



---

# The development process

## *A layman's workbook*

---

**Author:** Jon Barnett

**Document version:** 1.0

# Table of Contents

1-Introduction.....	1
2-Thinking about system development.....	2
2.1-What is a system?.....	2
2.2-Collecting requirements.....	3
2.3-Building models.....	4
2.4-Adaptive and agile development processes.....	4
2.5-Documentation.....	6
2.6-Construction tools.....	6
2.7-People.....	7
3-The process in action.....	8
3.1-Requirements.....	8
3.2-The system in context.....	10
3.3-Use cases.....	11
3.4-Analysis.....	14
3.4.1-Partitioning and architecture.....	14
3.4.2-Analysis patterns.....	15
3.4.3-Scope of representation.....	16
3.4.4-Use-case realizations.....	17
4-Pitfalls.....	18
4.1-Partitioning a system.....	18

# 1 Introduction

It is vital to have a development process. But the process is a means to an end. I quote the Agile Manifesto - "Working software is the primary measure of progress." So it is important to keep in mind that it does not matter what modelling paradigm you employ as long as it helps deliver software system that meets the task for which it has been designed and that the system continues to do so during the lifetime of the system.

With that in mind, I present one case study based on my experiences. I hope that it allows people to see the process of systems analysis and decomposition, and that the examples are not tied to a particular means of implementation. Purists may not like the approach as I borrow from a few methodologies and this results in a hybrid process. However, the notion I hope to promote is that you choose the appropriate tasks, the appropriate models and the appropriate tools to accomplish the job. Should the approach to decomposition and development prove to be very difficult, perhaps the model and its implicit analogy is inadequate for the particular task.

So even in studying the examples here, the reader should be aware that there is no single right way of creating a system. Individual designers organize their thoughts and concepts of a system in different ways. Models and processes are neither right or wrong.

## 2 Thinking about system development

It was six men of Indostan  
To learning much inclined,  
Who went to see the Elephant  
(Though all of them were blind),  
That each by observation  
Might satisfy his mind

The Third approached the animal,  
And happening to take  
The squirming trunk within his hands,  
Thus boldly up and spake:  
"I see," quoth he, "the Elephant  
Is very like a snake!"

The Sixth no sooner had begun  
About the beast to grope,  
Than, seizing on the swinging tail  
That fell within his scope,  
"I see," quoth he, "the Elephant  
Is very like a rope!"

The First approached the Elephant,  
And happening to fall  
Against his broad and sturdy side,  
At once began to bawl:  
"God bless me! but the Elephant  
Is very like a wall!"

The Fourth reached out an eager hand,  
And felt about the knee.  
"What most this wondrous beast is like  
Is mighty plain," quoth he;  
" 'Tis clear enough the Elephant  
Is very like a tree!"

And so these men of Indostan  
Disputed loud and long,  
Each in his own opinion  
Exceeding stiff and strong,  
Though each was partly in the right,  
And all were in the wrong!

The Second, feeling of the tusk,  
Cried, "Ho! what have we here  
So very round and smooth and sharp?  
To me 'tis mighty clear  
This wonder of an Elephant  
Is very like a spear!"

The Fifth, who chanced to touch the ear,  
Said: "E'en the blindest man  
Can tell what this resembles most;  
Deny the fact who can  
This marvel of an Elephant  
Is very like a fan!"

### 2.1 What is a system?

When thinking about systems, it is useful to describe the concept of a system. I like the definition by Blanchard and Fabrycky [BLA90], which I reproduce verbatim here:

Systems are composed of components, attributes and relationships. These are described as follows:

1. Components are the operating parts of a system consisting of input, process, and output. Each system component may assume a variety of values to describe a system state as set by control action and one or more restrictions.
2. Attributes are the properties or discernible manifestations of the components of a system. These attributes characterize the system.
3. Relationships are the links between components and attributes.

A system is a set of interrelated components working together toward some common objective. The set of components has the following properties:

1. The properties and behaviour of each component of the set has an effect on the properties and the behaviour of the set as a whole.
2. The properties and behaviour of each component of the set has an effect on the properties and behaviour of at least one other component in the set.
3. Each possible subset of components has the two properties listed above; the components cannot be divided into independent subsets.

A system is more than the sum of its components and has characteristics that cannot be reproduced by any of the subsets of its components. Components, themselves can be systems.

The purpose of a system can be defined and expressed succinctly. A system is an active component in a larger system that encompasses itself and the external systems with which it interacts.

