built given a specific set of resources, capabilities and constraints. For a single specification, we may have a number of different designs depending on what we have available to work with.
4. **Construction:** The construction phase is where the product described in the specification is built according to a given design and construction or project plan. Having a design alone is not sufficient for something to be built, especially as the size and complexity of what you are building increases. You also have to employ the appropriate construction techniques and plan out the development activities.
5. **Deployment**: The completed solution is deployed when it goes into use. The questions that we have to ask for the deployment phase deal with might be called operational issues such as: *"How are changes or repairs to be made?"* or *"How are users and maintenance people to be trained?"* or *"How do monitor the product for possible malfunctions?"*
6. **Retirement:** All products and systems have to be taken out of service at some point, whether it's a nuclear reactor, a make of car or a software application. This phase is concerned with how we move the product out of the deployment phase and replace it, if necessary, with its successor.

2.1 The Generic Software Development Process

What I'm calling the generic software development process is just the translation of the production process we just described into the language of software. There is no actual generic software development process but it is the common core of all successful software development processes from waterfall to Agile to DevOps.



This natural progression is captured by the flow from requirements through specifications and design to building. In order to be effective at each step, we need to have at least a partial solution to the previous problems.

The stages are:

1. **Requirements:** Remember the purpose of the requirements phase is to answer the question *"Why are we doing this?*" We want to identify the overall business need that is driving the software development. We want to identify who has requirements (stakeholders) and who has constraints that set limits on what we can do, whether these are budgetary, resource availability or process or business rules. And of course, we collect and manage the requirements in a robust enough way to feed into the analysis stage.

2. **Analysis:** The analysis stage produces a specification that describes the solution to the problem and meets the requirements identified in the previous stage. The specification describes the system in terms of its features: what it should look like and what it should do. In addition to the specification, we may also produce a high-level architecture that can be used as a starting point for a detailed design.
3. **Design:** Given that we know what we have to do, how are we going to turn this into reality given what we have to work with? This is the role of design; to create a step by step plan as to how the specified solution will be built. The design will require choices on architectures, technology, and other proven techniques.
4. **Construction:** This represents the bulk of the time and cost of the project; the writing, testing and deployment of the application based on the design.

The generic software development process does not typically carry on through deployment and retirement. This reflects the historical fact that most software projects were considered to be completed once the working software was delivered. Several processes were extended into the full engineering process in what were termed “enterprise” versions of the process. In recent years, a different way of looking at software as a product has emerged called Application Lifecycle Management (ALM), which is an attempt to apply general product management principles to software. In addition, the DevOps methodology also tries to support the full production process.

2.2 Requirements, Specifications and Designs

Before things get confusing, we are going to clarify our terminology about exactly what we mean by requirements, specifications and designs. Unfortunately, in the real world, these terms are not used consistently, unambiguously or clearly. In this material, these terms will be used unambiguously and consistently based on the descriptions in this section.

One of the first problems we have to sort out is what the term requirement means. According the IEEE Standard Glossary of Software Engineering Terminology (1990) a requirement is:

1. *A condition or capability needed by a user* to solve a problem or achieve an objective.
2. *A condition or capability that must be met or possessed by a system* or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. *A documented representation* of a condition or capability as in 1 or 2.

The problem is that under this scheme the term “requirement” has three definitions which all refer to similar but different things. My experience is that this ambiguous terminology can often create confusion in a development project.

Therefore, for the purposes of this course, we will use the following terms:

1. A **requirement** is a condition or capability needed by a stakeholder to solve a problem or achieve an objective. (*"I need to be able to to find all the invoices associated with a customer, and I don't care how you do it, just do it."*) Requirements do not describe solutions or features but describe something about the stakeholder's need.
2. A **feature** is a condition or capability possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. (*There is a "search for invoices" button on the customer information screen.*)
3. A **specification** is a documented representation of a feature. ("*The system shall have a search for invoice option which, when clicked, will display a list of all invoices associated with that customer in a new window. The display will be sortable by date issued, date payment received, invoice number and amount."*) The specification is the design target for a feature.

2.3 Specifications versus Designs

A specification is a description of what the system will look like and how it will behave after it is built. It describes how the finished product functions and how it interacts with other systems and users. It describes the logic and algorithms that are incorporated into the design as well as a description of the system interfaces. The specification describes the goal of the design and construction processes.

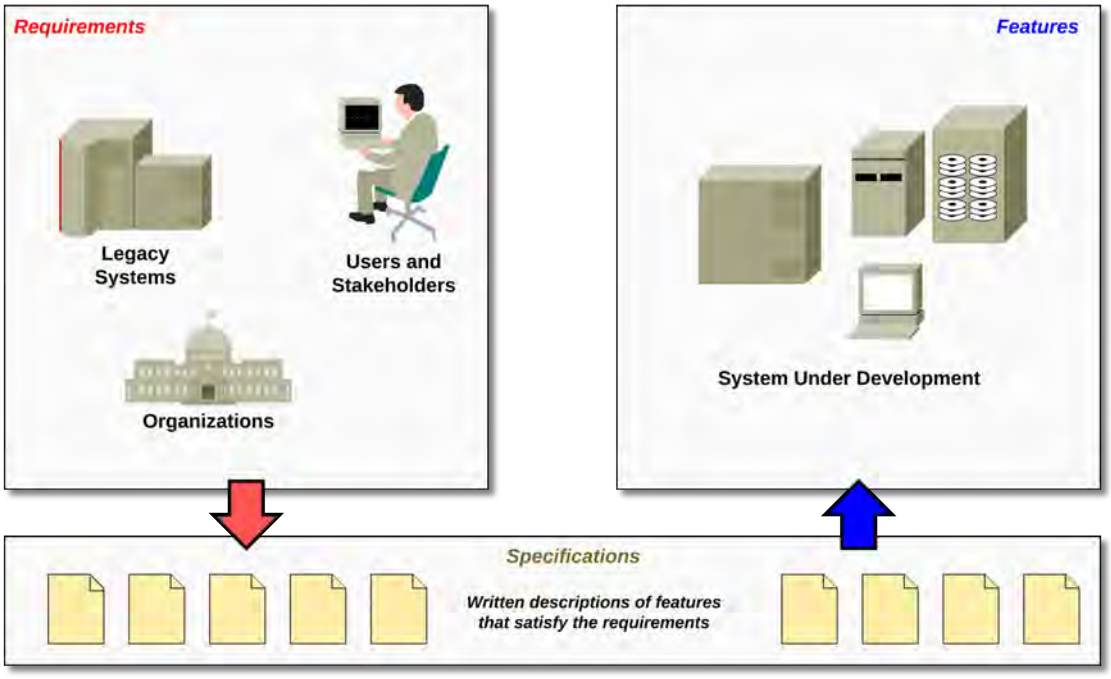
The goal of a specification is to paint a picture of the final result. There are no restrictions as to what form a specification can take as long as it communicates effectively. A specification may include text, diagrams, screenshots, acceptance tests, videos and any other kinds of data.

A design is a blueprint. A design describes how a specification is to be built given a specific set of resources and a given set of architectural constraints. The specification is architecturally neutral and one specification can be used to produce a number of different designs. A design is always expressed with respect to a given architecture or technology.

For example, a series of screen shots that describe how a website should look and a set of scenarios describing how a user interacts with it is a specification. To make this into a design we have to choose a technology, Angular perhaps, and then plan the steps to make those screen shots and scenarios into working web pages using ANgular. The same specification can also be used as the basis of another design, this one perhaps using PHP or Ruby on Rails.

This distinction is very important if the specification is used as the basis of a Request for Proposal when a software project is being contracted out on a competitive basis. If the specification suggests a design then any of the losing bidders could challenge an awarded contract on the basis that the suggested design in the specification gave another bidder an unfair competitive advantage.

For example, a specification that suggested using Java Server Pages for a website would be called biased because it gives an unfair advantage to developers who specialize in Java over those that use Angular.



To summarize:

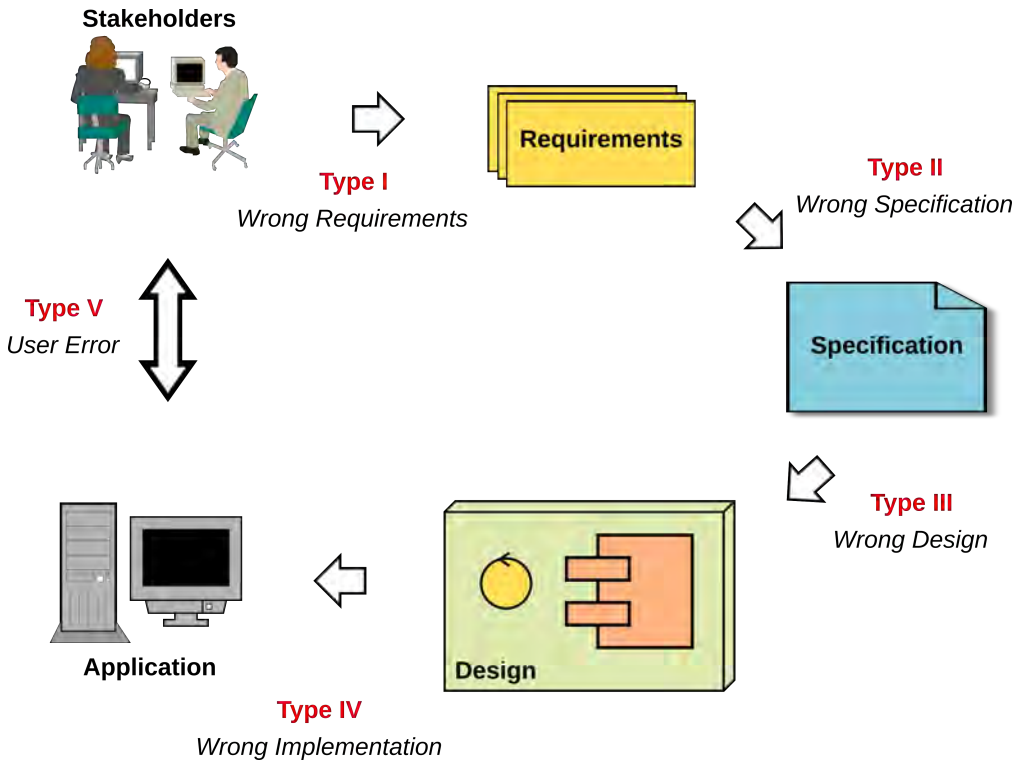
Requirements describe the stakeholders. Requirements describe the needs, wants, wishes and desires of the stakeholders.

Specifications describe the solution. Specifications describe the system to be built. It serves as a design target and a basis for discussion between the stakeholders and developers about what the system should look like and what it should do. It also serves as a baseline for testing and quality assurance. The solution described in the specification should satisfy the requirements of the stakeholders as much as it feasible to do so within resource constraints.

Designs are blueprints that describe how to build what the specification describes. A design describes how the to build the solution given a specific set of resources. This means that a single specification may be implemented by a range of designs. This is the old idea that there's more that one way to skin a cat – presuming that cat skinning can be somehow described by a specification.

2.4 Where Things Go Wrong

Five types of systemic errors can commonly occur in building a system. Each of these error types represents a risk factor that has to be controlled for in the project development life cycle.



Type I: Requirements are wrong. This is the most difficult kind of error to correct since it cannot be detected within the project but can only be detected with the participation of the stakeholders who are outside the development effort. In the worst-case scenario, identification of the error takes place after the finished product is deployed and being used by the unhappy and angry stakeholders.

Type II: Specifications do not satisfy the requirements. Our system may work perfectly, but it does the wrong thing perfectly. Generally, a type II error occurs if there are requirements that are not addressed by the specification (omissions), features proposed that provide functionality that does not meet any requirement (surprise functionality), or a proposed feature that does not satisfy the requirement correctly (failures).

Type III: System design does not meet the specifications. Following the design leads to a system that does not operate as specified. For example, the specifications may require the use of encryption for security, but the encryption methodologies were omitted in the design by mistake. Or the architecture proposed cannot handle the projected throughput or exceeds the allowable response time.

Type IV: The system build and artifacts do not conform to the design. This generally means that someone made mistakes in building the system whether writing code or performing some other construction task that was specified in the design – omitting error checking code for example.

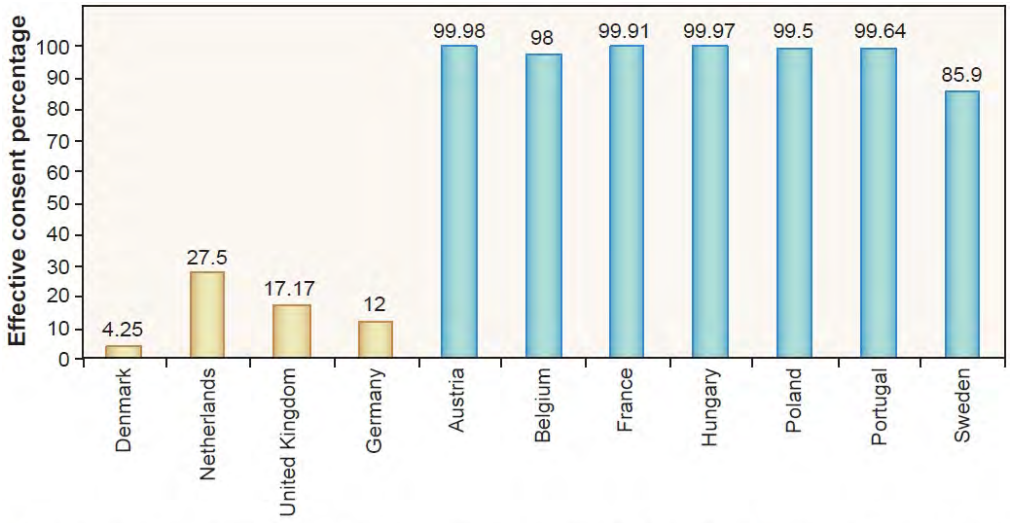
Type V: User Error. This type of error is an unavoidable human error. The problem is that people do not always behave rationally and, while we may have designed and implemented the system perfectly, it may be used in an irrational or unanticipated manner.

Type II - IV errors are called transcription errors, a term coined back in the 1970s by Glenford Myers, because they generally occur when someone has misunderstood (i.e. incorrectly transcribed) deliverables of the previous stage of the process. In other words, we get a wrong design because someone misunderstood the specifications, and we get wrong specifications because someone misunderstood the requirements.

2.4.1 Behavioural Economics

One of the confounding factors that we have to deal with, especially for the type V errors mentioned before, is that people do not behave rationally. The science of economics makes a number of predictions about how markets and people will act based on a primary assumption that that people behave in rational ways to maximize their own utility in the choices they make. Very often, game theory is used to model choices because its models are based on that same assumption.

However, we discover that in real life these predictions often fail. A primary reason for this failure is that while people often do behave rationally, people also behave in predictable but irrational ways when they make choices. The following is an example of what we mean.



Effective consent rates, by country. Explicit consent (opt-in, gold) and presumed consent (opt-out, blue).

In a paper entitled The meaning of default options for potential organ donors by Shai Davidai, Thomas Gilovich, and Lee D. Ross, the authors studied the underlying reasons for different rates of organ donation in different countries. The following is from their paper and describes the study[1]:

> *Rates of participation in organ donation programs are known to be powerfully influenced by the relevant default policy in effect ("opt-in" vs. "opt-out"). Three studies provide evidence that this difference in participation may occur in part because the requirement to opt-in or opt-out results in large differences in the meaning that individuals attach to participation.*
>
> *Some countries (e.g., Germany and The Netherlands) have explicit consent, or "opt-in," policies whereby citizens must indicate their willingness to have their organs harvested in the event of a fatal accident. Other countries (e.g., Austria and Belgium) have presumed consent, or "opt-out," policies whereby citizens must indicate their unwillingness to participate. This difference across countries provides a very telling natural experiment. The difference in organ donation rates – typically exceeding 90% in opt-out countries and failing to reach even 15% in opt-in countries – astonishes most readers.*

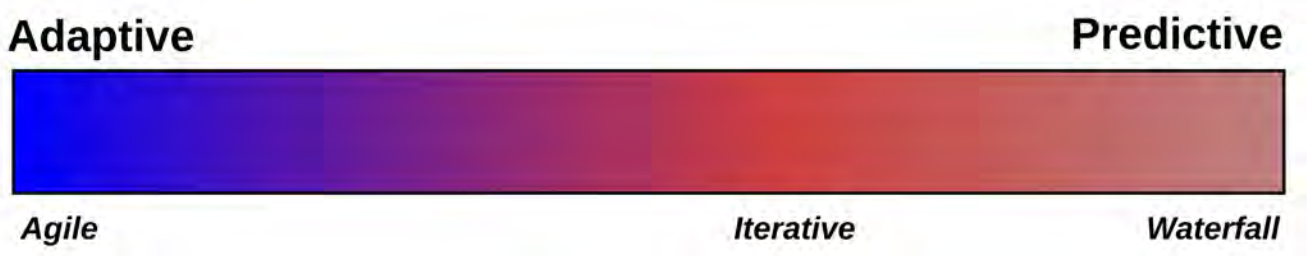
What this study shows is that the deciding factor on the donation rates is whether or not a person has to check a box to opt in or opt out. And later, based on what choice the donors made, they viewed the value of organ donation differently, or one would say they rationalized their behaviour after the fact.

The new field of Behavioural Economics, founded by Richard Thaler, studies these irrational behaviours. What we understand at this point is that every decision people make is for a reason, just not always rational reasons. Behavioural Economic models are allowing us to understand this irrational behaviour so that we can understand how our users and stakeholders will actually use our applications.

3 Process Types

While all processes follow the generic steps, they are often implemented in different ways to respond to different situations. One of the main ways processes differ is along what we call the adaptive-predictive dimension.

3.1 Predictive versus Adaptive Processes



Highly adaptable (agile-like) methodologies are designed to respond quickly to changing requirements and architectural constraints which makes planning, at best, guesswork since the changes and unexpected discoveries between the development of a plan and the product delivery will probably invalidate planning assumptions.

Highly predictive methodologies work by planning the future in detail. The project plan is developed for a specific known development target. Changes to that plan almost always cause rework. Predictive teams usually use a change management process so that only significant changes are considered for incorporation into the project.

The essential difference between the two approaches is that predictive methodologies know what is to be built beforehand while adaptive methodologies do not, often because they are trying to discover a solution to an existing problem.

In terms of requirements, predictive methodologies can identify all the requirements in the early stages of a project and can assume those requirements will not change significantly during the project. Adaptive methodologies are used when only some of the requirements are known at the start of the project or it is expected that requirements will change while the project is underway, often in unpredictable ways.

One of the decisions many organizations have to make is "How agile should we be?" There is a common misconception that agile is the same as having no process. It is not, it is just a different kind of process. One of the key factors that motivates the different methodology types is how well we know the requirements at the start of the project.

3.2 Predictive Processes

Many IT projects are highly predictive which allows for accurate planning in terms of time, budgets and resource allocation, although we tend to find these sorts of projects in more engineering-oriented contexts – embedded control systems, device and sensor automation and other similar areas.

In the Agile world, the waterfall process gets a bad rap as being fundamentally flawed or broken. This is not a justified criticism. The waterfall and its variants have been used successfully for thousands of years. While the waterfall had been very successful in the engineering world, it was not as effective in the software development world for highly user interactive projects because:

1. Requirements change much faster for software projects than for traditional engineering projects and most software organizations have no or little experience with change management.
2. Users are not often sure of their requirements until they actually start to use the software. As they use the new technology, they tend to come up with new requirements and change their existing requirements.
3. Software technologies and their economics change rapidly, often during the development project. In particular, the introduction of different functional services into the marketplace changes the overall set of user expectations about what applications should do for them. For example, the popularity of Google makes users expected that they should have some kind of search functionality in their applications.

3.3 Iterative Processes

An iterative process is one in which a significant portion of the requirements are known at the start of a project, but not all of them. The expectation is that during the project, requirements and other factors will be discovered which may have a significant impact on the final result.

To accommodate this flux in requirements, iterative methodologies tend to look like predictive methodologies but with “checkpoints” during the project where the final goal can be reevaluated and re-planned if necessary.

This is usually done by chopping the project up into mini-projects. A specification is produced early on that is either:

1. **Incomplete:** That means there are pieces missing from the specification since we don't know all the requirements.
2. **Putative:** While we can propose a specification, we expect that as we discover more requirements, the specification will change accordingly.

The most successful of the iterative methodologies was the Rational Unified Process.

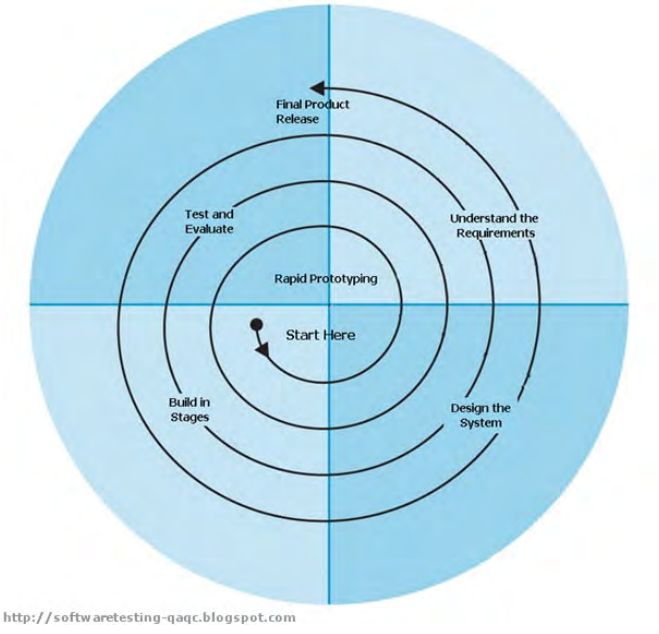
3.4 Adaptive Processes

Highly adaptable processes are needed when the final solution cannot be accurately predicted. Adaptive processes are used when a problem needs to be solved, or when there are very few identifiable requirements and no obvious solution at the start of the project. In this case, the developer often has to take a few known requirements, craft a solution to those requirements, then present that partial solution to the user. Using this early prototype, the developer works with the user to find more requirements to incorporate into a second prototype. This cyclic process continues until the user declares that the result is “good enough” so that the development process can be considered as over and the software delivered.

Adaptive processes are characterized by the requirements being in a state of constant flux. Multiple attempts at a solution for a single requirement may be necessary until we get something that works. Projects are usually complete when the problem is solved or when the business side essentially says “good enough.”

3.4.1 Non-Agile Adaptive Processes

Just as a point of interest, adaptive processes in software development have been around since the 1970s. One of the more popular adaptive processes that gained prominence in the past was Barry Boehm’s Spiral methodology from 1986. It defined a series of iterations with the objective of producing prototypes in each iteration which were used to provide inputs into the subsequent iterations.



Another popular adaptive methodology was James Martin’s Rapid Application Development (RAD) which he developed in the 1980s while he was at IBM. The model was built around the idea that for some sorts of development, like working with user interfaces, the requirements are too fluid for a predictive approach. The RAD approach, like the Spiral methodology, cen-

tred around getting prototypes into the hands of the users to start generating feedback that would be used to continuously improve the product.

3.5 Agile Development Processes

In the late 1990s there was a rebellion against the high level of ceremony and formalism of RUP, which had become quite popular. Starting with Extreme Programming, several methodologies started to emerge that shared a common core of ideas:

1. Close collaboration between the development team and business experts.
2. Face-to-face communication as opposed to written documentation.
3. Frequent delivery of working prototypes to facilitate developer and customer communication.
4. Small, tight, self-organizing teams.
5. Sets of techniques to "craft" code and organize the team to anticipate the inevitable "churn" of constantly changing requirements.

These methodologies referred to themselves as "light-weight", "agile" and "low ceremony" and were intended for smaller projects with fast turnaround times that did not need the formality or complexity of the RUP style methodologies.

Martin Fowler notes[2]:

> *Agile methodologies developed as a reaction to these [heavy-weight] methodologies. For many people the appeal of these agile methodologies is their reaction to the bureaucracy of the engineering methodologies. These new methods attempt a useful compromise between no process and too much process, providing just enough process to gain a reasonable payoff. The most immediate difference is that they are less document-oriented, usually emphasizing a smaller amount of documentation for a given task. In many ways they are rather code-oriented: following a route that says that the key part of documentation is source code.*
>
> *Agile methods are adaptive rather than predictive. Engineering [heavy-weight] methods tend to try to plan out a large part of the software process in great detail for a long span of time, this works well until things change. So their nature is to resist change. The agile methods, however, welcome change. They try to be processes that adapt and thrive on change, even to the point of changing themselves.*
>
> *Agile methods are people-oriented rather than process-oriented. The goal of engineering methods is to define a process that will work well whoever happens to be using it. Agile methods assert that no process will ever make up the skill of the development team, so the role of a process is to support the development team in their work.*

On February 11-13, 2001, the famous Snowbird conference took place where seventeen representatives from a number of methods like Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development and Pragmatic Programming, and others that were attempts to satisfy the need for an alternative to documentation driven, heavyweight software development processes. Out of that meeting came the "Agile Manifesto."

The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions	over	Processes and tools
Working software	over	Comprehensive documentation
Customer collaboration	over	Contract negotiation
Responding to change	over	Following a plan

We follow these principles:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

3.6 Extreme Programming

Many of the concepts and techniques in Agile testing, such as TDD and using acceptance tests to drive overall development, originated in an Agile methodology called Extreme Programming or XP. The following brief overview of XP is derived from the description of the methodology by its creator, Kent Beck.

Extreme Programming (XP) is an agile methodology created by Kent Beck. XP is a set of values, principles and practices for rapidly developing high-quality software that provides the highest value for the customer in the fastest way possible.

XP is extreme in the sense that it takes 12 well-known software development "best practices" to their logical extremes, and these, in turn, are the XP core. TDD or Test Driven Development is one of those core practices as is the concept of refactoring.

XP is intended to be easily used for projects of up to a dozen programmers and twice that with some difficulty. While XP itself does not scale well, the idea to large scale XP development is that a project is organized overall along more traditional models, but is then split into multiple smaller XP projects.

The other point that we have to keep in mind is that we can have both traditional and Agile methodologies within a single organization or project. The two are not mutually exclusive.

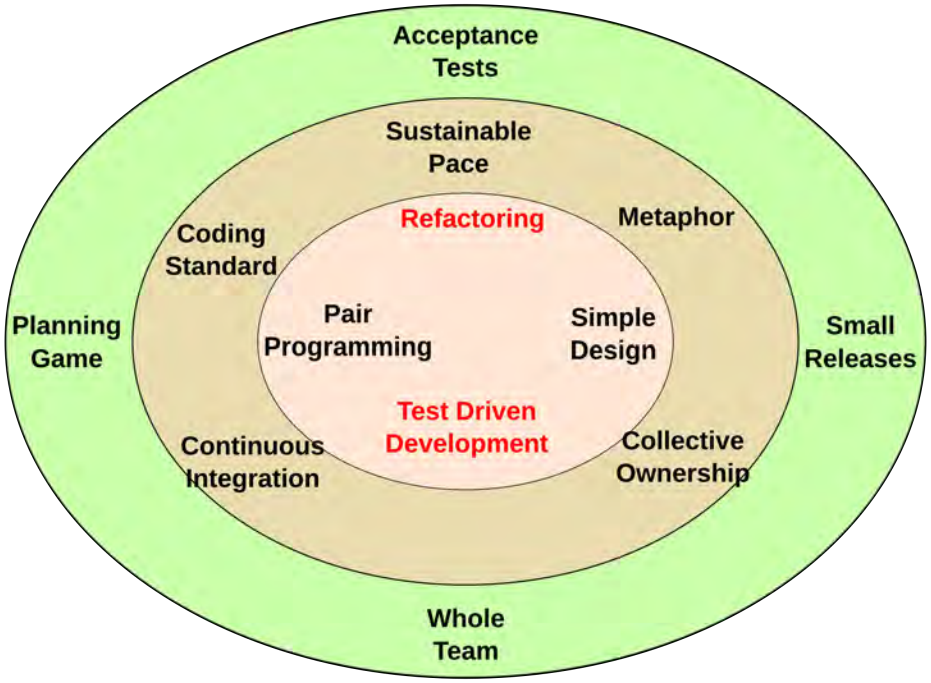
For example, we may have a project to automate a customer management system for a bank. Banking development projects are usually high governance with layers of stakeholders putting in their requirements (state banking commissions, state and federal agencies, risk management departments, security departments, etc.)

The overall project probably will be done by a waterfall approach (typical in banks since they are very risk-averse and tend to require reams of documentation, sign-offs and all those other "fragile" constructs). However, suppose that with in the project is a requirement to have an "Account Manager Workbench" – a computerized interface that the account managers will use to do their work, which was up to this point done manually.

The requirements people talk to the account managers who do not have any idea what this should look like (after all, they've never seen one before). Suddenly, we have an Agile subproject! Using Agile methods, we can develop a workbench in concert with the account managers who will evolve their requirements as they see the technology unfold.

3.6.1 The XP Onion

XP is seen as a sets of practices within practices. Each layer of the onion feeds into and defines the activities in the layer beneath it.



The Planning Game: Customers and developers cooperate to produce the maximum business value as rapidly as possible. Customers come up with a list of desired features for the system. Each feature is written out as a User Story, which gives the feature a name. Developers estimate how much effort each story will take, and how much can be done an iteration. Customers prioritize the stories to be implemented, as well as when and how often to produce a production release of the system.

Small Releases: Start with the smallest useful feature set. Release early and often, adding a few features each time.

Metaphor: Each project has an organizing metaphor, to provide an easy to remember naming convention.

Simple Design: Always use the simplest possible design that gets the job done. The requirements will change tomorrow, so only do what's needed to meet today's requirements.

Continuous Testing (TDD): Tests are written for a feature to be added, then the code is written. When the suite runs, the job is done.

Refactoring: Refactor out any duplicate code generated in a coding session. You can do this with confidence that you didn't break anything because you have the tests.

Pair Programming: All production code is written by two programmers sitting at one machine. Essentially, all code is reviewed as it is written.

Collective Code Ownership: No one "owns" a module. Anyone is expected to be able to work on any part of the codebase at any time.

Continuous Integration: All changes are integrated into the codebase at least daily. The tests have to run 100% both before and after integration.

40-Hour Work Week (Sustainable Pace): Programmers go home on time, up to one week of overtime is allowed.

On-site Customer Tests: Development team has continuous access to a real live customer, that is, someone who will actually be using the system who does continuous test generation and acceptance.

Coding Standards: Everyone codes to the same standards so that there is no way to tell by inspection who on the team worked on a specific piece of code.

3.6.2 XP Evolving Architecture

XP embraces change which means that an architecture and design "evolve" rather than being initially specified as something the application has to "fit into," in XP architecture has the sense of “the essential structure.” (XP tends to like Zen-like statements.)

Architecture and design are expressed in XP through:

1. Spike
2. Metaphor
3. First Iteration
4. Small Releases
5. Refactoring
6. Team Practices

In XP, design evolves because it is driven by the requirements of the customer, which usually evolve and change as the system is developed. Refactoring is not just writing code, evolving design is not the same as a haphazard or unstructured design.

In fact, if the XP developers do not apply good design principles to the emerging design, it will become brittle and non-functional in a very short period of time. We will look at a couple of fundamental principles a bit later on.

One of the important uses of refactoring is to simplify the code. One of the benefits of refactoring for design is that the simplified code is usually better designed and more robust so that it is more readily changeable or resistant to changes in other parts of the system.

3.6.3 Spikes

During the release planning game, the team develops "spikes" which are quick throw-away potential solutions. Based on the user stories and the spikes, an architectural approach will emerge that will suggest the system’s structure.

Stories suggest constraints that guide architectural choices. One story might identity security and access constraints while another might suggest performance levels for multiple users. This may suggest a spike simulating multiple simultaneous users that might provide insight into replication and rollover designs.

The spikes allow the developer and customer to examine the costs and trade-offs of the various proposed solutions. Spikes are used to experiment with implementations of functionality and how the solution would impact the existing system structure.

As an example of a spike, we might sit down with a customer that talks about his clients being able to place an order online and have the results feed into the existing corporate order database. A spike might be produced that looks at developing a solution round a web server, and REST web services that would allow back end connections to the existing database as well as to a new "on-line customer" database. Issues explored in the spike might revolve around the security and encryption of customer data, as well as firewalls and other system security features.

3.6.4 Metaphor

From the initiation of the project, an effective metaphor is developed that helps guides the solution. The metaphor is like the analysis model or the high-level design in RUP in that it describes the solution in the language of the problem domain. The metaphor describes the main conceptual objects and their relationships and provides a "check-point" or a high level view of a system that keeps the design oriented or on track. The metaphor also provides a common language for naming and talking about the artifacts of the system.

The metaphor may change as more is discovered about the customer's domain. The metaphor is not tied to the architecture but suggests and influences the architecture. The system design supports the metaphor.

As an example of a metaphor, consider a project to develop web access for a bank's customers. In the metaphor we talk about customers, accounts and other "things" that have conceptual reality to the customer. However, at the system architecture level, there is no real "account" but rather just a record in a database, and this architecture allows us to support the metaphor. For example, we may now have a requirement to allow a customer to list all transactions on their account. By keeping the account metaphor, this extension of functionality is easier to implement than if we started at the database view.

3.6.5 First Iteration

Michael Hill calls the first iteration a ZFR (“ziffer,” Zero Feature Release). The first iteration does nothing but puts an initial architecture (often called the architecture spike) into place. The first iteration is guided by a selection of user stories that have architectural significance – i.e. they force the whole system to be considered. This is the working skeleton since the first release has no functionality. The focus is making sure that the solution is installable and configurable.

The idea of a first iteration is well known to every programmer. If you are starting to work in a new programming environment, the very first program you are likely to write is the "Hello World" program. It does nothing, but it usually does force you to make sure the environment is set up correctly, the compilation and link tools work, and all of the other configuration issues have been resolved.

In the RUP approach, the architecture defines the final form of the system, the design process implements the design into that skeleton. In the XP approach, the first iteration defines the starting form of the system and will be modified as the system evolves. The first iteration is used also to identify risk as we walk through the architecture.

3.6.6 Small Releases and Refactoring

By creating a constant stream of small releases, we are able to monitor the impact of design changes and additions as we add the implementation of the user stories. Each release is a checkpoint: new functionality may break our existing design, so we have to go back and make modifications to the design to accommodate the new requirements.

Remember that a spike is a throw-away solution to see how to solve a particular problem – spikes help us choose low-risk ways to extend or modify the architecture and design. Refactoring allows us to change the design as we move forward. As we add new functionality, we may have to change the design of parts of the system without changing their functionality. Constant refactoring allows us to keep the design fluid and flexible while not giving up robustness.

3.6.7 Team Practices

Each pair of programmers is responsible for integrating their code into the code base at opportune moments, for example, when the unit tests all run at 100% or another chunk of the planned functionality is finished. Only one pair integrates at any given time.

4 Process Maturity

The Software Engineering Institute (SEI) kick-started the process maturity movement in software with the publication of the Capability Maturity Model or CMM. We are not going to deal with the CMM except in a very superficial way as a lead into a discussion of OOAD development processes.

Unfortunately, the concept of process maturity has been taken over to a degree by faddish gurus who sell it as a magic formula or silver bullet that will fix whatever software ills bedevil your organization, which unfortunately has detracted from the incredible importance that process maturity has in making the engineering process work effectively – not just for software development but in any sort of development.

4.1 Crisis? What crisis?

In the 1980s, the US General Accounting Office study on large government projects found that for contracted software:

2 %	was used as delivered
3 %	was used after changes
19 %	was used but extensively reworked or abandoned
29 %	was paid for but not delivered
45 %	was delivered but not successfully used

The US Department of Defence realized that this poor quality of the software components of its systems was impacting the development of US military systems. They understood the need to correct this situation and “to advance the practice of software engineering because quality software that is produced on schedule and within budget is a critical component of U.S. defence systems.”

The DoD put out a tender with the goal of finding a solution to this problem. The tender was won by Carnegie Mellon University which then created the Software Engineering Institute to address this issue.

4.2 The Software Engineering Institute

From the SEI Website www.sei.cmu.edu

The Software Engineering Institute (SEI) is a US federally funded research and development center sponsored by the U.S. Department of Defense through the Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics [OUSD (AT&L)] The SEI contract was competitively awarded to Carnegie Mellon University in December 1984.

The SEI is a preeminent software engineering R&D technology center. The SEI provides the technical leadership to advance the practice of software engineering so the DoD can acquire and sustain its software-intensive systems with predictable and improved cost, schedule, and quality.

The SEI mission includes four objectives:

1. accelerate the introduction and widespread use of high-payoff software engineering practices and technology by identifying, evaluating, and maturing promising or underused technology and practices
2. maintain a long-term competency in software engineering and technology transition
3. enable industry and government organizations to make measured improvements in their software engineering practices by working with them directly
4. Foster the adoption and sustained use of standards of excellence for software engineering practice

The Two-Part Solution.

The research carried out by the SEI pointed to the fact that organizations that produced quality software were consistently characterized by two factors:

1. They have an explicit and well-defined software development process; and
2. They always followed the process. The more they stuck to the process, the better they were at producing software.

SEI defines a software development process as:

> *The means by which people, procedures, methods, equipment, and tools are integrated to produce a desired end result.*

4.3 Defining Maturity

The second finding of the SEI should not have been as surprising as it was to the software development community. Just having a defined process was not enough, it had to be part of the organization's culture so that it became “The way we do things around here.” In other words, merely defining a process or using it intermittently, even if it is an Agile process, does not produce results – it is the commitment to using the process consistently that produces results. When an organization adopts a way of doing things as the norm (in the sense that deviations from this norm are sanctioned in some way) we say that the process has become “enculturated” or has become part of the corporate culture.

Process maturity can be thought of as always following the process, and when a new situation arises that the process does not handle, the process is modified so that it can handle that situation the next time it is encountered. In a mature organization, the process is not fixed or static or followed blindly, instead it is constantly evolving as the organization adapts to changing internal and external factors.

Process maturity is process agnostic. Any process, as long as it is appropriate for the organization and its needs, can be used to produce high-quality software provided the organization follows it. This came as a bit of a blow to those organizations that kept switching processes looking for the right process that would solve their problems – the SEI data demonstrated that it is not the process you adopt that is the critical success factor but rather it is using the process and sticking to it that makes the difference.

Looking at other industries, we see process maturity as being the factor that often separates the excellent from the so-so organizations, whether it’s in construction, medicine or government.

Consider that we learn from experience as individuals; when we encounter a new challenge, we figure out how to meet it so that the next time we come across a similar challenge, our experience tells us how to deal with it, we don't have to re-invent the wheel every time we encounter that situation.

In the same way, a mature process provides the same capability for an organization to learn from its experience; the challenges that arise that are outside what the process can handle are the stimuli for improving the process so that it can handle those challenges the next time the organization encounters them.

4.3.1 Mature versus Immature Organizations

The SEI describes the difference between mature and immature organization as follows.

Immature Organizations

1. The process is often improvised by the developers and management during the course of the project.
2. Defined processes are often not followed or enforced.
3. Project activities are reactionary and often focus on solving immediate crises.
4. Schedules and budgets are based on unrealistic estimates and routinely exceeded.
5. Bringing a project in line with deadlines or costs causes the functionally and quality of the product to suffer.
6. Product quality is difficult to predict and there are no objective standards to assess product quality.
7. There are no standard ways for solving problems, everything is done on a “fire-fighting” basis and the same problems keep recurring with the same negative impacts.
8. The defined process is not realistic and the organization is characterized by the attitude “Following the process won't get the job done.”
9. The defined process is often created in part by those who are not part of the development staff and tends to be more about reporting and management activities than work tasks.

Mature Organizations

1. The software process is standardized across the organization and communicated to all existing staff and new employees.
2. All work activities are carried out according to the planned process.
3. The processes are usable and realistic because they describe how work actually gets done.
4. Defined processes are updated as necessary.
5. Improvements to the process are planned and evaluated through pilot tests or cost-benefit analyses.
6. Roles and responsibilities are defined clearly in both the organization and in each project.

7. Management continuously monitors the quality of the products and the processes using objective and quantifiable criteria.
8. There are defined processes for analyzing and solving problems with the product and process.
9. Budgets and schedules are realistic because they are based on historical performance data.
10. The infrastructure of the organization supports the process.
11. A disciplined process is consistently followed because all of the participants understand the value of doing so and enforce following the process at a peer level.

4.4 The Capability Maturity Model

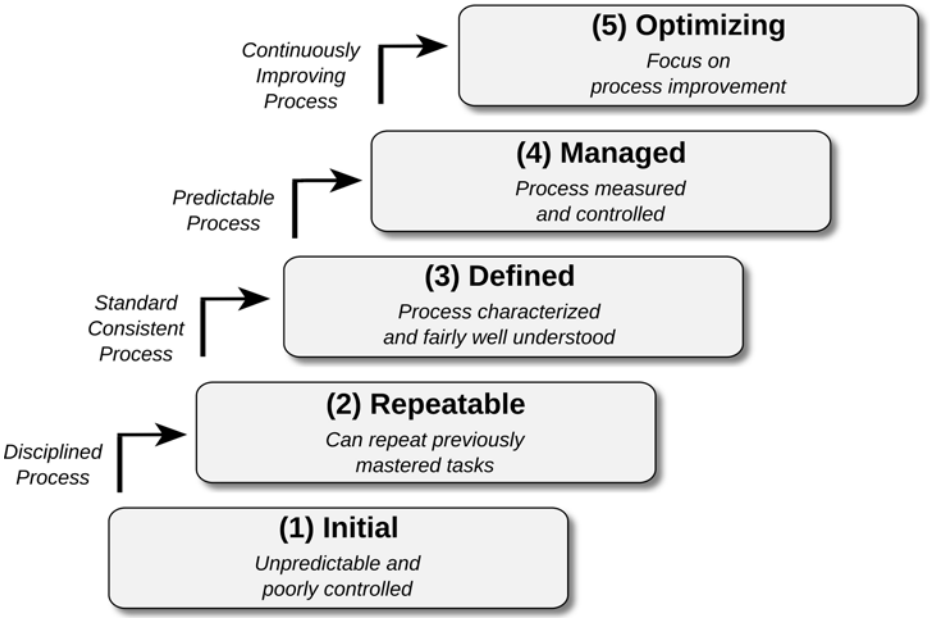
The Capability Maturity Model (CMM) has been one of the most well-known results of the SEI process maturity studies and has led to a slew of different maturity models. The maturity level concept has been applied to other areas: People CMM, Software Acquisition CMM, Software Engineering CMM, Integrated Product Development CMM for example. However, in this course we are interested more in the essence of the maturity models and leave exploring the details of specific formal maturity models as a self-directed activity.