## 2.2 Collecting requirements

The purpose of collecting system requirements is to form a basis for understanding the system that you are intending to develop. However, many factors can impact the requirements gathering process:

- Unfamiliarity with the domain by the requirements gathering team
- Inability of the domain experts to impart their knowledge in a timely fashion
- The breadth and depth (complexity) of the domain
- The inability of users and designers to envision the detailed use of the system and the detailed use of its outputs

In order to facilitate a transfer of knowledge it often helps to have a definition of the areas for which you want requirements but then to conduct an unstructured exploration of the domain with the domain experts. Many people know this as a discovery workshop. This approach is vigorous and is usually more appropriate for eliciting information from domain experts, since their knowledge is often organised in an associative way. Having a group of domain experts participating in the exercise can help increase the speed at which information is exposed as the interaction helps trigger knowledge associations amongst the experts. The process if self-feeding can reduce prompting from the requirements gathering team who can actively learn about the domain and the system, without disrupting the information flow.

This technique is useful when the requirements gathering team has little domain knowledge. The issue lies in discovering the domain so you can ask relevant questions about the domain and the system. This lack of knowledge can lead to a stilted, lengthy and incomplete extraction of the major features of the system and the domain within which it resides.

The template I use for this requirements gathering technique is based on the ISO 9001 requirements document template. The antecedent for this is the practice of engineering design. It covers the basic input criteria for the system:

- What is the system to achieve in operations and functional performance (throughput, storage, total records, speed of performance, and so on)?
- When is the system required? What is the operational lifetime?
- What are the hours of operation, the downtime and so on?
- How is the system to be physically distributed, and will there be changes to this in the operational lifetime?
- What measures of effectiveness will be used on the system (reliability, dependability, ROI or other cost effectiveness measures, availability, resilience, maintainability, supportability)?
- What environmental requirements will be imposed (OS policies, security policies, supported hardware platforms, network capability)?
- How will the system be supported during its operational lifecycle?
- What safety or security risks may a faulty system pose?

Requirements gathering is an on-going process. A discovery workshop may uncover seventy to eighty percent of the requirements. You may not get the details in some areas and there will most likely be uncertainties, particularly with regard to the physical user interfaces for the reasons listed previously. However, the information gathered should provide sufficient information to succinctly state the aim of the system to be built. It provides important information on the system and its relationship with the environment.

## 2.3 Building models

Models are an important tool in developing an understanding of the system and successively refining the level of detail that you have about the system. A model is a visual representation of what is required of the system. A model helps clarify the interplay of components, giving sufficient level of detail to understand the salient features of the interaction without overwhelming the person studying the model. You might also have several models of the system or the component, depending on the features of interest. For example, you may have a functional component model of the system, with each component having attributes describing its processing characteristics in order to identify bottlenecks and throughput stresspoints on the system. You most likely will have design pattern models describing how the system solution will be implemented in broad object-oriented terms. Each model will have a certain purpose.

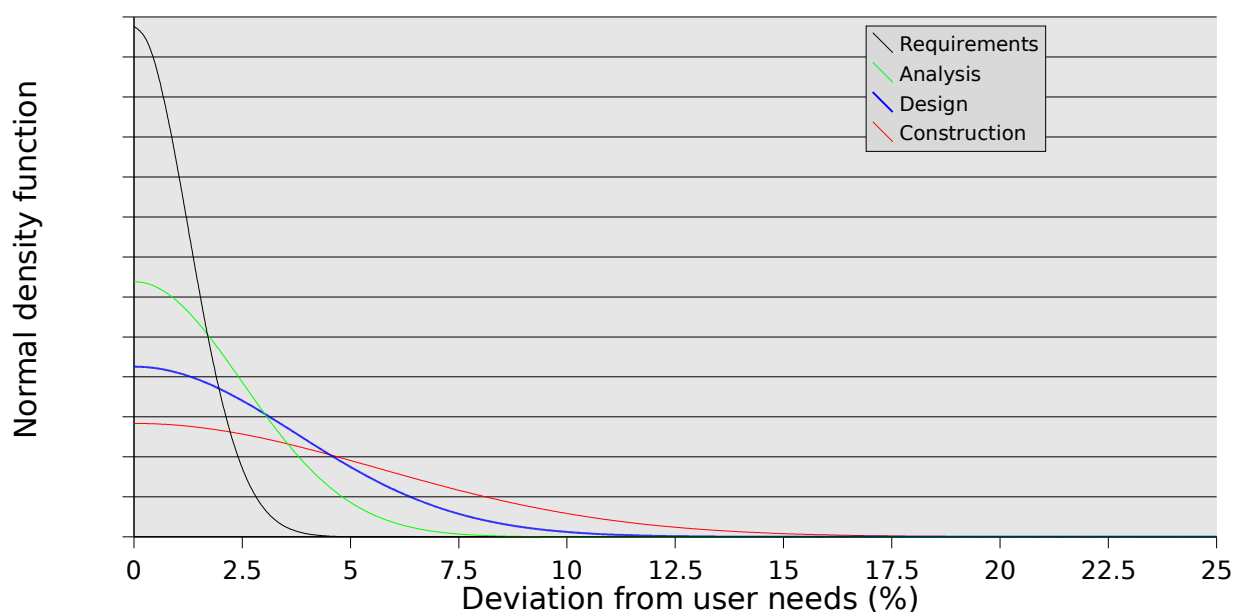
Although some may feel otherwise, I believe it is more important to understand when to apply different modeling techniques rather than learning parrot-like individual common component configurations. You have a compendium of published useful configurations and should refer to them as necessary. Perhaps through experience you may remember certain often used configurations. Electronics engineers use products databooks to determine specific component configurations to employ, and retain a knowledge of the important skills of being able to analyze and partition systems. It is also useful to have a wider range of analogies available in your vocabulary. Natural systems offer a diverse range of design solutions and architectural solutions for study, and it seems almost arrogant not to accept this open offer.

## 2.4 Adaptive and agile development processes

An adaptive development process should include not only the adaptation of tasks and their sequence but also the tools used to develop the system. Adaptation requires frequent monitoring of the useability gap between the tangible artefact of the development process and the user of that artefact. This measure gives the best possibility for error correction as it marks progress against an end goal rather than to an intermediate goal.

For example, assuming that successively each stage of the process of translating user needs to a working system can incur a cumulative five per cent variation, and that the probability for variation can be modeled by a normal distribution function, the deviation can be characterized by the following curves.

Deviation of successive stages

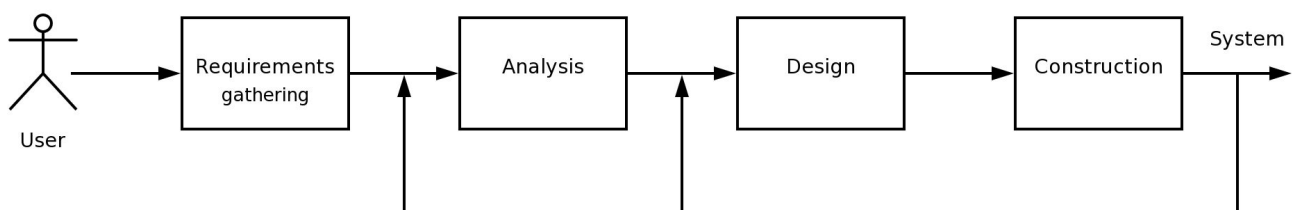


The models produced at the various intermediate stages only capture salient points of reality. This is because systems are complex and we breakdown the interactions and focus on specific areas using our modeling tools. A good systems architect and a good designer will always employ a number of modeling tools and understand the limitations of the models they use.

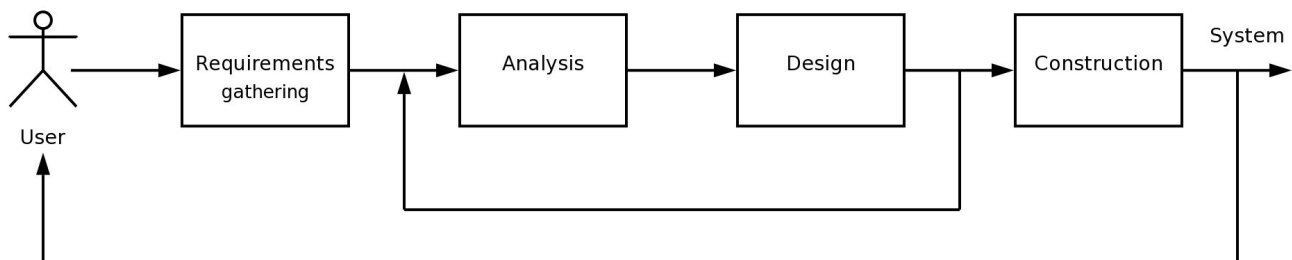
As can be seen, the process of developing a system can incur uncertainties, particularly in interpreting and translating the outputs of previous stages and these uncertainties can dilute the ability of the final system to meet the actual user needs. The probability of attaining the system that exactly meets the user's needs drops while the possibility of returning a system that significantly deviates from the user needs increases. I use the term "user needs" to distinguish it from the requirements, as I believe requirements are an interpretation of the actual user needs.

As the systems engineering process may itself be considered a system, it may be characterized with the following feedback loops. I present the "traditional" development model and the agile development model. I also include the requirements gathering phase as part of the cycle since this is interpretive and a source for errors.

### "Traditional" Development Cycle



### Agile Development Cycle



The agile development cycle benefits from having an error correction loop that includes the "needs generator" as represented by the user, rather than performing error correction on first and second order derivations. It is important to note that our representations are simplified, as all models are to emphasize the features we are examining. The traditional models can have more numerous and complex feedback paths, but all variations omit the user from the error correction loop.