4.4.1 Maturity Levels

As an organization matures, it passes through five different maturity levels. There is more formal definition involving a set of process goals and other metrics, but this is really beyond the scope of this course. Instead, we want to just get the fundamental essence of each level. The levels are evolutionary -- organizations cannot “skip levels” because the levels are not just based on the software development process, but describe the underlying organizational culture, its commitment to their process, and its overall approach to quality and continuous improvement.

The SEI describes the levels this way:

1. **The Initial Level**: Also referred to as the tribal level. The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics. There is a reliance on “gurus” or key people who know the product and/or process as opposed to documentation or any form of corporate memory. There is little or no learning from past experience.
2. **The Repeatable Level**: Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.



3. **The Defined Level:** The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.
4. **The Managed Level:** Detailed measures of the software process and product quality are collected so that the process and products are quantitatively understood and controlled.
5. **The Optimizing Level:** Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

Think of the stages of growth of a person. While a person is at the child stage, they are taken care of by a parent, are not expected to have a job nor make important decisions themselves. This stage of maturity is entirely appropriate for a six-year-old. Where it becomes dysfunctional is when a person is still at this level of maturity when they are 25 years old.

Similarly, all organizations start at level one as start-up. But as they grow, they have to mature or else they tend to not be able to meet the challenges of an established business. A lot of businesses fail because they experience a lot of growth, but the company is still be managed as a start-up – usually because the founder insists on still running the company that way. The founder can make all the decisions when there are three employees at the beginning but not when there are 3,000 people on staff.

I find that it is easier to conceptualize the levels like this:

Level 1: No Process Level

In a lot of the literature, this is often made to sounds like some dystopian, chaotic free for all. However, when you take a look at a lot of organizations at this level we can separate out two basic types.

The first is the start-up where the emphasis is on the personal skills and productivity of a few key people. It's counterproductive to have a software development process when there are only two people – the programmer and someone running the business side. In fact, it is probably counterproductive to have a formally defined process while the organization is still trying to figure out what it's doing.

The second is the organization that has grown past the start-up stage and is still hasn't made the transition to process-based production. The organization is still being run as a start-up with reliance on key individuals (*"We can't let Chani go on vacation because she is the only one who knows how our database works."*) Eventually, these organizations either have to mature or they tend to fail.

Level 2: Repeatable Results

What characterizes organizations at this level is that every project is on time, on budget and to spec. Projects are not late, over budget or defective in any way. The reason is that the organization has mastered all of the development processes needed to replicate success on every project from project management to vendor management to software testing.

This is also the level that is hardest for most companies to reach since it requires a major reworking of the corporate culture. As one of my clients pointed out *"It took us 10 years to get to CMM level 5, but 8 of those years were spent getting to level 2."* This is fairly typical – most organizations spend most of their time on their way to level 5 trying to get to level 2.

Level 3: Standardized Process

This level is characterized by a standardized process that has various implementations; however, what really defines a level 3 maturity is that the process is enculturated. What that means is that it is seen as *"the way we do things around here"* and is enforced by the people in the organization, not by management. In a level three organization, if you are not following the process, then it is your peers that are likely to call you out on it.

The result of having a standardized process is that the organization is able to collect data on their process – identifying what works and what doesn't. Having a standardized process (not just defined but actually in day to day use by everyone) allows QA to actually be done effectively.

If you think of the organization like a basketball team, level one is a bunch of guys showing up at the court and just throwing the ball at the hoop now and then. To get to level two, each individual player has to master the skills of dribbling, passing, shooting, defending as well as

learn the rules of the game. At level three, we can take those five level two players and integrate them into a team that can run plays on the court.

Level Four: Proactive Quality and the Review Culture

Once an organization has been a level 3 for a while, it can start to prevent defects from occurring by developing best practices and establishing a review culture. For example, if we know from experience that certain design decisions result in certain kinds of errors, we establish a best practice that says something like *"Don't do x because these bad results will happen; instead, do y because that will work."* These best practices are based on the organization's past experiences and what they have learned from previous projects.

The reviews follow naturally from having these standards or best practices. When work is done, it is reviewed to ensure it has followed the best practices or standards of the organization. However, these best practices and standards themselves are also constantly evolving to keep the corporate knowledge base of best practices current.

The key characteristic of level four is that everything is reviewed as a normal part of the process activities.

Level Five: Optimization

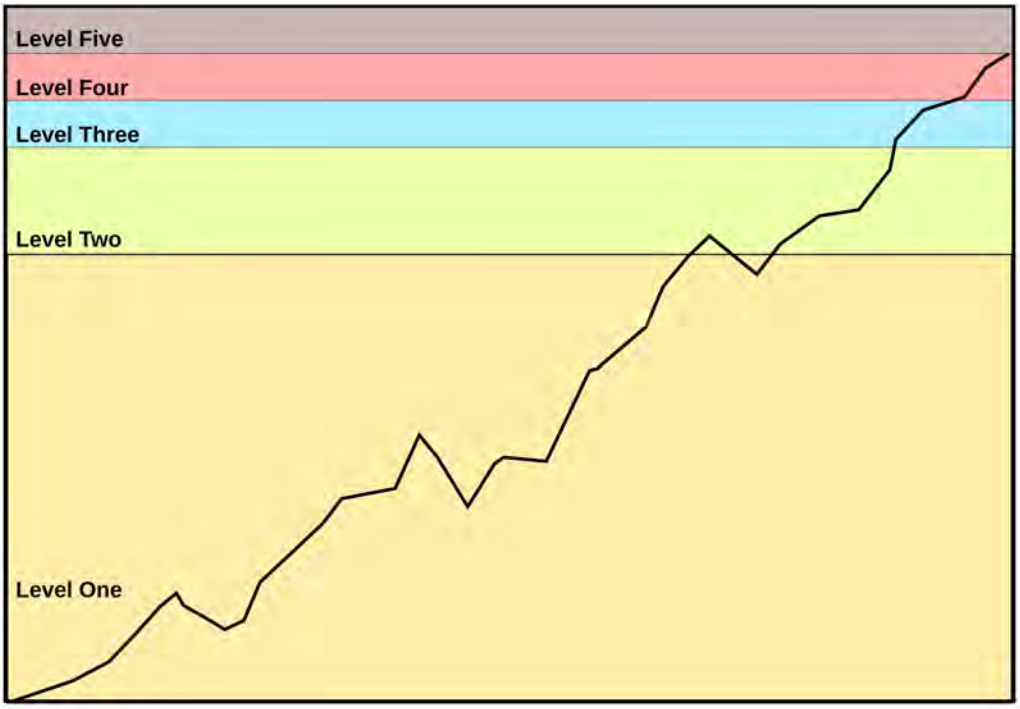
At level 4, work is done then reviewed – at level 5, the best practices and reviews are integrated into the work itself. Level 5 can be thought of as work being reviewed and tested as it is being done. A simple example of this is a spell checker. Level 4 spell checking is analogous to running a spell checker on your document while level five is like having the spell checker identify misspellings as you type and offer on the fly suggestions or automatically correct common typos.

4.5 Observations on CMM

While the material so far presented is from the academic point of view, one of the aspects of CMM I have observed is that it works because at some point the people in the organization embrace it and refuse to go back to the old way of doing things.

One aspect of the move towards CMM that is not apparent from the nice even presentation of levels is that it is a long and difficult journey and that getting to level two represents a major accomplishment for many organizations.

In fact, a typical path might look life diagram on the next page where CMM level is on the vertical axis and time is on the horizontal axis.



4.5.1 Getting to Level Two

This is often the hardest step for most organizations. To be a level two organization, they have to master all the aspects of software development processes and the supporting disciplines like software testing, configuration management, contract management and project management. At level two, organizations run projects very well and are consistently delivering on time, on budget and to spec.

At level two, we have a professional organization that has good process practices, the only problem is that they may not consistent throughout the organization. Why does it seem so hard to get to level 2? The fundamental reason is that there are a lot of basics to master from testing to project management to technical expertise. Remember our basketball team – to get to level two, every player has to develop mastery of the individual basketball skills. This sort of mastery requires sustained effort.

4.5.2 Getting to Level Three

Once the organization has mastered the skills necessary for good process management, it can start to define a corporate wide process. Usually this does not go well initially because there is a tendency for groups within the organization to revert to what works – their own individual processes because those high quality level two processes got the job done efficiently and effectively.

However, as the new process is used, even though it hurts initially, the organization starts to learn from past mistakes and is able to improve the process until it starts to return clear benefits to the people in the organization. When this happens, the people start to use the process,

embrace the process, and make the process the “way we do things around here.” The process becomes part of the corporate culture.

An organization arrives at level three when the process is supported and enforced by everyone. If someone doesn’t follow the process, their peers are more likely to correct them than their superior is.

Once this happens, moving to levels four and five are merely a matter or time.

Anecdotal Observations

I had the following conversation, or fairly close to this, with two twenty-something software developers in a CMM level five organization in Palo Alto CA.

> **Me:** What's it like working in a level five organization?
>
> ***Developer:*** *It is onerous dude. Like, if I want to make a code change there are all these steps I have to go through. First I have to justify why I want to make the change, then I have to do an impact analysis on what the change will do, and if that gets approved, then I have to do a code proposal, and if that gets past a review then we do a code walkthrough, write the test cases and then, and only like a third of the time we get this far, I get to write the code.*
>
> **Me:** And that is onerous?
>
> ***Developer:*** *Dude, when I did my PhD at Stanford, I would just like wire myself up on Red Bull and let the code flow in a three-day coding binge. But this is so onerous I quit.*
>
> **Me:** You mean you just quit now?
>
> ***Developer:*** *No, I quit last year and went to work for a start-up.*
>
> **Me:** But you're here. Now.
>
> ***Developer:*** *Yeah, here I work nine to five from Monday to Friday. At the start up I was pulling 60 hour weeks and spending my weekends debugging and handling all sorts of crisis crap.*
>
> **Me:** So why is this better? I thought you would like the excitement of a start-up.
>
> ***Developer:*** *Hey. I'm 25 and single in California. I don't want to spend my weekends writing code when there are so many beautiful women out there on a Saturday night looking for guys as cool as me. I'll take onerous for that trade off.*

The other employees commented that while they didn't share his specific reason for working in a level five company, they did agree in principle that being in a level five organization meant that they didn't fight fires, manage crises, work over time, get emergency calls or work on trying to fix bugs. As one person put it, *"when I go home, I go home. As long as everyone follows the process, my job is under control, but woe be to the person who improvises and makes my life stressful as a result."*

4.6 Process Immaturity

The five CMM levels can be thought of as describing the aspirations of a normal organization. It has been suggested by some analysts that the CMM level scale can actually extend in the negative direction. This work was first proposed by Finkelstein as an immaturity model that was appropriate for dysfunctional organizations or companies in crisis.

This work was extended by Capt. Tom Schorsch of the US Air Force in his CIMM (The Capability Im-Maturity Model[3]) who suggested that a number of the companies identified as being at level one are in fact even worse since there was no measure of dysfunctional processes in the original assessment. He also points out that level one assumes the organization wants to get better, but he proposed negative levels for organizations that just don't care about improving.

Schorsch suggested that there are four levels below level one which can be described as:

0. Negligent: Indifference

Failure to allow successful development process to succeed. All problems are perceived to be technical problems. Managerial and quality assurance activities are deemed to be overhead and superfluous to the task of software development process. Reliance on silver bullets.

-1. Obstructive: Counter Productive

Counterproductive processes are imposed. Processes are rigidly defined and adherence to the form is stressed. Ritualistic ceremonies abound. Collective management precludes assigning responsibility. Status quo über alles.

-2. Contemptuous: Arrogance

Disregard for good software engineering institutionalized. Complete schism between software development activities and software process improvement activities. Complete lack of a training program. *“Industry-wide processes and standards won’t work for us because we are unique and unlike anyone else.”*

-3. Undermining: Sabotage

Total neglect of own charter, conscious discrediting of peer organizations software process improvement efforts. Rewarding failure and poor performance.

5 Application Lifecycle Management

Application Lifecycle Management (ALM) is a set of concepts that define a collection of best practices in software development. ALM is not a methodology or a process – like Agile or waterfall – but focuses instead on broader management concerns that are common to all development process methodologies, whether Agile, waterfall or iterative.

The primary difference between a software development process and ALM is that most development processes only deal with activities up to the delivery of the finished product. ALM takes the broader perspective of "software as a product" which means managing not just the development phase but also the period leading up to the start of the development process, the time the software is in production, being maintained and then eventually retired. ALM adds in the stages of the engineering process that are missing from most development methodologies. ALM is intended to support and enhance existing development methodologies.

The problem with most standard software development processes that ALM addresses is that these development processes terminate as soon as the product is deployed. The product is handed over to a support team or, even worse, the users are left to fend for themselves. ALM is intended as an over-arching strategy for managing development no matter what sort of process is in play.

There are a number of different implementation sof ALM currently in use. IBM, for example, has incorporated ALM as a framework called "Collaborative Lifecycle Management" into their Rational Team Concert and Software Engineering product lines.

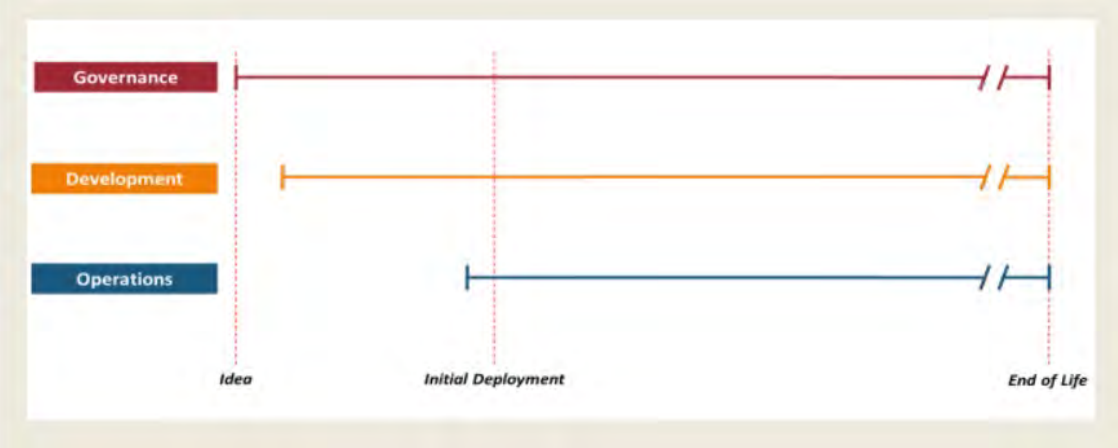
5.1 Software as Product

This material is derived from James Chappell's excellent summary of ALM from his 2014 paper sponsored by Microsoft[4].

A common description of an application's lifecycle is "the entire time an organization spends money of the product from the initial idea to the end of the application's life when it is no longer in use." The diagram following shows that there are three events that are considered milestones in an application lifecycle:

1. **Idea or Inception:** This occurs in advance of the start of the development process and usually involves evaluating whether to develop an application, examination of the business case, deciding whether to build or buy, as well as other portfolio management issues and feasibility considerations.
2. **Deployment:** This occurs when the finished product is moved into production and made available to users. This event marks some sort of hand-off between the development team and a support team.

3. **Retirement:** At some point the application has to be retired. In the simplest case, the functionality the application provides is no longer needed and the application can just be removed. Typically, retirement is a more complex event. For example, it may involve migrating massive amounts of data into a new application or ensuring that automation support for some critical process is not interrupted.



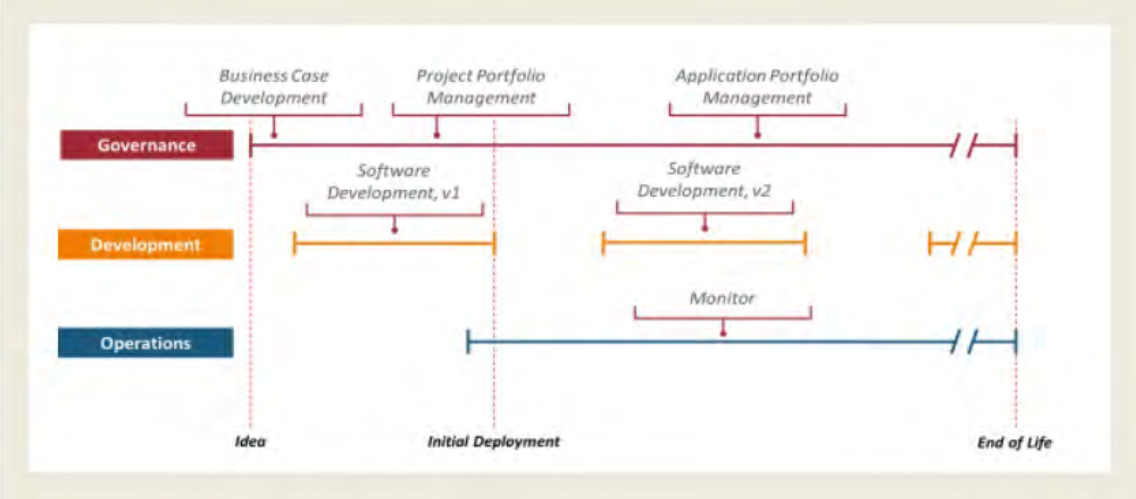
Spanning these milestone events are three main areas of concern. These focus on broader aspects of a product lifecycle than the concerns of the software development process being used. The development process can be thought of as addressing the tactical issues of software development from the perspective of the developers and others directly involved. ALM, on the other hand, addresses the strategic issues of the development process from the organizational perspective.

5.2 Governance

Governance starts with business case development and feasibility studies. Once the development begins, governance is implemented through project portfolio management where the role of the application in the overall business context and IT architecture is continuously evaluated, often with respect to organizational-wide metrics, standards and valuations. Once an application is deployed, it becomes part of the organization's portfolio of applications, and as such is an organizational asset like any other and is subjected to an ongoing assessment of its associated benefits and costs.

Some of the decisions that are the responsibility of portfolio management are deciding when updates and revisions, or even retirement, make business sense, ensuring that the application does not duplicate the functions of other applications, or leaves gaps in functionality that could create business liabilities.

One role of governance that is becoming critical for many organizations is the assessment of security risks and liability arising from issues like hacked data, compliance with new regulatory frameworks or other legal and social concerns. These issues may result in a decision to mitigate risk by retiring an application and migrating functionality to other applications.



5.3 Development

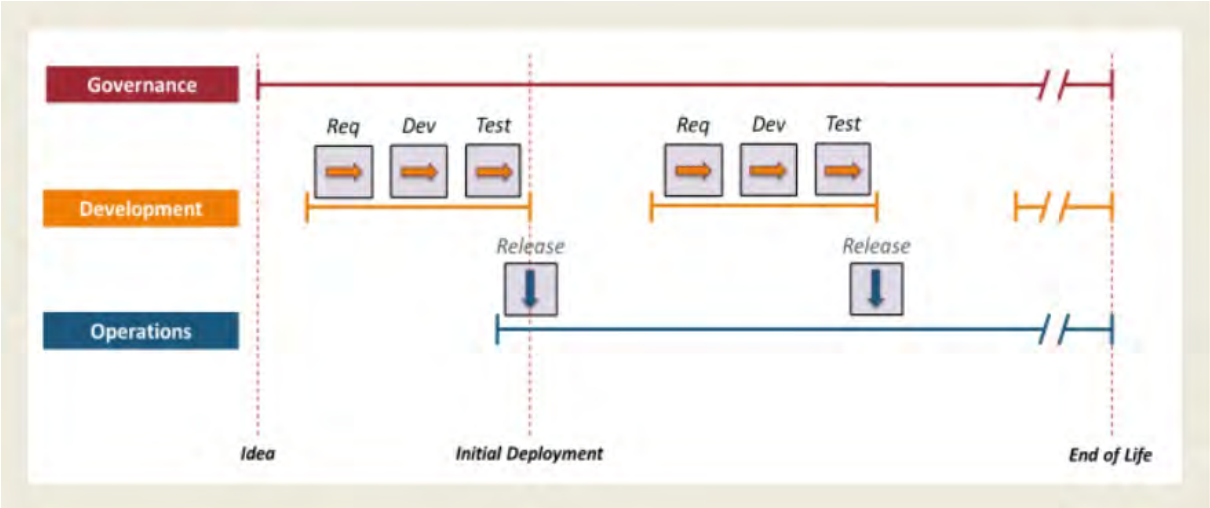
ALM does not dictate the actual tasks done in a software development process, but takes a more strategic view of the development period. A typical issue that ALM deals with is the choice of software development processes to be used both in development and during deployment for support and enhancement programming; these may be different processes.

ALM is also concerned with the transmission of information and artifacts to ensure those working in development at any point in the product lifecycle have the correct and accurate information they need to accomplish work in a timely and cost-effective manner.

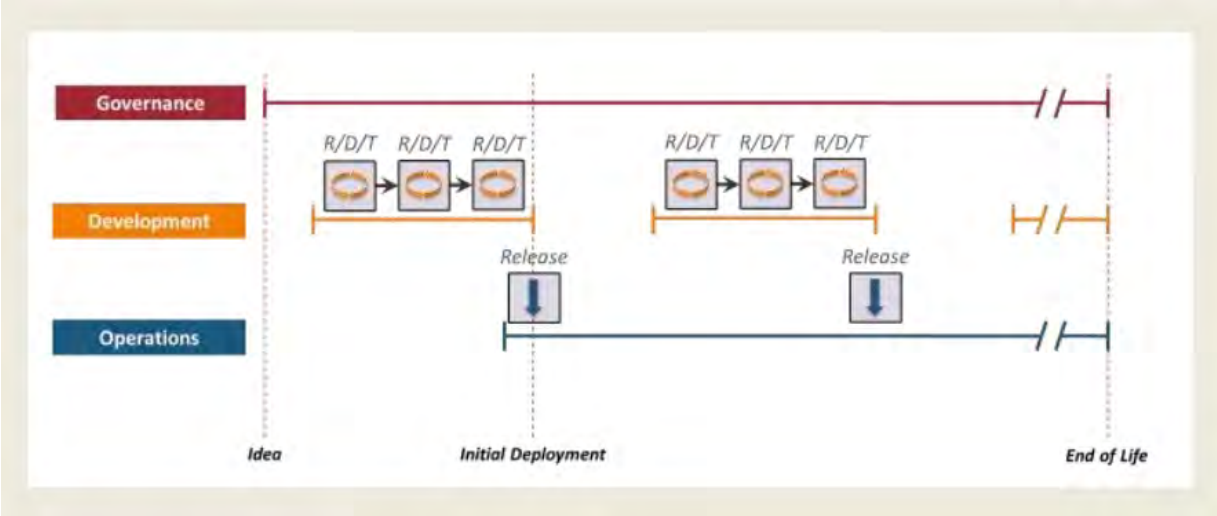
The current version of ALM integrates with the three main SDLC approaches used today: traditional waterfall, Agile methods and DevOps style continuous development. Each model of integration breaks all the SDLCs down into three fundamental activities:

1. **Requirements:** working out in detail what the application should do.
2. **Development:** building the code and other software artifacts.
3. **Test:** ensuring that what was built is in fact what was supposed to be built.

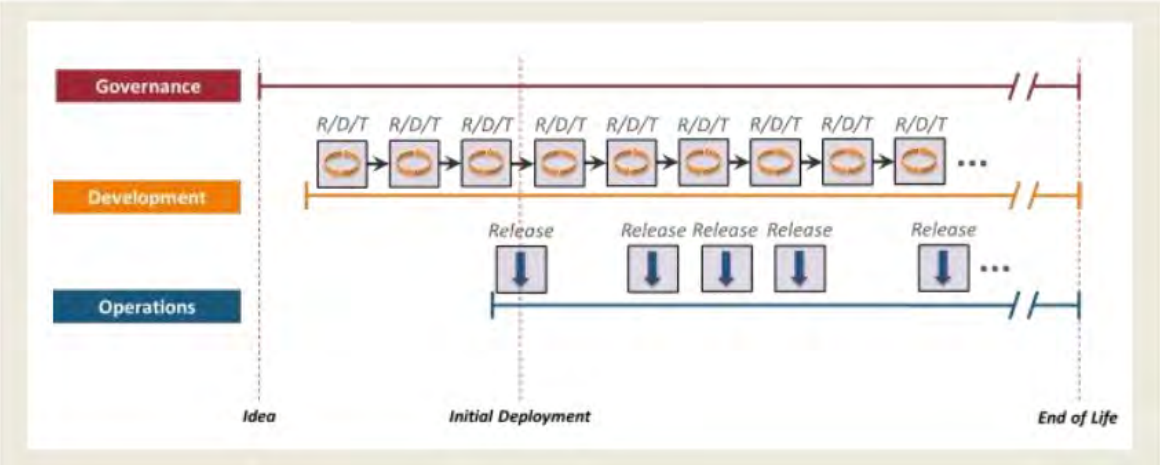
Where the SDLCs differ is in the timing and organization of the activities. The diagram following shows the ALM with a traditional waterfall SDL:



The next diagram following shows ALM with a standard Agile development where the three activities are cycled through iteratively.



The final diagram show a continuous delivery (DevOps like) process integrated with ALM.



5.4 Operations

During deployment, governance makes decisions and development makes changes. Operations provides the context for making those two threads work together. Operations monitors usage, performance, service level targets, change requests and bug reports. Operations also monitors changes in the application's technical environment, such as operating system changes or architectural changes, that could impact the application.

This data is used by governance to make a decision to effect some change, and by development to quantify and scope the needed changes. Operations also manages the integration of the product into its operational environment, including the installation of any upgrades and enhancements. We usually find that the feedback from operations is used as input into any ongoing requirements activities.

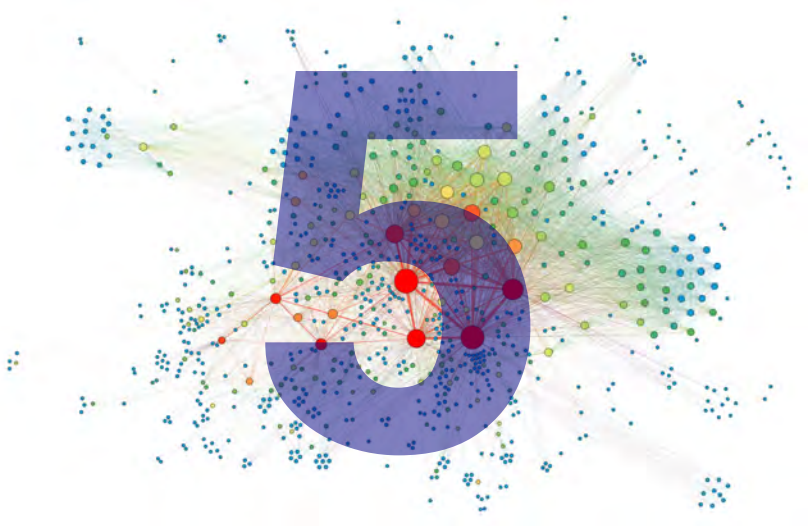
Notes and References

1. Davidai, Shai & Gilovich, Thomas & Ross, Lee. (2012). The meaning of default options for potential organ donors. Proceedings of the National Academy of Sciences of the United States of America. 109. 15201-5. 10.1073/pnas.1211695109.

2. https://www.martinfowler.com/articles/newMethodology.html

3. "A Software Process Immaturity Model," ACM SIGSOFT, Software Engineering Notes, Vol. 17, No. 4, October 1992, pp. 22-23.

4. http://davidchappell.com/writing/white_papers/What-is-ALM--Chappell.pdf



Object-Oriented Analysis and Design: Visual Modeling with UML

USDP and UML

Standards are nature's way of codifying obsolescence.
Anonymous

Those who cannot remember the past are condemned to repeat it.
George Santayana

If history repeats itself, and the unexpected always happens,
How incapable must Man be of learning from experience.
George Bernard Shaw

We know why projects fail, we know how to prevent their failure;
so why do they still fail?
Martin Cobb, Treasury Board of Canada Secretariat

exgnosis

1 Introduction

In module one, we briefly recapped the historical forces that resulted in the emergence of object-orientation as the dominant software development paradigm in the 1990s. We also looked at how, as a paradigm is applied to production environments, the paradigm activities have to be integrated into a production process.

While the structured paradigm was the dominant mainstream paradigm in the 1980s, there were still problem domains where it just didn't work. A number of these, like artificial intelligence, robotics, real-time processes and event-driven systems, were of interest to the defence departments in various countries because they were the specific applications that were needed to computerize the battlefield.

The US DoD began the so-called "Star Wars" program through DARPA which funded research into these areas in both academia and the private sector. DARPA and defence agencies in other countries weren't the only source of funding, private corporations with deep pockets also started pouring money into what they hoped was the technologies that would transform their businesses.

During the 1990s, there was an explosion of OO development processes and methodologies, each with its own strengths and weaknesses. There were a number of attempts to create some sort of synthesis and simplification of the processes with the primary goal of developing a process that could be used at scale in industries with highly complex, mission-critical and industrial-strength computing requirements.

1.1 Origins of the Unified Software Development Process

The primary developer of the process was Ivar Jacobson who in the 1980s developed the Objectory Process while working with the telco Ericsson. His company was later bought by Rational Software whose principals, Grady Booch and Jim Rumbaugh were also trying to develop an OO development process which they called the Unified Method.

After Ivar Jacobson joined with Grady Booch and Jim Rumbaugh, his Objectory Process was merged with the Unified Method to become, after some refinement, the Unified Rational Process for Software Development.

In 1999, the Unified Software Development Process was released as a template for organizations to use to develop their own processes. This was, at the time, declared to be the official OO standard process by the Object Management Group (OMG) – an IT industry standards association.

Ivar Jacobson describes the Unified Process this way:

> *First and foremost, the Unified Process is a software development process, a set of activities needed to transform a user's requirements into a software system. However, the USP is more than a single process, it is a generic process framework that can be specialized for a large class of software systems, for different application areas, different types of organizations, different competence levels, and different project sizes.*

1.2 The Rational Unified Process (RUP)

The USDP is often confused with RUP, Rational Unified Process. According to Philippe Kruchten:

> *RUP is a process product. It is developed and maintained by rational Software and integrated with its suite of software development tools. It is a software engineering process developed and marketed by rational. RUP is a specific and detailed instance of a more generic process described by Ivar Jacobson, Grady Booch and James Rumbaugh in the text book The Unified Software Development Process*

The USDP is an industry-standard template an attempt to define a generic OO software development process that includes the "best practices" or "lessons learned" from both the structured type projects and the various OO methodologies that were developed by the different OO theorists.

For organizations, the USDP represents a starting point for the creation of their own specific process. Or they can buy the RUP "off the shelf." However, the current status of the standard is in some confusion because it seems to have disappeared. Some cynics have suggested that Rational did not think through the business implications of supporting a standard while trying to sell a process based on the standard that they, well, gave away for free.

2 Phases and Releases

The USDP repeats over a series of cycles that make up the life of a system. Each cycle concludes with a product release which is a product ready for delivery. Each phase terminates with a milestone that is defined by the availability of certain artifacts – models and documents – that have been brought to a particular state of completion.

The product, according to Jacobson, is defined as:

> *The finished product consists of a body of source code embodied in components that can be compiled and executed, the manuals and the associated deliverables. It includes the requirements, use cases, non-functional specifications and test cases. It includes the architecture and the visual models – artifacts modeled in UML. It is all the things that enable the stakeholders – customers, users, analysts, designers, implementers. Testers and management – to specify, design, implement, test and use a system. Moreover, it is these things that enable the stakeholders to use and modify the system from generation to generation.*

The cycle consists of four phases:

1. **The Inception Phase:** Define the scope of the project and the vision for the system. Often an associated business case is developed.
2. **The Elaboration Phase:** Plan project, specify features, baseline architecture.
3. **The Construction Phase:** Build and test product.
4. **The Transition Phase:** Deliver product to the end-user community.

2.1 The Project Phase Deliverables

Each phase ends when a particular deliverable is available. In many ways, the use of the term “deliverable” is misleading. I prefer to say that we know when the project reaches a particular point of development because a particular deliverable can now be completed. These deliverables by phase are:

2.1.1 The Inception Phase: Vision for the System

The concept for the system is developed into a vision of the end product and the business case for the product presented. The deliverable answers the following questions:

1. What is the system primarily going to do for each of its major users?
2. What could an architecture for the system look like?
3. What is the plan and what will it cost to develop the product?
4. The important risks are identified and prioritized, an outline for the critical subsystems is roughed out and the next phase planned out in detail.

As we will see in the next module, this phase deliverable incorporates a number of requirements best practices.

2.1.2 The Elaboration Phase: Baseline Architecture

Most of the use cases are specified in detail and the system architecture is designed to implement the most critical use cases.

The relationship between the architecture of a system and the system is paramount. A simple way to put it is that the architecture is analogous to a skeleton covered with skin but with very little muscle (the software) between the bones and skin – just enough muscle to allow the skeleton to make basic movements. [Booch99a:9]

2.1.3 The Construction Phase: Initial Capability

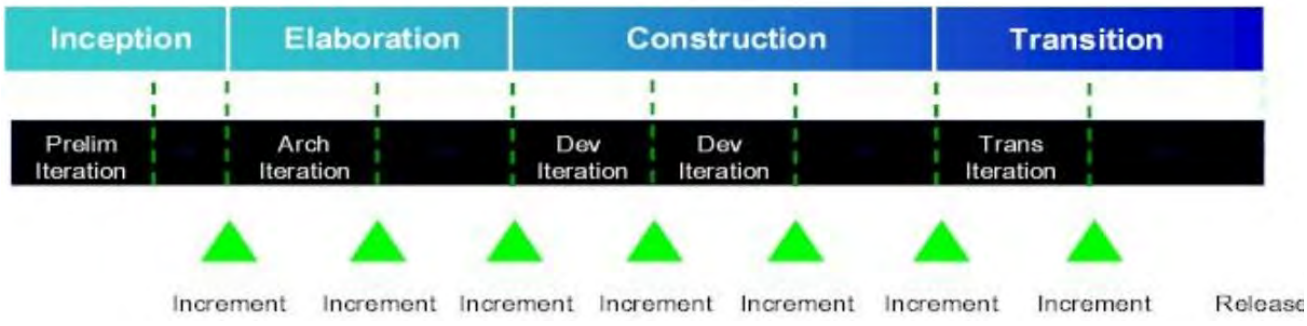
A system that is working well enough and contains all of the use cases that management and the customer have agreed on to implement (i.e. the muscle is added to the skeleton). This phase consumes the bulk of the project resources.

2.1.4 The Transition Phase: Product Release

The product moved to beta, users trained and other delivery issues resolved.

2.2 Iterations and Increments

Each phase is further subdivided into iterations. Each iteration can be thought of as a mini-project that produces an increment, or a chunk of growth, in the project.



Each iteration builds on the state of the development artifacts from the state they were left in at the end of the previous iteration. Iterations can range from two weeks to several months – it depends on a wide range of factors related to the goals of the project and the resources available.

The iterations allow us to do reality checks on the project at regular intervals The sorts of things that are evaluated for each iteration are:

1 **Requirements:** Have the requirements be re-prioritized? Have new ones been added? .

2. **Context Analysis:** Have we uncovered any other aspects of the problem domain that have a bearing on our design? Have we found something that has an effect of the priority of a requirement?

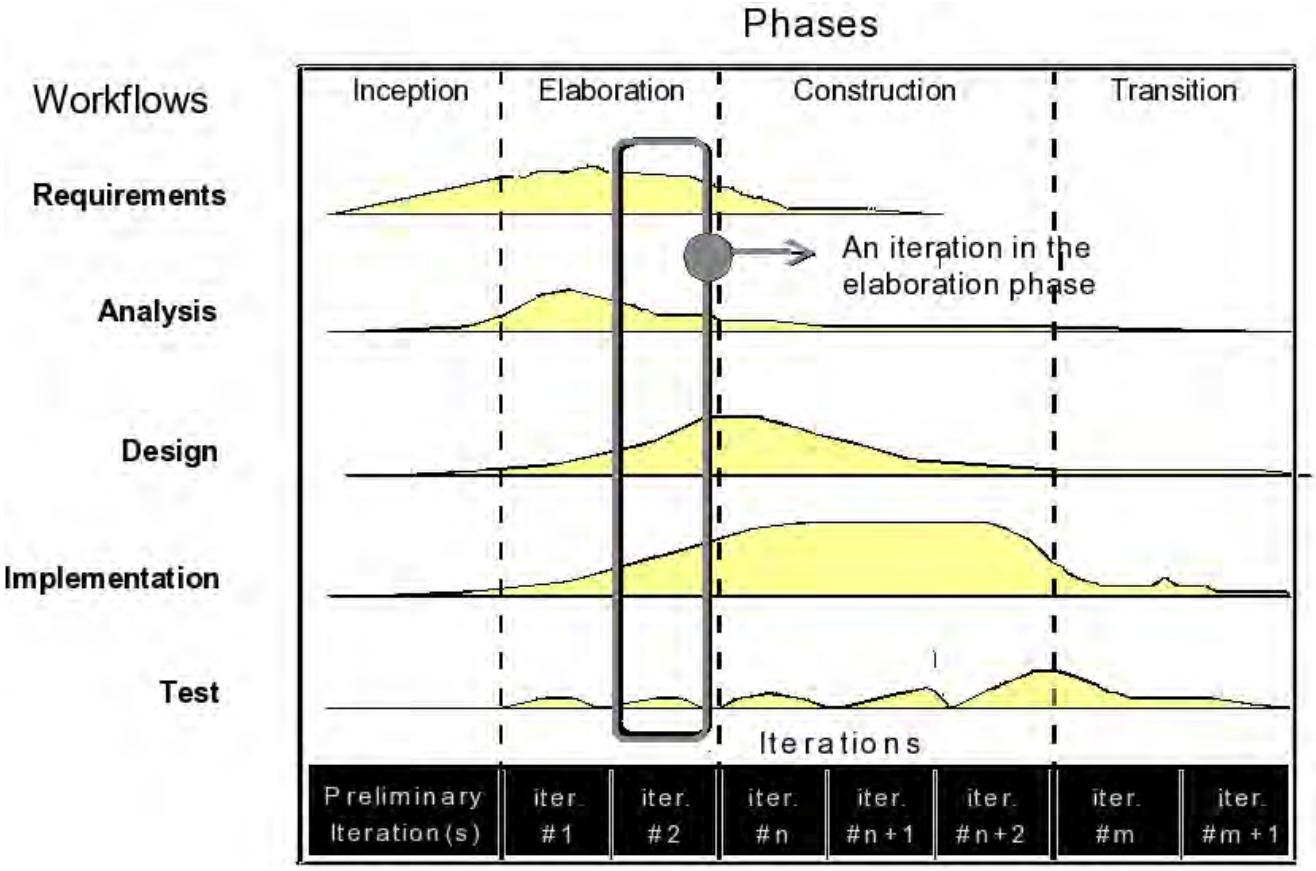
3. **Risk:** Have there been any internal or external developments that may require us to modify our original risk assessment?

4. **Architecture and Design:** Is our design and architecture still viable or have we uncovered confounding factors that may increase the likelihood of project failure?

5. **Test:** Is the system been tested to a level appropriate for its level of development?

3 Workflows

The innovative feature of the USDP that was a major break from the standard waterfall SDLC was the decoupling of the workflows from the phases of the project. The diagram below (prepared by Jacobson) shows the workflows along the left-hand side.



The curves roughly indicate the extent to which each workflow is being carried out in each iteration. Depending on where in a cycle an iteration occurs, the development activities being done in that iteration will be made up of different workflow to varying degrees.

The transition from one phase to another takes place when the software product reaches a particular stage of development or when certain milestones are reached. However, workflows are activities that don't have a specific start or end point. Instead, a specific workflow's activities peak during certain phases throughout the project.

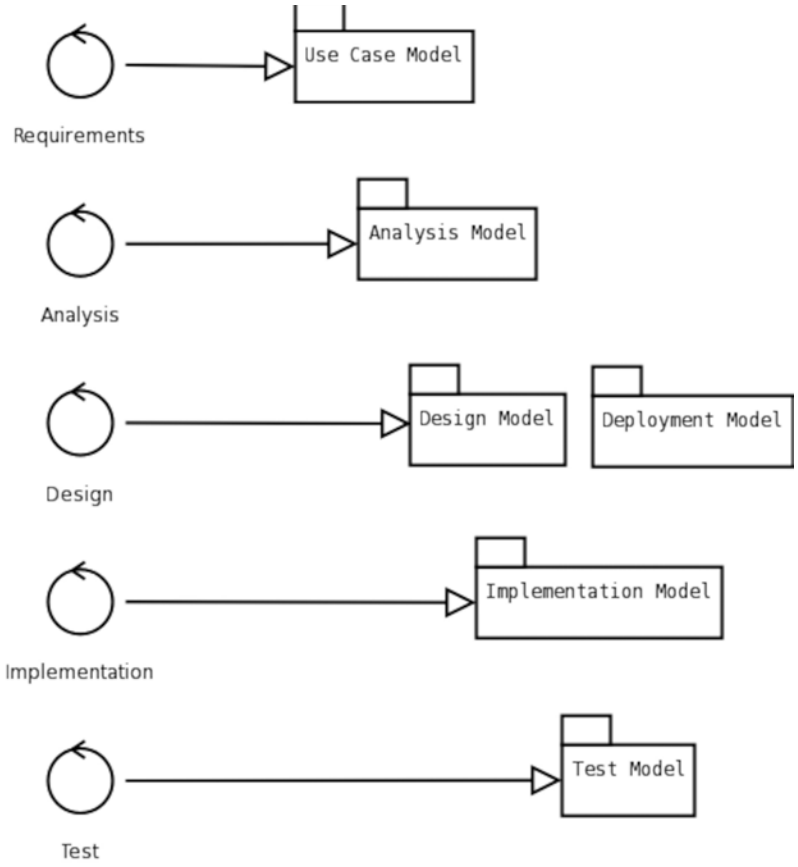
A workflow does not have a deliverable but is focused on developing and refining a model or set of models that will be in various stages of detail and completion during each phase milestone.

In another major break from the waterfall, the test workflow indicates testing activity happens within each iteration, even those that take place before an executable is available for testing. This was the first formal acknowledgment of the idea of continuous testing and quality control

throughout the development of the product.

3.1 Workflow Outputs

Workflows create models. Each workflow's models will be in different stages of detail and elaboration as each phase milestone is reached.



1. **Use Case Model:** Defines and describes system functionality through use cases.
2. **Analysis Model:** Refines the use cases and makes an initial allocation of the behaviour of the system to a set of objects that provide the behaviours. Creates an initial architectural overview by partitioning the system into packages.
3. **Design Model:** Defines the static structure of the system as subsystems, classes and interfaces. and realizes use cases as collaborations among the structural elements. Defines an implementation of the design model onto a specific physical architecture.
4. **Implementation Model:** Defines the software components (i.e. Source and compiled code modules, scripts, data files and other physical artifacts) and the mapping of classes to components.

5. **Deployment Model:** Describes the topology of the physical nodes of the computers and the mapping of components onto these nodes.
6. **Test Model:** The set of test cases and procedures required to fully test the system.

3.2 Use Case Driven

The Use Case model defines the system's functionality from the point of view of the users. All the other models go through intermediate stages while developing the system that finally delivers the functionality described in the use case model. These intermediate models are said to realize, literally "make real," the use cases. Each model is tested for conformance to the use case model so that all development activity is focused on implementing the specified functionality to avoid spurious "feature creep."

Each artifact in each model has a trace dependency to an artifact in the previous model, which in turn has a trace dependency to an artifact in its preceding model all the way back to a use case in the use case model. If we cannot justify the inclusion of a model element by showing a trace dependency back to some aspect of system functionality, we omit it.

The rationale behind the trace dependencies is that we do not want to design and implement components that will never contribute to the system. Trace dependencies are reality checks on how focused the design and testing efforts are.

3.3 Architecture-centric

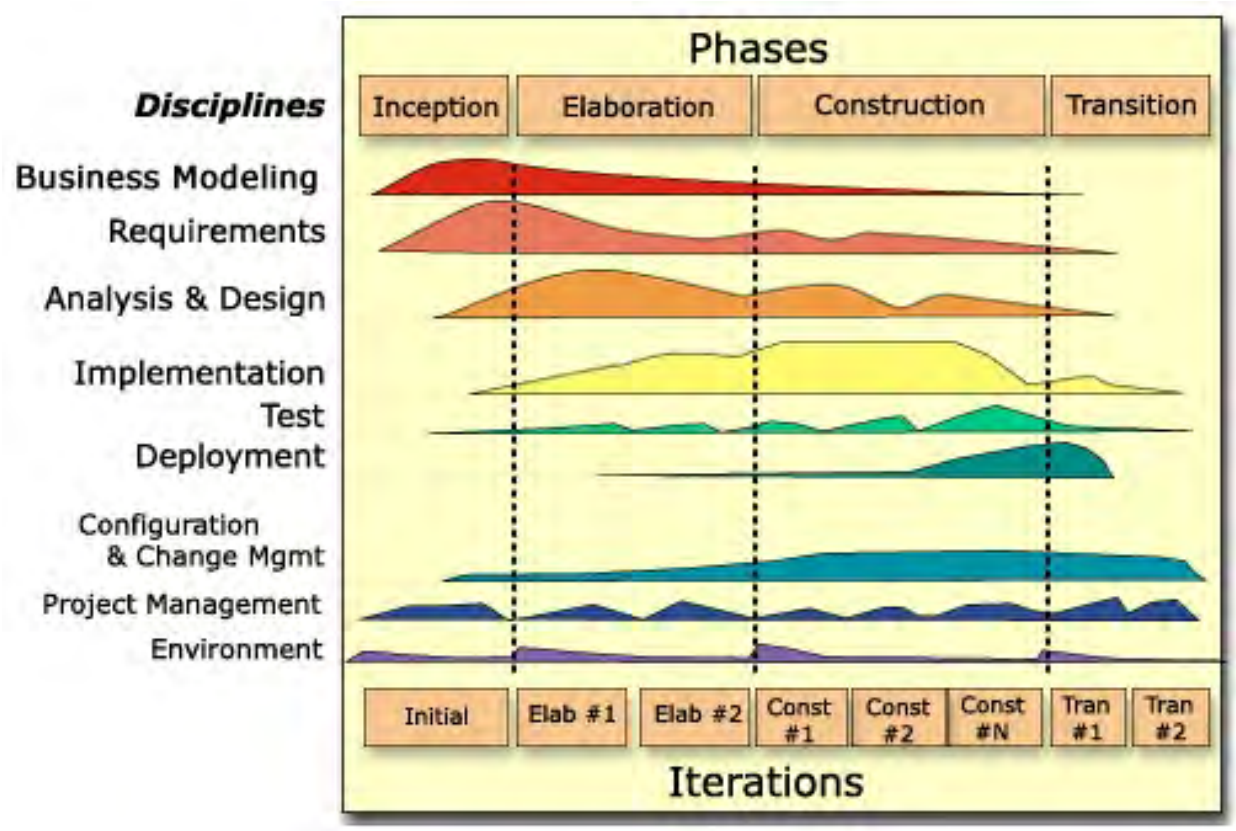
The use case model describes function, but the architecture of the system describes its form. The architecture takes into account the non-functional constraints such as: hardware platform, OS, DBMS, network protocols, deployment considerations, legacy systems and non-functional requirements.

Architects start with the key use cases that supply the core functionality of the system, usually only 5-10% of the use cases. These guide the architecture development by suggesting what other use cases to develop. The architecture in turn lets us select which of the remaining use cases can be reasonably implemented.

Architecture guides use cases, and use cases drive architecture.

3.4 The Rational Unified Process

For comparison purposes, the following diagram illustrates how the USP process framework has been adapted by Rational for their proprietary RUP.



One of the major differences between USD and RUP is how the USDP workflows are re-organized into disciplines in RUP. These are:

1. **Business modeling discipline:** Business modeling describes the business architecture, structures, data and processes that the system under development will be constrained by and have to interact with. This is often called the "Business or HighLevel Requirements" work flow.
2. **Requirements discipline:** Requirements describes what the system should do and allows the developers and the customer to agree on that description.
3. **Analysis and design discipline:** Analysis and design shows how the system will be built in the implementation phase. This is the work flow where we create the blueprint for the system development.
4. **Implementation discipline**: Implementation is where we build the system which consists mostly of programming and code development.
5. **Test discipline:** Test includes all the testing and QA activities performed in each iteration.

6. **Deployment discipline:** Deployment centres around the delivering product releases, to the end users. This includes creating builds, packaging, distribution, installation and troubleshooting.
7. **Configuration and change management discipline:** This discipline covers three subareas:configuration management (SCM), which is the identification of artifacts, versions and dependencies among those artifacts; change request management (CRM), which keeps track of the proposals for change; and status and measurement management.
8. **Project management discipline:** PM involves the planning and management of the phase plan, which describes the entire project, and the Iteration plans which describe the iterations in each phase.
9. **Environment discipline:** The focus of the environment discipline is on the activities necessary to configure or tailor the software development process for a specific project.

However, the major difference between USDP and RUP was in terms of what was envisioned to be. Users of the USDP were expected to adapt their existing process and paradigm activities into the USDP process structure. One of its primary goals was to allow those who already were using a waterfall or similar process to migrate to an OO based iterative process.

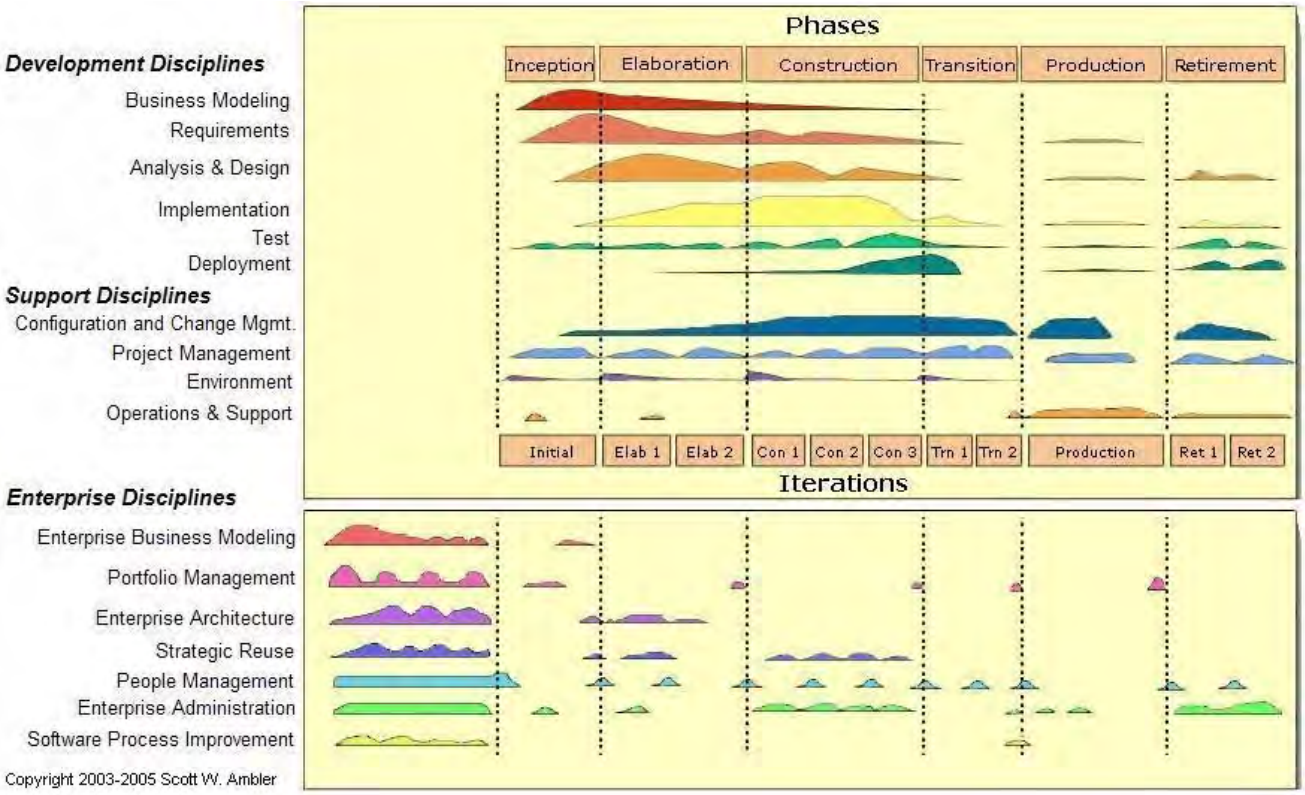
But many organizations that wanted to use USDP did not have an existing process and found the prospect of having to build one from scratch overwhelming. RUP was an out-of-the box process implementation of the USDP with a full set of detailed activities, templates and guideline.

It was not the USDP that the Agile developers claimed was too cumbersome to use, it was the specific RUP implementation. And they were right.

3.4.1 Enterprise RUP

Even and USDP and RUP became popular, the lack of ALM type activities started to prove problematic. A number of authors like Scott Ambler and Larry Constantine tried to extend RUP to cover the entire ALM or engineering development process.

The diagram following shows the proposed structure of the the Enterprise RUP.



Copyright 2003-2005 Scott W. Ambler

4 The Unified Modeling Language

In the early 1990s, there were a number of OOAD methodologies that were attempts to replace the structured methodology. OOP was well entrenched with languages like CLOS, Smalltalk and C++, so there was a scramble by each theorist to develop an OOAD methodology to develop those artifacts that feed into OO program development.

Why was there a scramble? The kinds of distributed systems that were being developed looked a lot more like the agented systems that OOP had been perfecting itself on rather than monolithic mainframe batch processes systems that had been built with the structured approach.

In other words, OO was going mainstream and not only getting the business credibility it needed but was also something that clients would actually pay for. So, starting in the late 1980s to the early 1990s, a number of theorists started popularizing their own brands of methodology.

Some of the key players and their methodologies were: Jim Rumbaugh's Object Modeling Technique, Grady Booch's Booch Method, Peter Coad's OOA and OOD, Ivar Jacobson's ObjectOriented Software Engineering and Rebecca Wirfs-Brock's Responsibility Driven Design. There were many others, like Edward Berdard and Bertrand Meyer who also had significant contributions to the subject.

Each theorist's methodology was used to some extent within the industry. While the methodologies shared a common foundation in OO conceptually, they all different notations, processes, methods and diagramming techniques to express those concepts and apply them in a practical manner.

Some methodologies had six or seven basic diagrams (like the Booch methodology) while others had up to twenty-five. The popularity of OOAD had created a methodological tower of Babel.

4.1 Standardization

OO theorists had the benefit of hindsight from watching a similar situation in the structured analysis and design (SA&D) notation wars of the 1970s and 1980s where hundreds of diagrams in the early years eventually were distilled down into four basic modeling diagrams: the data flow diagram, the flow chart, the entity-relationship diagram and the state transition diagram. Nobody doing SA&D today can probably even remember what a Michael Jackson diagram was.

The OO theorists knew that OO would not be widely accepted as a viable system development approach until standardization on a methodology occurred. The problem was that everyone naturally wanted their methodology to be the standard.

The first move to standardization came when Jim Rumbaugh left General Electric to join Grady Booch at Rational Software. Between the two of them, many estimated that users of the Booch method and OMT accounted for more than half of the OO practitioners, although some claimed that it was as high as 80% of all practitioners.

At the OOPSLA '95 conference, Booch and Rumbaugh unveiled version 0 .8 of the Unified Method and made the rather provocative statement that “the methodology war is over". They also added a third member to the team, Ivar Jacobson, whose company Objectory, had been bought by Rational. And thus started the legacy of the “three amigos.”

The response from the OO community to the announcement of the end of the methodology wars was swift and loud. First up to the microphone were the theorists who basically stated in one way or another “Who died and made you guys God?” Most theorists believed their methodology to be as good as or better than the Unified Method, which they rightfully saw as being mostly based of the OMT and Booch methodologies. Of course, you don't become a leading theorist without a healthy ego.

However, it was the companies that made software that competed with Rational's tools (like Rational Rose) that made the strongest argument against Rational Software being the keeper of the industry standard. Any industry standard created, maintained and controlled by Rational would give Rational products an unthinkable competitive advantage. A number of pointed similarities to Microsoft were drawn.

4.1.1 The OMG takes the lead

Into this fray stepped the Object Management Group (OMG), an industry consortium that had been responsible for maintaining the CORBA standard and ensuring that CORBA did not fall prey to the same scenario now being envisioned by tool vendors for UML. It also helped that many of the vendors who raised objections to a Rational controlled standard were also members of OMG.

Two of the leading non-Rational employed theorists, Mary Loomis and Jim Odell, took on the responsibility of heading up a working group at OMG to look at drafting a standard using the original Rational Unified Method and soliciting input from others in the industry.

The result was the Unified Modeling Language 1.0 which was released as a proposed standard in January 1997. At this point, the name of the standard was formally changed to Unified Modeling Language.

One of the reasons why the change was made from Unified Method to Unified Modeling Language was the Unified Method had a much grander vision than just creating a notation. According to the UML specification, the goals of the Unified Method were:

1. Enable the modeling of systems (and not just software) using object-oriented concepts.
2. Establish an explicit coupling to conceptual as well as executable artifacts.

3. Address the issues of scale inherent in complex, mission-critical systems.
4. Create a modeling language usable by both humans and machines.

Devising a notation for use in object-oriented analysis and design is not unlike designing a programming language. There are trade-offs that have to be made between simplicity and completeness, complexity and usefulness and other issues.

As it turned out, defining the modeling language was the easy part (well at least one that could be used by humans – they never did the machine part), getting it accepted as a standard was much more of a challenge. The rest of the goals of the Unified Method had to wait a few more years until the Unified Process for Software Development was released by the three amigos (although primarily Ivar Jacobson's creation).

After some revisions, the OMG adopted the UML 1 1 as an official OMG standard, the first standard for diagramming in the OO world. After several minor revisions, the standard stabilized for a while at version 1.4. UML version 2.0 is a major revision and the one we are using in this course.

UML is currently at version 2.5 however these materials have not been updated to include the new revisions.

4.2 UML Specifications and Structures

UML is described and defined in a set of specifications. These are available for free at www.uml.org, the website maintained by the OMG to support the UML standard. While you probably will never have to actually read the UML specification, it is helpful to understand what they are.

In fact, it is strongly recommended that unless you have a specific reason for reading the specification, you avoid it since it is quite formal, technical and laid out more as a set of detailed rules rather than a document for self-learning.

4.2.1 Design Goals of the UML

As mentioned earlier, the UML is not intended to be a universal modeling language but it is intended to be a generic meta-modeling language. In other words, it is a set of tools for creating modeling languages as well as being used to model directly. UML is used in many different ways and many different domains to specify different kinds of concepts.

For example, with the appropriate application of the core UML concepts and structures, UML can be used as

1, A visual representation of programming language implementations (Java, C++, Smalltalk, CORBA IDL etc.)
2. Directly executable models (e.g. XUML)
3. Language-independent software specification
4. High level architecture and framework description.
5. Process (re)-engineering
6. Website structures
7. Workflow specification
8. Business modeling

For each application, the same set of concepts, symbols and diagrams is used, but their interpretation of those the meaning of UML symbols and concepts is different. A main design goal of UML 2.0 was to enhance this extensibility mechanism of UML to facilitate it role as a meta-modeling language.

4.2.2 The OMG Architecture

For UML 2.0, the OMG has described UML in terms of a four-layered modeling architecture:

1. **M0:** This is the level of no abstraction – the actual objects out there in the real world.
2. **M1:** User abstraction models – this layer corresponds to the actual UML diagrams that you draw or your domain-specific abstractions.
3. **M2:** UML metamodel – this is where the UML is defined and is where we define our extensions to UML. This is a layer where we model how we create models and define how to use the basic diagrams. For example, how to use them to make models at the M1 level.
4. **M3:** Meta object facility – this is a model for creating modeling languages. It is used to create meta-models like UML. This level is used to describe UML in terms of what diagrams are allowed and how they relate to each other and the other artifacts of UML. Clearly, this level is “above” UML.

Our course focuses on using the M2 concepts to make M1 models. We will not look at the meta-models at all in this course.

4.2.3 Structure of the UML Specification

Starting with UML 2.0, the specification is split into separate sub specifications. These are:

1. **UML Infrastructure Specification.** This specification is concerned with the basic structure of UML and is of no interest to those of us who just want to draw diagrams. This defines the core elements of UML in a way that is almost impossible for most people to understand on a casual reading.
2. **UML Superstructure Specification.** This specification defines the basic diagrams and their syntax in excruciating detail. This is called the “user level” specification.
3. **UML Object Constraint Language.** The Object Constraint Language (OCL) is a notational used with UML that allows modelers to create constraints and queries over object models. These constraints are used to create a highly specific set of rules that govern the aspects of an individual object. We are not going to be concerned with OCLL in this course.
4. **XML Metadata Interchange (XMI)**. This is a specification that defines a common XML format for UML diagrams that allows them to be portable between different modeling tools.

4.3 Customizing and Extending UML

The main elements of UML are:

The UML Metamodel: This is essentially the UML specification which describes the syntax and semantics that make up the UML. We already touched on this earlier.

Packages: The UML allows for grouping of model elements into packages. A package can be thought of as a grouping of elements according to some logical criterion which can then, in turn, be treated as a single unit. For example, in design terms, a subsystem is a package. There is also a package diagram specification that shows how to document these groupings.

This may seem a bit confusing but packages can occur both at the M1 Level, which is where we will be using them in this course, and at the M2 level, where they are used to define new kinds of models or diagrams.

Keywords and Stereotypes: Keywords are UML labels that are used to distinguish elements that use the same symbols. Stereotypes are user-defined labels used to distinguish different types of element that use the same symbol. Stereotypes are denoted either by a user-defined graphic symbol or a label enclosed in guillemets, e.g. «boundary». Stereotypes are used to allow UML to be extended by users with specific modeling requirements.

Expressions, Constraints and Comments: Expressions are computable statements that can be evaluated. A constraint is a predicate expression that is applied to some model element. It is denoted by strings representing executable expressions in some language, enclosed in braces. e.g. {record.number==0 || file.isClosed()}. Comments are constraints described in natural language.

Notes: Notes are little sticky notes – boxes with turned down corners – which contains meta-information about the diagrams but are not part of the model themselves.

UML Profiles are a generic extension mechanism for working with UML models for different platforms (such as J2EE or .NET) or domains (such as real-time or business process modeling).

For example, the UML testing profile is described in the UML Testing Profile specification as:

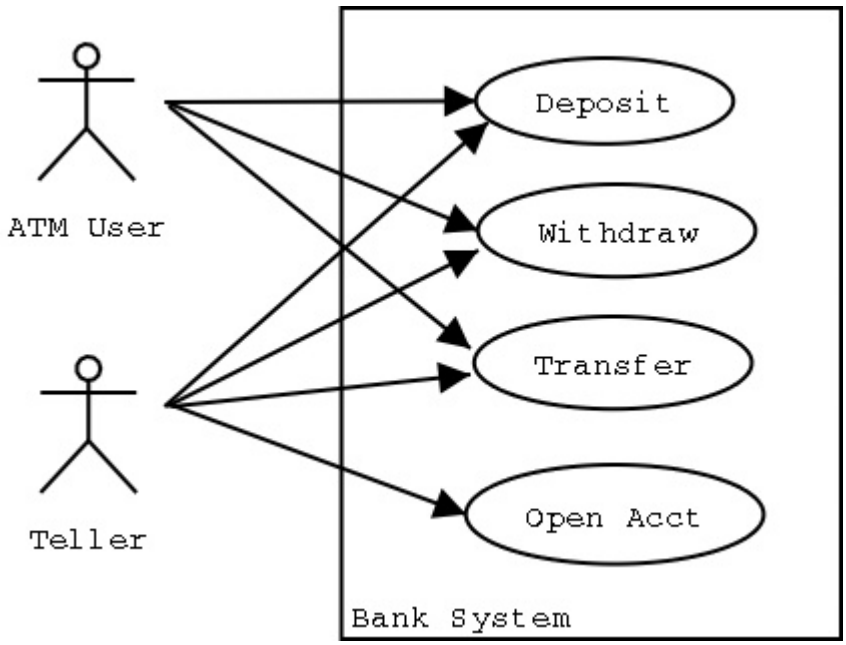
> *The UML Testing Profile defines a language for designing, visualizing, specifying, analyzing, constructing and documenting the artifacts of test systems. It is a test modeling language that can be used with all major object and component technologies and applied to testing systems in various application domains. The UML Testing Profile can be used stand alone for the handling of test artifacts or in an integrated manner with UML for a handling of system and test artifacts together.*

5 The Basic Diagrams

UML defines the following diagrams which are organized around architectural views (see, that stuff about architecture is starting to pay off).

5.1 The User Model View

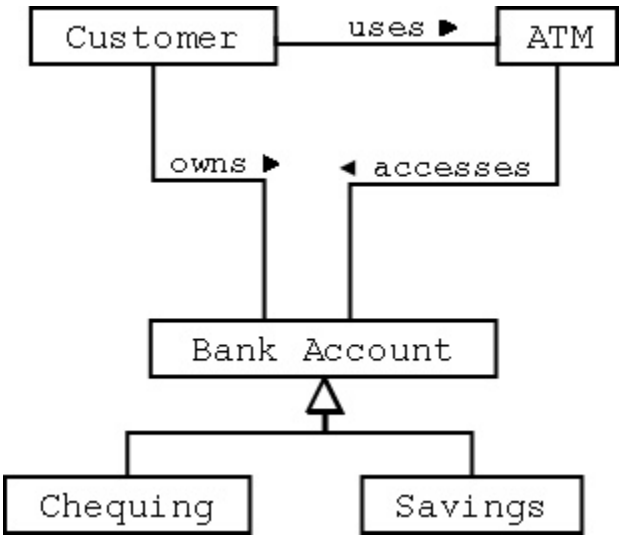
Use case diagrams depict the functionality of a system. This is that view that ties all of the other views together.



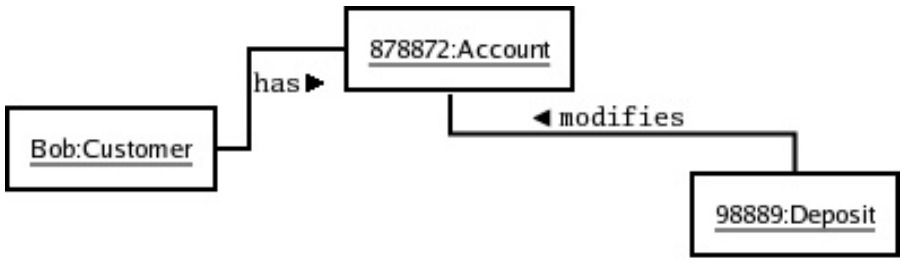
5.2 The Structural Model View

Class diagrams depict the static structure of a system showing the logic relationships between the types of things in a system that are time independent. This is the same sort of thing that an Entity Relationship diagram shows.

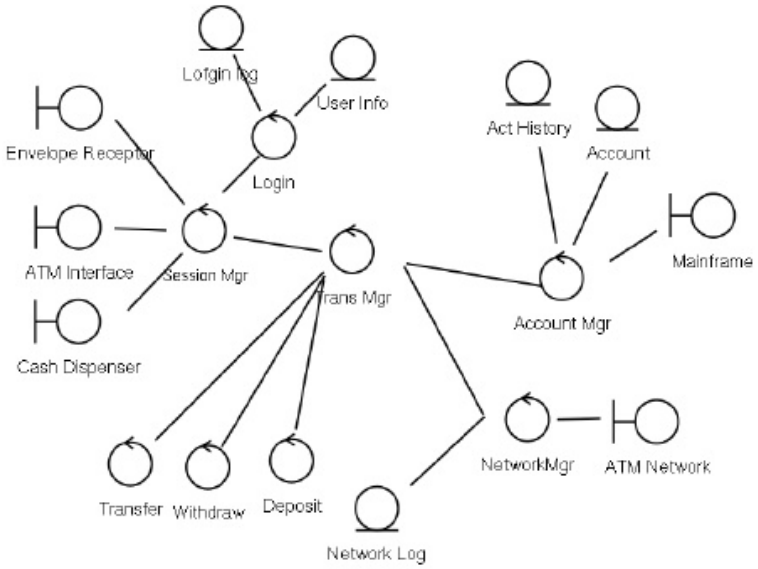
Object diagrams depict the static structure of a system at a particular time. An object diagram is like a snapshot of the objects instantiated from the classes at a particular point in time. For example, we can say that “husbands are married to wives,” which is a class relationship, and the corresponding object relationship would be “Dagwood is married to Blondie.



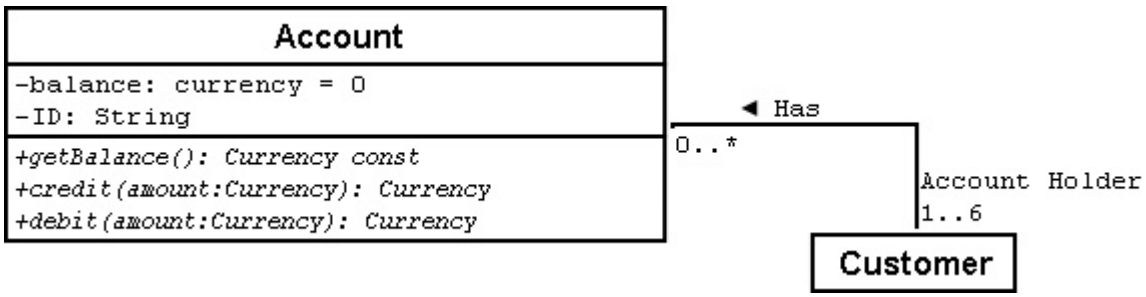
The diagram above is a sketchy class diagram, and below is an object diagram derived from the class diagram.



We also have what we call an analysis class diagram:



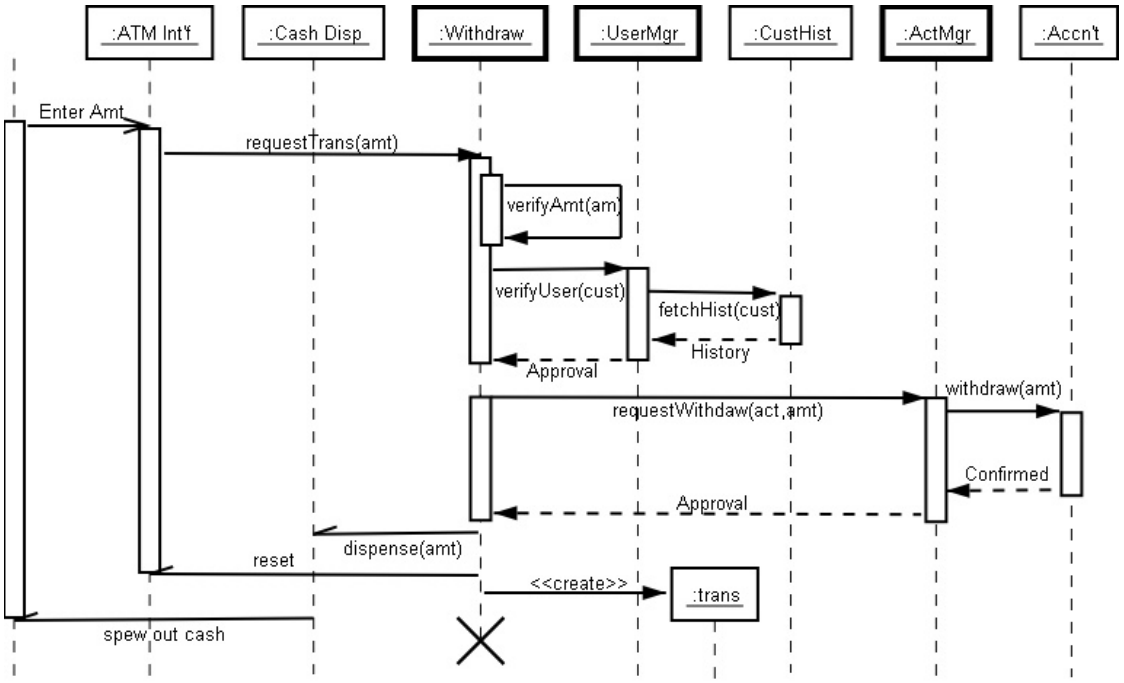
And a design class diagram.



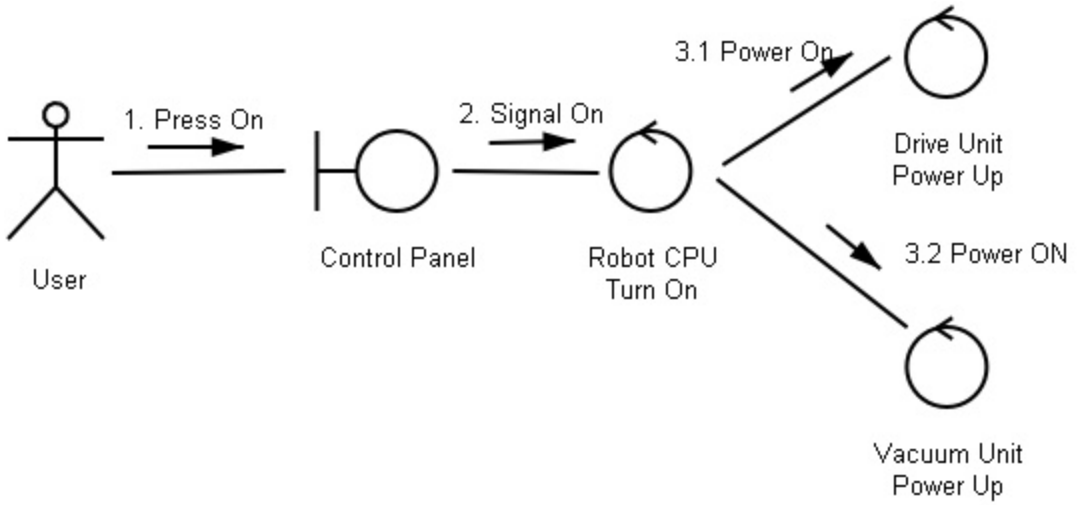
5.3 The Behavioural Model View

There are several diagrams that are used to show how objects interact with each other.

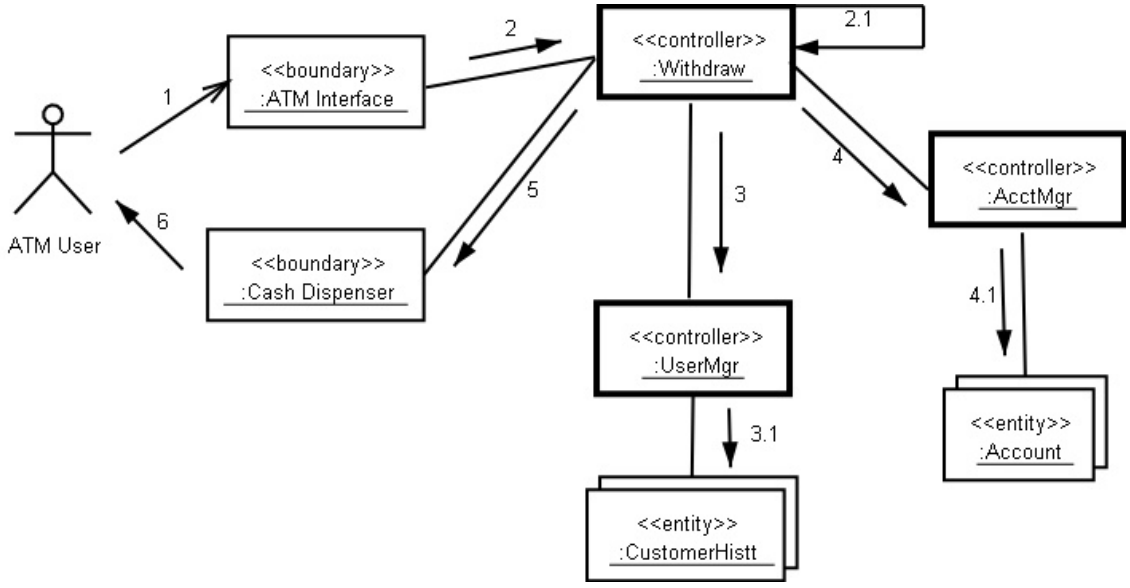
Sequence diagrams depict an interaction among objects in a system organized in time sequence.



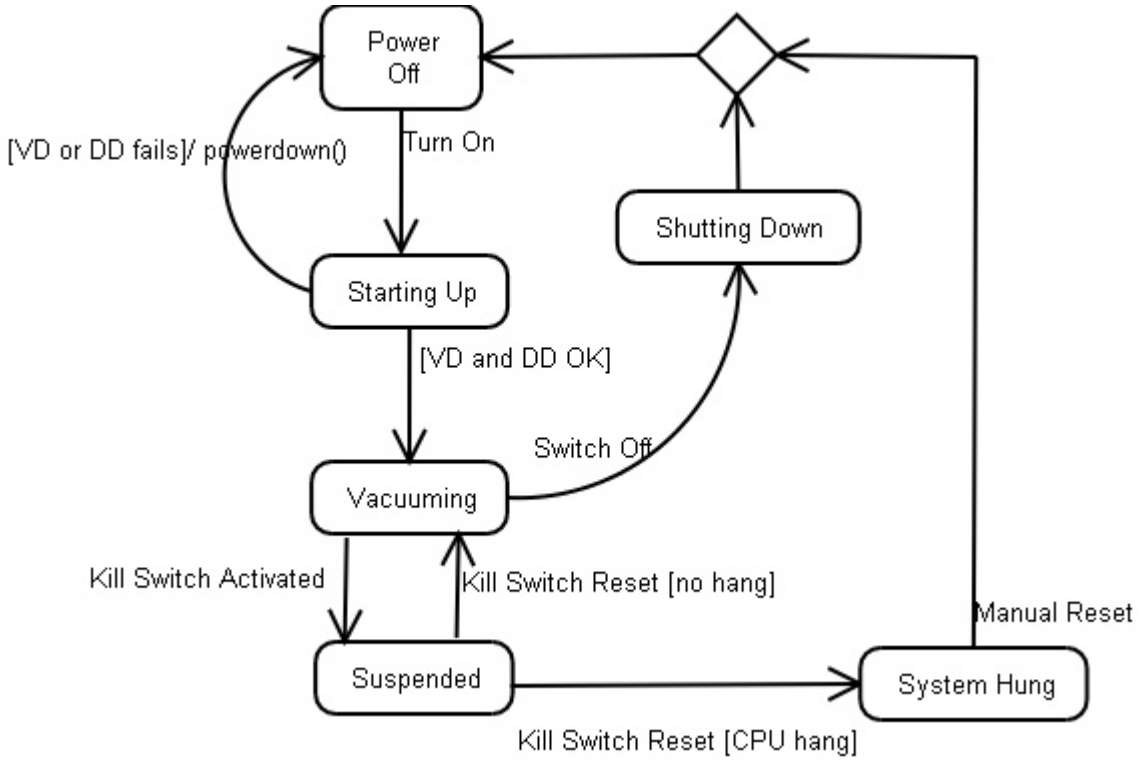
While communications diagrams are used to depict the flow of information through a system.



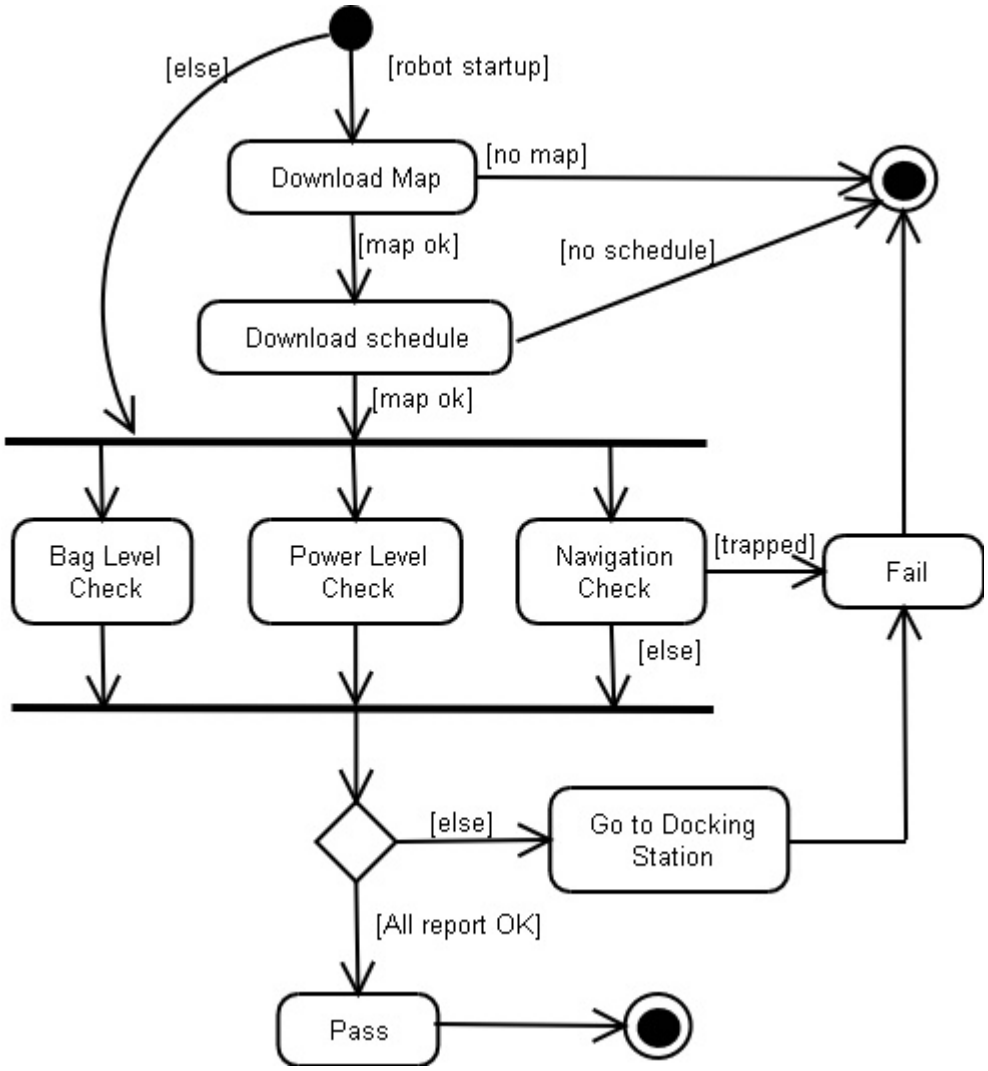
Or in a slightly different notation.



State diagrams, or more accurately statecharts, depict the status conditions and responses of elements of a system.

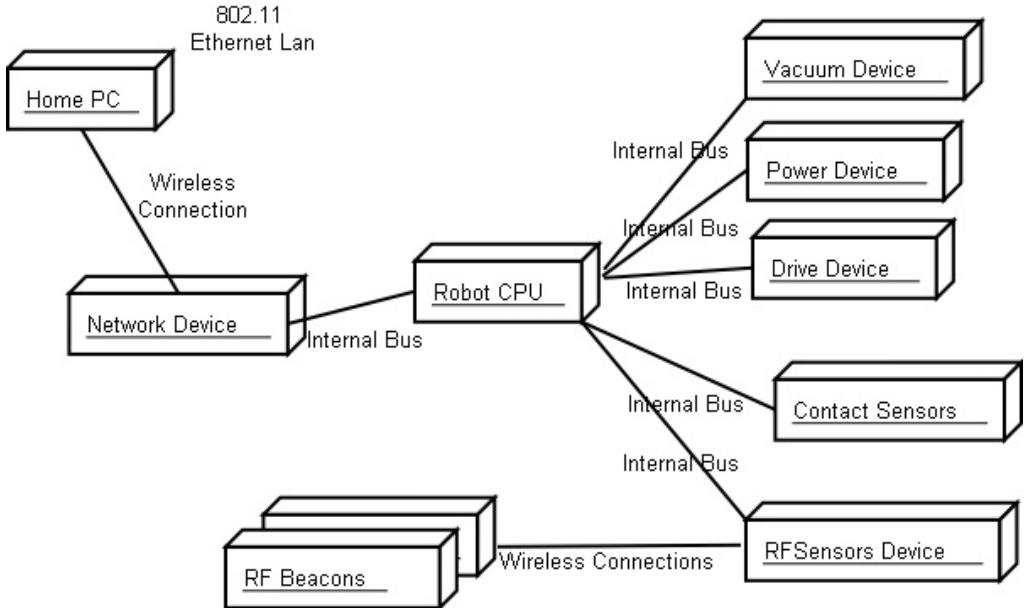


Activity diagrams depict the activities of elements of a system. These are like flowcharts and data flow diagrams rolled into one and then enhanced.