As detailed by Martin Fowler in "The New Methodology" [FOW03], requirements are unpredictable. What we transcribe as requirements and translate through the development process can lead to divergence from actual user needs. Where the user cannot adequately visualise and convey the need, or visualise the use of the completed system, the result of the development process may be system that is unable to be applied to the need it is meant to satisfy. Even when the requirements are thought to be well captured, the vagaries of natural language can introduce errors of interpretation. Without including the user in the feedback loop, deviation from the user needs can be expected.

Would you have a house built for yourself purely by giving the details of your requirements to the architect and builder, or would you be involved in the development process for the house, viewing the progress as the foundations are laid, the walls erected and so on? Would you make suggestions to shift a window to take advantage of a better view or move a doorway because a structural feature prevents easy access?

Traditional software systems development may have followed a waterfall process, but modern proponents favour incremental delivery. However, in traditional engineering[BLA90] and in recent studies on the historical practice of incremental development[LAR03], there is evidence that this is not a new concept in systems engineering or even in software development. It may be that certain interpretations of the systems engineering process have omitted the iterative nature of the process.

One other thing to note is that the system development processes presented imply an all or nothing approach to system delivery. In fact, the process model cannot adequately communicate the idea that successful partitioning of the development should allow various parts of the system to be presented for user evaluation and correction. I sacrifice these details in the diagram in order to retain the clarity of the message – the user is the absolute reference point for error measurement of the resultant system to the user needs.

## **2.5 Documentation**

Documentation on decisions made during the development process is important. It provides useful information for architects and developers on choices made and avenues eliminated. When you need to revisit designs during future iterations, it helps to have notes about past choices and their reasons.

However, documentation should not be considered a deliverable of the development process, excepting documentation for the operation of the system. Notes, drawings, and other by-products of the development process are useful only to the developers, present and future. A good set of design documents is not a guarantee that the final system meets the needs for which it was created. The only measure of success is the usefulness of the system for its intended purpose as judged by the users of the system.

For the same reasons as above, the content of the documents is more important than how it is presented. Simplicity, conciseness and elegance in communication should be pursued. Unless your documentation is destined for publication, whatever tools and media used for documentation should not hinder the capture of thought processes. Should you be spending more time generating documents rather than actively thinking about the development of the system, then consider a less cumbersome means of recording your thoughts.

Documents and models are tools to help you think. They can be used to help order and clarify your thoughts. Tools do not do the thinking for you.

## **2.6 Construction tools**

Construction tools should help automate the work of building systems. They should simplify the process of converting a design to a physical implementation. However, you should always assess whether it simplifies the construction or whether it removes degrees of freedom in the construction of a system. For example, many Java 2 Enterprise Edition (J2EE) development tools hide the fine granularity available in separable components. Much of the advantage of the J2EE framework is that components may be considered loosely coupled as encouraged by traditional software metrics. Loosely coupled components also encourages re-use as the same component could be employed by many systems. The J2EE development cycle encourages the separation of component implementation from system deployment. The deployment engineer takes the components produced by developers, and can bind the components into system configurations just prior to deployment. However, some tools hide the “complexity” of this concept by encouraging tight physical bundling of components at development time.

It is important to use construction tools that help simplify the construction process. Tools should simplify repetitive tasks. However, a tool should not remove fine-grain control from the developer. Simplicity should not translate to simplistic [DEB98]. A tool should not do the thinking for a developer. Tools that hide the construction process or hide details from the developer are counter-productive in the long term. You employ developers for projects because of their thinking abilities, their creative abilities, for their problem solving abilities and for their abilities in creating the physical system. The ability to monitor, control and override where necessary the construction of the system is a vital part of the process.

## **2.7 People**

People are the single most important component in the development process. The act of creation in any system is non-linear. The described process of development covers the linear part of the process. It does not ensure success in the delivery of a system that meets the needs of the user within the environmental constraints given.

Similarly, knowledge of design practices does not guarantee the success of delivery. For the developer, knowing how to think is more important than knowing what to think. The developer, the architect, the designer bring such qualities to the process. They transform the results of the analysis, an evaluation of what needs to be achieved, into a system by defining the best manner in which it can be achieved. It is one of the reasons I value experience over purely theoretical excellence. There are many environmental factors to consider and integrate into working designs that cannot be accommodated by our modelling techniques.

Modern construction tools help prevent language and syntax mistakes at the time of construction. Compendiums of design patterns are available for perusal. Such information may be learnt by heart, much as multiplication tables can. The usefulness of committing to memory such information as opposed to having the ability to research, obtain and adapt such knowledge on a needs basis has been often debated. However, I believe that the ability to assess the need, assess the multi-dimensional constraints of the problem and adapt previous solution fragments is much more important. People who have built successful systems tend to have a better appreciation of the more multi-dimensional problems encountered and tend to have a better ability to adapt theoretical models into working systems.