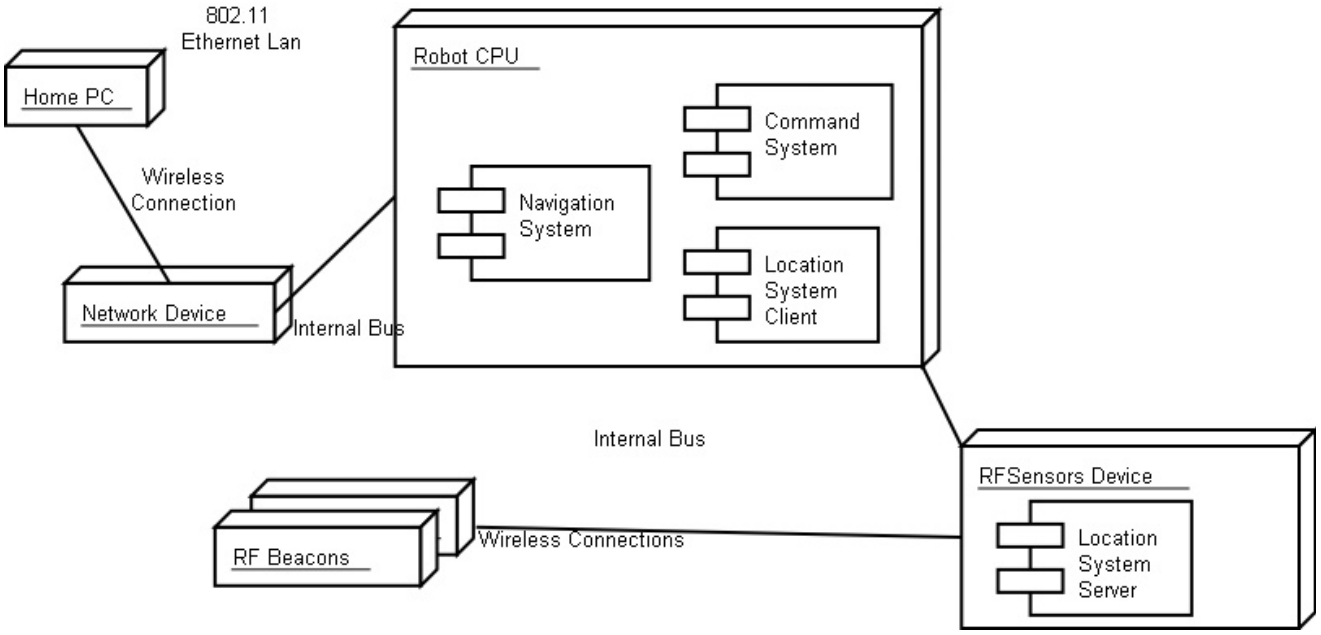
5.4 The Environment Model View

Deployment diagrams depict the configuration of environment elements and the mapping of elements realizing a system onto them.



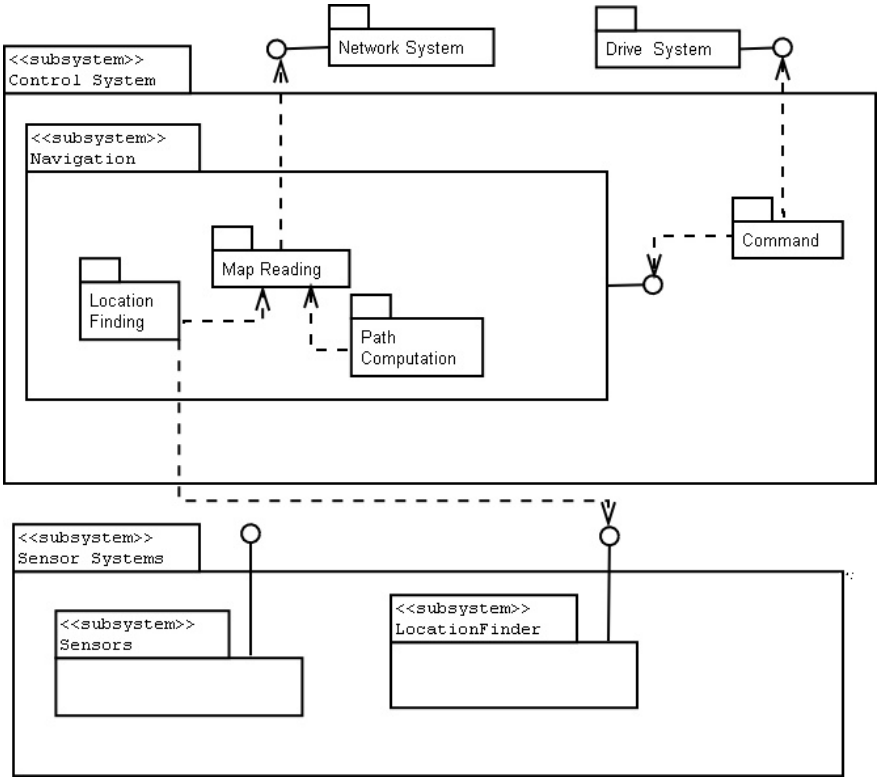
5.5 The Implementation Model View

Component diagrams depict the organization of elements realizing a system

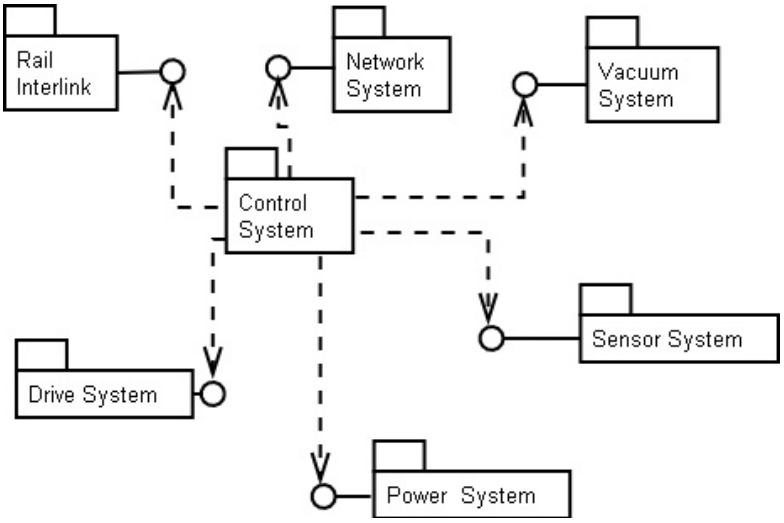


5.6 Other Diagrams

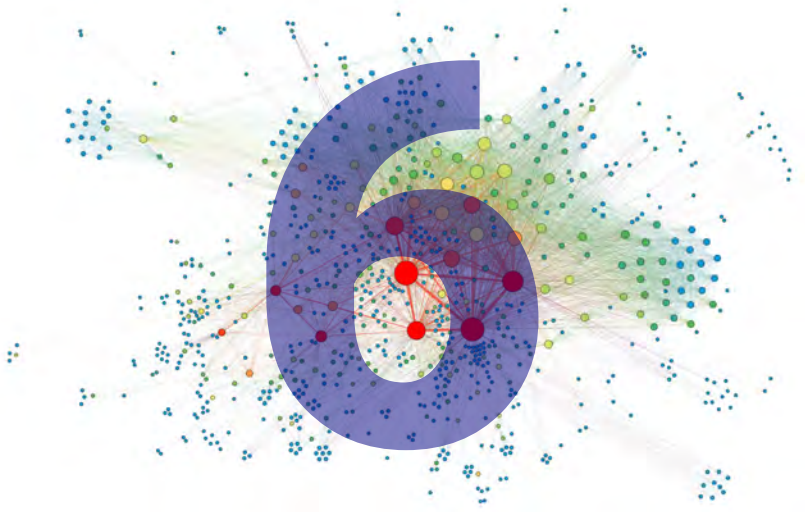
While not associated with any particular view, the package diagram is used more to give us a zoom in and out facility for our diagrams. A package is like a box that we can stuff a whole bunch of detail int so that we can treat it as a single unit.



Or depict it at various levels of detail.



UML 2.0 also introduces several new kinds of diagram beyond the ones mentioned so far which are also defined in UML 1.5. In a sense, these new diagrams are not really new, but are extensions of the notation of the existing diagrams to accommodate more specific application domains. For example, a timing diagram is a sequence diagram with additional timing information added with a specialized notation.



Object-Oriented Analysis and Design: Visual Modeling with UML

High Quality Requirements

Over the years, we have fooled ourselves into believing that users know what the requirements are and that all we have to do is interview them.
Ivar Jacobson

[requirements gathering] is frustrating, full of complex interpersonal relationships, indefinite and difficult. In a word, it is fascinating. Once you're hooked, the old easy pleasures of building a system are never again enough to satisfy you.
Tom DeMarco

No, you can't always get what you want,
But if you try sometime, You just might find you get what you need.*
Mick Jagger and Keith Richards

exgnosis

1 Doing Requirements Right

As we saw in the module on software development processes, the requirements phase of a project is part of the process, not part of the paradigm. However, depending on the paradigm being used, we may engage in different activities within each phase of a project. During the requirements phase, we discover what the stakeholders want and need the system under development to do for them. Independent of any specific paradigm, we still have to ask people what their requirements are in pretty much the same way. People's needs and requirements don't change when we decide to change paradigms.

The differences in paradigms show up in the types of models we build. For example, the emphasis on iconicity and domain modeling is something that object-oriented emphasizes a lot more than other approaches. The actual activities, like interviewing users, are common engineering practices that we find in all paradigms, but the way in which we document, analyze and use the information gathered, and often the type of information we want, is paradigm specific.

In this module, we will be focused on the requirements activities that are associated with the production process; then focus on the specific types of models that are paradigm dependent in later modules. These process activities are also found in other sorts of production processes that have nothing to do with software, which reinforces the fact that these activities are related to the underlying engineering and not the paradigm.

We will first look at the requirements process. As people and organizations get better at getting requirements right, they tend to develop very similar processes for doing requirements. The first step to doing requirements right is to understand the best practices that have been proven to work in the trenches.

Next, we will explore the deliverables that are common to all requirements processes and are the outputs of the stages of the requirements process. These are:

1. A System Definition (also called vision document or project charter) that describes:
    a) What the client need is that the system must meet is and a suggested solution or range of solutions that meet the need.
    b) Clearly and unambiguously what the scope of the project is.

2. A stakeholder analysis that describes:
    a) Who the stakeholders are for this project including why they are stakeholders: specifically what their vested interests in the project are and the sanctions that they can bring to bear on the project.
    b) How to contact each stakeholder.

3. A unified statement of requirements that contains:
    a) A complete record of all requirement requests and their dispositions.
    b) A prioritized baseline of proposed features.
    c) A set of identified technical constraints.
    d) Any business processes or rules that impact the requirements expressed in a process model.

In addition, there are two other deliverables that are characteristic of the object-oriented approach, primarily to support iconicity.

4. The domain object model that describes each class of stakeholder's view of the application domain.
5. The use case model that provides a set of scenarios that the user is familiar with from interacting with the enterprise and that will be replicated in their interactions with the system.

1.2 The Requirements Process

One of the most problematic areas for IT development teams are requirements issues. According to most of the books, it sounds pretty easy – you just go out and gather the requirements. It brings to mind the image of requirements just lying around “out there” waiting to be plucked like over-ripe fruit.

In reality, getting requirements is more like hunting a wild boar – they are elusive and usually have to be torn from the clutches of the stakeholders – and their quality often makes you wonder if it was worth the hunt and battle at all.

When we look at organizations that “do requirements” well, we find several factors that they all consistently have in common.

1. They have a plan and a process for getting requirements.
2. The activities at each step of the process are guided by what was learned in earlier steps.
3. The process is managed. Requirements are gotten whether the stakeholders want to provide them or not.
4. The stakeholders are managed to help them change their requirements so that they are realistic and lead to solutions that can be implemented within the scope of time, budget and available resources.

1.3 The Requirements Process Phases

Just like the development process, there are progressive stages to the requirements process.

1. Determining why we are doing the project. Answering this allows us to define and delimit the scope of the project.
2. Determining who has requirements.
3. Eliciting the requirements.
4. Analyzing the requirements to resolve inconsistencies, missing items and contradictions as well as establishing priorities among requirements.
5. Identifying stakeholders we have to negotiate with to “help” them modify their requirements and “manage” their requirements.
6. Identifying what the solutions are that would satisfy the requirements.
7. Producing a specification to describe the solution to be built. This is done in conjunction with the developers and other team members to produce cost and time estimates.
8. Identifying which stakeholders have to modify their requirements to keep the project within time and budget constraints and then help them “manage” their requirements.

Sometimes during the last step, we have to defer some of the requirements into the “next release” or have them incorporated as enhancement requests once the system is into production. The requirements process is iterative even if the software development process itself is not. That's because requirements come from people and people never get it right the first time through.

The last several stages of the requirements process start to make the transition into Analysis, but recall from the USDP module, the worklows do overlap and interact with each other.

2 System Definition and Project Scoping

In many cases, a need is known in terms of the effects it is having on the stakeholders but the causes of the problem may not be obvious or known. A wrong assumption about what the underlying problem actually is may result in us developing a totally ineffective solution.

For example, a client might state that they are having a problem with throughput because the processor they are using is not powerful enough. A more powerful processor is brought online as a solution. But what if the poor throughput (a result) is because of a poor user interface design (the real cause) and not the power of the processor (misidentified cause)?

In many ways, those who build systems are like sales people – both groups are trying to meet their clients' needs, one group by building something, one group by selling something. Both groups have the same problem "Is what I am building (or selling) going to meet the real need of my client?"

No client will ask for a new system to be built if they don't perceive a need for it: the challenge we have is to determine whether or not that perceived need is the real need. The System Defintion is a tool we use to explore, with the stakeholders, what the real business need is for the work we are going to do, and what they expect will the result of the development process.

A System Definition should not be just a list of features or requirements. It will involve describing some key features of the system, but we want to avoid turning the description into a "shopping list". Its strength is in providing an opportunity to address the core question of "What is the problem we are trying to solve?" in a collaborative way with the stakeholders.

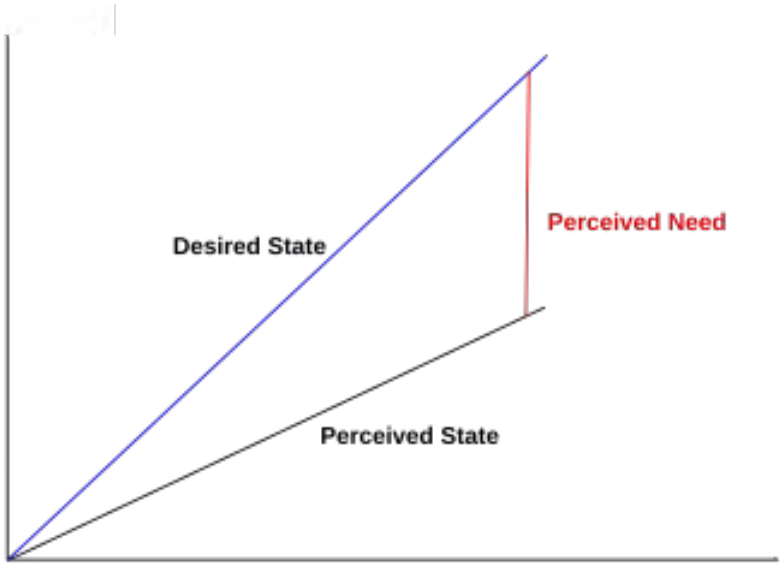
The System Definition should also identify any assumptions that we have made in proposing the solution since this will be useful later in our risk assessment. But the main role of the System Definition is to keep the project focused.

2.1 Identifying the Need the System Meets

Projects are initiated by a request from a stakeholder for a solution to a perceived need. The perceived need may or may not exist in reality, or the stakeholder may not have identified the need correctly. This means that we may have to undertake a considerable amount of investigative work to get to the reason we are doing the project.

In terms of needs, a client is going to be in one of the following categories:

1. They perceive that a need exists, usually because of the effects or results they see, but they admit that they don't know the real cause or underlying need.
2. They perceive a need where one does not really exist.
3. They perceive their real need accurately.
4. They perceive a need but the real need is something different than what they perceive.



Notice that unless a need is perceived, correctly or incorrectly, there will be no project.

In many organizations, there is often a formalized document that has to be signed off before a project can begin. Sometimes this is called a charter or statement of work or system proposal. In most cases, this document includes something that resembles the system definition, and sometimes includes more, like a full business case or proposed budget.

2.2 The System Definition

The system definition can be thought of as defining the target or ultimate goal of the project. It provides the answer to two critical questions:

1. Why does the client want a new system?
2. What does the client expect the new system to do for them?

Answering these questions is not as simple as it seems. A project can grind to a halt early on because the various groups of stakeholders cannot agree on why a system is being developed or what it is supposed to do.

We start with a simple, brief statement of the need and what the new system will do to meet the need. This can be a very difficult step because of the different perceptions among the stakeholders of what the real need is and what the new system “should” do.

The System Definition should also identify any assumptions that we have made in proposing the solution since this will be useful later in our risk assessment. But the main role of the System Definition is to keep the project focused.

2.2.1 Example: Friendly Bank System Definition

Background: The Friendly Bank of Minot ND is a nice country bank that has never used ATMS or web banking. However, its customer base of local farmers is starting to retire and move away. These older folk did not use technology and loved coming to the bank because it served as an opportunity to interact with their friends. However, their farms are being taken over by a younger generation of farmers who are technologically literate and expect their bank to offer the same services as an urban bank. A project is initiated to give the Friendly Bank system the capabilities to support these new technologies.

System Definition

The Problem: *The banking industry has undergone a tremendous transformation in the way it delivers services to the marketplace. Customers now expect to be able to access their accounts anytime from ATMs, by phone or on-line. Additionally, customers have come to expect the convenience of making purchases directly from their bank accounts using their debit cards. Our business customers are also under increasing pressure to accept debit cards as a payment for purchases.*

Currently, our existing system, while still functioning superbly, cannot deliver these new services. However, the marketplace has spoken and if we do not provide these services, we can expect our market share and customer base to erode and, in short order, collapse entirely.

Keeping track of the scope of the project can be difficult, and many projects suffer from "scope creep," which is the phenomenon of more and more features being added to a project as it progresses until the project collapses under its own complexity.

Our Solution: *We propose creating a middleware layer that will be responsible for providing these new services to our customers while still keeping our existing system intact and accessible.*

The new middleware will be responsible for processing ATM requests, debit card purchases, connecting to other ATM networks, providing web and phone access, as well as providing a new integrated desktop for our staff within the branch. We will be working closely with our customers to identify other services that they may like to see available (online loan applications for example) that we would be able to host on the new system. Once the middleware is in place, we will have the opportunity to replace our existing mainframe with a more modern system with greater capacity.

Key Requirements: *The new system must be able to deliver these new services with performance characteristics (response time etc.) that are in keeping with our reputation for quality. The new system must be able to integrate with our current internal processes and support our existing standards.*

Assumptions: *There will be a demand for these new services over a sufficient period of time that we will achieve a return on our investment. The technology is sufficiently mature that the risks associated with it are known and can be managed effectively.*

There is a clear problem definition expressed in what sales people call a feature benefit analysis, which is just another version of cause and effect. This is what has happened (cause/feature) and here is what it means to us (effect/benefit). We note that the industry has changed and FBOM has not, which means that they are out of step with the market (this is the cause or problem). If they don't do anything about it, then the bank will lose customers (this is the result or effect).

The key requirements here are fairly general and very important to the project from a business perspective. Both of these main key requirements have significant implications for the system architecture.

Another example of a key requirement might be the budget for the project. Key requirements should always be strategic in scope and not tied to specific system features.

2.3 The In/Out List and Scope Creep

Keeping track of the scope of the project can be difficult, and many projects suffer from "scope creep," which is the phenomenon of more and more features being added to a project as it progresses until the project collapses under its own complexity.

Once we have a System Definition, we can create an In/Out list to define scope. In the example, the list is a table with three columns, the left column containing any topic, although usually a requirement or request, the next two columns saying "in" or "out." Other information may also be kept on each item like who made the decision, justifications for the decision and so on.

Request	In	Out
On-line foreign exchange		X
Email notifications of overdrafts	X	
Employee time sheet automation		X
Customer mailing list management tool		X
Banking phone app	X	
Email transfer between customers	X	

Any topic that is a scope issue is added to the list and then a consensus is reached as to whether it goes into the in or the out column. If there is no consensus, then at least there is a point at which we have documented this as a scope issue.

While the list is most useful with requirement requests, we can also use it to scope out issues like who the project should report to or any other topic that makes you wonder "Does this really concern us?"

3 Stakeholder Analysis

With the emphasis on the system users and use cases, the non-user stakeholders and their requirements are often overlooked. System requirements are not just user requirements because there exist parties, either people or organizations, who are not necessarily users, but have a vested interest in how a system operates.

Their requirements can drastically alter the design of a system, and overlooking their requirements can result in an unusable system. We call these parties stakeholders.

A stakeholder is anyone:

1. who could be materially affected by the implementation of a new system or application or who has a vested interest in the project; AND
2. who has the ability, legal, political, economic or otherwise, to force the developers to protect those interests to their satisfaction.

Stakeholders have input into the project because they have the power to affect the project. This definition of a stakeholder is quite pragmatic. There is no rule that says that you cannot take into account the requirements of non-stakeholders – we often do this for business or political reasons or to create goodwill with groups or individuals.

However, ignore stakeholder requirements and you introduce a significant risk element to the project. We cannot totally identify the stakeholders until we have a system definition to clarify what the scope of the project is. If we wanted to, we could create an In/Out list of all potential stakeholders and then triage the list using the Definition to eliminate non-stakeholders.

This sort of hunting for stakeholder is called stakeholder analysis.

3.1 How to Find Stakeholders

Some questions to ask to help identify stakeholders

1. Who are the users of the system?
2. Who is the customer (who is paying for it)?
3. Who will maintain and support the system?
4. Whose processes will be automated or affected?
5. Whose data will be accessed or modified?
6. Who regulates the system’s activities?
7. Whose standards does the system have to be in compliance with?
8. Who needs to sign off on the system and its activities?
9. Who can say NO and make it stick.

Stakeholders do not have to apply legal sanctions to be stakeholders. For example, users as stakeholders can sanction a system by not using it and system support staff can sanction a system by not supporting it properly or by not providing the resources required for its operation.

3.2 Active and Passive Stakeholders

In looking at stakeholder requirements, we typically use two different strategies depending on the type of stakeholder we need to consider.

Active stakeholders derive a direct benefit from the system or have a direct responsibility to the system. Active stakeholders are, in a fundamental sense, aware or know about the system under development. This means that active stakeholders can articulate their requirements for the specific system under development and may have unique requirements for each system for which they have a vested interest. Normally we have to find and ask active stakeholders what their requirements are.

Passive stakeholders generally do not know about the system and do not derive any direct benefit from it but tend to become aware of the system when their requirements are not met. Typical passive stakeholders are regulatory agencies who only get involved when the system violates their requirements. Passive stakeholders are more concerned with regulating a class of activity, like collecting sales tax or ensuring conformance to security or other standards, that the system under development may engage in. Generally, passive stakeholders publish their requirements in some manner and it is up to us to find out what they are.

This distinction between active and passive is only for convenience because it suggests a strategy for us to follow in identifying their requirements

3.3 Example: Friendly Bank Project Stakeholders

A preliminary review of stakeholders for a bank might produce the following list:

1. **Users:** Includes internal users (bank staff), bank customers who use the system, non-customers who interact with the system (holders of other banks' cards who use bank ATMs), merchants who use the FBOM bank card system.
2. **VP Finance:** If this project does not work and come in on time and on budget, he is back to working the midnight shift at the drive-through teller window.
3. **Shareholders:** This project will have a significant impact on the bank's economic health over the short and long term.
4. **Bank Networks:** The FBOM system must interact with other bank networks according to the protocol that the network specifies. The FBOM system has to maintain network transaction logs as specified in the network user agreement.

5. **Accounting and Auditors:** The auditors and accountants need to be able to access information about the business activities of the bank.
6. **Federal Agencies:** Various agencies the bank interacts with, like the Federal Reserve, Treasury, IRS and others.
7. **State Agencies:** State banking commission regulations that have to be followed by banks.
8. **Police:** Want to be able to ensure the safety of ATM users at ATM locations.

Note that the police in point (8) could be a project killer unless managed properly. The police are an example of a passive stakeholder. They don't really care about the system nor do they derive a benefit from it, but if it causes a sudden increase in crime because people are being mugged at ATMs, their requirements are not being met. It's not FBOM bank ATMs specifically that they have a vested interest in, it is ALL ATMs.

An interesting real-life example about ATMs is that in the US, all ATM keypads have Braille encoding. Even the ones at drive-through ATMs. How might this be a passive stakeholder issue?

Another passive stakeholder issue. Until the late 1990s, web banking could not be done because to provide adequate encryption for customer security would have violated a stakeholder interest. Under US law, encryption technology was considered munitions and banned from export – you could have gone to jail as an arms dealer for building a web-banking site.

3.4 Types of Stakeholder Requirements

Stakeholders often have requirements that constrain the use of the system in some manner to preserve their interests. These constraints often take the form of:

1. **Data capture requirements**: The system may be required to maintain audit trails or transaction records of its activities. For example, an ATM may be required by bank security stakeholders to record every keystroke a user makes at the keypad to track whether or not they know the PIN number associated with a card.
2. **Access requirements**: The system may be required to limit or control access to certain features or functionality, or the system may be required to conform to certain access and interaction standards.
3. **Functionality constraints:** The system may be required to control or manage, usually via a set of business rules, the functionality of the system. This may take the form of requiring the system to follow certain business processes or workflows.
4. **Standards Compliance:** The system may be required to perform to certain performance standards, such as encryption levels or response time requirements.

5. **Contractual and Legal Compliance:** The system may be required to implement certain features in order to be compliant with legal standards, like privacy regulations, or to conform to contractual obligations, like offering opt-out provisions for certain users.

Stakeholder requirements are often expressed in terms of processes to be adhered to or in terms of features (“The system shall....”). However, in general there has been so much emphasis on use cases that the need to have a systematic way to document and manage these stakeholder requirements has been overlooked

4 Requirements Gathering and Elicitation

The process of actually getting the requirements from stakeholders is beyond the scope of this course so we will assume that this is being done effectively and correctly. Going out and gathering the requirements is a particularly difficult task. In fact, Ivar Jacobsen notes:

> *Requirements capture is the process of finding out, usually under difficult circumstances, what is to be built. In fact, it is so difficult that it is still not uncommon for project teams to start writing code (which is fairly easy) before they have firmed up just what the code is supposed to do (which is difficult).*

4.1 Problems with Requirements Capture

There are a number of reasons actually getting the requirements from stakeholders is difficult:

1. Different groups of users and stakeholders may have different or conflicting requirements for the same system.
2. Stakeholders may not know what their requirements are.
3. Stakeholders may think they know their requirements, but really don't.
4. Stakeholders may not trust us with their requirements.
5. Stakeholders may lie about their requirements.
6. Stakeholders may tell us what they think their requirements should be and not what they really are.
7. Stakeholders may not know how to describe their requirements.
8. Requirements may change at any time.

Jacques Vallee, the French computer scientist who became a UFO researcher is attributed with observing that in the empirical sciences having 99% of the data means that you are 99% of the way to the solution; but when dealing with people, if you have 99% of the data, it means that somewhere, someone is holding on to the critical 1% without which the other 99% is useless.

Depending on the complexity of the situation, eliciting requirements can require highly trained individuals with exceptional investigative techniques and interpersonal skills. In many ways, the same skills used to elicit requirements are the same skills that therapists, interrogators and highly trained sales people use (not the used car salesman type) since their success is dependent on eliciting customer requirements.

Two excellent sales training resources which are applicable to requirements elicitation are the books by Stephen Heiman *Strategic Selling* and *Conceptual Selling* and the books by Neil Rackham *SPIN Selling* and *Major Account Sales Strategy*.

4.2 Problems with New Technology

When being introduced to new technology, stakeholders generally do not understand how they could use the new technology and which aspects of their work could be turned over to the system. They need to experience the technology before they can start to think meaningfully about requirements.

Market research and predictions of how much users would use new technologies, and how they would use them, is always suspect because people being asked have no experience to base their answers on. Estimates of how much a technology will be used, and what it will eventually be used for, is very difficult if no one has any experience actually using the technology. All the following were accurate statements at the time based on feedback from their customers who might makes use of that technology.

> *"I think there is a world market for maybe five computers." Thomas Watson, chairman of IBM, 1943*

> *"I have traveled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won't last out the year." The editor in charge of business books for Prentice Hall, 1957*

> *"But what ... is it good for?" Engineer at the Advanced Computing Systems Division of IBM, 1968, commenting on the microchip.*

4.3 Requirements Gathering Activities

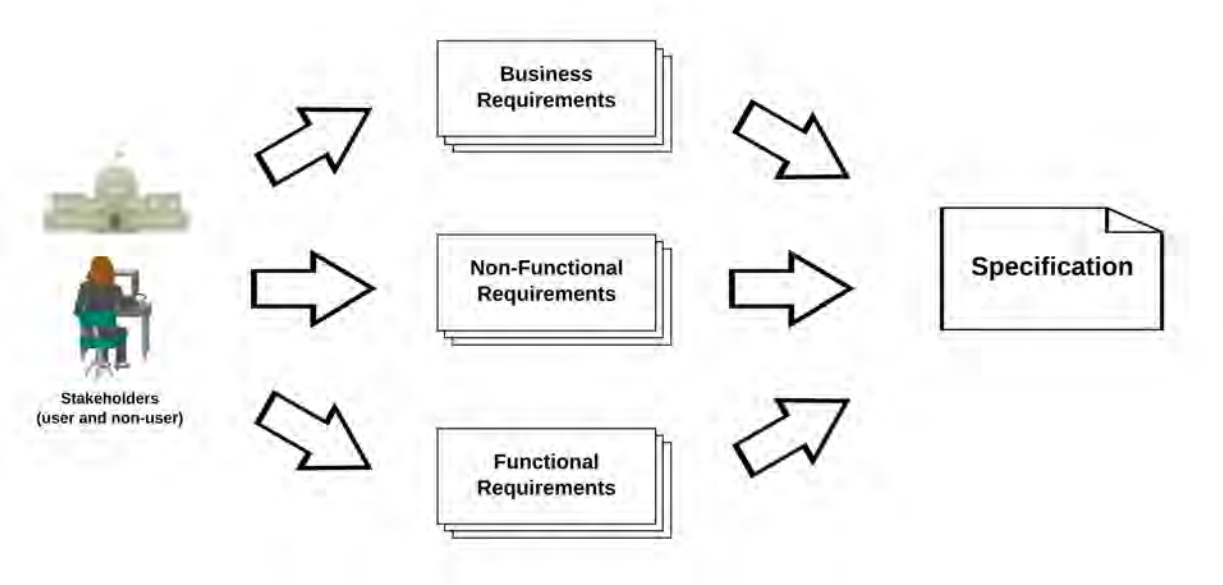
Requirements gathering involves a number of activities:

1. **Eliciting requirements:** Getting the requirements from the stakeholder.
2. Validating requirements: Reality checking to see if the stakeholder has not made mistakes in communicating their requirements to us. “Are you sure this is what you want?”
3. **Organizing requirements:** Stakeholders usually don't organize their requirements for us, nor do they organize them with reference to other stakeholders. Experience shows that a requirements managing system is critical to keeping on top of the mass of information. Automated tools are a significant help.

4. **Documenting requirements:** Requirements need to be written down, but they can also be effective in other documentation formats. Diagrams, screen-shots, storyboards, videos of work flows – the secret is to use the documentation that is easiest to understand and simplest to maintain. This includes using visual models to clarify the requirements.
5. **Verifying requirements:** Another reality checking process but this time we ask "Am I sure I understood the stakeholder correctly?"
6. **Maintain and update requirements as necessary:** Requirements change, and stakeholders will often remember things they have forgotten, make corrections and clarifications over time.
7. **Managing stakeholder expectations about requirements:** Part of what we have to do is often keep the stakeholder expectations realistic about what can be done with the capabilities available to us. Very often this takes the form of "We can't do that, but let's find another way that might produce the same or a similar result for you." Managing expectations takes an unrealistic or impossible requirement and reworks it so that it becomes a reasonable and do-able requirement. It may often involve helping stakeholders drop spurious or unnecessary requirements.

5 Types of Requirements

We traditionally group requirements into several basic types. As we can see from the diagram, the specification depends on requirements of all types.



Functional requirements specify what a system “does" or as the name suggests, describes the functions or operations of the system. Because the system has to have the functionality necessary to satisfy the requirements, there is usually a fairly obvious mapping between the functional requirements and the features of the system. The set of functional requirements tends to be unique for each system since each system satisfies a unique problem in a unique business context.

Non-functional requirements describe “how” the system delivers its functionality. Non-functional requirements describe the general performance characteristics of the system. Unlike functional requirements, which tend to be system-specific, non-functional requirements tend to be similar across systems. Some people find the name a bit misleading, as if these were somehow dysfunctional requirements, so sometimes the terms "operational requirements" or "performance requirements" are used instead.

Business requirements are the stakeholder requirements in the form of the rules that the system has to follow. These generally come in two forms. The first are the business rules and processes that a system has to follow, often called high-level requirements. The other form are the constraints and rules that are imposed on a project because of infrastructure or architectural factors, sometimes called "technical system" requirements.

5.1 Business Requirements

Functional requirements are usually straight forward, often of the form “use clicks mouse or presses button or swipes card… and the system responds with result X”.

Generally, business requirements define the logical workflows to be automated, the decision-making logic and organizational rules that have to be followed. Business requirements provide a process context that constrains what the features that implement the functional requirement can and cannot do.

For example, consider an ATM. A simplified set of functional requirements might be that the user has to be able to withdraw money and check their balance at an ATM. However, those two functional requirements have to interact with the following (hypothetical) business requirements.

1. The withdrawal must follow the standard bank process: if requested funds are available, a hold is placed on the account for the amount and an authorization sent back to the ATM, once a notification has been received that the funds were dispensed, the money is actually removed from the account, or else the hold is removed after 15 minutes and the transaction rolled back. (Bank Business Procedure)
2. The ATM must validate that the card used is current, not reported lost or stolen and has valid customer accounts linked to it. (Fraud prevention office requirement)
3. All transactions at an ATM must request a transaction ID number from the transaction journal, and then be recorded into the journal if the transaction succeeds. (Bank accounting requirement)
4. Any failed transactions must be recorded in the incomplete journal in the specified format. (Bank auditor requirement)
5. All ATM activity is to be recorded in an event log, including a list of all key presses and the latency between key presses. (Fraud prevention office requirement)
6. All ATM traffic has to follow protocols specified by the ATM network service provider. (External stakeholder requirement)

And we could come up with many more. One of the important things to notice is that the average user interacting with the ATM usually has no real idea of what is going on during the processing of a transaction.

One other thing to notice is that business requirements and functional requirements are sort of fuzzy. They represent the ends of a continuum rather than discrete categories. For example, it may be a business requirement to ask a user to click on a button to verify they are over 18, but one could argue that the requirement to have a button to click on makes it a functional requirement. In this case the professional answer to whether this a business or functional requirement would be “Yes. Sort of. Most of the time.”

It should be noted that the stakeholder requirement types we looked at earlier (data capture, contractual or legal compliance, etc) often are business requirements.

5.2 Non-Functional Requirements

Non-functional requirements tend to focus of general properties of the system, or they can be thought of as specifying the characteristics of the delivery of the functionality. Unlike functional requirements which tend to be system specific, non-functional requirements tend to be similar across systems. This has allowed a number of standardized specification standards for non-functional requirements.

5.2.1 FURPS: A Standard Model

The FURPS+ model was developed by Robert Grady in his 1992 book Practical Software Metrics for Project Management and Process Improvement. The acronym FURPS describes the major categories of requirements with subcategories as shown below:

In most cases, working with non-functional requirements is done using techniques from engineering and is often referred to as Requirements Engineering.

Functionality

These are the functional requirements discussed elsewhere.

Usability

The ease with which users can operate and learn the system. This includes task time and response time for user requests to the system, the availability of help, documentation, wizards, support tools and the degree of conformance with existing and familiar human-machine interaction standards (like Microsoft's Window application standards).

Usability is not a behaviour of the system and therefore cannot be a functional requirement. We can't have a use case for it, but it can be the critical feature that makes or breaks the acceptance of the system.

WordPerfect became the market-leading word processing software in the 1980s, knocking off the previous market leader WordStar because it was more usable. Unlike WordStar, the developers of WordPerfect realized that the end-users of their products were going to be computer illiterate secretaries. WordPerfect developed a tutorial that was tailored for secretaries to get them up and using the product within a couple of hours, as opposed to a lengthy and frustrating learning curve for WordStar.

Reliability

Reliability can be measured along a number of dimensions. Some common ones are:

1. **Availability:** Is the system available 24/7 or 99% available during business hours? Is there system downtime?
2. **Mean Time Between Failure**: How often can the system be allowed to fail? How long will it take to repair or become available? We may specify that the system has to be repaired within 5 minutes 99% of the time.
3. **Defect Rate:** What is the allowable defect rate – usually measured in bugs per thousand lines of code (bugs/KLOC)?
4. **Accuracy and Precision:** What standards for accuracy and precision are required for the system? For specific system tasks?

Performance

There are a number of measures for performance. Some common ones are:

1. **Response times:** per transaction or user event reply time. On average and the maximum allowed.
2. **Throughput:** the number of transactions per second.
3. **Capacity:** the number of simultaneous transactions that can be supported on average consistently (load) and the maximum intermittently (stress).

Supportability and Maintainability

This describes the capacity of the system to accommodate enhancements, modifications and changes and how well new technology and requirements are supported.

In the banking industry, the IRS required that, for joint accounts, the social security number of the primary account holder had to be recorded but not the social security numbers of the other account holders. Data bases were designed and implemented to meet this require-

ment. Then the IRS changed the requirement so that ALL the account holders' social security numbers had to be recorded. This produced a major problem for banks in reworking data bases to add new fields or tables to support this new requirement.

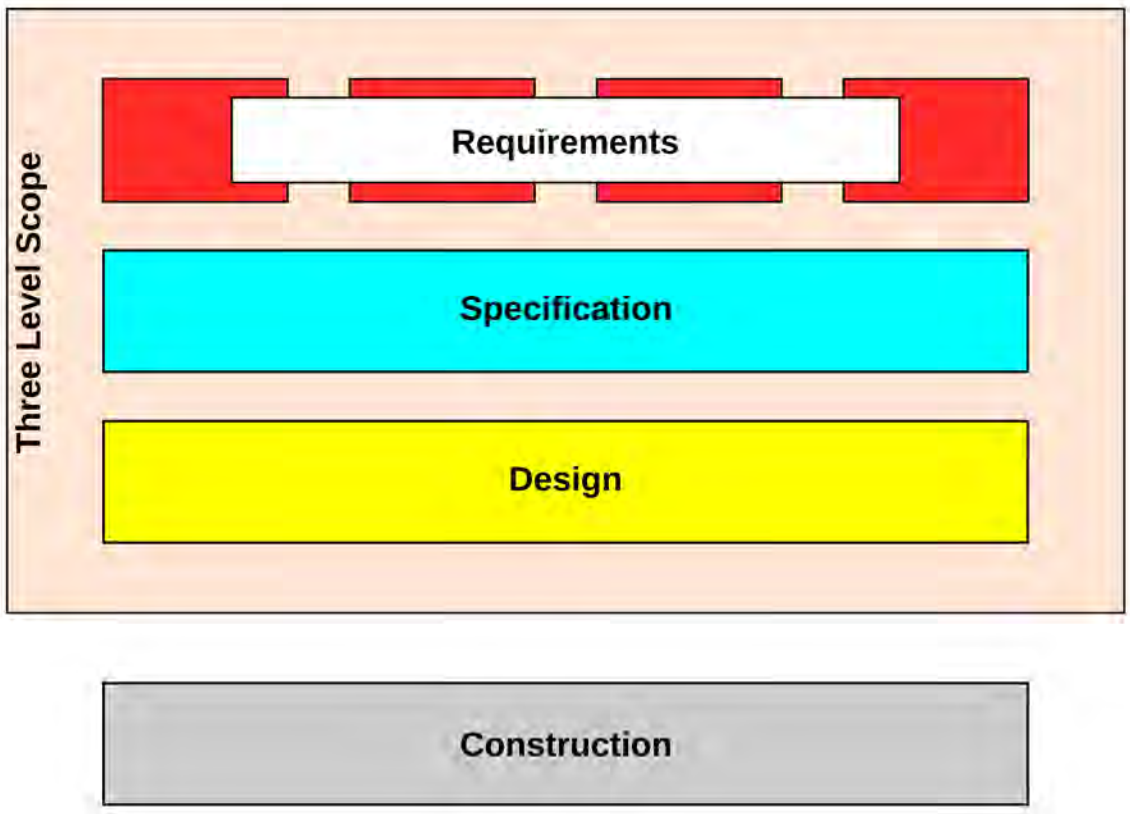
The "+" Requirements

The "+" requirements that were added onto the FURPS model were things like: design constraints, implementation requirements, interface requirements and physical requirements like operational temperature ranges or EM shielding.

Many other models exist such as Boem's model.

6 Developing High Quality Requirements

The title of this section is not “gathering high quality requirements.” There is a very simple reason for this: requirements that are just gathered or accepted uncritically as they are offered are almost always (except in the case of very astute stakeholders) of poor quality. They inevitably are incomplete, inconsistent and contradictory.



We will be delving into the model above in the following modules but the important thing to remember at this point in the course is that we are going to have to convert those varied and contradictory requirements models into a single specification model in order to transition from the requirements workflow to the analysis workflow. What we discuss in this section is part of the process of making that transition.

6.1 Specs: What we eventually have to produce

In order to figure out what we need to produce during our requirements work, it is helpful to examine what we have to turn the requirements into – a specification. We might say, we are looking at the "requirements" for specifications.

Unlike requirements, we have a very good idea of what a specification looks like and what the properties of a good specification are. We know exactly what properties of good specifications have in order to successfully to build a system. We can then use this knowledge to cre-

ate what are called robust requirements that can feed effectively into the development process, and eliminate those type I and type II errors we saw earlier.

6.2 The SRS Document

We start by looking at the SRS – software requirements specifications document.

An SRS is the basis of a complete and unambiguous document designed to facilitate the communication of information between the different parties involved in a systems development. The intent, as noted by the IEEE, is to:

1. *Establish the basis for agreement between the customers and the suppliers on what the software product is to do.* The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs.
2. *Reduce the development effort.* The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.
3. *Provide a basis for estimating costs and schedules.* The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates.
4. *Provide a baseline for validation and verification.* Organizations can develop their validation and verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured.
5. *Facilitate transfer*. The SRS makes it easier to transfer thOver the years, we have fooled ourselves into believing that users know what the requirements are and that all we have to do is interview them.

Ivar Jacobson

[requirements gathering] is frustrating, full of complex interpersonal relationships, indefinite and difficult. In a word, it is fascinating. Once you’re hooked, the old easy pleasures of building a system are never again enough to satisfy you.

Tom DeMarco

No, you can't always get what you want,* But if you try sometime,* You just might find you get what you need.

Mick Jagger and Keith Richardse software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.

6. *Serve as a basis for enhancement.* Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation.

The SRS provides the basis for a contractual agreement between contractors and customer that defines exactly what the final delivered product is to do without having it being a blueprint or design document.

In order to accomplish these objectives, the specification has to have certain properties, or else we will have quality failures when we design and build the system. Remember, the specification describes the system to be built, it does not contain any requirements – it is derived from the requirements.

There have been a number of studies as to what constitutes a “good specification” While it easy to list these properties, it is considerably more difficult to decide whether or not a specification has them or to write a specification that has them. Why this is of interest to us in requirements is that if we know what our requirements have to eventually turn into (specifications) then we can ensure we gather the right information for our requirements to make the transition as painless as possible for ourselves.

The material in this section is taken from the IEEE Std 830-1998: recommended practice for software requirements specifications.

6.2.1 Complete

A complete specification must precisely define all the real-world situations that will be encountered and the system’s responses to them. It should not include situations that will not be encountered or unnecessary capability features. It should define all of the responses of the software to all possible classes of input data in all possible classes of situations. Note that it is important to specify the responses to both valid and invalid input values.

If there is not a response to a given condition specified, when the developers are implementing that feature in the system, they will have no idea how the system ought to behave. Instead, they have to make a guess as to what it should do. Fortunately for many projects, developers can be good guessers, but it does introduce a substantial element of risk when you force them to make those judgment calls. Especially if it involves some aspect of the business domain.

6.2.2 Consistent

A consistent specification is one where there is no conflict between individual specification statements.

If inconsistencies in different sets of specification items are not resolved, developers are forced to chose one alternative over the other. For example, we might have a specification item that states that only HTML forms are to be used, and another specification statement that says every keystroke must be captured. Another example is where a report is described in one place as textual and another place as tabular. The problem is often traceable back to the fact that different stakeholders have conflicting requirements. In a consistent specification, this conflict is resolved during the requirements phase before the specification is developed.

6.2.3 Modifiable

Specifications are modifiable when related concerns are grouped together and unrelated concerns are separated. This characteristic is exhibited by a logical structuring of the requirements document. Simply put, the specification is organized, easy to use and easy to modify.

6.2.4 Correct

For a specification to be correct it must accurately and precisely encode any external business processes, rules and logic so that the system behaviour is consistent with real word operations of the organization. For example, if we are modeling a business process, the model is incorrect if the logic in our model produces different results than the business process. Incorrectness often emerges when the model is originally correct, but the subject being modeled changes and the model is not updated to reflect the change.

6.2.5 Ranked

The ranking of items is established in the development of the feature baseline, which we will look at in the next section.

The IEEE recommendation notes:

One method of identifying requirements uses the dimension of stability. Stability can be expressed in terms of the number of expected changes to any requirement based on experience or knowledge of forthcoming events that affect the organization, functions, and people supported by the software system.

Another way to rank requirements is to distinguish classes of requirements as essential, conditional, and optional.

> *Essential* Implies that the system will not be acceptable unless these requirements are satisfied in an agreed manner.
>
> *Conditional* Implies that these are requirements that would enhance the software product, but would not make it unacceptable if they are absent.
>
> *Optional* Implies a class of functions that may or may not be worthwhile. This gives the supplier the opportunity to propose something that exceeds the SRS.

Software Quality Engineering Checklist

Complete	*The specfication covers all possible inputs and conditions, both valid and expected as well as invalid and unexpected..*
Consistent	*No two specification items require to system to behave differently under the same conditions, inputs and system state.*
Correct	*The described functionality for a spec item the is consistent with the appropriate business logic, process and documentation.*
Testable	*Each specification item is quantified and measurable so that a pass/fail test can be written for it..*
Verifiable	*There is a finite cost effective process for ensuring each specification item is in the final product and can be evaluated.*
Unambiguous	*There is only possible interpretation of each specification item.*
Valid	*All stakeholders can read, understand each specification item and has formally approve the item.*
Modifiable	*The specification items are organized in a way so that they are easy to use, modify and update.*
Ranked	*The specification items are in a priority order that both the business and technical sides agree on.*
Traceable	*Every specification item can be traced back to the original requirements criteria that it is intended to satisfy.*

6.2.6 Testable

In order for a specification item to be testable, it must be stated in such as manner that pass/fail or quantitative assessment criteria can be derived from the specification itself and/or referenced information. For example, requiring that a system must be easy to use is subjective and therefore is not testable. Testable usually can be considered the same as "measurable" or "quantified." A non-testable specification means that it hasn't been quantified ("the system shall be fast") and thus must be subjectively interpreted by the developer. Items that are not testable are vague and impossible to design to in any consistent or effective manner.

6.2.7 Traceable

Each specification item should be uniquely identified to achieve traceability. That means that every feature in the system can be traced back to the requirement it satisfies, and every requirement can be traced forward to the feature that satisfies it. This becomes very important when change requests are received for a system in production: changed requirements are easily traced to features that need to be changed, and the impact of those changes on other requirements can easily be computed.

6.2.8 Unambiguous

A statement of a specification item is unambiguous if it can only be interpreted one way. If an item can be interpreted in more than one way, then the developer has to make a decision as to which interpretation to implement, which introduces considerable risk that the end result will not be what the stakeholder had envisioned.

6.2.9 Valid

To validate a specification, all the project participants, managers, engineering and customer representatives, must be able to understand, analyze and accept or approve it. This is the primary reason that most specifications are expressed in natural language. Valid means that all the stakeholders and developers agree on what is to be done. Notice that this is related to the system definition that is created early in the requirements process.

6.2.9.10 Verifiable

A verifiable specification item is one for which there is finite and cost-effective way run tests on the resulting feature. It is not enough that we can write a pass or fail test for each specification item, we need an efficient way to actually execute those tests. Testability is about being able to write tests, verifiable is about being able to actually run those tests.

6.3 Robustness Analysis

Ivar Jacobson's work in software development processes formalized the process of requirements analysis as a necessary precursor to the development of a specification. At the time he introduced the concept, there was a tendency to move right from requirements to design in the development of a system, which inevitably led to all sorts of rework situations. According to Doug Rosenberg:

> *Ivar Jacobson introduced the concept of robustness analysis to the world of OO in 1991. It involves analyzing the narrative text of your use cases and other requirements and synthesizing the requirements models into a single conceptual specification model. [Rose99; pg 61]*

What are these "requirements models" that Jacobson refers to? Each class of stakeholders has a model of how they want the system to behave and work. This model is a collection of ideas about how it should look, how people should interact with it, the processes that it uses and the various requirements they have for their vision of how the system will work for them.

We know from experience that these stakeholder requirements models have certain characteristics:

1. A requirements model describes the system in terms of how a class of stakeholders wants to interact with it or constrain the system’s behaviour, it does not describe the features of the system.

2. A requirements model is subjective and provides only the view of the system that that particular stakeholder has deemed important.
3. There are multiple requirements models because there are multiple stakeholders.
4. The requirements from different models are usually inconsistent since each requirements model is developed without reference to the others.
5. Requirements are not “testable.” It is impossible to determine if a system meets a requirement since a requirement is subjective.
6. Requirements often do not address any of the non-functional issues of performance, reliability, etc.
7. There is no sense of the level of detail or effort required for the allocation of resources. Minor functionality may consume a disproportionate level of resources.
8. The requirements are often missing critical information that the stakeholder assumes others "must" know. They are often ambiguous since the stakeholder knows what they meant and may not even be aware of the ambiguity. Often the requirements are incorrect since they may depend on a possible flawed interpretation of business processes and policies by the stakeholder.
9. The requirements from within each model are usually not complete, or even consistent and may require impossible system’s behavior. Generally, stakeholders do not consider all possible situations but confine the requirements to the most common or expected situations.

Rosenberg states that robustness analysis ensures that the inputs from the following four sources are all integrated and checked for robustness.

1. The functionality described in the business or other processes requested or needed by the various stakeholders.
2. The domain objects identified by the users and the use cases (specifically we need them to be described sufficiently for iconic interface design and database definitions).
3. Stakeholder and user functional requirements.
4. System constraints and factors that will constrain our eventual design.

In robustness analysis we critically examine the specification that is being developed with the following checks being performed:

1. **Sanity check:** make sure that the specification is correct and that doesn’t require unreasonable or impossible system behaviour given the constraints that we have to work with.

2. **Completeness check:** make sure that the requirements address all the necessary alternate courses of action and that all of the stakeholder requirements are fulfilled.
3. **Consistency Check:** make sure that the various parts of the specification are not contradictory and that the same requirement is not fulfilled differently in different parts of the system.
4. **Exception Check:** make sure that the system recovers appropriately from all the abuse cases and exceptions identified in the business processes and requirements, including time-out conditions.

The reason that we focus on these areas is that experience has shown that if we allow these kinds of problems our specification, we will eventually wind up doing rework – a robustness analysis ensures that we find these problems before we commit resources to the development of the specification.

6.4 The Baseline

Given the list of candidate requirements, we produce a baseline, which is:

> *"an itemized set of features, intended to be delivered in a specific version of the system"*

The baseline is described in terms of features that meet stakeholder requirements. The objective of the baseline is to create list of features that:

1. Meet the most important requirements – or it is acceptable to the customer; and
2. Have a reasonable probability of success from a development perspective.

This is the point at which customer or user management becomes important. A good analyst should probe the proposed features with the customer to determine the requirement that the feature is supposed to meet.

In many cases, a proposed feature does not meet the requirement the stakeholder thinks it will, but another feature that the analyst might propose will meet the requirement. Remember, analysts should always be asking "Why do you need that feature in the system?"

6.4.1 Planning the Baseline

In order to establish the baseline, we assign planning values to the candidate features. Planning values can be assigned in a variety of ways -- there is no one right way to do it -- what is of critical importance is to actually assign planning values.

Planning values are computed by looking at a number of different criteria and determining how each contributes to the overall ranking or score for that feature and should be reviewed every iteration.

6.4.2 Ranking Criteria

Some common ranking criteria are:

Importance to system

Since features satisfy requirements, we can rank a feature in terms of how important to the system the requirement it satisfies is. We could rank them on a scale of 1 to 5. A very common ranking is: critical, important, useful and ancillary.

Ranking a feature in terms of its importance to the system requires an understanding of the benefit that feature provides -- how critical the requirement is that the feature satisfies.

This is not something that can be decided in isolation but requires an understanding of the business context as well. A very common type of criteria is “Does the feature provide a core service?” Another is “Does the feature generate revenue?” This requires stakeholder input since their needs sets the ranking.

Feature Status

Qualifies the disposition of the features. Values could be: proposed, approved, incorporated, or deferred.

The status of the feature will probably change as we go through the various iterations. However, the status of the feature certainly enters into the planning; for example, we will not consider features in the planning process which are already incorporated into the system.

Difficulty to Implement / Cost

This is a measure of how expensive or how much effort would be required to implement this feature. Possible rankings for this might be: low, medium and high.

Some features may be very desirable, but the costs may be prohibitive. This measure of cost is intended to help us avoid getting trapped by “black hole” features – those which seem to eventually consume all the available resources and never get implemented.

Risk

The risk that we are measuring here is not the same as the business risk analysis we do later for the project as a whole. The risk we are measuring here is just the risk to the completion of the project.

Deciding to implement a feature based on risky or unstable technology (a biometric log-in system for example) may jeopardize the whole project because we may never be able to get the login system to work successfully. The project never gets finished because the high-risk feature could not be implemented successfully.

Stability

This is a measure of the probability that the feature will change. Features that are mutable require more analysis before they are incorporated into the system. We don't want to start developing features or trying to incorporate them into the design of the system if they will change in the next iteration. Then we are trying to hit a moving target.

The less stable a feature is, the less we want to incorporate it into the system design because changes later in the project will involve rework.

There is an exception to this. Sometimes the user is unclear about a particular feature, often because it is new or unfamiliar to them. In these cases, implementing an early version of the feature (prototyping) helps users get some hands-on experience with it and nail down exactly what it is they want.

In other words, we implement an unstable feature in order to get enough data and stakeholder feedback to make it stable.

Obligatory Requirement

There may be a specific legal or contractual requirement to have this feature in the system even if there is no appreciable benefit to any stakeholder. Often we are bound to provide certain features through the legal or contractual requirements we have with certain stakeholders.

For example, if we are selling online copies of a book, we may be required under the licensing agreement to provide details of each sale – which means that we are obligated to have a sales tracking feature in the system which captures information we may not otherwise need to capture.

6.5 FBOM Example

The following ranking criteria are defined for the requirements.

1. Critical Functionality – scale of 0-5.
2. Generates Revenue, has business value – scale 0-5.
3. Difficulty to implement (time and costs) – scale 0-5.
4. Stability – scale 0-5.
5. Risk Level to project completion– scale 0-5.
6. Business risk (introduces business risks) – scale 0-5.
7. Obligatory – yes/no.

The following features have presented so far.

1. ATM users would like to have frequent flier miles for their ATM use.
2. Bank's insurance company wants us to use facial recognition technology for ATM users. (missed these guys on our first iteration of stakeholders).
3. The police would like all ATMs to keep photographic records of users.
4. The Treasury Department wants the bank to track certain kinds of transaction patterns to help to identify money laundering -- this may become a legal requirement in the future.
5. The VP Finance would like to track all debit card purchases in order to develop buying pattern profiles of bank customers.
6. The customer service manager wants an ATM usage profile to show the types of transactions for each ATM by time and location so the bank can develop an ATM placement strategy.
7. Web users want to be able to transfer to other customer's accounts online.
8. The ATM network has provided us with a set of conformance standards that we have to implement if we want to use the network.
9. Merchants want customers to have the ability to add a "tip" to debit card purchases.
10. Management wants to sell commercial space on the ATM and Web banking interfaces.

We also have the following additional information.

The Bank is going to be using standardized ATMs purchased from a manufacturer in Indiana. There will be two models. The "White Knight" model will be positioned around town and use a standard network or dialup connection to link up to our ATM server (note: we are going to have to write our own). It comes with standard software that allows withdrawals and account queries. It is configurable to a certain degree. The "RoboTeller" model will be located at the bank branches and connected right into the bank network. They have the hardware and software to deliver full services, but we have to write the specialized code for any customer services and the code to connect it to the bank system.

Under municipal bylaws, ATMs are technically vending machines and the police have the final say on their placement. In an interview with the police chief, he suggested that they might be open to us just placing ATM in places where there is existing video surveillance.

A possible ranking is

Requirement	1	2	3	4	5	6	7	status
Frequent flier miles	0	3	1	3	1	0	no	defer
facial recognition	0	0	5	2	5	2	no	reject
photographic records	0	0	5	1	4	2	no	prop.
Treasury Department	0	0	3	4	3	0	yes	accept
track all debit card purchases	2	5	2	4	2	5	no	reject
ATM usage profile	3	5	2	3	2	3	no	accept
other customer's accounts	2	2	3	3	3	4	no	prop.
conformance standards	5	2	4	5	4	0	yes	accept
add a "tip"	4	5	3	5	2	3	no	defer
commercial space	0	5	4	2	4	4	no	reject

Possible statuses are:

- Defer – wait until more information available before making a decision.
- Reject.
- Accept.
- Proposed – still being evaluated

Object-Oriented Analysis and Design: Visual Modeling with UML

Logic Modeling

The significant problems we face cannot be solved by
the same level of thinking that created them.
Albert Einstein

There are known knowns. These are things we know that we know.
There are known unknowns.
That is to say, there are things that we know we don't know.
But there are also unknown unknowns.
There are things we don't know we don't know.
Donald Rumsfeld

Life is pleasant.
Death is peaceful.
It's the transition that's troublesome.
Isaac Asimov

exgnosis

1 Introduction

There are a number of different ways to model processes: flow charts, UML activity diagrams and standard Business Process Modeling Notation (BPMN). This particular manual will not go into details about any specific notation since in this class we will rely on the abundance of resources detailing the nuts and bolts of these notations. Instead, this material is going to focus on the problem of actually modeling a process or logic.

Most of the challenge in modeling logic occurs when we are trying to capture business or process logic. The primary reasons for this are:

1. The business processes may be broken. They may be impossible to execute but it is up to us to identify those broken processes before they get automated into the system.
2. The process we are trying to model may not be a real process, but is some other sort of activity labelled as process. For example, in some organizations, what might be called a "procurement process" is in reality people just improvising and then winging it.
3. The official processes may not be what people are actually doing. You often hear people talking about the official process and then the real way things get done – often expressed as "If we followed the process, no one would ever get anything done."
4. There may be competing processes because different groups all have their own personal or group processes that are unique and to not match up against what others are doing. ("That may be how you did expenses in that other team, but that’s not how we do it here.")
5. Most importantly, decision-making processes in use may not be logical processes.

This sort of critical analysis is essential as part of the robustness analysis we discussed earlier in the section on producing high quality requirements.

2 Logical Models

A fundamental activity that occurs all the way through the development process is the modeling and analysis of logic. In previous modules we have used the term "process" to apply to software development processes, but in this module we will be using the term "process" in a different sense – to represent the more general idea of a logical process. When we use the term "process" without qualification, we will mean a logical process, otherwise we will qualify it in some way as in "development process."

We are also going to be looking at state modeling since it is intimately associated with process modeling since every logical process can be written as a set of states and transitions, and vice versa.

2.1 Logic and Processes

Even though we don't necessarily consider logic, such as in business rules, and processes, as in workflows, to be the same, when we look at them from a modeling perspective, we can see that they are really just different views of the same underlying type of subject.

1. Logic and business rules are often implemented as an ordered sequence of decisions, which we could consider a process for executing the logic.
2. Processes are a sequence of logical decisions made at specific points, usually based on some sort of business rule.

No matter if we start from the point of view of modeling logic, or the point of view of modeling a process, we seem to wind up coming back to models that resemble each in form, content and style. Depending on our modeling objectives, we can emphasize one of these views over the other. For example, in structured analysis and design, a flowchart is used to emphasize the logical and business rules while a data flow diagram emphasizes the movement of data through a process. A work flow in a structured model tends to be optimally modeled when both these perspectives are used.

2.2 The Model Representation

We often use either an activity diagram or flowchart to model process logic throughout the development cycle.

1. During the requirements phase the diagram is used to model business processes and logic.
2. During the requirements phase the diagram is used to model user interactions, scenarios and use cases.

3. During the analysis phase the diagram is used to model the system processes and algorithms that implement the system functionality, and to help map functionality onto system objects.
4. During the design phase the diagram is used to model the program implementation logic used by the developer.

2.3 What is a Logical Process?

For the purposes of this course, we will use the following operational definition of a process:

> *A process is made up of a series of actions that can be executed in a number of possible sequences. This series of actions must have the following properties to be considered a process:*

1. **Well defined values:** All variable data and information used in the process is defined as inputs to the process. Any values used in the process that are not provided as inputs are hard-coded and invariant and do change between execution instances.
2. **Multiple alternatives:** At specific points in the sequence, conditions may be specified that allow for alternative paths through the logic. There is no requirement for all of the specified actions to be performed during the execution of the process.
3. **Invariance:** The same inputs always produce the same outputs.
4. **Deterministic:** The same inputs always follow the same path through the sequence of actions.
5. **Unambiguous:** Every decision point unambiguously specifies an alternative path for every possible input.
6. **Complete:** A path through the sequence is defined for all possible values of all the inputs to the process.
7. **Decomposible:** Each action that makes up the sequence can either be completely described atomically or can be described itself as a process.
8. **Concurrency:** The sequence of actions can split into sub-sequences so that each sub-sequence can be followed independently of and concurrently with the others provided each sub-sequence itself is also a process.

While seeming complex, this definition just formalizes what you already know about processes intuitively. There are a lot of things that look like processes but fail to be so under this definition. For example, business procedures that require a subjective decision are not invariant since different people following the same procedure may come to different outcomes depending on what each person subjectively decides.

We often see ambiguity in procedures ("Don't sell fireworks to people who aren't mature enough") which often leads to subjective interpretations. However, the most common reason why what appear to be processes but are not is because of incompleteness. ("We don't know what to do in this case; our policy doesn't cover it.")

The invariant condition just means that an activity is performed in a consistent manner. It should not be done differently on Thursday than it was on Wednesday unless it has different alternatives specified for Wednesday and Thursday.

The deterministic and unambiguous conditions means that the sequence of the activities is always clear, we never get to a point in the process where we do not know what activity to do next. The decomposability condition says that we should never reach an activity and not know how to do that activity. The last condition could be considered more optional than the others – it just means that we can subdivide our process into independent sub-processes that can be all done at the same time.

2.4 Descriptive Processes

A process is descriptive when it describes how something is done. A descriptive process does not limit what must be done, but is more like a guideline.

Normally the process being described is documented in some manner so that when the process changes, the related documentation must be changed so that it continues to describe the process.

Example: A recipe is a descriptive process. It describes how to cook something, Chicken Tandoori for example, but the cook can make changes in the recipe as they see fit. However, if changes are made in the process of cooking Chicken Tandoori that the cook wants to keep, then the written recipe (documentation) needs to be updated to include those changes.

The main problem that many people run into when working with descriptive processes is confusing the documentation with the process. Changes in the process are resisted because the documentation is seen as being sacrosanct.

To summarize descriptive processes think "Follow the process, more or less, if you want to get the job done."

We generally encounter two kinds of descriptive processes.

1. **Workflows:** These are purely descriptive and describe how activities are sequenced and organized. The emphasis is on the sequence of steps and the logic of how to select the alternatives.
2. **Algorithms:** An algorithm is also a sequence of steps by the emphasis is on describing how each activity is performed. The emphasis is on the content of the steps rather than on the sequence of steps.

This is not an either/or distinction, a process may be both and workflow and an algorithm to varying degrees. The difference is what we are emphasizing in the process.

2.5 Proscriptive Processes

A process is proscriptive when is describes how something must be done. A proscriptive process sets limits on what can be done, and can be thought of as the rules that define and delineate the actions that can and must be taken.

Normally the process being proscribed is regulated in some formal manner so that to change the process, the related regulations must be changed so that process can be allowed to change.

Example: If you are applying for a loan at a bank and do not follow their process, you will not be allowed to get a loan. However, even bank employees must follow the process or there will be sanctions applied by the bank, or perhaps even criminal charges.

Proscriptive procedures define what must be done and are usually intended to protect the interests of those involved in or impacted by the execution of the process.

In a nutshell, proscriptive process are rule oriented. They are there to describe how to follow the rules or policies that are in place, usually to ensure that something is done correctly or to protect the interests of the stakeholders.

For example, a significant percentage of the regulations that make up our legal system deal with procedural law, which defines how something must be done to be legally valid. For example, how to arrest someone and advise them of their rights, how to execute a search warrant, etc. Procedural law ensures that the rights and interests of the various stakeholders who could be affected are preserved and also provides a way of deciding if something was done correctly.

To summarize proscriptive processes think "Doing it by the book"

We can distinguish between two kinds of proscriptive processes.

1. **Procedures:** A procedure is a process that has been mandated by an organization as "How we do this process here." A procedure must be followed to ensure that the organization's policies that are relevant to a process are applied. A policy can be thought of as a rule.
2. **Protocol:** A protocol is a procedure that transcends organizational boundaries. Procedures generally are set by an organization at some level; e.g. corporation, division, state or federal government, but a protocol is a set of trans-organizational procedures that deal with a specific subject area.

Procedure example: When hiring musicians for an orchestra, the policy of the orchestra is that the only basis or hiring must be the quality of playing and not gender or age or any other factor.

To apply this policy, the procedure for auditioning a musician is to have them play from behind a screen with no information about them supplied to the adjudicators. Failure to follow this procedure, including even being able to hear the clicking of high heels, is a violation and the audition is declared void.

Protocol example:The field of biology has established "aseptic" protocols which are used in medicine, veterinary science, dentistry and other biological related fields. The field of aviation has established pilot protocols such as the preflight inspection.

Failure to follow protocols usually results in sanctions but more likely could result in some kind of systems failure or disaster.

Protocols differ from procedures in that they often codify algorithmic information related to a subject area as opposed to a procedure that enforces organizational policies.

As we look around, we can see that we follow many procedures in the course of our daily lives, from following the traffic signs and regulations to how buy lunch to how we do our banking.

Procedures are not bad things in and of themselves. We rely on people following procedures to ensure that we all can all live together safely. As a result, we sanction those that don't follow procedures (In Canada, we sanction those that do not follow the procedure of driving on the right-hand side, while in the UK, driving procedure demands we follow the policy of driving on the left- hand side).

The primary difference between a procedure and a protocol is one of scope. Protocols usually mandate certain practices which a community or profession has identified as a "best practice" or a "necessary practice." For example, the protocols on the treatment of epidemics, or how an archaeologist should document a new discover or how a crime scene investigator should collect evidence.

An organization may have to establish procedures to ensure that protocols are followed. The general rule of thumb is that no one enforces protocols, but if you don't follow them and something bad happens, you are held responsible whereas if you had followed the protocols, you would not be considered responsible, or at least not as responsible.

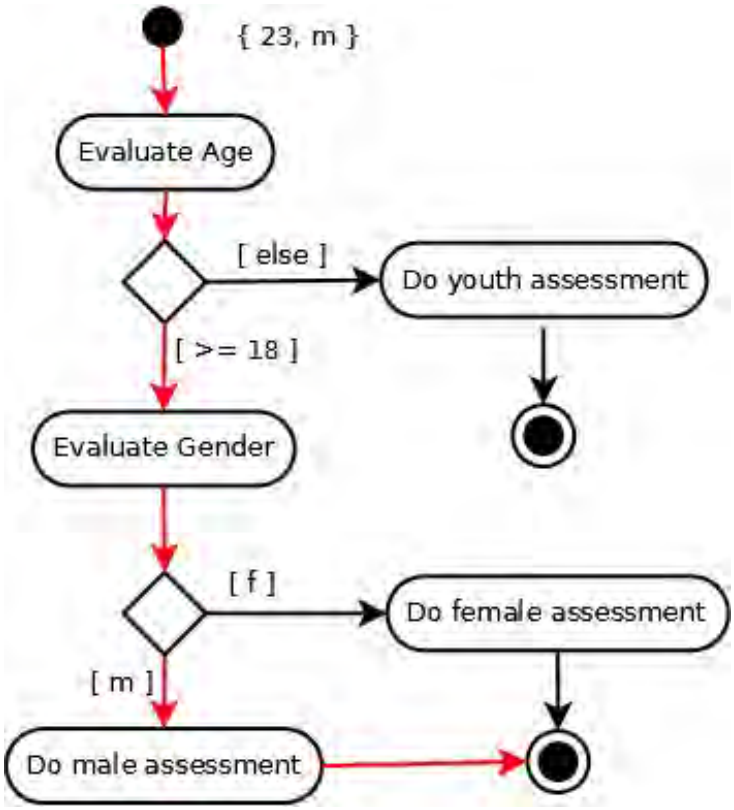
Generally procedures are arbitrary, like driving on the right as opposed to the left, while protocols are based in some best practice or body of evidence, such as having your doctor wash his hands before he operates.

3 Verification and Validation

In a previous model we introduced the IEEE best practices in terms of correctness, completeness, etc. This sort of analysis is especially important when we are dealing with logic diagrams since the possibility of errors in either the subject of our model or model is increased (remember the type I through type V errors)

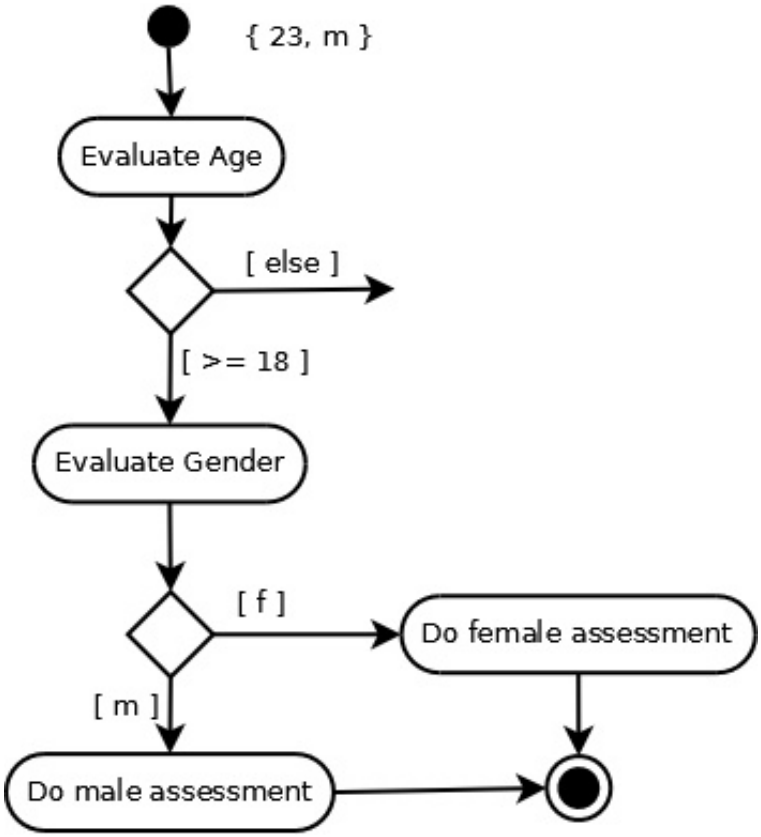
An important technique for validating our logic is the idea of exercising an activity diagram. To exercise a diagram is to provide a set of test cases that causes every set of transitions in the diagram to execute.

For example, in the diagram below we can see the transitions exercised by the test case {m.23}



A diagram is completely exercised (100% coverage) we have a set of test cases that exercise all of the transitions ({m, 23 } {f, 23 } {m 17})

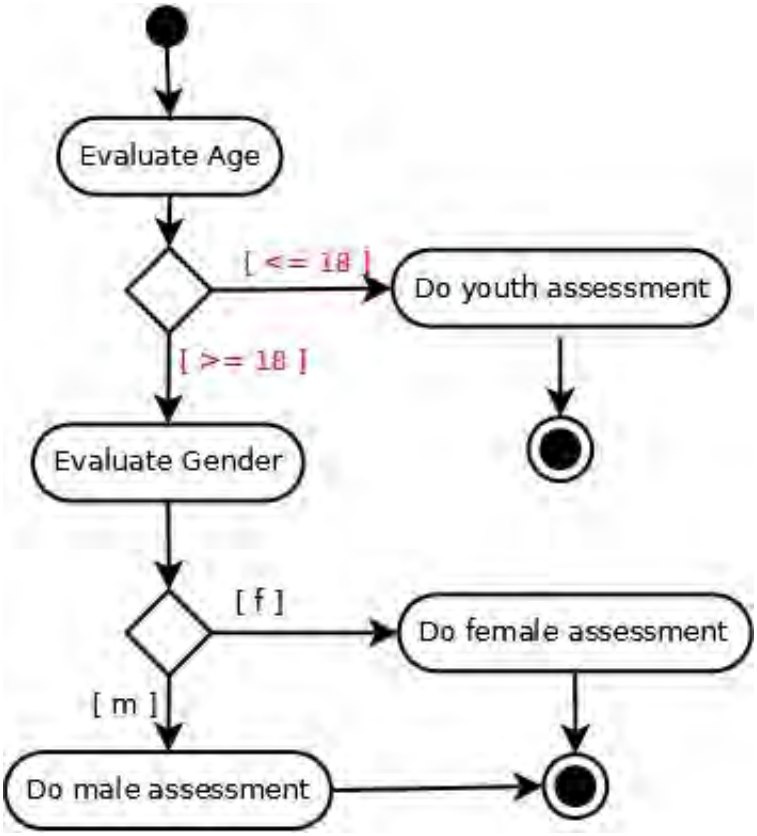
3.1 Activity Diagram Incompleteness



A diagram is incomplete when one or more branches are missing.

3.2 Inconsistent and Ambiguous Diagrams

Usually this is due to poorly formed decision conditions.

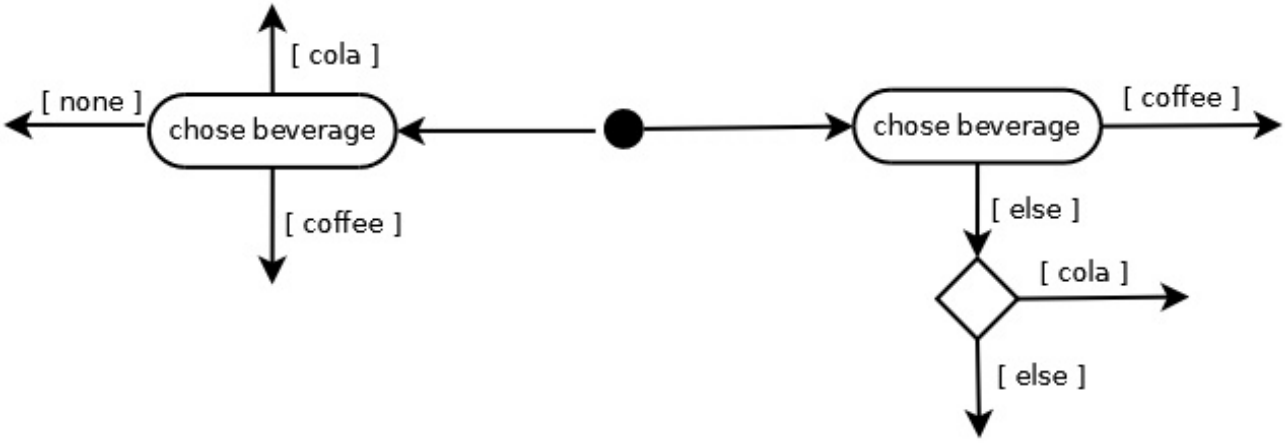


The above diagram shows the ambiguity that comes from poor boundary conditions for the guard. We could have produced the same problem with the conditions [<18] and [>18].

In these situations, there is often a specific test case that can be chosen (usually by following the functional testing technique of boundary value analysis) that show us the flaw in the diagram. For both the cases just mentioned, a test case that would show the error is {m,18} or { f,10 }

3.3 Incorrect Diagrams

Incorrect diagrams are those that do not describe either the existing process accurately or define a process where some aspect of the process cannot produce that required results.

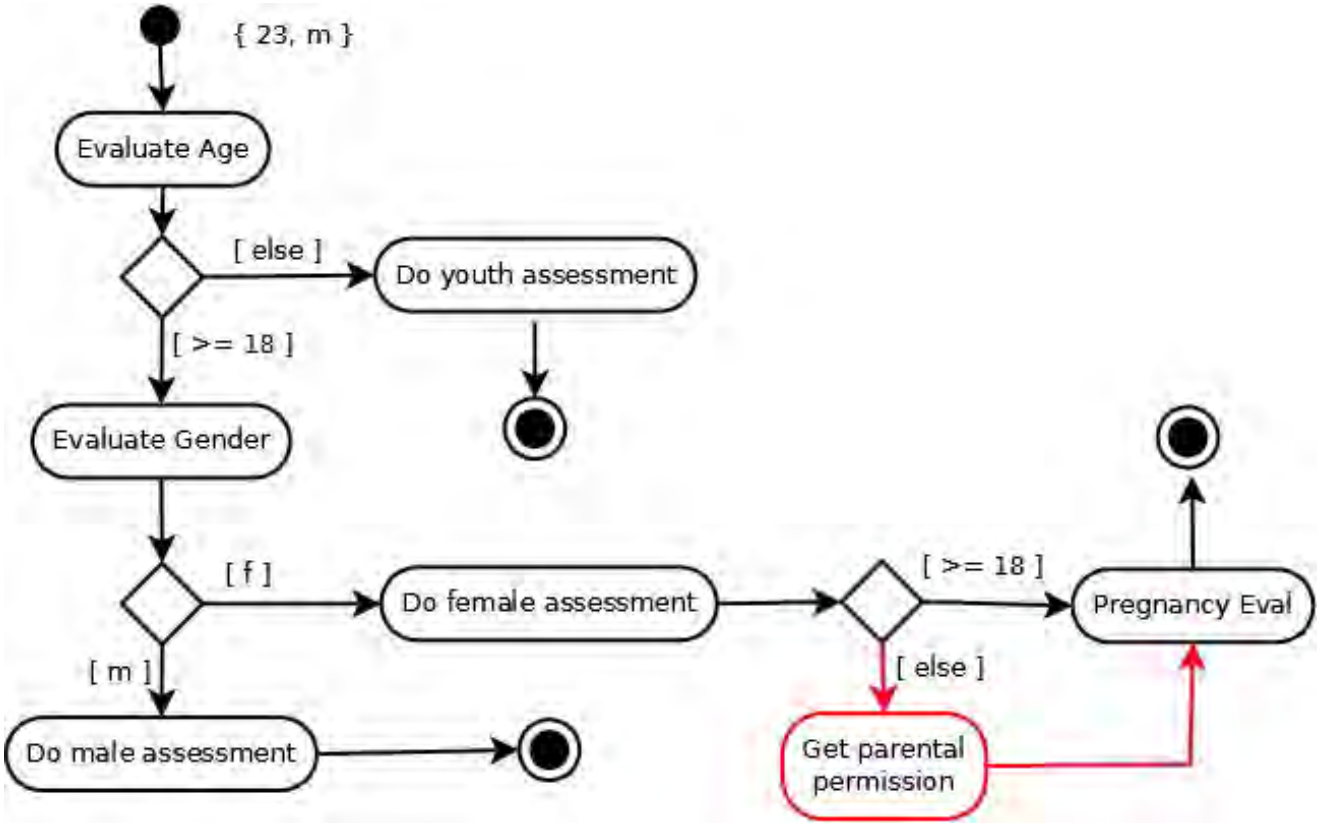


The branch on the right is familiar to us and codes the logic (look for beverage, if you find coffee, do that, otherwise if you find cola then do that, or else.) What you will see a lot in poorly executed UML is something like the left branch.

The left branch is incorrect because it does not capture the sequence that coffee is chosen before choosing cola. Instead, when both coffee and cola are found, no decision is possible because of the missing logic. Note that cannot be fixed by correcting a guard condition but instead the underlying logic has to be corrected.

3.4 Unreachable Logic

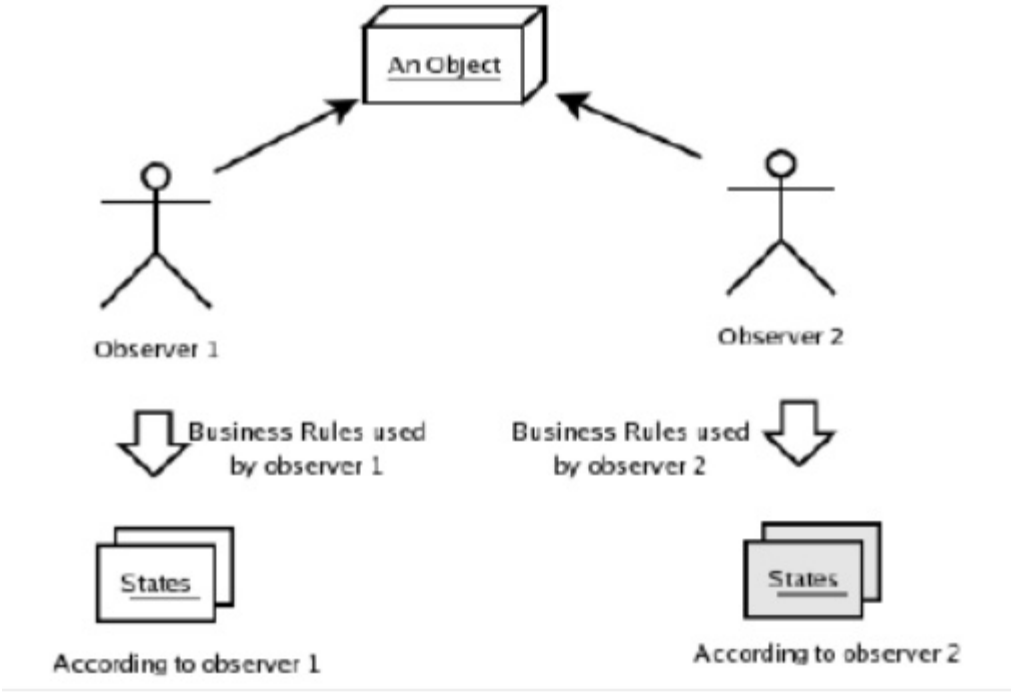
This occurs when no test case can be designed or chosen that will cause a specific transition to be exercised. Consider the following diagram and note the part of the logic that can never be executed.