System building is non-deterministic. The adaptability of people and their ability to think greatly enhance the success of building a working system. Methodologies are sometimes seen as a guarantee that the system built will be successful in meeting the user needs. A methodology is only a general plan for developing a system [DEM87].

## 3 The process in action

I will work through the development process using examples from past projects to help illustrate what I think are useful considerations. In particular, I will focus on a purchasing solution and the However, as businesses are unique and remain so to maintain a differentiation from their competitors, so too will there be uniqueness in the solution you must manufacture. Therefore, this section should be used as an illustrative guide rather than a complete and comprehensive definition.

### 3.1 Requirements

Requirements are a structured interpretation of the user needs and as much as possible try to cover the breadth and depth of the system to be developed as well as the environment with which it will interact. As discussed earlier, there can be a wealth of information but lack of understanding in the business domain can make it difficult to determine the important facts.

It often helps to understand the business forces that help create the need for the system you are developing. The diversity of business operations make it difficult to anticipate exactly the information that will be necessary and relevant to your situation.

An example of business factors that may influence the focus of the solution is shown in a section that focuses on goals set by management. We use this because it often directs attention to measurable goals by which to judge the success of the solution delivery.

#### Management goals

The Business has a goal to create efficiencies in the procurement processes and these are being addressed in different ways. Primarily, the focus of current concern is with the indirect purchasing and the distribution of the function throughout the organisation. The aim of distributing the function is to:

1. Allow end-users to manage their own orders, reducing centralised involvement and associated inefficiencies in processing
2. Simplify the purchasing process such that end-users can easily place their own orders
3. Reduce the amount of non-catalogue purchases and capitalize on negotiated pricing and terms with suppliers
4. Improve the visibility of the process to those involved in the purchasing, including the approvers, without a heavy reliance on the physical paper trail

Changes in the environment must also be considered so the system remains a relevant solution during the expected operational lifetime. The template we use covers this under a section on future considerations.

#### Future considerations

There are no plans for future electronic integration with suppliers of indirect goods, either in exchanges of purchasing information, including confirmation of order and capability to supply, exchanges of expected delivery timings or invoicing detail.

Although no time-frame has been supplied, it is expected that the Australian corporate unit will seek inclusion in the indirect purchasing processes. This however, will only comprise a change in the user population of approximately twenty persons.

Of greater significance is the requirement for the system to handle multiple corporate cards as well as the ability to handle personal liability cards. There is no time-frame by which these requirements are to be met.

You may also discover performance information in the course of the initial requirements gathering phase.

### **Corporate purchase cards**

There are approximately x hundred Corporate MasterCard issued by the Business to staff of which n percent are active during any particular month. Corporate purchase cards have a total expense value of over y million dollars per month. Typically there are three to four line items per month on statements, resulting in z thousand general ledger lines per month. The corporate card is intended to provide for:

1. Low-value, one-off, non-catalogue items
2. Purchase of travel-related expenses
3. Entertainment expenses as it relates to business activity

Beyond such influences on the development processes and the definition of expectations, there are the more obvious direct interactions of the environment that need to be explored. The aim is to establish the relationship of the environment with the system to be developed. More often than not users will relate these in terms of scenarios, so this is well suited for the application of use cases.

Most users will visualize and hence talk about the interactions in terms of processes and functions, and use a domain specific language. A discovery workshop with the encouragement of free-flowing thought will allow rapid transfer of domain knowledge through continued exposure to the use and application of domain specific language. The workshop will result in a lot of business structured workflows, processes and functions. Analysis should not be undertaken during the course of the workshop, but should focus on the recording of all the information as it comes out. After the discovery workshop, you can begin to structure the interactions and begin to establish the boundaries of the system.

For example, based on extracts from many pages of notes taken, we assembled the following definition in the “personnel and roles” section of the requirements document.

### **Requesters**

Requesters are those employees of the Business who make a request for the purchase of goods or services either for approval before or after the consumption of the goods or services, depending on the mode and nature of the purchase. As appointed representatives of the Business, these requesters have the authority to place orders with suppliers on behalf of the Business, with either explicit or implicit approval.

The issue of purchase cards (corporate cards) to employees authorises card holders to make legitimate company expenses on the card account and later have such purchases approved as part of the statement reconciliation process. This is considered an implicit purchase approval as at the time of acquisition of the goods or service, there is no required approval. Similarly, re-imbursements are treated as an implicit approval as other than a verbal approval, no requirement for formal approval of purchase is necessary at the time of acquisition. In both these cases, formal approval is sought after the acquisition of goods or services.

In all other cases for indirect purchasing, formal approval must be sought prior to submission of an order to a supplier.