4 States and Objects

The collection of an object's attributes and their values are often used to compute an object's state at a particular moment. However, a common error that we see in UML books and courses is the assertion that the state of an object is defined by its attribute values. This gives the impression that the state of an object is a property of an object.

In fact, we use state in OO and UML in the same sense that we use the idea of a state in our everyday lives. The state of an object is not a property of the object; it is a judgment made about an object by applying business rules to the values of the object's attributes. Changing the business logic used to determine state while leaving the object's attributes unchanged may change an object's state.



States are often used to map a continuous domain of precise data values into a relatively small set of fuzzy categories. If we take a person object with a data value "birthdate," we can use that data value to compute the property called "age" which is still measured by a numeric value. We can also have a state called "old." We all know that being old is subjective – it depends on the rules we use. Is someone born in 1960 old, for example?

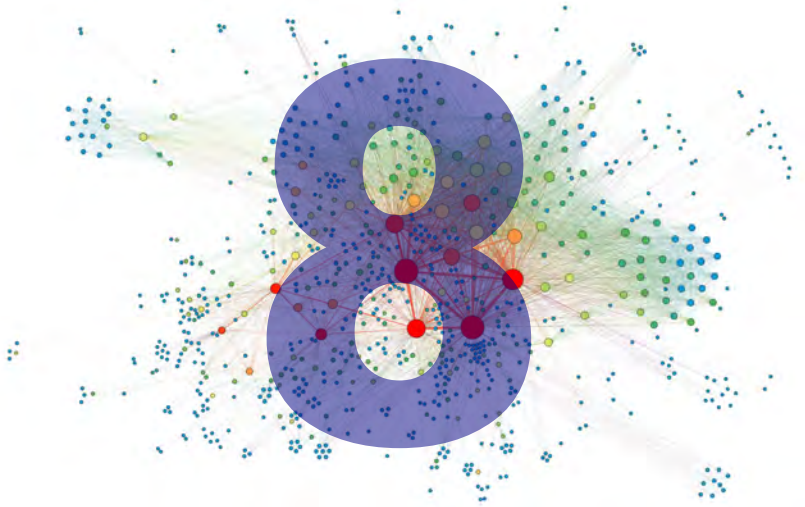
A person's height is a property but whether they are tall or not is a state, and that depends on a rule that defines what tall means – different rules apply for an NBA player than for a jockey. For users, states are a mapping of object attributes to fuzzy categories, like the height attribute getting mapped to the fuzzy categories of tall and short. These mappings can idiosyncratic with each observer or group of observers has their own.

There is no dispute over the value of an attribute no matter who is measuring it, a person's height is always 5'8" and their age is 32. But we do find disputes about states – is this a tall old person or a short young person? It depends on who is doing the evaluating.

One point that tends to confuse the issue are methods that supposedly report attributes but in fact report states. For example, a method that returns a colour returns an attribute if it returns an RGB value, but returns a state if it returns a value like "magenta."

The critical difference between states and attributes is that when the object remains unchanged, attributes are invariant across observers, but states can vary depending on the observer. We often see users and domain experts arguing over the interpretation (state) of data (attributes).

Users and objects use states to manage how we interact with other objects, so the idea of state is very critical to OOAD. Users think in terms of states and tend to think of states as belonging to particular objects in the same way that properties are seen to belong to objects.



Object-Oriented Analysis and Design: Visual Modeling with UML

Use Cases and User Stories

One of the more remarkable aspects of use cases is that they have achieved such wide currency despite an almost complete lack of precise definition. Entire books have been devoted exclusively or primarily to use cases without even so much as offering a definition.
Larry Constantine

The greatest problem in communication is the illusion that it has been accomplished.
George Bernard Shaw

If there is any great secret of success in life, it lies in the ability to put yourself in the other person's place and to see things from his point of view - as well as your own.
Henry Ford

exgnosis

1 Introduction to Use Cases

Larry Constantine's comment is an interesting observation on how use cases went from being a useful requirements tool to being hyped as the magic bullet that would solve everyone's requirements or system development problems. In reality, they haven't been that solution and IT projects today are faced with the same sort of woes that they gnashed their teeth over before use cases came along.

Over the last decade or so the following claims have been made for use cases:

1. All requirements should be written as use cases.
2. All business processes should be rewritten as business use cases.
3. The use case model should be the basis for the system design.
4. Developers should design and develop the system from use cases.

All of these statements are nonsense, and many projects have failed directly as a result of adopting one or more of these assumptions as a working principle, yet there continue to be many books, courses and consultants who push these ideas.

This course is about pragmatics, which means that theory takes a back seat to what works. There isn't a particular point of view pushed in this course, no specific methodology or approach deemed to be “correct” or superior to any other. Instead, the course looks at what works, what people do that gets results regardless of the doctrinal purity of their techniques. It's also important to look at what people do that doesn't work and the common mistakes that people make with use cases. While we all should learn from our mistakes, it's a lot less painful to learn from the mistakes of others.

Some of the more common use case errors include:

1. Using use cases to document information that is more effectively documented in other forms or formats.
2. Using use cases at the wrong time in the development process.
3. Not capturing the information that should be captured in a use case.
4. Capturing information in a use case that should not be captured in a use case.
5. Using a use case form that becomes difficult to use, usually resulting in a significant waste of project resources.
6. Adhering blindly to a particular theoretical approach or use case format even when it is not applicable to the project at hand.
7. Not providing traceability from the use cases to the system specifications.

8. Not utilizing effective records management for the use case documents and the use case model.
9. Not using the use cases in the development of a unified requirements model.

1.1 Defining the Use Case

Use cases were very clearly defined by their inventor, Ivar Jacobson.

> *A use case describes or defines how the system will behave under various conditions from the point of view of the user. Use cases treat the system as a black box, and the interactions with the system, including system responses, are perceived as from outside the system.*

A use case is a like a script that describes how the system responds to allow the user (actor) to accomplish some goal through their interaction with the system. The use case collects together all the different scenarios or variations that could occur between the user and the system – again from the user's point of view.

Like the Dickens character who found out, much to his surprise, that he had been speaking prose all his life, what we call a use case is just a formalization of something that we all do all the time. Our knowledge about the world can be classified into two types:

1. **General knowledge:** facts and relationships about the world; like dogs wag their tails when they are happy, erasers are not good to eat, the sky is blue, etc.
2. **Specific knowledge:** episodic information that tells us how to respond in a situation.

A seminal work on episodic knowledge appeared in 1977 in a book called Scripts, Plans, Goals and Understanding: An inquiry into human knowledge structures. by Roger Schank and Robert Abelson. They provided this description of episodic knowledge [Shank77; pg 17].

> *An episodic view of memory claims that memory is organized around personal experiences or episodes rather than around abstract semantic categories. If memory is organized around personal experiences, then one of the principal components of memory must be a procedure for recognizing repeated or similar sequences... Some episodes are reminiscent of others, as an economy measure in the storage of episodes, when enough of them are alike, they are remembered in terms of a generalized episode we shall call a script.*
>
> *We use specific knowledge to interpret and participate in events we have been through many times. Specific detailed knowledge about a situation allows us to do less processing and wondering about frequently experienced events.*
>
> *A script is a structure that describes appropriate sequences of events in a particular context. A script is made up of slots and requirements about what can*

> *fill those slots. This structure is an interconnected whole, and what is one slot can affect what can be in another.*

Scripts are used as a way of structuring, remembering and communicating procedural information. In this sense, a use case is an intuitive structure.

For example, given the question "How do I get pay a bill on my web banking site?" the answer is usually given in the form of a script.

> *A customer goes to the bank website and enters his login name and password into the login screen. At the main screen, the user selects Pay Bills from the menu bar. At the pay bills screen, the user selects the vendor to be paid from the drop-down list, selects the account to take the payment from another drop-down list, and enters the amount to be paid in the appropriate text field. After selecting "proceed," the user will see a summary of the transaction and request to confirm the payment. Clicking on the confirm button causes the payment to be submitted and the user then sees a screen showing the details of the payment just processed.*

This script tells us what happens when everything works as it should, such as: There are sufficient funds in the account to pay the bill, the vendor we want to pay is selectable form the drop-down list, the account we want to make the payment from is selectable from the drop-down list, our login id gets us to the right set of accounts, we enter the password correctly, and so on.

This is called the "garden path" or "main success" scenario. We don't describe all the what-if's and the various alternatives that are possible, usually because we just want to the point.

These stories are familiar to all of us. Use cases are a special kind of these user stories, done to a more thorough level of formality, rigor and detail.

So why do we need use cases now and why didn't we need them before now? To answer that we have to remember that the use cases need to be iconic so that the users know how to interact with the system in ways that are familiar to them. In this context, the use case does two things for us:

1. It documents during the requirements phase what sort of interactions the users have with the business that would be familiar to them.
2. It provides a basis for the interface design in the analysis and design phases so that interfaces are iconic and usable without training.

Why Use Cases? In the olden days (like when Disco ruled), most systems were batch systems. A program started, read in data, transformed it in some manner and then outputted the results. There was usually a human or corporate intermediary who took a user request and executed a number of steps to provide a service to the user (e.g. a telephone receptionist, a bank teller or a gas station attendant).

However, modern systems are interactive and event-driven. They tend to rely on the user interacting directly with a system interface to access the system's services. Modern systems that are event-driven are designed to respond to external stimuli from the environment.

This means they are always on and waiting for someone or something to interact with them, then they respond to that event. When finished, the system goes back into "wait" mode until the next event happens.

There is no flow of control in an event-driven system, unlike a batch system, but there are usually multiple threads of execution (often running on different processors) that are initiated by responding to these external events.

Users interact with these new systems directly instead of through an intermediary like an operator, as was often the case in older systems. Technology has supported a move towards highly interactive systems that has been going on for decades, starting with the elimination of elevator and telephone operators.

1.1.1 Use Cases as Facilitator

If you go to a bank and interact with a teller to do something, make a deposit, for example, the teller walks you through the process step by step. The teller shows you or teaches you how the process for depositing money works at that bank – the teller facilitates your interaction with the bank.

If you automate that process with an ATM, the ATM now has to walk the user through the deposit process. The ATM takes over the role of the teller to facilitate the transaction.

What we cannot do is assume that the user knows enough to be able to drive the process step by step. This is exactly why so many VCRs never stopped flashing 12:00 all over North America. Users have to be able to so something iconic (which means that what they have to do resembles something they already know how to do) rather than having some interface where users have to be trained to use it.

1.1.2 Use Cases as Scripts

Users have goals and interact with systems to accomplishing those goals. Informally, use cases are stories of how a system uses a system to accomplish goals. The purpose of a use case is to initially document how a user will interact with a system in a step by step manner to achieve a goal. It gives us a record of what our users know and can do.

1.1.3 Use Cases as System Focus

The collection of all use cases taken together comprise the "Use Case Model" which defines the scope of the system by identifying everything that the system does for each user.

The basic logic behind the use case model is: if we identify all the users who interact with their system, then identify each of their interactions, we have described the totality of the systems functional requirements. However, this point of view ignores stakeholders who are non-users. Some theorists try to get around this issue by converting business processes into use cases or by some other reworking of the definitions of actors or use case to fit everything into a use case model.

The use case model serves an important function – to keep the system development focused on delivering the required functionality, and prioritizing features so that the most critical functionality is delivered early in the development process. One of the common errors seen in the field is the assumption that once we have the use cases, we have all the requirements. However, most reputable requirements experts, like Dean Leffingwell, agree that the use case model often makes up only 30%-50% of the requirements.

Use cases cannot exist in isolation. A system is made up of processes, and at various points in a process, the system process has to interact with a user. Those interactions are use cases.

1.1.4 Use Cases as User Story

Use cases treat the system as a black box. They do not describe what goes on in the system but instead describe the interaction from the point of view of the user. The user does not see what is going on inside the system and so neither does the use case.

1.2 Use Case Forms

Use cases are not diagrams, they are textual forms. As Alistair Cockburn, points out:

> *UML has had little impact on these ideas (use cases) – and vice versa. Gunnar Overgaard, a former colleague of Jacobson's, wrote most of the UML use case material, and kept Jacobson's heritage. However, the UML standards group has a strong drawing-tools influence, with the effect that the textual nature of use cases was lost in the standard. Gunnar Overgaard and Ivar Jacobson discussed my ideas, and assured me that most of what I have to say about a use case fits within one of the UML ellipses, and hence neither affects nor is affected by what the UML standard has to say. That means you can use the ideas in this book quite compatibly with the UML 1.3 use case standard. On the other hand, if you only read the UML standard, which does not discuss the content or writing of a use case, you will not understand what a use case is or how to use it, and you will be led in the dangerous direction of thinking that use cases are a graphical, as opposed to textual, construction.*

Use cases are not specifications. The textual form of a use case is appropriate for gathering requirements since it is like a worksheet for gathering requirements or a tool for allowing the analyst and stakeholder to explore requirements in a structured way.

A use case is a user story about how the user interacts with the system. It is told from the user's point of view where the system is treated as a black box. They usually start with a simple narrative that describes the use case in a couple of lines. As the requirements process progresses, the use cases are refined and detailed until they are, in some authors terms "fully dressed"

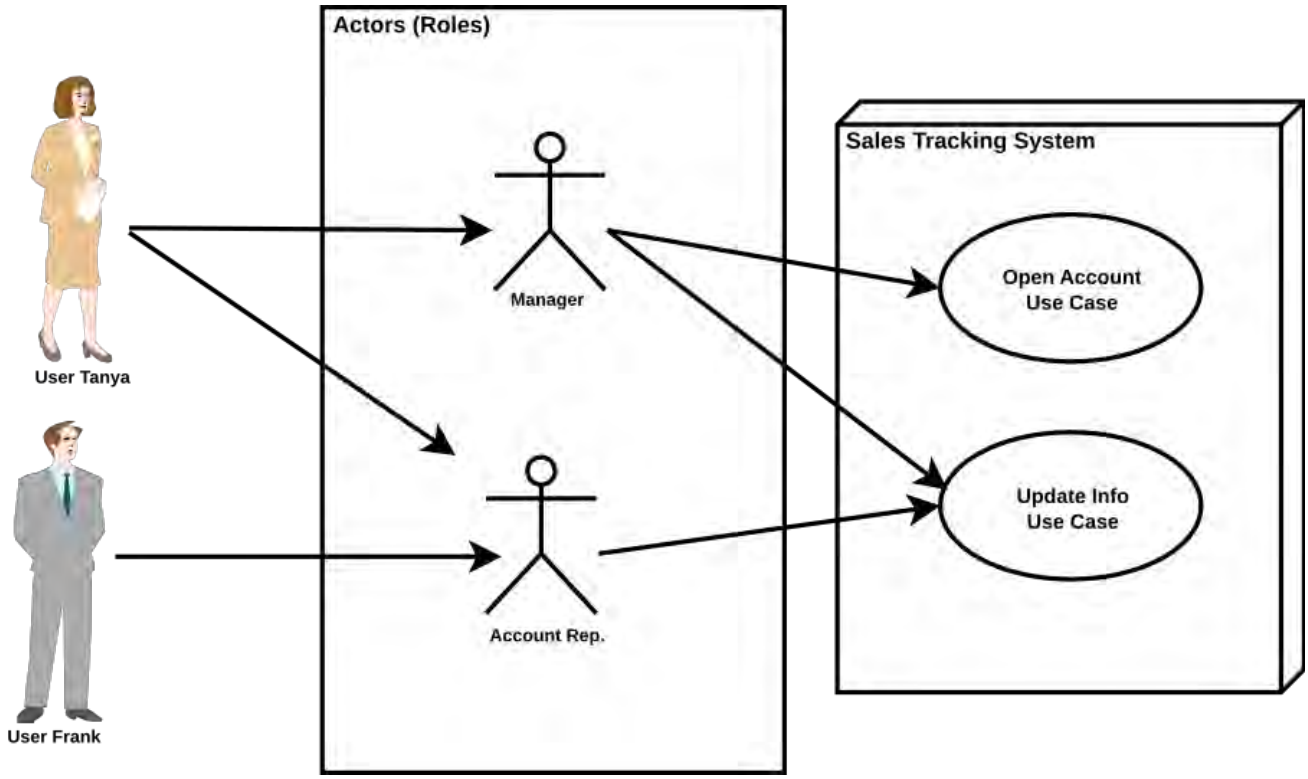
Once a use case is "fully dressed" it can start becoming a specification. Textual use cases should not be used as specifications because once they are fully dressed, they are highly complex and difficult to read. For a specification, the use case textual forms are replaced with process definitions, usually depicted in a flow chart or UML activity diagram.

We also have to remember that since they are user stories, they are not written from the point of view of the developer.

2 Actors and Goals

An actor represents a role that a user plays when it interacts with the system. An actor can represent a person or another system or a database or any other entity that resides outside the system.

The term actor is a bit of a mistake and can be somewhat misleading. Ivan Jacobson did the seminal work on use cases in Swedish. The Swedish term for “role” got mistranslated into English as “actor.” If it makes more sense to you to think “role” instead of “actor” you are not alone.



In the diagram above we have two use cases in a sales tracking system: "Open Account" and "Update Info." There are also two roles associated with the system, the role of the manager and the role of the account rep. What the diagram tells us is that if Tanya is a user, she has to be in the role of a manager in order to open an account. In other words, the use cases are associated with the roles people play when interacting with the system ("administrator", "user", "guest", etc.) as opposed to specific users.

2.1 Goals

Use cases are identifiable by looking at each actor's goals in their interaction with the system. Goals are explicit statements of what an actor wants from the use case – why they initiated it in the first case. For example, a bank customer does not initiate a withdrawal use case with an ATM unless they have a goal of "get cash."

An actor will initiate a use case to accomplish the goal, and the use case will continue until the goal is reached or until it becomes impossible to achieve the goal.

Actors goals are stated from the actor's point of view, not the system's point of view.

Secondary actors' goals (we will look at secondary actors later) are often related to ensuring that the interests of other stakeholders are preserved while the primary actor is interacting with the system.

For example, the bank teller enforces the bank rules while assisting the customer who wants to make the deposit (not accepting counterfeit money for example). The teller's goal is to ensure the transaction is done properly.

An secondary actor may have a series of sub-goals which have to be reached to assist the primary actor in reaching their goal. For example, the sub-goals of the teller may be: verify customer identity, ensure slip is filled out properly, etc.

2.2 Primary and Secondary Actors

There may be more than one actor in a use case.

For example, making a phone call is a use case which requires two actors. Both the caller and callee are external actors -- outside the system -- but a phone call use case cannot succeed with only one of the actors.

The primary actor in a use case is the actor who seeks to achieve a goal, often by initiating the use case.

A secondary actor is one whose assistance the system requires to successfully complete the use case. Secondary actors do not exist if there is no primary actor.

Another factor which has to be accounted for in a use case is the goals of any secondary actors involved. Secondary actor goals are often centered around preserving or safeguarding the interests of other stakeholders or satisfying non-functional system requirements.

In the phone systems, the actor placing the phone call is primary actor – the caller has the goal of phoning the other actor. However, the system cannot complete the use case successfully unless the other secondary actor, the callee, answers the phone. The callee cannot, by definition, exist unless there first is a caller.

Similarly, the primary actor in applying for a loan a bank is the customer. The customer has the goal of “get money”. The loans officer, a secondary actor and, from the point of view of the customer, part of the system, cannot complete the application until the Credit Bureau returns a credit report. The Credit Bureau is also a secondary actor.

Notice that if we change perspective from the customer’s point of view to the loan’s officer’s point of view, we have a loan officer’s use case called “get credit report” in which the officer is the primary actor. The credit bureau is still a secondary actor if the loans officer is accessing the credit report through the bank system.

The goal of a use case might have to be accomplished only by the achieving of a series of sub-goals by the main actor and the secondary actors. The process of achieving the sub-goals defines a set of use cases for the secondary actors. In the use cases which are implemented to achieve these sub-goals, the secondary actors are now the primary actors.

The customer at the bank cannot get the loan unless the secondary actor (loan officer) accomplishes the sub-goal of getting a credit report. The use case for the goal “getting a credit report” has the loan officer as the primary actor.

The secondary actor may also support non-functional requirements, for example the loans office may satisfy security requirements by keeping the application locked up.

2.3 Actor Personalities

An actor personality is a description of how a particular actor is involved in a use case -- this helps discover actors in the use case. Some personalities are:

1. **Initiator:** The external entity that initiates a use case by performing some triggering event.
2. **Server:** An external entity that responds to requests from the system.
3. **Receiver:** An external entity that receives information from the system.
4. **Facilitator:** An actor who interacts with the system on behalf of another actor.

The identification of the various personalities involved in a use case allows us to figure out who the secondary actors often are. However, some actors have a personality that overlaps several of these categories.

For example, a loans officer is clearly a facilitator when helping a customer and the credit bureau is a server. Is the person who receives a phone call a receiver or a server? Depending on how we chose to define the actions (is answering the phone a service to phone system or receiving information) we might go either way.

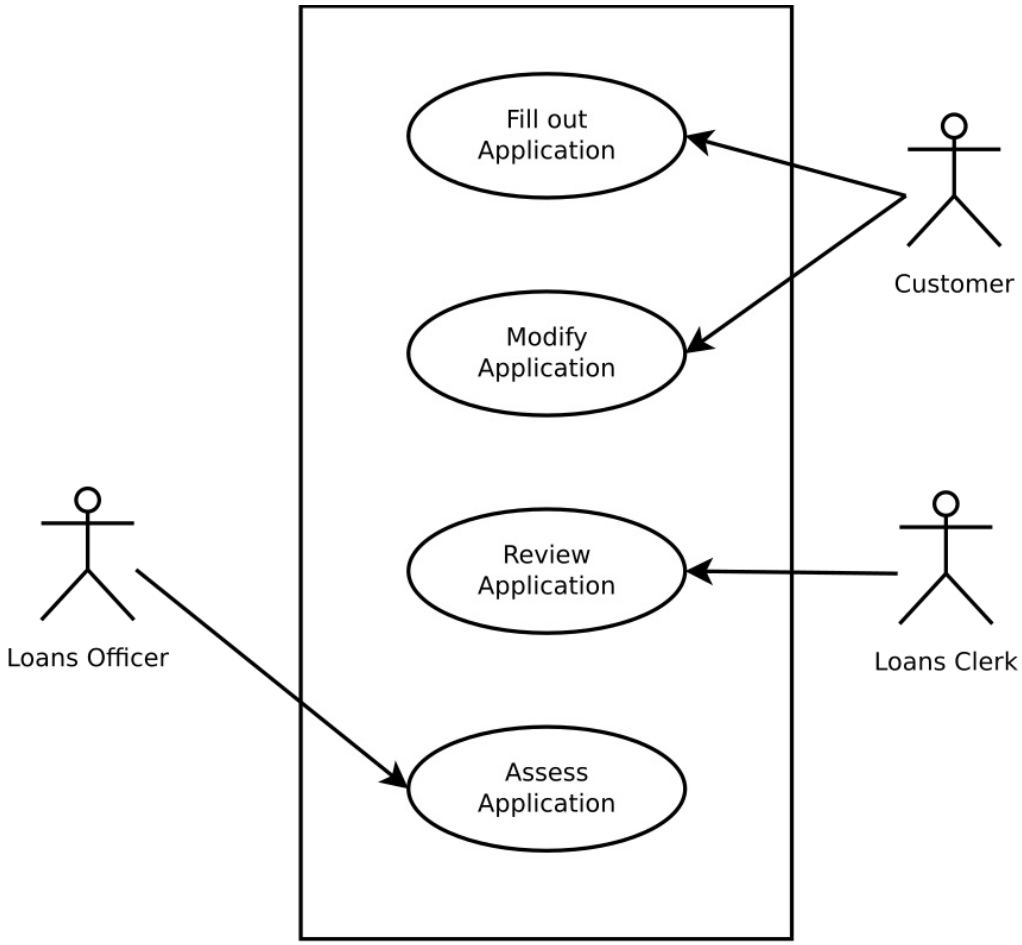
Don’t get hung up on these details – this is a tool to help your analysis, not an end in itself.

3 Bottom Up Use Case Development

This is the more commonly seen approach in the use case books. We have a bunch of users out there who are going to want to interact with the system, but we have no idea, or little idea, exactly what those interactions are. We go through the following steps:

1. We make a list of actors that will be interacting with the system.
2. For each actor, we create a list of goals for that actor.
3. For each goal, we create a use case.
4. The collection of use cases form the Use Case Model, we design the system to deliver the functionality defined in this model.

By this point, it should be clear that this approach only covers the point of view of the users which can lead to unwanted complexity later on as we try to accommodate the other requirements.



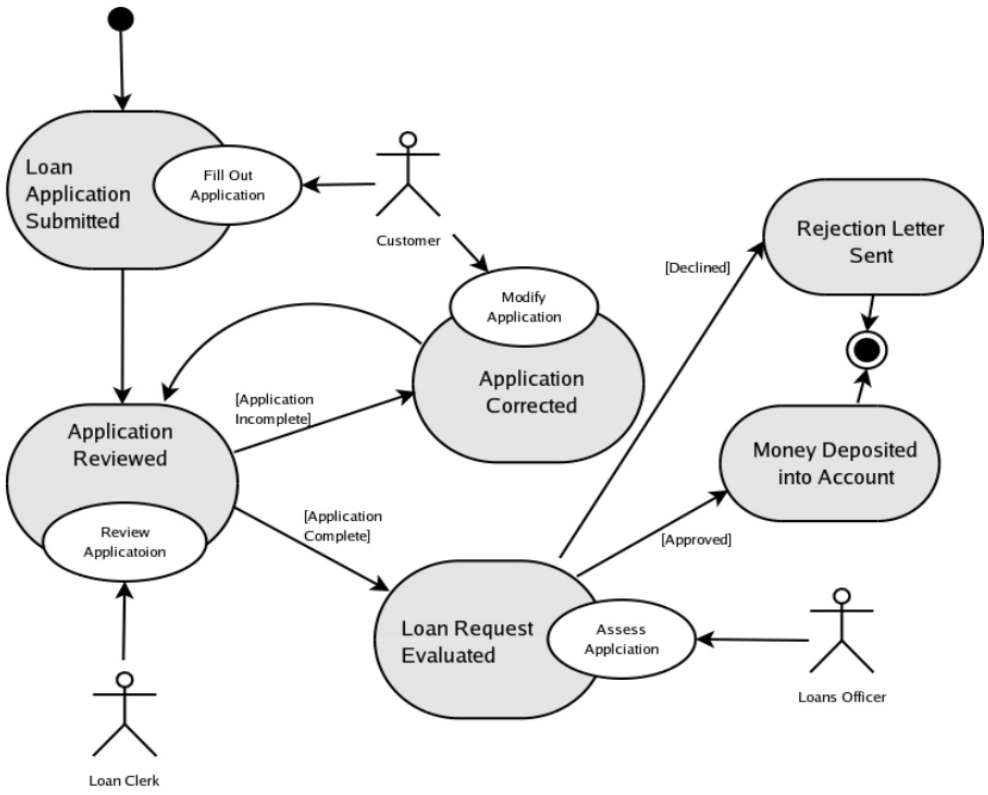
4 Top Down Use Case Development

In many systems, we are externalizing an existing business process, which means allowing users to interact with it directly via an interface. In this situation, the use cases describe the points where the process is exposed to the users. We do not need to hunt through stakeholders for actors and identify their goals and implied use cases.

Each use case represents a point in the process where some entity interacts with the system. These actors can represent a person or another system or a database or any other entity that resides outside the system and is inferred from the process definition or description.

In many projects, we will find ourselves using a mixture of top-down and bottom-up development to develop our use case model.

In a top down approach, we need to use a use case to ensure that the user will be able to interact with the system when we are done. The goals of the interaction are determined by the process, but the form of the interaction still has to be described in the use case.



5 Use Case Forms

A use case is usually textual. It can take the form of a sequence of steps, or a narrative or dialog, however, there is nothing that says a use case must be textual. A use case generally evolves from a simple brief narrative to a complex script. Use cases can also be represented as flow charts, activity diagrams or data flow diagrams. In some cases, It might even be appropriate to use pseudo-code.

UML use case diagrams are used only to identify the use cases and their interrelationships. In fact, many theorists call use case diagrams "cartoon models" and claim that their only value is as a brainstorming tool, not a documentation form.

There is no standardized form for a use case. However, as they have been used out in the real world, there has been a continuous process of refining the techniques of writing use cases. There are a number of use case “templates” in circulation. We will look at two in this course, the RUP form and the essential form.

Use cases evolve. They usually start with a simple narrative that describes the use case in a couple of lines. As the requirements workflow proceeds, in conjunction with the actors, the use cases are refined and detailed until they are, in some authors terms, “fully dressed”.

Use cases can be thought of as scripts. This fits in very nicely with users predilection for remembering or recording business processes, or interactions they have with the system, in episodes or scripts. “First you make an appointment with a loan officer who asks you how much money you make…”

5.1 The Summary Form

In the first iteration, a use case consists of identifying the primary actor, the goal and the scope of the use case and a brief narrative paragraph describing the use case activity. The summary is brief. One common error is to let the summary form description turn into a long rambling monologue. We want to defer the details of the use cases until after they have been ranked and prioritized.

Different sequences of behaviour, or scenarios, can unfold depending on the choices the actor makes at different points and the responses the system makes to the actor. These different scenarios, or “variations on a theme” make up the use case.

The primary success scenario is a sequence of actions that result in the primary actor achieving their goal. This is also called the “garden path” scenario or the “sunny day” scenario – it is what happens when everything works as it should.

This is usually going to be the scenario described in the summary form.

In many situations, the primary success scenario will account for the bulk of the actual instances of the use case. For example, most of the occurrences of the withdraw cash use case result in the customer actually obtaining the money from the ATM.

Use Case Name	Withdraw Money from ATM
Use Case ID	ATM-001
Primary Actor	ATM Customer
Goal	Get Cash from Account
Scope	Customer Activity

The customer swipes their card, enters the pin number, selects withdraw, enters the amount, selects the account to withdraw from, then confirms the account and amount. The customer takes the dispensed cash and receipt to conclude the transaction.

Normally, when elicited, the use case described first by domain experts is usually this sunny day scenario, or the primary success scenario. In some cases, it may be hard to select a single primary success scenario, there may actually be several candidates. This might suggest that either more investigation is required or that we are confusing two similar use cases.

For example, we may have a use case called withdraw which has two scenarios, one involving a teller and one involving and ATM. Upon reanalysis, we may decide these are actually the primary success scenarios of two different use cases “Withdraw from ATM” and “Withdraw from teller window”.

The main criteria for deciding if a scenario belongs to a use case is the actor’s goal – all of the scenarios in a use case share a common goal – they represent alternate paths or ways of getting to that goal or failing to get to the goal. As Ivar Jacobson points out:

> *A proven technique is to choose one complete basic path from the start state to the end state and to describe that state in one section of the description. Then we can describe the rest of the paths as alternatives of deviations from the basic path, each in a separate section. Sometimes, however, the alternatives or deviations are small enough to be explained inline as part of the basic path description. Common sense determines whether to inline the description of an alternative or to create a separate section for it.*

5.2 Detailing Use Cases

There are two standard forms of presenting the details of the use case:

1. **RUP Sequence of events:** the flow of activities is presented in a numbered list of steps (used by Jacobson).
2. **Dialog form:** the activities are presented in the form of a dialog between the actor and the system. (used by Wirfs-Brock and Constantine)

The RUP form will not be dealt with much in this class for two reasons. First, there is an incredible amount of material already available on the RUP form; and second, the essential dialog form tends to the one most used out in the real world.

Use Case	Withdraw Money
Use Case ID	ATM-001
Primary Actor:	ATM Customer
Goal:	To get cash from ATM
Scope	Customer Activity

1. Customer swipes card
2. Customer enters PIN at prompt
3. Customer selects withdraw from presented options
4. Customer enters amount at prompt
5. Customer selects account from presented list
6. ATM dispenses cash and receipt
7. Customer takes cash and receipt and ATM resets

The RUP form is not popular for a very practical reason. While it looks good when used for small, simple use cases, it does not scale well. As a use case becomes more complicated and complete, then the RUP form starts to become unworkable. One of the contributing factors to this is the requirement that the use case form acts as the main place for capturing all the scenarios in a single textual document and associated data, which results in large cumbersome documents. The essential form is preferred because it uses diagrams to capture much of the complexity of documenting multiple scenarios, and also does not try and document factors other than the dialog between the user and system.

The sequence of events is used in most of the literature; however, there is no use case standard which says that this is the “right” way to do it.

The dialog form has the advantage that is clearly identifies the interaction between the user and system which can lead to a more natural GUI design. Proponents of this form claim that it forces the developer to think in terms of how the systems has to respond to each actor action. The essential form of the dialog is the most commonly used use case form, at least according to research in the field.

User Intentions	System Responsibility
Identify self as customer	
	Present Transactions Available
Choose deposit	
	Present available accounts
Select Account	
	Request amount
Confirm	
	Dispense Funds
Take cash	

Diagrams are often better for use cases where the actor is another system – it captures the flow of information involved in the interface between the systems.

The important thing to remember is that all the techniques are available and you should always choose the one that provides the clearest description of a specific use case.

We will call this development of the descriptions of the flows "detailing." We will see how to do this in the next module; however, most theorists agree that detailing process should initially focus on describing the primary success scenario from beginning to end, ignoring (for now) any alternatives or branches.

5.3 The Essential Use Case

Popularized by Larry Constantine, the idea of an essential use case is to strip down the use case to the barest description of the interaction between the user and the system. This essential use case can then be used as a starting point for integrating the other stakeholder features into a model that can be fully tested. Larry Constantine defines an essential use case as

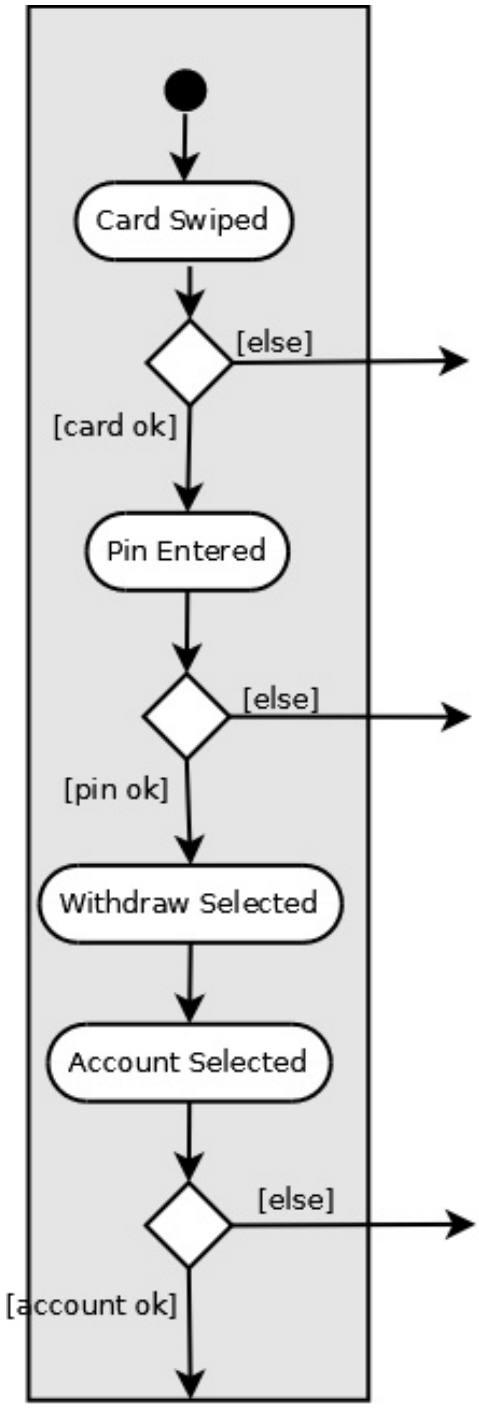
> *An essential use case is a structured narrative, expressed in the language of the application domain and of users, comprising a simplified, generalized, abstract, technology-free and implementation independent description of one task or interaction that is complete, meaningful, and well-defined from the point of view of users in some role or roles in relation to a system and that embodies the purpose or intentions underlying the interaction.*

For example, the essential use case for the withdraw cash from an ATM would look like this:

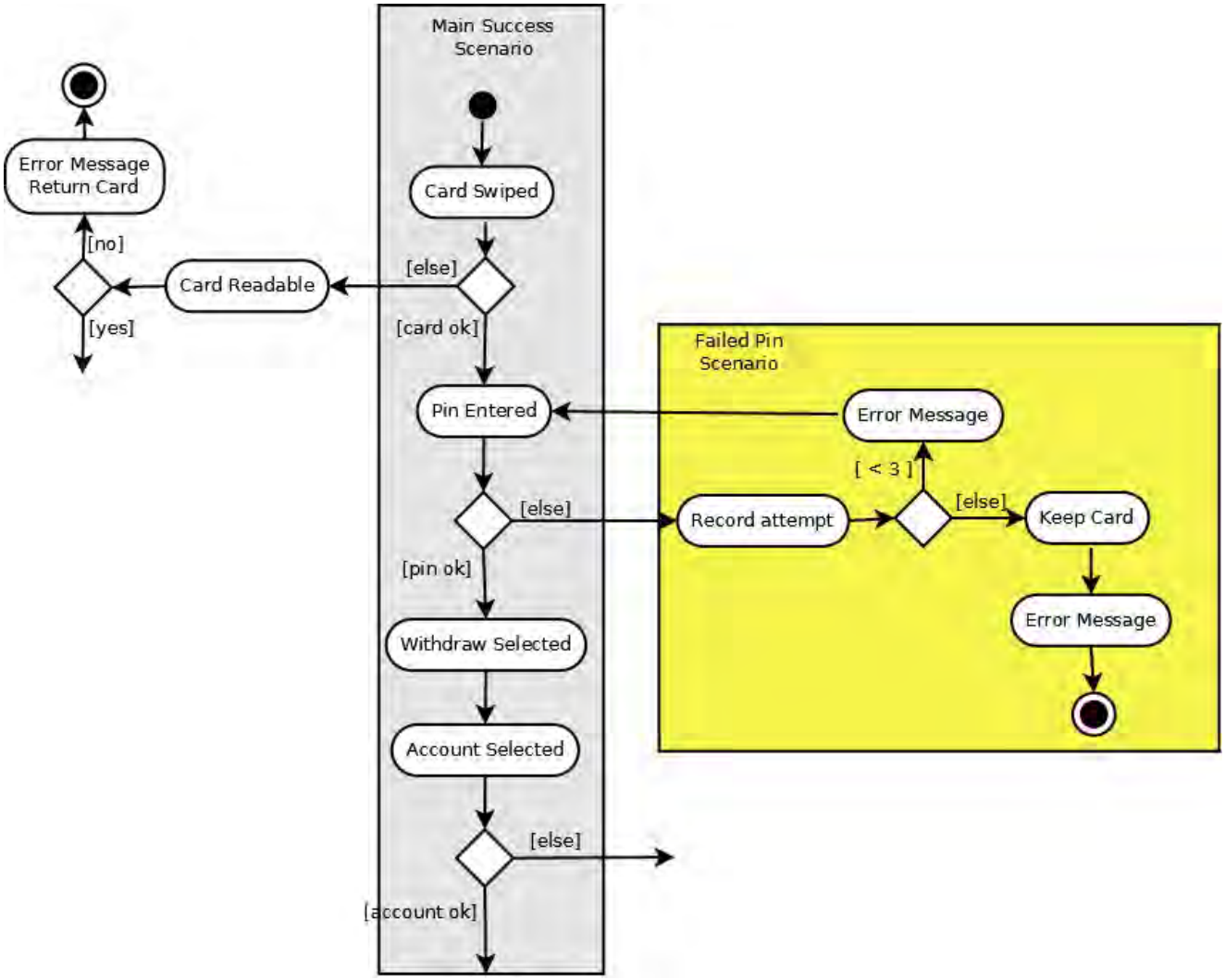
User	***SYSTEM***
Swipe card	
	Read mag strip / request PIN
Enter pin	
	Verify Pin / display transaction menu
Press key	
	Display account menu
Press Key	
	Prompt for amount
Enter amount	
	Display amount
Press key	
	Return card
Take card	
	Dispense cash
Take cash	

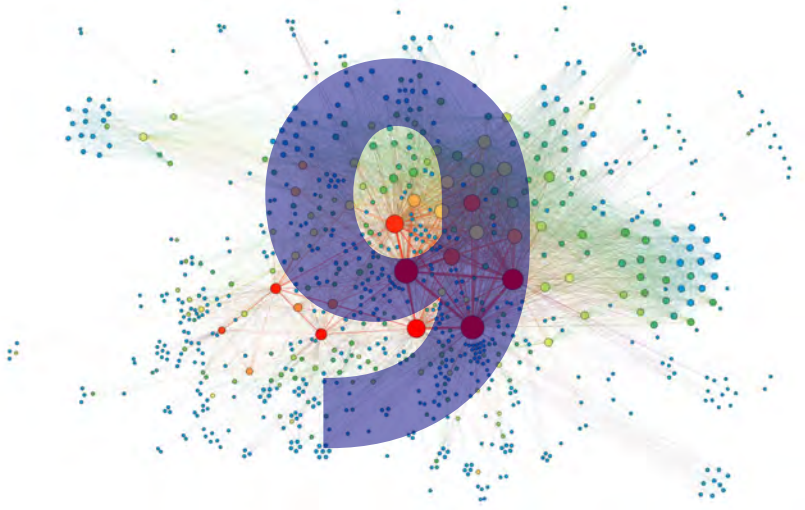
5.4 Alternate Flows

One of the reasons why the RUP and similar forms are not often used in the real world is that they do no “scale” well. The essential use case avoids this by not trying to document the alternate flows in textual form. Instead, the use case alternatives and scenarios are documented by a logic diagram.For example, we can start with the use case on the previous page to lay down the diagram backbone with the main success scenario.



Now we can add in the alternative flows. For example, we can start to look at what happens if the card is invalid, perhaps unreadable or maybe reported lost or stolen. Or maybe it just isn't valid. And we can look at the use case for a wrong PIN number entry.





Object-Oriented Analysis and Design: Visual Modeling with UML

Solutions Analysis

The most important single aspect of software development is
to be clear about what you are trying to build.
Edsger Dijkstra

First, solve the problem.
Then, write the code
Donald Knuth

There are two ways of constructing a software design.
One way is to make it so simple that there are obviously no deficiencies.
And the other way is to make it so complicated that
there are no obvious deficiencies.
C.A.R. Hoare

exgnosis

1 Understanding Analysis

In the engineering process model, we defined the Analysis as part of the process where we generate solutions. It is not generally true that we do requirements and analysis work independently, rather they tend to by synergistic. As we explore requirements, we start to think about solutions, and these in turn give us deeper insights into the problem domain.

The nature of the relationship between analysis and requirements reflect the way we solve problems. In fact, this problem-solving mechanism is so common that it has now been formalized into processes called “design thinking.” Design thinking processes, like the Stanford Design Process the Delft Process have put a structure to the process so that it can be used by development teams on a consistent and regular basis. Design thinking is being used successfully across industries to solve problems and design new products and services.

Design thinking, and by implication solutions analysis, is iterative by nature. This means that the iterative USDP approach, Agile methods and DevOps all fit nicely into our natural problems solving ways of thinking.

There are two deliverables that we should be producing during the analysis.

1. A specification of the system to be built.
2. A high-level design or system architecture.

As we saw in the requirements module, the specification is a reality check on what we are planning to build by forcing us to ensure that our solution is complete, correct, consistent and all the rest. Especially in Agile methods, the specification will evolve as we deploy prototypes of our solutions, which in turn produce a deeper understanding of the domain and the requirements.

The high level-design has become increasingly important as systems get more complex. Designs can get very large very quickly. Our high-level design is like a proof of concept before we start to invest any significant amount of resources detailing a design that has a fatal flaw.

In this section, we will be looking at some of the types of activities and artifacts that we often find in analysis activities. There is no “right” way to do analysis – every way we do analysis that leads to a solution that works. The actual challenge is to find the most efficient and effective ways to get to that working solution.

Some of what is covered here could also be considered part of the requirements process or part of the design process – but trying to do this sort of classification is not productive.

2 You Can’t Design to Requirements

The main reason that we stress the development of the specification is that it is impossible to design a system based on requirements (GASP! Heresy!!).

This primarily has to do with shifting how we think about the system being developed. During our requirements work, we kept stressing how the system looked and acted from the user’s point of view. For example, a use case describes a user interaction with the system from the user perspective while treating the system as a black box.

But we have to design what goes into that black box. We have to think about what that use case looks like from inside the system – and it looks very different. The analysis process is characterized by thinking about the functionality of the system from the system’s point of view and not the stakeholders’ perspectives.

2.1 A Few General Design Concepts

This is a well-known principle in design which states that a component or sub-system should have one and only one responsibility. This is also called cohesion. Te more logically related a component’s structure and functions are, the more cohesive the is.

Another way to think of cohesion is in terms of specialization. A cohesive design is where each component specializes in one function, and only that component performs that function. That last part means that the "stuff" that does a specific function is localized in only one place and is not replicated in other locations.

A practical way to envision this is in terms of a project team. The whole team has the responsibility of building an application. The team is divided up into groups; one group is responsible for coding, another for testing, another for database design and so on. It would be odd if the testers then went and started writing code, or the database design group started writing test cases. Each group is defined by their mandate which identifies what they are responsible for.

Within the coding group, assume there are three programmers. The first coder has the task of writing the user interface code, the second one has the task of writing the database access code, while the third is writing the business rules. Within the programming group, there is specialization of the programmers so that only one programmer is writing the user interface code and writing the user interface code is all he does. And only the intern makes the coffee run to Starbucks.

As an aside, note that database normalization is another version of this concept of cohesion. Normalization can be informally thought of as “each table contains data about only one thing, and any data item appears in only one place in the database.”

A standard step in any design or refactoring process, not just software, is to perform a high-level analysis which describes a functional decomposition of the system or application into cohesive sub-systems. Once there is an understanding of the functionality that the system is

supposed to provide from the user perspective (through an examination of the requirements, use cases, user stories and the feedback we get from earlier prototypes), we have to shift our point of view to thinking how a system would have to look like in order to deliver this functionality.

2.1.1 Non-software Example: Designing a hospital

Consider the patient's view of going to a hospital. It may be described as follows.

1. The patient arrives at Admitting and does the paperwork.
2. The patient is sent to the emergency room to wait to be seen.
3. The patient waits.
4. A doctor arrives and orders tests.
5. The patient waits.
6. The patient goes to x-ray, gets radiated, and is taken back to the ER.
7. The patient waits even longer.
8. A nurse comes by and takes blood.
9. The patient tries to amuse himself while waiting.
10. The patient is taken to the MRI unit and has an MRI.
11. The patient is taken back to the ER.
12. Patient dozes off while waiting.
13. A nurse comes by with medications and takes the patient to admitting.
14. The patient fills out forms and is released.

This is a use case or a user story. It describes the hospital visit from the point of view of the patient without any reference to any other patients' experiences that may be co-occurring, or any knowledge about what is happening anywhere else in the hospital outside of this patient's experience. We cannot design a hospital based only the patient’s description of how hospitals work because the hospital is a black box to the patient.

If we are designing a hospital, we need a lot more than just this patient use. We have to think in terms of design components. The hospital as a collection of departments (components), each of which specializes in a set of cohesive functions.

We might put together a first functional architecture where we might define this in terms of the following departments.

1. **Admitting and Administration:** responsible for keeping track of all administrative and legal issues involving a patient.
2. **Emergency:** responsible for providing medical care for patients and a place to keep them, specific tasks are allocated to doctors, nurses and other members of the department.
3. **X-Ray:** responsible for taking x-rays and MRIs for patients on request.
4. **Laboratory:** responsible for taking and processing blood work and other medical tests.
5. **Nursing:** Responsible for allocation of work to nursing staff.
6. **Medical:** In charge of doctor work assignments.
7. **Pharmacy**: responsible for supplying and controlling drugs for the hospital population.

Each department has a set of responsibilities, which in turn can be delegated to other subgroups or individuals in the department. Different departments also have to co-operate with each other to get things done. For example, if we want to give a patient a drug, the pharmacy can't just give it out. A doctor has to prescribe it, the pharmacy has to dispense it and a nurse has to administer it. This grouping of a system (or hospital) into divisions that specialize in delivering a cohesive functionality is called a functional decomposition or a functional architecture. The term high level design is also used.

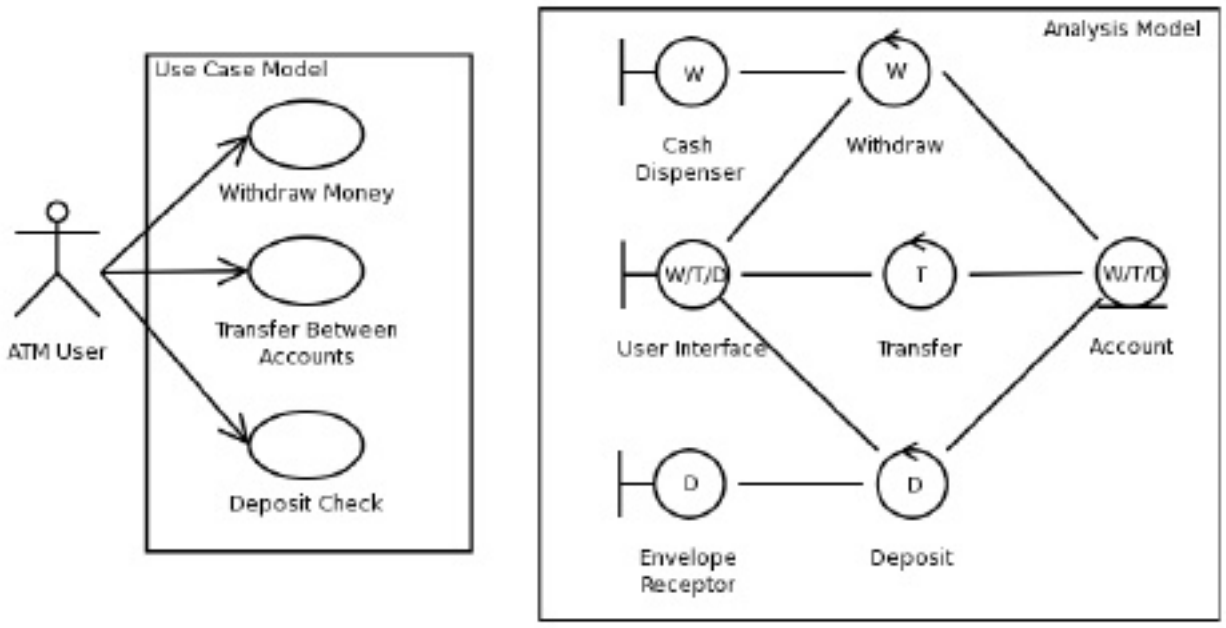
Once a functional architecture has been nailed down, it can be verified that the hospital will have the right “pieces” do all the things that a hospital is supposed to do. Once this has been done, the non-functional requirements, such as “how do we design the x-ray department to process an average of 10 patients at a time with up to a peak of 15 at once with no patient waiting more than 5 minutes?” can be used to plan out how to design the individual functional sub-systems. The non-functional requirements are often not easily solvable until we have a functional architecture because the scope of the system as a whole is too large.

This is a modular as opposed to monolithic approach to systems design. We start with packages of cohesive functionality and then work downward, refining each package's design until we finally are ready to write code.

3 A High-Level Design Example

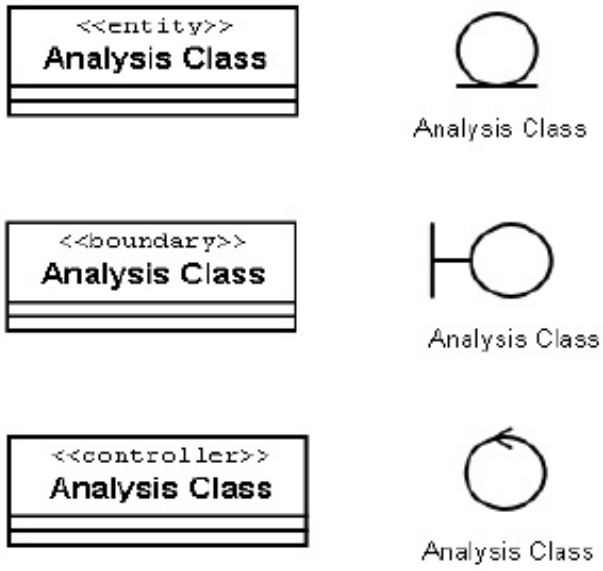
Consider the standard first-year OO problem of designing an ATM system. Don't take the problem too seriously, all that is intended to be shown here is the general process of performing a functional decomposition while creating a high-level design..

In the first diagram following, there is a use case model on the left describing the functionality of the system from the point of view of a customer. On the right, there is a starting point for our functional decomposition of the system in terms of the standard UML class stereotypes.



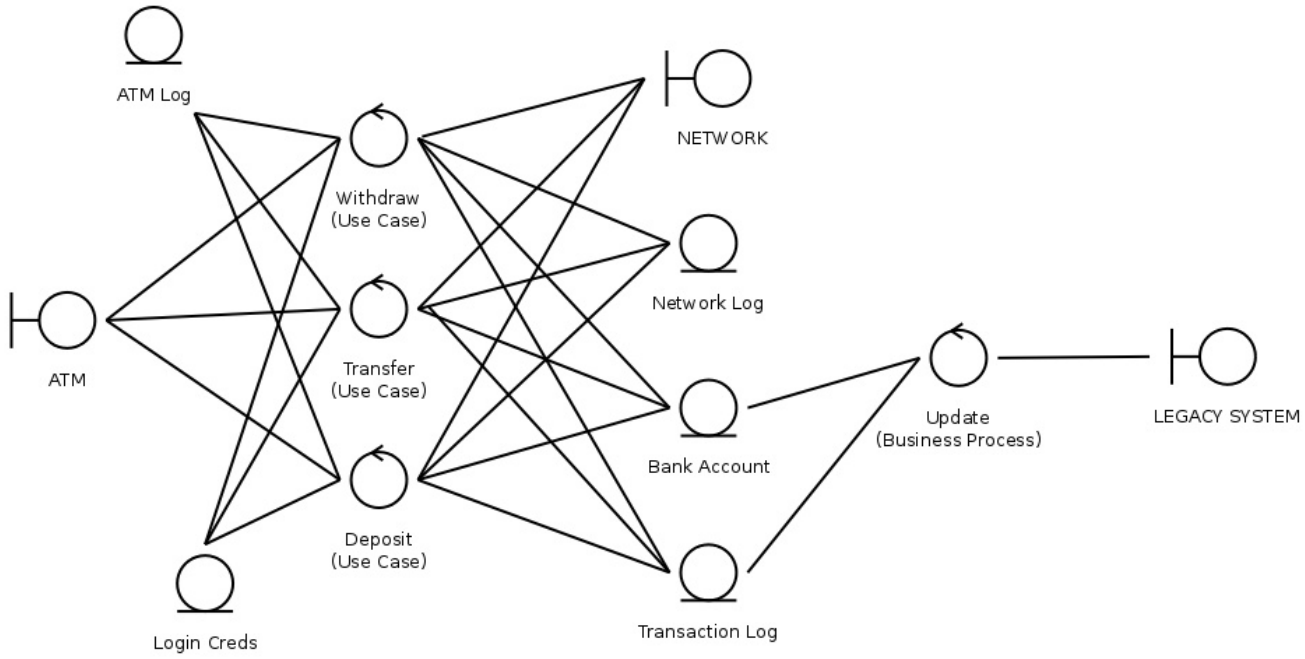
1. **Entities:** Entity classes are used to generally model long-lived or persistent information associated with some event or object. Entity classes are introduced to satisfy requirements for data to be stored or accessed. Entities are abstractions of databases, flat files and other data storage mechanisms.
2. **Boundaries:** These are the objects through which users interact with the system. At this point in the design, they are very high level and can be thought of as abstractions of terminals, user interfaces, network connections and physical devices.
3. **Controllers:** A controller can be thought of as a running program. They contain the logic that runs our system, delegate work to other objects in the system and manage the dynamic work flows.

In the preceding diagram, the cash dispenser, user interface and envelope receptor are all boundaries, the account is an entity and the withdraw, deposit and transfer classes are controllers which execute the logic to process each of those transactions respectively.



This first staring point is not very cohesive since the controller objects have multiple responsibilities. For example, each of them has the logic for managing accounts entities, as well as the login logic necessary for interacting with the user. Each controller has multiple responsibilities and there is logic replicated in multiple places; for example, each controller contains the same user validation logic.

A first refinement is to see what else might be missing from this first try. We can see that the model is missing the network connection that is needed for the customers to access their accounts from other banks' ATMs over the network. Investigation of requirements might lead to the discovery find there is a security requirement to log all network activity.

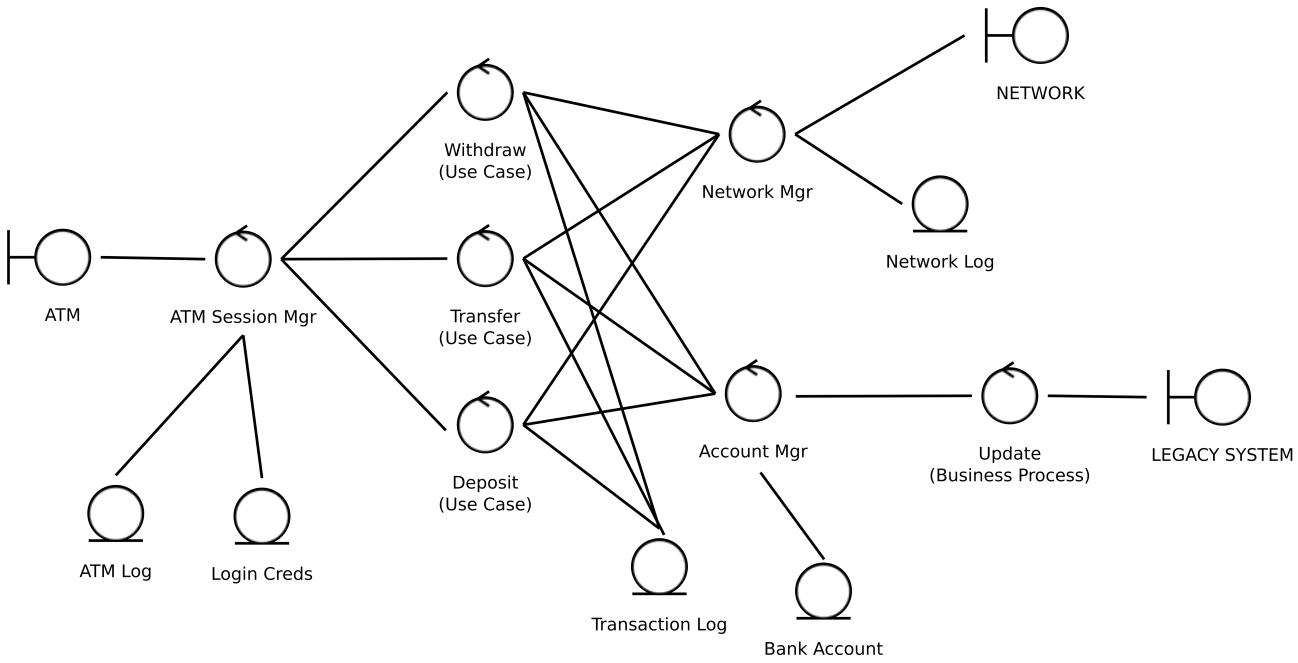


Similarly, investigation of the existing environment might reveal that the official account information is not held on the system being built, but is on the legacy bank system that the new application will access over the internal LAN. This suggests that there should be an update process to sync the two sets of data (we should be getting a bad feeling about that).

Thinking through the log-in process, customer credentials are needed to validate their log-ins, credentials that have to be kept somewhere. This suggests there might be other requirements to log ATM activity and all transactions that take place. This might lead us to explore requirements we might have missed from the Fraud Prevention Group in the bank.

Refining the model uses the same process that would be used to eliminate replicated code in a program – the replicated functionality would be encapsulated into a subroutine and then whenever that functionality is needed, a call to the subroutine is executed. This is where thinking of the controllers as executing programs, gives a strategy for improving their cohesiveness.

The model can be refined even more with another iteration.

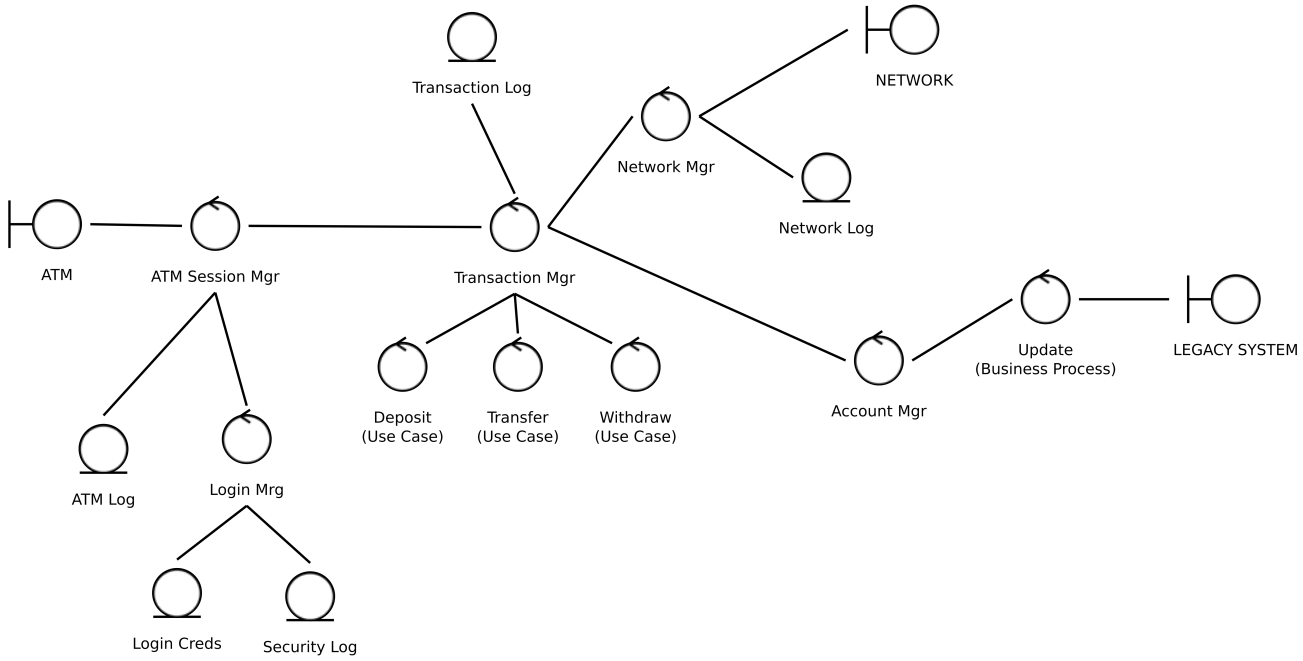


At this point, we can introduce another design concept, that of coupling. The coupling between two components is the number of connections between them. Tightly coupled systems with lots of connections are very brittle since the smallest change in one might have a devastating effect on the other.

Good design practices in all branches of engineering agree that the lower the coupling between components, the more robust and efficient the system is. One of the main sources of high coupling is low cohesion. When functionality is spread across multiple components, then every time that functionality is invoked, the components need to collaborate and require coupling to do so. The higher the cohesion, the less the components need to be coupled with each other.

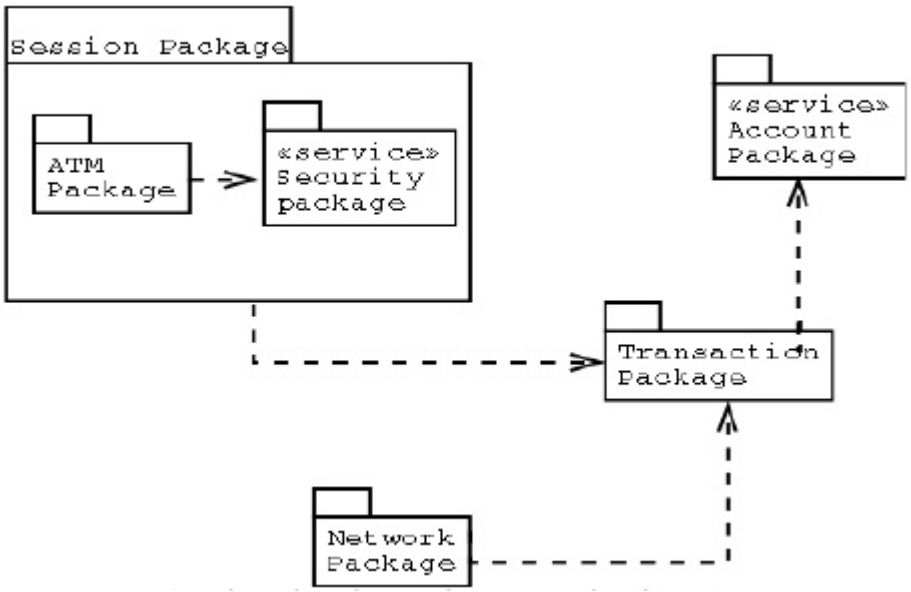
Back to our design. Now there is an account manager that contains all the logic for interacting with the account, and a session manager that handles the user log-in and validation. However, notice all those crossing lines. This suggests the system has higher coupling than it should. This suggests that there still are some low cohesion components. Another iteration produces:

In this latest refinement , the common logic in each of the transactions have been extracted into a transaction manager so the deposit, withdraw and transfer don't have to worry about dealing with a session manager or an account manager. It is also apparent that there is no need for a bank account object since the account manager can interact with the customer accounts directly on the official mainframe.



This would now be a functional architecture since it decomposes the system's functionality into loosely coupled and highly cohesive packages. Another way to show this in UML is with an analysis package diagram shown on the next page.

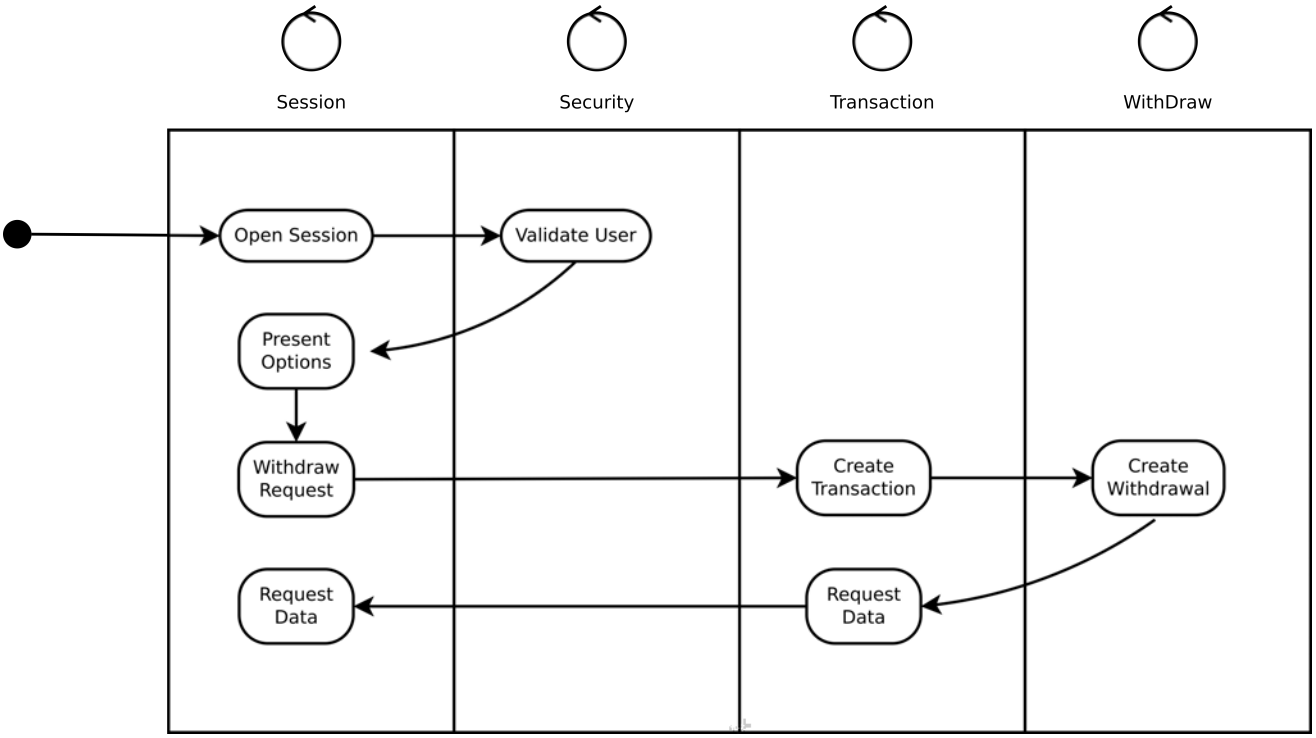
From a design point of view, as the development team starts to work on the transaction package for example, another team can start working on the session package. This same sort of process can be followed to decide what design classes are needed, what their responsibilities should be and what their inter-dependencies should be.


Session Package
ATM Package
«service» Security package
«service» Account Package
Transaction Package
Network Package

4 Logic Validation with Swimlanes

A swimlane is a column that overlays an activity diagram. The swimlane represents some analysis entity. If the activity lies in within an entity's swimlane, then that entity is responsible for the activity. A set of swimlanes partitions an activity diagram so that each activity must lie totally within a swimlane, although transition may cross swimlane boundaries.

Swimlanes are used to verify and validate an allocation of activities performed in a scenario to a collection of classes or entities. Remember that a system is a collection of objects working together to accomplish a goal. The process of getting to that goal can be described in an activity diagram. The swimlanes help us associate specific tasks with specific objects.



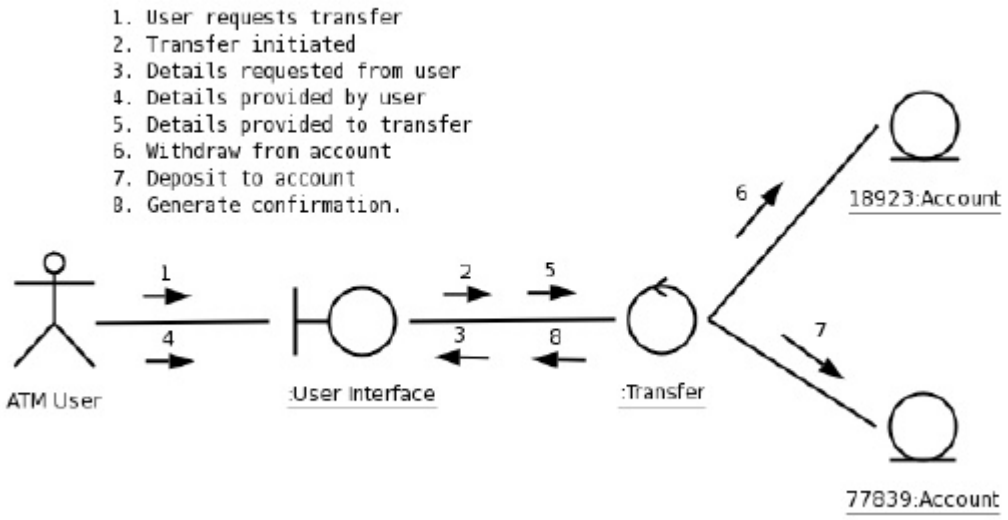
5 Communications Diagrams

The analysis objects we have proposed in our analysis class diagram “collaborate” to accomplish the goals of the systems as defined in the use case model.

These specific collaborations are diagrammed out in communication diagrams. A collaboration takes place when a use case or business process executes. The objects involved in the collaboration have to work together by exchanging specific messages in a specific order in order to provide the required functionality.

The communication diagram is an object diagram because the stereotypes in the communication diagram stand for objects, not classes. The communication diagram is intended to depict how a specific set of objects work together, not how classes are related. Logically, it is impossible for classes to collaborate, only objects can collaborate.

The communication diagram diagrams out what is happening as a scenario executes using instances of each of the classes associated with the scenario.



The diagram above shows the collaboration of the objects in the transfer use case. Specifically, we have two account objects involved in the transfer.

Each communication diagram corresponds to a scenario in a use case or one possible outcome. If the funds were not available in account 18923, then a different collaboration would have resulted with different messages. Generally the rule of thumb is to use communication diagrams to do a robustness analysis of a scenario, specifically:

1. Is all the functionality needed in the collaboration provided by some object or are we missing functionality?

2. Do the objects specified account for all the alternatives required in the scenario?
3. Does each object have clear and contributory responsibilities that are part of this scenario?

The lines connecting the object are links. They show the paths over which messages can travel from object to object. We want to show the flow of information from object to object as the scenario progresses, but we don't want to get too detailed too soon because the high-level architecture may change. Changes to the analysis model will probably result in our communication diagram changing as well. When we create our communication diagram, we should always be very careful that out objects and links actually correspond to classes and associations in the analysis class diagram.

5.1 Communication versus Sequence diagrams

There is a trend in UML books and courses to avoid communication diagrams and to jump right into sequence diagrams. Communication diagrams and sequence diagrams are both classified in UML and interaction diagrams and they do show essentially the same thing, but each diagram shows an interaction from a different perspective.

From The UML Users Guide:

1. An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them.
2. A sequence diagram emphasized the time ordering of the messages. A communication diagram emphasizes the structural organization of the objects that send and receive messages.

In the analysis workflow, we are still sorting out which objects have to collaborate and what information they have to share to accomplish a use case, so a communication diagram is more useful.

Later on in the design workflow when the interfaces of the objects have been stabilized, the sequence diagram is more appropriate because it focuses on the form and sequence of the messages.

In small scale systems where the amount of functionality and detail is quite small, it is often possible to do analysis informally, sort of on the fly, and not use analysis class diagrams and communications diagrams.

6 Domain Modeling

So why do domain modeling at all? As a practical and effective designer, I should be asking that question about any technique or methodology before I adopt it – the technical version of "but what's in it for me?" In this section, we will be looking at domain modeling from the perspective of a technique called Domain Driven Design, which is an excellent starting point for looking at real-world issues in the development of domain models. Domain Driven Design is not a complete modeling methodology but it not does pretend to be. It was developed by Eric Evans to address one particular problematic aspect of software development that occurs in large scale projects – managing complexity.

We will also be looking at other techniques as well that address the aspects of domain modeling that Evans does not address in Domain Driven Design, as well as the improvements in techniques since the introduction of Domain Driven Design in 2005.

6.1 Mission-Critical and Industrial-Strength Software

Software is said to be mission-critical when an organization has become so dependent on the software that, if that software fails, the operations of the business also fail or are seriously degraded.

Sometimes these failures are very public and news worthy – like the meltdown of the Delta Airlines software in August 2016 – but sometimes the public never really is aware of the impact of these failures, like the September 1983 software failure that brought the world to literally seconds from a global nuclear war. The important feature about mission critical software that concerns us, especially if we build it, is that we don't want it to fail.

The problem facing developers was pointed out by Grady Booch in 1994 that when software becomes mission-critical it has often grown to a level of complexity that makes it what he calls industrial-strength software.

> *The distinguishing characteristic of industrial-strength software is that it is intensely difficult, if not impossible, for the individual developer to comprehend all of the subtleties of its design. Stated in blunt terms, the complexity of such systems exceeds the human intellectual capacity. Alas, this complexity we speak of seems to be an essential property of all large software systems. By 'essential' we mean that we may master this complexity, but we can never make it go away.*

Over twenty years ago software engineers already had identified this trend to large, complex, mission-critical, industrial-strength software as a challenge that software developers would have to deal with, whether they want to or not.

6.2 Complexity and Failure

Just to be clear, there are two kinds of failures that we are lumping together here:

1. **Operational Failures:** These occur when an existing mission-critical system fails and has a negative impact on an organization or the larger community.
2. **Project Failures:** These occur when some sort of IT automation project is never completed and abandoned, or is sort of completed but the final result never really works like it should have.

The relationship between complexity and both kinds of failures has been established by a number of researchers and authors. For example, Roger Sessions points out that the rate of IT failures is :

> *The United States is losing almost as much money per year to IT failure as it did to the [2008] financial meltdown. However, the financial meltdown was presumably a one-time affair. The cost of IT failure is paid year after year, with no end in sight. These numbers are bad enough, but the news gets worse. According to the 2009 U.S Budget [02], the failure rate is increasing at the rate of around 15% per year.*

Sessions claims that the primary cause of these IT failures is complexity.

> *Is there a primary cause of these IT failures? If so, what is it?…. The almost certain culprit is complexity…. Complexity seems to track nicely to system failure. The more complex a system is, the harder and more costly it is to work on that system. And while complexity can correlate with functionality, there are many examples of highly functional systems that are organized much simpler than other systems with much less functionality. Empirically, we have all experienced that the difficulty of maintaining a system is much more related to how functions are organized than to the number of functions.*
>
> *Once we understand how complex some of our systems are, we understand why they have such high failure rates. We are not good at designing highly complex systems. That is the bad news. But we are very good at architecting simple systems. So all we need is a process for making the systems simple in the first place.*

Other researchers have also implicated complexity as a root cause of IT project failures (such as the Whitney and Daniels 2013 paper which is appropriately titled *"The Root Cause of Failure in Complex IT Projects: Complexity Itself"*).

In fact this problem with building complex systems is not just confined to software but any attempt to create complex systems. Attempts to build complex systems in one go seems to inevitably wind up in project failures, even when we try to emulate a naturally occurring complex systems such as those found in biological systems. One insightful observation from Gall is:

> *A complex system that works in invariably found to have evolved from a simple system that works… A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system.*

This is the problem that Domain Driven Design tries to address, how to manage complexity in a software project. The goal of Domain Driven Design is not to manage all kinds of complexity but to focus on a specific subset of what makes systems and software complex in order to provide a partial answer to how we "master this complexity."

6.2.1 Where Does Complexity Come from?

In a very insightful observation, Dijkstra noted that:

> *While we all know that un-mastered complexity is at the root of the misery, we do not know what degree of simplicity can be obtained, nor to what extent the intrinsic complexity of the whole design has to show up in the interfaces. We simply do not know yet the limits of disentanglement. We do not know yet whether intrinsic intricacy can be distinguished from accidental intricacy.*

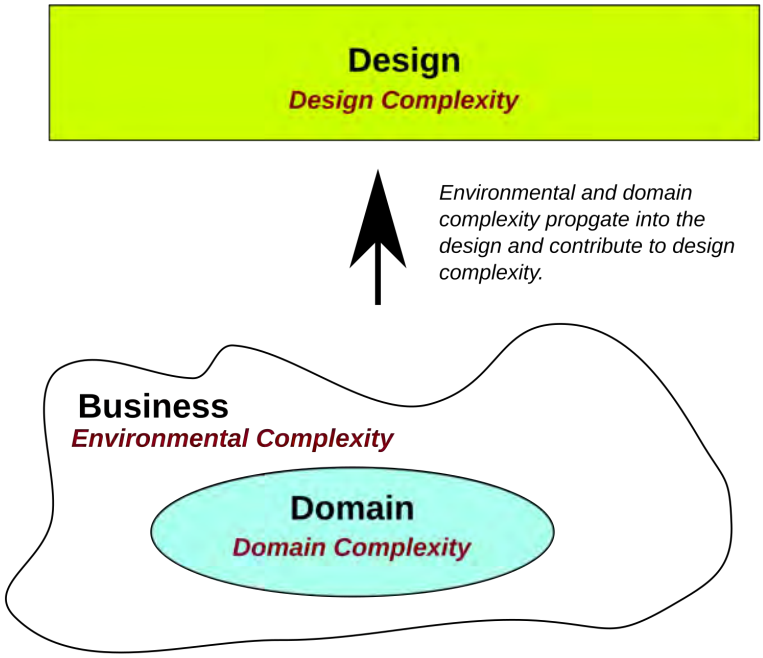
He quite rightly observes that there are two sources of complexity in software which he calls implicit and accidental. For the purpose of this course, we will call these design complexity and domain complexity, but I am also going to add a third source of complexity which I will call environmental complexity. These three types of complexity can be defined in the following way:

1. **Design (accidental) complexity**: The system is complex because we have designed a complex solution. In this case, the complexity is manufactured by us as a result of design, code and other implementation decisions we have made in building the software.
2. **Domain (intrinsic) complexity:** Sometimes the domain we are automating is itself very complex which means that no design decisions we make can reduce the complexity of the system since they do not impact the complexity of the domain. The complexity of the software is a reflection of the complexity of the underlying domain.
3. **Environmental (incidental) complexity**: In some cases the domain subject matter itself is not complex but the way the organization deals with the domain subject matter is chaotic, inefficient and ineffective. The complexity we see is a result of the ineptness of the organization which creates a kind of conceptual fog that makes it difficult to see directly how the underlying domain works.

These sources of complexity are not mutually exclusive which means that we usually need to employ a variety of methodologies and practices to deal with the different kinds of complexity.

6.2.2 Propagation of Complexity

The diagram following shows that the overall complexity that we deal with in a design has contributions from each of the three sources. The design complexity partially comes from trying to build software that manages both the complexity of the underlying domain subject plus any layers of complexity imposed by the attempts to understand the subject through the filter of the organizational chaos and complexity. However, there is also design complexity caused by making poor design choices that lead to unnecessarily complex solutions.



In this course, we will not be concerned with complexity that has its roots in pure design decisions because that topic is already well discussed in the software development community in conversations like the role of heavy-weight versus light-weight solutions. That topic is definitely out of scope for this course.

What we are focused on in this course is reducing the propagation of complexity from the domain and environment into the design.

6.2.3 Environmental Complexity

Domain Driven Design does deal tangentially with some of the environmental complexity issues but only when they impact on the domain modeling process. There already are a number of other techniques and methodologies that are well developed to deal with a range of environmental complexity issues. These are often not technical methodologies but more organizational oriented like developing process maturity, implementing Agile development process, effective requirements management techniques and many others, some of which

we deal with in other parts of the course.

The only reason that we deal with any of these environmental factors in domain modeling is that they may present barriers to our modeling the domain, but we only will deal with them in that context. If we are in an Agile environment, we generally assume that domain modeling is being deployed in conjunction with other techniques, like Acceptance Test Driven Development, which deals with these environmental and organizational issues.

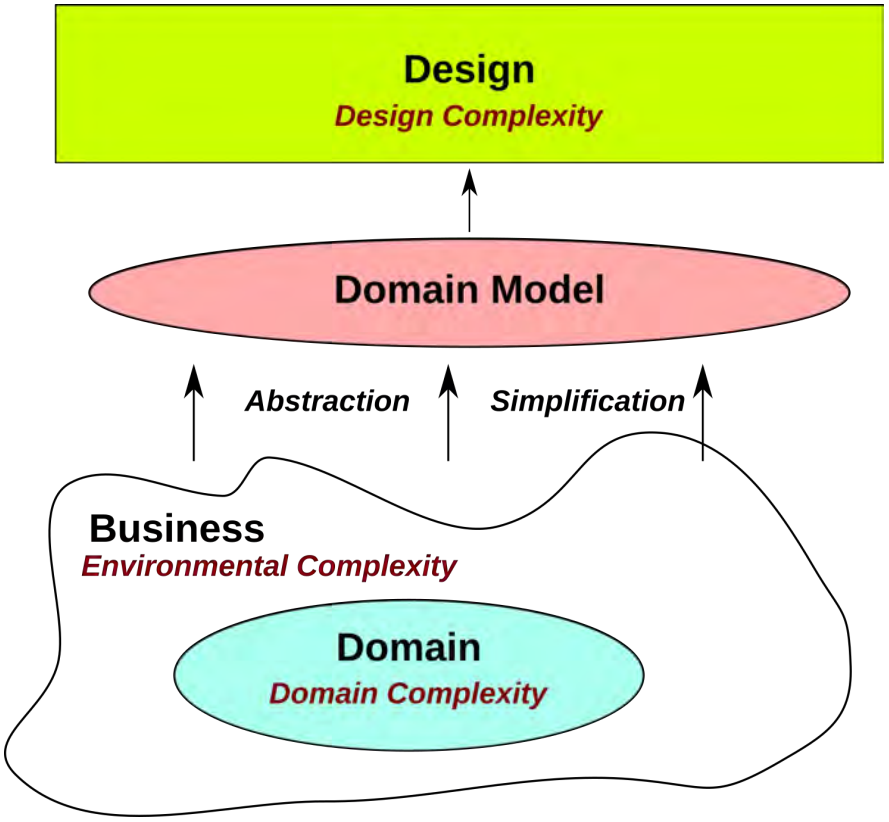
Some of these sources of environmental complexity that may overlap with our modeling listed below. This is not a definitive list but just intended to give you a sense of what we mean by environmental complexity.