With the exception of purchase cards, any employee of the Business may be a requester. For purchase cards, only card holders may be a requester. Requesters may have access to any of the following methods of purchase:

1. Purchase of goods
2. Purchase of services
3. Re-imbursement for personal expenses incurred on behalf of the Business
4. Purchase of goods and services using the purchase card

Functional and procedural requirements can also help expose underlying relationships and identify additional systems. For example,

### **Approvals**

The operation of approvals in the context of the Business is such that:

1. An approver is a nominated position, rather than a person
2. The person who is currently holding the position will perform the approval
3. Approvals for purchases made against a cost centre will be directed through the chain of approvers who have delegation for that cost centre for the document
4. The requester will be able to select the starting point for approvals against a cost centre from a list of approvers who have delegation for that cost centre for the document
5. Where multiple cost centres result in multiple chains of approvers, approvals will traverse chains in parallel
6. Where an approver appears in multiple chains, that approver will only be petitioned a single time for approval
7. The sequence of approval through multiple chains will ensure that an approver who is referenced in multiple chains will only be petitioned, when all other approvers ranked below that approver in all the relevant chains have given approval
8. A request will only traverse an approval chain sufficient for an approver with sufficient financial delegation to approve the requested expense against the cost centre
9. Where a requester is also an approver for cost centres involved in the request, the approver and those with delegation below that of the approver for those cost centres will be removed from the approval chain for that particular request
10. There is at least one approver for any one document
11. Approvers will be notified appropriately of a document awaiting their approval but only when at the time their approval is being sought

This process detail identifies that there is a source for cost centre information and there is some organization for approvers of a purchase request. It also produces textual quality for the process.

We use the requirements document as a structuring tool. Although I have stated previously that I believe the documentation is of less importance than the delivered system, the requirements document has a special significance in that it organizes and records the first interpretation of the user needs. We use it to deliver to the client a tangible checkpoint to show we have listened and we have understood what has been communicated. We've learnt not to use it as the absolute contract for deliverables.

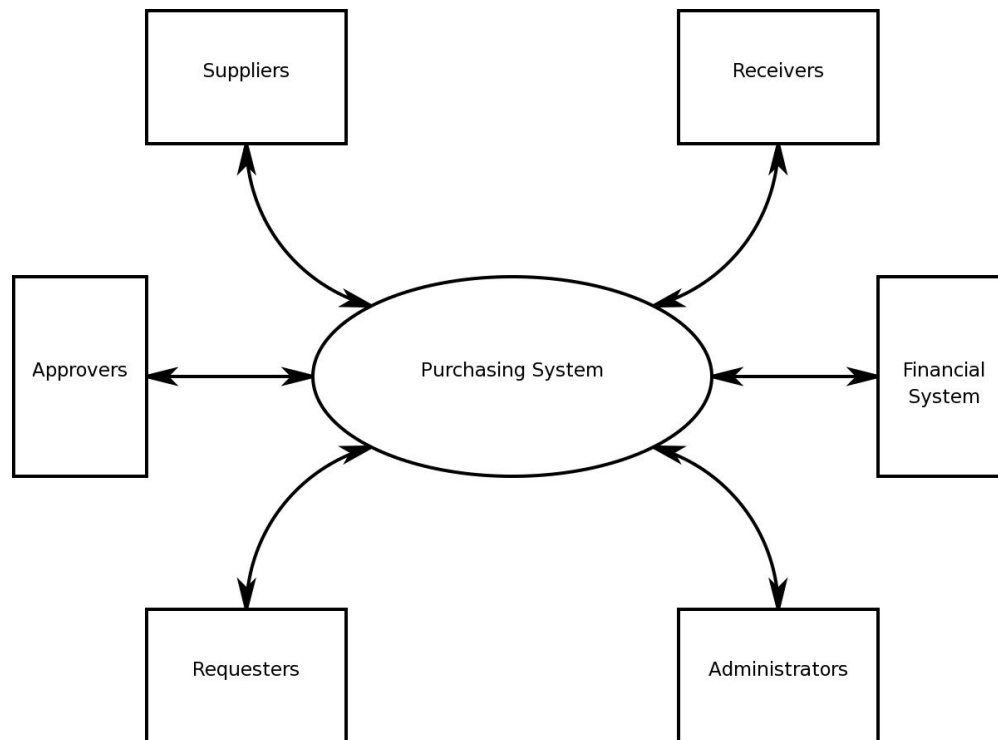
The requirements document should be considered a live document until the completion and acceptance of the system. The content is more important than the formatting. While we use a general outline of topics based on past experience, we do not rigidly adhere to the content headings. Dependent on the project, we will add or delete headings as necessary. As much as anything, we use the outline as a loose framework for discovering the boundaries of the system and the influences on the system. Use your requirements document template as a prompt for asking questions about the system. The requirements document is there to help you think about the system. It is not a meaningful deliverable. No one remembers your well written requirements document when the system deliverable was a failure.

## **3.2 The system in context**

Many people like to immediately develop and analyze the use cases obtained from the requirements gathering process to produce the design templates. However, I prefer to at least produce an overview of the system and the external interactions it has with the environment. This provides a good visual aid to which to refer when sanity checking the completeness of the analysis.

I still use the context diagram from Tom DeMarco's Structured Analysis and System Specification [DEM78] for this purpose. I also feel it gives a basis or reinforcement for the arrangement or grouping of many of the external use cases. It also clarifies your systems thinking because developing the context diagram forces you to think about the single purpose of the system. You should be able to express the purpose for which you are building the system in no more than a paragraph, and you should be able to develop the context diagram to express with simplicity and clarity the system, its purpose and the external systems with which it interacts, including the users.

### Purchasing system context



The purchasing system receives and processes purchase requests, creating the necessary notifications to the financial system, approvers and suppliers to complete the action of purchasing goods and services for the Business.

The context diagram makes no implications about the exact nature of the relationship or the composition of information that passes between entities. It does identify the main groups of systems that form the external environment and it also identifies the system that is the focus of the development and its single purpose. I find this helps galvanize the thought process around the system to be developed.

### 3.3 Use cases

Use cases document the user goals interpreted from the requirements gathered from the business users. It is important to separate the use case from any reference to how the activity is to be implemented. The aim of the use case is to define what must be done, not how it might be achieved. You must avoid implementation descriptions in a use case. Martin Fowler describes this as being the difference between user goals and system interactions [FOW97a].

The dangers in defining how you will achieve a goal can obscure from the developer and the user the actual needs, and sets design decisions in motion while still in the requirements definition phase. The ramifications of these early choices may create disharmony and strong data impedances between components in achieving the total delivery of a solution. This can lead to inefficient or more complex designs to overcome these impedances.

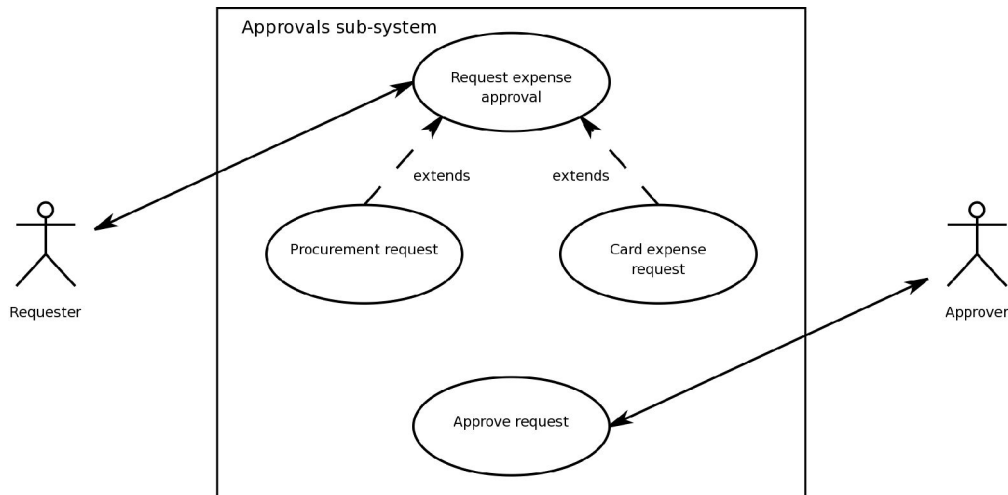
The context diagram also provides a classification of the general actors for the use case studies. This can help provide a visual generalization for the activities expected and hence act as a prompt for the completeness of the use cases. Many of the simple and apparent use cases will have been established in the initial requirements gathering phase. There is a symbiotic relationship between use cases and the context diagram as the initial requirements gathering exercise inevitably produces use cases and these help the analyst form the general organization of the context diagram.

Throughout the iterations, more knowledge will be gained and will lead to more detailed questions, but many will be prompted by questions related to those groups of actors identified in the context diagram. For example, "What does the group, Requesters do?" As I said, initially you will have probably obtained seventy to eighty percent of the information about Requesters. As a follow up, you will elicit some of the important alternative courses. The more obscure alternative courses will probably not be discovered until working code that supports the main use case has been tested by the user.

As the number of development cycle iterations increases, the remaining information pertinent to the system will be revealed as you gain more in-depth knowledge about the problem domain and the goals of the "users". This should be reflected in the folder of use cases you have for that group of "users". I find that organizing the major "users" of the system, including other computer systems into folders where you can slot use cases and their alternative courses belonging to that group of "users" is a good organizational tool.