1. *Poorly documented processes.* A lot of the domain logic and concepts may exist only as idiosyncratic knowledge and experience in the minds of some people in the organization. We can study the documentation, policies and other business artifacts, if they exist, but without accessing the real understanding of how things are actually done by talking with those experts who know, we have no idea what is actually happening in the domain.
2. *The documented business is not the real business*. In many cases the "official" way do doing things is not the real way things are done, usually because doing things the official way doesn't work – so people do what works instead. This is something we only become aware of as we start to explore how things actually work with the domain experts.
3. *The organization may be complex.* This might be called the bureaucracy problem. The domain may be managed in a complex manner with the imposition of organizational issues, reporting requirements, overlapping departmental responsibilities, excessive oversight and other institutional artifacts. Often once we get to the underlying domain itself, we discover that it is a lot simpler that it initially appeared and what appeared to be domain complexity was in fact administrative complexity.
4. *The business is disorganized.* What may be relatively simple domain subjects are dealt with or processed in a chaotic manner. Work may be allocated poorly, records not maintained properly, tasks performed sloppily or incorrectly requiring a lot of rework, conflicting or contradictory policies or processes may make it unclear exactly what is going on in the domain. Essentially, we have a simple underlying domain and a badly managed environment.
5. *There is no single domain*. The domain may be fragmented among departments or groups each of which interacts with domain in their own way resulting in a patchwork of processes and business logic, terminology and concepts – especially if there is no co-ordination between the different sub-groups.

6.3 Domain Driven Design

Evans bases the idea of Domain Driven Design on the following two axioms which are now considered to be best practices:

1. For most software projects, the primary focus should be on the domain and domain logic.
2. Complex domain designs should be based on a model.



The domain driven design solution can be depicted in the following way:

As Eric Evans puts it:

> *Many things can put a project off course: bureaucracy, unclear objectives, and lack of resources, to name a few. But it is the approach to design that largely determines how complex software can become. When complexity gets out of hand, developers can no longer understand the software well enough to change or extend it easily and safely. On the other hand, a good design can create opportunities to exploit those complex features.*

Software models the real world, which means that when we design software, we extract from the domain what is relevant for our application so that we can build a conceptual model on which we can base our design and code.

However the more complicated the domain becomes, the more and more difficult it gets to work with a simple and intuitive model. Remember the discussion on industrial-strength software earlier in this module.

Suppose that we are automating the US Tax Code with its almost 100,000 pages of rules and regulations that form a tangled web of logic that those in the know say is almost totally non-understandable. If we were to try to automate the US Tax Code, there is nothing we can do at the design level that would simplify the complexity of the resulting application. This is domain complexity – it is inherent in the Tax Code itself, not in our software design.

Domain driven design is a process of extracting a simplified domain model from a complex domain to allow us to avoid having the complexity of the domain propagating into the design. In one sense the domain model filters out the complexity of the domain or at least transforms it into a model that is simpler to work with in the design process.

However his domain model has to have two critical properties or this whole activity is pointless:

1. The simplified model has to reduce the complexity in the domain model so that we can work towards an elegant design. If the model does not simplify, then we have gained nothing in the design process and just wasted a lot of time doing pointless modeling.
2. The simplified model must contain the domain critical information we need to build the application correctly. There is no point in having a simplified model of the domain if much of the important information about the domain that we need has been discarded during the simplification process.

6.3.1 Domain Complexity Creates Design Complexity

Software models the real world, which means that we extract from the domain what is relevant for our application to create a conceptual model on which we base our design and code. For simple domains, the model that we use is usually obvious and intuitive, both properties make it easy to use in making good design decisions.

However the more complicated the domain becomes, the more and more difficult it gets to work with a simple and intuitive model. As we can see in the figure above, increasing complexity in the domain drives the design to become more and more complex until our software starts to become brittle and ineffective.

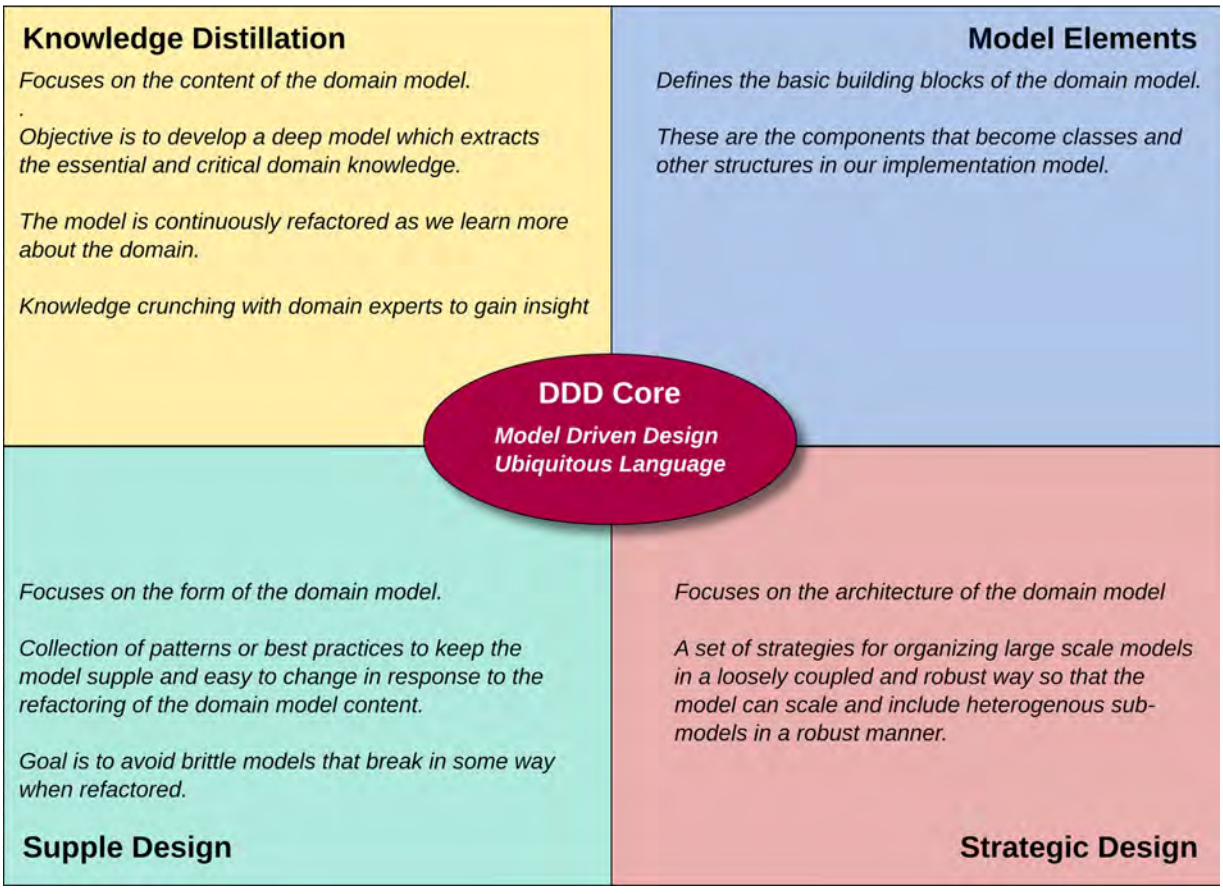
Evans describes domain driven design in the following context:

> *Some design factors are technological. A great deal of effort has gone into the design of networks, databases, and other technical dimensions of software. Many books have been written about how to solve these problems. Legions of developers have cultivated their skills and followed each technical advancement.*
>
> *Yet the most significant complexity of many applications is not technical. It is in the domain itself, the activity or business of the user. When this domain complexity is not handled in the design, it won't matter that the infrastructural technology is well conceived. A successful design must systematically deal with this central aspect of the software.*
>
> *Domain-driven design is both a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains… [it] presents an extensive set of design practices, techniques, and principles.*

6.4 The Domain Driven Design Quadrants

The diagram below is a summary of the “ways of thinking” that make up domain driven design

Domain Driven Design Components



6.4.1 Knowledge Distillation

Evans refers to this as knowledge crunching. It is the driver of the domain driven design process for three basic reasons, although there are lots of other reasons in addition to these three fundamental ones.

First, good domain models emerge through continuous refactoring as our understanding of the domain improves in an iterative manner. We are not going to take a one-time look at a complex domain, understand it all, and then churn out an elegant domain model. Our brains just don't work that way.

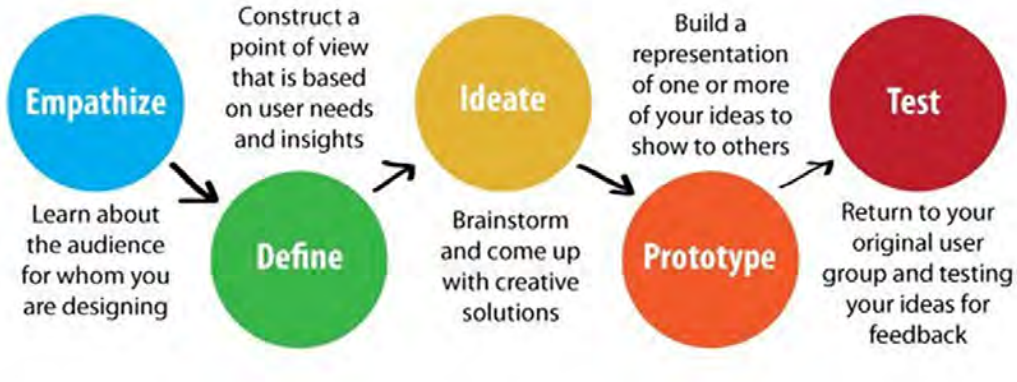
Our brains work most effectively after we immerse ourselves in a domain and cram information into our minds, and then go and focus on something else for a while. This period of inattention allows our brain to go about organizing that knowledge to produce insights and deeper understanding. Then when we go back to the domain, we now understand it at a deeper level based on these new insights.

As this cycle is repeated, we eventually hit a critical iteration, which Evans calls the breakthrough iteration but others have called the "eureka" effect or the "ah-ha" moment. We have all experienced those breakthroughs when we work hard on a problem with no obvious solution, and later when we are doing something totally unrelated, a solution just pops into our mind.

However, we now understand from a cognitive point of view what drives these eureka moments. It involves being stuck on a problem after you immerse yourself in the problem for an extended period of time, then relaxing your mind and focusing on something else. Most scientific breakthroughs have occurred this way from Einstein's insight on special relativity to Linus Pauling's breakthrough conceptualization of protein structures and folding.

In fact, this process has been formally organized into something called the Stanford Design Process which has been adopted for use at Capital One. Knowledge crunching process is totally consistent with this kind of design process.

The diagram below is a typical depiction of the Stanford Design Process, and it is also an iterative process.



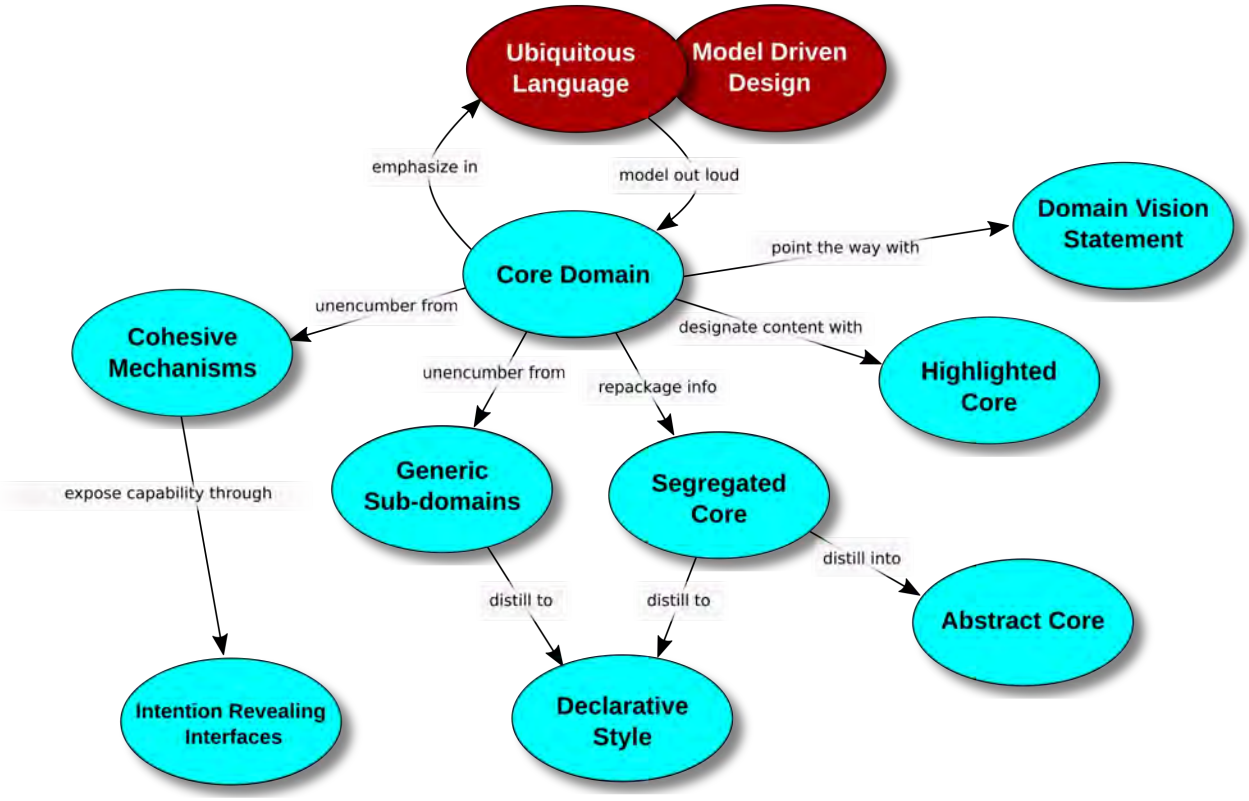
Second, getting a deep insight into the domain requires a close and regular collaboration between the developers and the domain experts. This collaboration has to be iterative to be effective because as we understand more about the domain, then we understand what sort of questions need to be asked, or where we need to drill down into the domain – things that we would not been aware of earlier in the process.

This dialog is iterative by necessity because it usually has to fit into an iterative development process to accommodate the reality of scheduling collaborations. Each time we collaborate, our latest insights into the domain allow the next collaboration to be more focused and productive.

Third, in each iteration we can see how the domain model would suggest a design. With a general idea of what our solution might look like, we can go back and explore the issues in the domain that would affect or impact the putative design we are considering. The design starts to emerge in a natural way from the domain model. We can also identify designs that would be problematic early on in the development process. We can avoid those alternatives and the inevitable design rework later.

However, along with this general process are a number of strategies to help distill this knowledge – specific techniques that can be employed to make the knowledge crunching process more effective. These are depicted in the diagram below.

Strategic Distillation

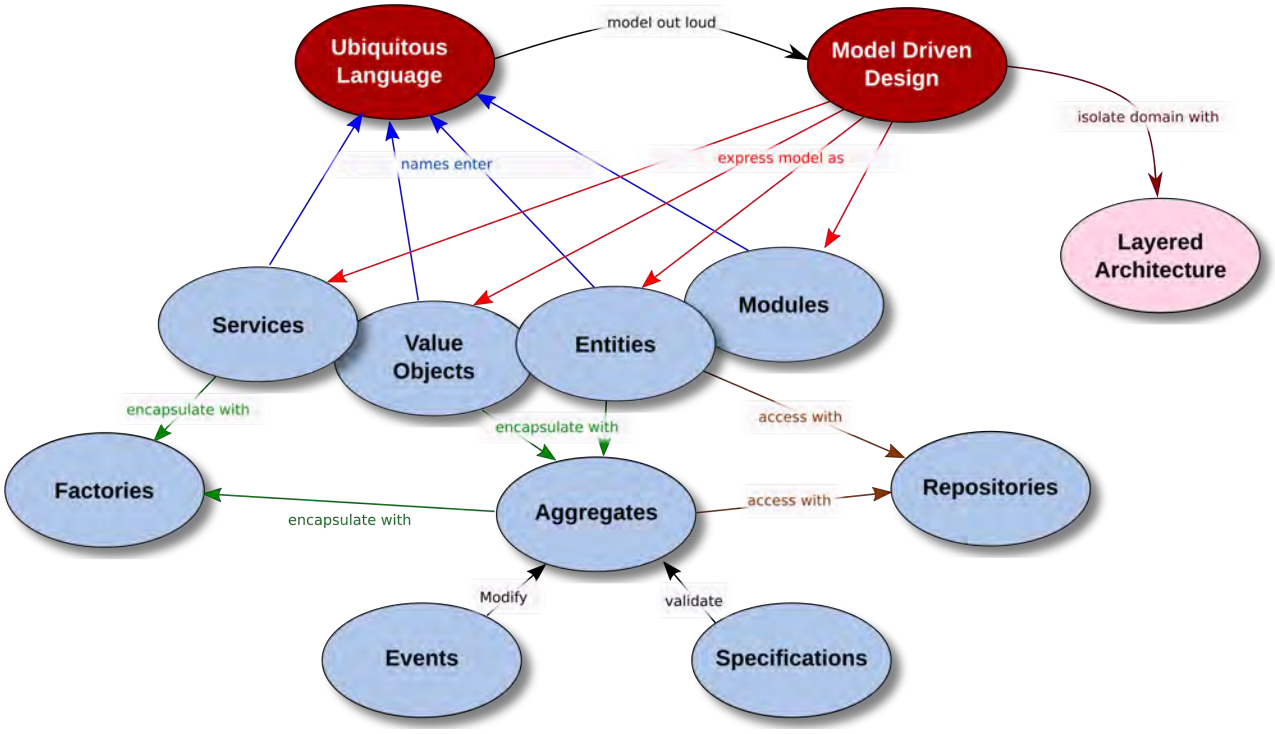


6.4.2 Model Elements

Domain driven design modeling is based on well-known modeling best practices and uses OO modeling concepts as a starting point. The modeling elements used in domain driven design are derived from the traditional OO modeling elements, but are just a bit more specialized. These specializations refine the more general OO artifacts into something more concrete in order to make domain modeling easier. This is not a theoretical divergence from OO modeling, but is rather an extension of it. Evans states a number of times that domain driven design models must meet the criteria of good OO models.

The model elements in domain driven design are depicted below..

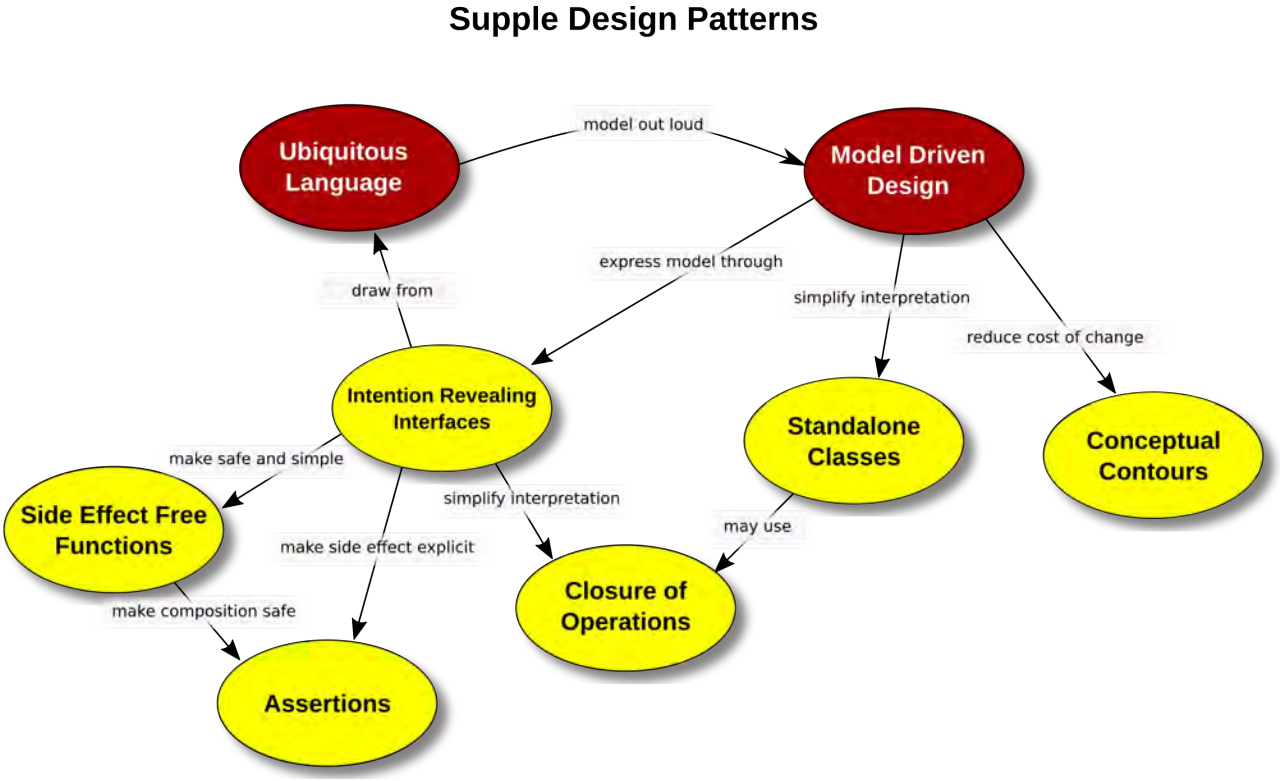
Fundamental Model Building Blocks



6.4.3 Supple Design

The core of the domain driven design process is the continuous refactoring of the domain model as our understanding of the domain improves. The supple design practices are ways to keep the model supple – which means that model is assembled in a way that is robust, flexible and easy to change and modify. These supple design practices are essentially modeling best practices, including OO best practices, which are implemented in the context of working with a domain model and with the domain driven design model elements.

These are depicted in the diagram below.



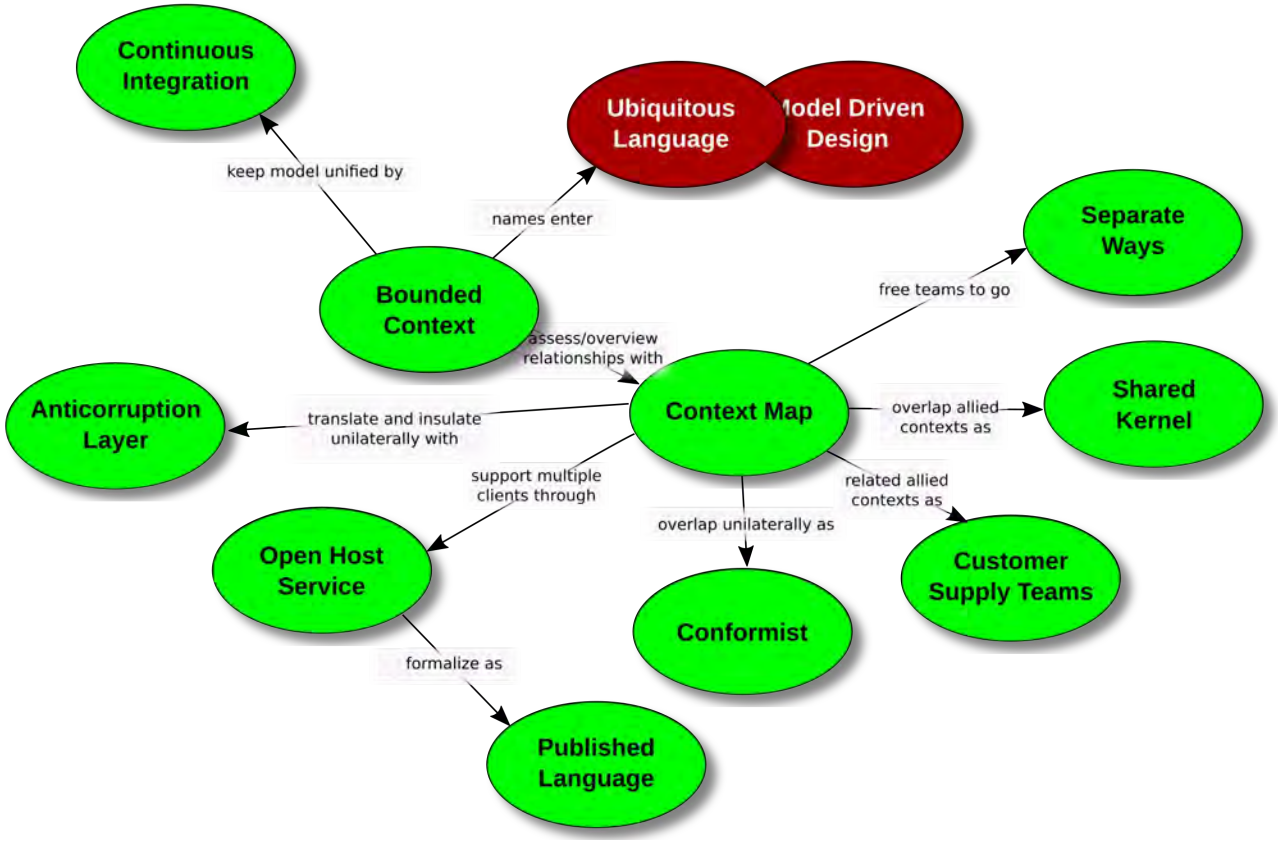
6.4.4 Strategic patterns

Where the supple design strategies create a robust local domain model in terms of the elements and relationships between the elements, the strategic design works at a higher level to create a supple architecture. A domain model is rarely monolithic – it is often a combination of sub-domains, where each sub-domain may have its own unique local domain model. We have to integrate these at a higher level so that they all work together and in a way that allows for structural changes in any of the local domains to be incorporated seamlessly into the overall macro model of the domain.

This sort of heterogeneous structure is fairly common in a lot of organizations: the domain models in the accounting department are different that than in operations which is different than those in marketing – but the whole enterprise has to work in a unified and supple way. That is where strategic design comes into play.

The strategic patterns are depicted in the diagram below.

Strategic Design



6.5 Synergies with the Business

As we go through the domain driven design process, it's a common experience to find places where the existing domain is broken in some way – dysfunctional processes, poor data organization, faulty communications channels and transmission of information. One of the strategic goals of our project should be to not just automate the existing domain processes and artifacts, especially if they are broken or inefficient, but to allow an opportunity for the business and developers to fix some of those problems in the new design going forward by creating an improved "domain model to be implemented.".

During this process we often have opportunities where a domain expert might say something like "It would be so nice if we could just have this information automatically sent to these other departments, the process we go through now is so archaic and time-consuming it really is a pain. We are still doing this activity the way we did it in 1950 but we just don't have any other way to do it, perhaps we can finally fix that." Now we have an opportunity to actually do that.

The process of developing the domain model in conjunction with the domain experts allows us to develop a model of the domain “as it should be” in their expert opinion rather than as it is now. Then our automation project implements a better version of the existing business, not one with all the current problems, issues and limitations blindly copied into our design.

The domain model we are developing is not intended to be a snapshot of what is because it is the basis for the design of the automation. It is more like a refactoring of the domain – generally we are automating to produce the same outputs that the current domain does as currently exists and those that planned to exist in the future – a step forward that produces a return on investment for the business. We are not producing a slavish automated version of the cruft and accumulated historical artifacts they are currently working with in the existing domain.

When the business domain experts see how this kind of modeling can benefit them, they generally become enthusiastic participants because they now see value in being able to contribute to making the project finally do what they think the business should have been doing all along..

7 Domain Specific Language

Now we take a bit of a detour to talk about domain specific languages. This section will have a lot of background not covered in class, but I have not found any good source out there to refer anyone to, so I compiled one here. This is one of those sections you can read on a Saturday night for something to do, when there is nothing good on TV and your Internet is down.

Behaviour Driven Development incorporated the idea of a ubiquitous language from Eric Evans Domain Driven Design. Evans identified the cause of many problems in developing systems was a mismatch in terminology between the business facing and the technology facing sides of teams. Other design problems could be traced back to the same issues that we have identified in this section as communications issues – or in simpler terms, we don't all speak the same language.

His solution was to suggest a designed vocabulary of terms that precisely articulated exactly what was meant by terms that related to key concepts and ideas in domain, and whose defined meanings would be valid only within the scope of the project.

This concept of a defined vocabulary is not restricted to domain driven design but does appear in a wide range of disciplines and industries although the names for this technique are quite varied. For example in law, we speak of legal definitions and in the social sciences we often speak of operational definitions.

In general, we can think of a domain specific language as a kind of jargon that meets several important criteria, which will explore in a moment.

7.1 Reasons for using Ubiquitous Language

Miscommunication, semantic differences, misunderstanding, vagueness and the other sorts of pitfalls of everyday communication are the source of many errors that cause projects to go off the rails.

By using a correctly defined domain specific language, we are able to eliminate many of the errors that arise from using natural language directly in our design, but not all. A DSL is not a magic solution for communication errors, just a useful tool. Some of the problems in natural language are described in the material that follows:

7.1.1 Shibboleths

In linguistics a shibboleth is a word or phrase that is used to identify different groups based on how the two groups differ in the meanings they assign to the shibboleth.. For example, the term “pudding” is a shibboleth because in North America it refers to a specific kind of food while in the United Kingdom the term refers to what North Americans call “dessert.”

I first became aware of this problem when I was listening to two of my colleagues discussing the design for a legal case management system.

> Brit: At this point we need to identify the current location and status of each homicide.
>
> *Yank: Ummm we already know that. They are in the morgue.*
>
> Brit: Why would they be there?
>
> *Yank: Because they are homicides. Where else would they be?*
>
> Brit: In a jail cell. (ed. perhaps I should have said gaol cell?)
>
> *Yank: Why would you keep a corpse in a cell?*
>
> Brit: Why are you talking about corpses?

The problem here was that the Englishman was using the term homicide to mean a person who commits a murder while the American was using the term homicide to mean the victim of a murder. They then proceeded to waste an entire afternoon arguing over which one was the “right” definition.

7.1.2 Polysemous Ambiguity

Ambiguity occurs when there are multiple definitions for a word or expression and different people don’t apply the same definition to a single use of the word in some context. The technical term for different meanings for the same world is polysemy.

A number of years ago I was asked to consult on a database problem. Two different analysts for a government agency were coming up with different results when they tried to query the number of jobs at that agency. The problem stemmed from one analyst assuming that jobs meant the number of people on the payroll while the other analyst assumed that jobs meant the number of positions on the organizational chart. Since some people were sharing a single position and some positions were not filled, as well as having a few people filling several positions, it was no wonder the numbers didn't match.

This sort of problem is more likely to occur when the polysemous definitions can be confused in context, like the term jobs in the preceding example. When this confusion is unlikely to occur because the definitions are dissimilar (like pen meaning both a writing implement and an enclosure) we tend to call them homonyms.

To close off with an example, if someone asked you to bring a light, it would only be in context that you could determine if they meant something to provide illumination or something to ignite a fire.

7.1.3 Specialized Meaning

There are many words that have a general meaning, like “credit”, and also have a specialized meaning withing a specific domain, such as “credit” in a University setting which refers to a course taken, or “credit” in an accounting setting which refers to a particular type of financial transaction.

Problems start to arise when the business side assumes the specialized meaning by default and the technical side assumes the general meaning instead.

For example, a number of years ago someone broke into my car and stole a few items. I reported the incident to the police in a conversation someone like this:

> Me: Hi, I've been robbed.
>
> *Police: Can you describe the person who robbed you?*
>
> Me: Umm no, when I went to get in to my car this morning and the driver's window was smashed and a bunch of stuff had been taken.
>
> *Police: That is not a robbery.*
>
> Me: But I was robbed.
>
> *Police: No, you weren't. That's a theft. A robbery is when someone accosts you.*
>
> Me: So I was thefted?

This dialogue is a simple case of the police using the specialized meaning encoded in police jargon and me using the more general use of the term robbery.

However, specialized meaning occurs in a number of other ways. One of the more common forms is something called metonomy where something is not referred to by its usual name but by the name of something associated with it. If one does not understand the association, then using the default meanings can produce problems.

For example: Washington announced today new rules governing the relationship between Wall Street and Hollywood.

This one sentence has three uses of metonomy that is obvious to anyone who understands the references. Specifically that Washington refers to the US federal government, Wall Street refers to the US financial services industry and Hollywood refers to the US film industry.

Another specialized meaning form is called synecdoche, where a specific part of something is used to refer to the whole, or the whole stand is for a specific part. The following are a couple of examples:

> *A lot of hungry mouths to feed.* Mouth is a part of the whole person who is hungry,
>
> *America celebrates!* The people of America celebrate, the whole stands for the part.

7.1.4 Meaning Clusters

Often a single word represents a cluster of related meanings. The word commonly represents a sort of ideal prototypical case but also can cover cases that do not fit the ideal case where the special meaning is clear from context. George Lakoff provides an excellent example of

this with the term mother. What makes “mother” a complex trems is that different groups of people may have different ideal prototypical definitions.

In the 1950s version of the ideal mother was someone who is female, who gave birth to the child, supplied her half of the child's genes, nurtures the child, is married to the father of the child, is one generation older than the child, is the child's legal guardian and does not work outside the home.

Deviations from this ideal are generally qualified in some way to indicate the whole definition does not or cannot apply. The terms working mother and single mother indicate some deficiency in how the person meets the ideal definition.

In fact, we can identify each of parts of that cluster of meanings with qualified versions of the term mother:

> **Genetic mother:** The individual who contributes genes to the zygote that becomes the child.
>
> **Biological mother:** The person who contributes the egg that becomes the zygote – with genetic engineering, the biological mother does not have to be the same person as the genetic mother.
>
> **Birth mother:** The person who hosted the fetus and gave physical birth to the child.
>
> **Stepmother:** The woman married to the child's father and is not the genetic mother.
>
> **Adoptive mother:** The woman who is the child's legal guardian.
>
> **Family mother:** The closest female ancestor (not common in western cultures).
>
> **Nurturing mother:** The woman who provides care for the child.

It would be possible to say “This is the woman who I got my genes from, but this is the woman I got my mitochondrial DNA from, and this is the woman who gave birth to me, and that is the woman who is married to my father, but this is the woman who raised me although that woman adopted me but this is the woman I call mommy.“ – and each one is a different person.

So why care? Because in specific domain contexts, if specialized meaning of mother is used without qualification which can create confusion. For example, when working in a genetic disease clinic, a person's mother clearly refers to the genetic mother while in juvenile court, it would refer to the person who is the legal guardian.

7.2 Defining Ubiquitous Terms in Other Domains

We do not define all the terms in the domain, only the core terms that are both potential sources of error and critical to understanding the domain concepts. Fortunately, we have a number of examples of ubiquitous language in other disciplines that face similar sorts of issues which provide us with some techniques that we can borrow or steal.

7.2.1 Legal Definitions

At the start of each well written piece of legislation is a section that defines the terms used in that act or statute. For example, at the start of the Ontario Landlord Tenet act is the following:

> *2. (1) In this Act,*
>
> *"Board" means the Landlord and Tenant Board;*
>
> *"care home" means a residential complex that is occupied or intended to be occupied by persons for the purpose of receiving care services, whether or not receiving the services is the primary purpose of the occupancy;*
>
> *"care services" means, subject to the regulations, health care services, rehabilitative or therapeutic services or services that provide assistance with the activities of daily living;*
>
> *"guideline", when used with respect to the charging of rent, means the guideline determined under section 120;*
>
> *"land lease community" means the land on which one or more occupied land lease homes are situated and includes the rental units and the land, structures, services and facilities of which the landlord retains possession and that are intended for the common use and enjoyment of the tenants of the landlord;*
>
> *"land lease home" means a dwelling, other than a mobile home, that is a permanent structure where the owner of the dwelling leases the land used or intended for use as the site for the dwelling;*

One of the more interesting demonstrations of the importance of a legal definition is the birther controversy. According to the relevant legislation, the President of the United States must be a natural-born citizen however this term is never definitively defined which has resulted in both sides of the controversy supplying their own definitions and arguing that their definition is what was actually intended.

7.2.1 Operational Definitions

The problem of ambiguity is one that many different industries and disciplines face. For example, if we want to measure intelligence, then the first challenge is to define what exactly intelligence means. Similarly, if we ask someone to build a flat metal plate, then we need to determine exactly what flat means.

One of the founders of the modern quality movement, W. Edwards Demming, stressed the importance of using quantifiable definitions or operational definitions. He defined these as “a procedure agreed upon for translation of a concept into measurement of some kind.”

For example, when we discuss the hardness of a material, there is no inherent sense of what that means quantitatively so we use an operational definition, which is the Mohs scale for scratch harness; but the Mohs scale is only one of several operational definitions, for example another is the Brinell indentation hardness test.

7.3 Properties of a Domain Specific Language

The domain specific language is the precise definition of key terms, concepts, processes and data. Not every term in a domain needs a DSL definition. In fact is we do too much and carried away with a DSL we actually wind up wasting a lot of time and resources. The trick is to develop the right amount of DSL.

A DSL definition has the following four basic properties.

1. **It is Limited in Scop**e. The definition is intended to be used only within the scope of a particular domain so a critical part of the definition is a description of which contexts or domains in which the definition is intended to apply.
2. **It is Testable.** The property of testability is the ability to write the definition in terms of a pass or fail test. A more expedient way of thinking of this is Peter Chen's notion of defining the concept in terms of logical predicate which always evaluates to true or false when applied to any potential object that could be in the domain.
3. **It is Verifiable.** It is one thing to have a test, but it is another matter to be able to apply the test. Something is verifiable when there is a cost-effective, finite process for evaluating the predicate. This is operational component of the DSL definition.
4. **It is Valid.** Validity here means that the various stakeholders all can read can understand the definition in a way that makes sense to them. This means that the terms defined in a DSL have obvious logical and semantic relations to the terms as used in the domain.

Consider for example the concept of being a US citizen. The average person has a general and intuitive idea of what being a citizen means but that definition is fuzzy. They may say a person is a citizen if they are born in the US or are naturalized. But that definition does not cover children of foreign government officials who are posted in the US when the child was born, nor does it cover children of US citizens who were posted by the US government overseas at the time of the birth. While the average person has a general idea of what a US citizen is, there is also a lot of variation in the individual definitions that different people hold.

We see this in statements like “I don't care if he was born here, he is not a real American.” If pressed for a definition of what being a US citizen means, many people would not be able to describe what they mean by US citizen, and among those that could, there would be a range of probably contradictory definitions. (“Someone who speaks English, loves the flag and baseball and has a gun.”)

However, the US Border Services requires a more useful and impartial definition of US Citizen to determine whether or not someone who shows up at US Customs wanting entry to the county is a citizen or not. We could allow each Customs Officer make a judgment call or just believe whatever the person tells them, but that obviously would be unworkable.

Instead, the Customs officer uses a DSL like criteria. This consists of two main parts.

1. **A testable predicate**. A person is a US citizen if they meet the criteria of the US legislation defining citizenship. The problem is that when someone shows up at the border, while we have a test that tells us whether or not a person is citizen, there is no way to apply that predicate to the person standing in front of us.
2. **A verification process.** The operation for testing the predicate is to ask to see a current US passport because a person cannot be issued a US passport unless they satisfy the citizenship predicate.

Bibliography

Booch94 **Object-Oriented Analysis and Design with Applications, 2nd Edition**; by Grady Booch; The Benjamin/Cummings Publishing Company Inc;Redwood City CA; 1994; ISBM 0-8053-5340-2

Booch99 **The Unified Modeling Language User Guide**; by Grady Booch, Ivar Jacobson and Jim Rumbaugh; Addison-Wesley; Upper Saddle River NJ; 1999; ISBN 0-201-57168-4

Booch99a **The Unified Software Development Process**; by Grady Booch, Ivar Jacobson and Jim Rumbaugh; Addison-Wesley; Upper Saddle River NJ; 1999; ISBN 0-201-57169-2

Gerbr01 **The E Myth Revisited**; by Michael E. Gerber; HarperCollins Publishers; New York NY; 2001; ISBN 0-88730-728-0

Heima98 **The New Strategic Selling**; by Stephen Heiman, Dianne Sanchez; Time-Warner Books; New York NY; 1998; ISBN 0-446-67346-3

Kuhn70 **The Structure of Scientific Revolutions, second edition;** by Thomas Kuhn; University of Chicago Press; Chicago IL; 1970; ISBN 0-226-45804-0

Lakof87 **Women, Fire, and Dangerous Things: What Categories Reveal About the Mind**; by George Lakoff; University of Chicago Press; Chicago; 1981; ISBN 0-226-46804-6

Leffl00 **Managing Software Requirements: A Unified Approach;** by Dean Leffingwell and Don Widrig; Addison-Wesley; Upper Sadle River NU; 2000; ISBN 0-201-61593-2

MIlle05 **The New Conceptual Selling: The Most Effective and Proven Method for Face-to-Face Sales Planning**; by Rober Miller, Stephen Heiman; Time-Warner Books; New York NY; 2005; ISBN 0-446-67449-4

Rumba91 **Object-Oriented Modeling and Design**; by James Rumbaugh, Micheal Blaha, William Premerlani, Frederick Eddy and William Lorensen; Prentice Hall; Englewood Cliffs, NJ; 1991; ISBN 0-13-629841-9

Shank77 **Scripts Plans Goals and Understanding: An Inquiry into Human Knowledge Structures;** by Roger Schank and Robert Abelson; Lawrence Erlbaum Associates; Hillsdale NJ; 1977l; ISBN 0-470-99033-3