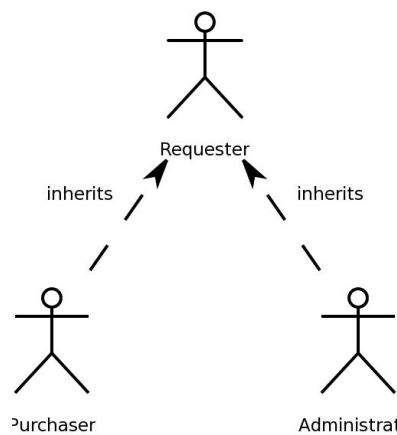
This paper-based filing method also allows rapid development and organization of the use cases. Many may find the graphical tools useful but for me, these are a hinderance. I can hand draw the scenarios faster than I can create them on the screen, and I can shuffle the scenario sheets around as I work towards some cohesive and logical clustering of the activities. For use cases that seem on the boundary of groupings, I can file them away for a revisit or pin them to the wallboard as a prompter. Particularly in the formative stages of discovering the natural organization of the system, I think the paper-based approach provides speed and flexibility. Perhaps when the system form starts to stabilize, your drawings and hand-written notes can be committed to an electronic repository.

One other important set of sheets I like to create contains all major system states that should never occur. These states may be noted for reasons such as dangerous output conditions, major impact on outlying systems and so on. I draw these from experience in the design of digital systems. We actually take it one step further in digital system design since we carefully mark "don't care" states, from normal states. However, the inherent complexities of software systems results in a large number of states. So I restrict the requirements tracking to the union of the set of normal use cases which handle discrete events and the set of general system states that should not occur. Sometimes these overlap but highlighting events that should never occur helps ensure that there are no major system vulnerabilities. These might also be known as system assertions. Having a folder for these conditions can help remind developers to actively search for these conditions.



During the process of building your initial set of use cases, you may notice the emergence of natural sub-system organization. Whether you label these or not is optional. I label for convenience and if I feel reasonably confident of the boundedness of the functionality. This should not imply identification of an object at this stage, and definitely should not imply any implementation details. Note that this only describes the visible interactions and does not uncover any relationships with other parts of the system. Further into the process, we will need to establish the component relationships of the system. Should there be no interaction with any other components or sub-systems in the system, then based on the definition of a system presented earlier, we might conclude that our use case does not belong to the system we are developing.

**Requester of an expense approval**



A corporate purchaser or a system administrator can make a request for approval of an expense that has been incurred or will be incurred on behalf of the Business.

With use cases you may also have the need to define responsibilities for actors. During the course of some of the elaboration, you may encounter phrases such as “a system administrator may also make a request for the approval of an expense”. For most reasonable-sized systems, it helps to depict this in case the words are overlooked. I also usually have some annotation as shown here to quickly explain what purpose this serves. This does not imply that an implementation would contain specific class representations of an administrator or a purchaser.

As hinted at, an arrangement of interactions can be based around sub-systems. However, you usually only start creating an idea of a sub-system by seeing enough interactions that suggest the organization around such a functional cluster. We will cover sub-systems and partitioning in the next section.

Remember to review and re-organize your collection of use cases. When your system form stabilizes, you probably can move forward with analysis of the system. Make notes on major re-organizations or removed use cases and the reasons. I keep another folder for discards and notes on changes. I don't retain erroneous use cases but I do keep use cases that were refactored or split. Balance between trackability and clutter. Too much information will make it difficult to track back to major decisions. In accordance with the Agile Manifesto, I suggest that this decision is up to the skill of the practitioner.

## **3.4 Analysis**

Analysis is intended to expand the designer's knowledge and understanding of the system. It is a part of the creative process. For the same reason that an artist develops sketches of the subject matter, with different sketches exploring different levels, from general form and structure to in-depth studies of particular features, a designer develops various models of the system. Each model provides a particular perspective to understanding the relationships and interactions in the system. None can claim to be a whole and complete facsimile of a system that would meet the user needs.

### **3.4.1 Partitioning and architecture**

Jacobson's text on object-oriented engineering [JAC93] does make reference briefly to subsystems in the analysis model. However, it doesn't emphasize the importance of defining the high level structural features of a system. "A System of Patterns" [BUS96] better explains and distinguishes architectural patterns from design patterns and design patterns from idioms. It is a matter of scale.

Patterns describe solutions to a particular problem in terms of a set of relationships. The solution describes a finite set of components and their spatial configuration. Many designers prefer working up from medium-scale component organizations such as design patterns. Some prefer working down from architectural patterns. I prefer using a mixed approach, trying to identify significant relationships and categorizing these in terms of known patterns. My technique is an emphasis on finding major relationships and naming them, rather than forcing relationships into a perhaps preconceived pattern. Perhaps this should be termed a search for natural configurations.

Natural configurations can reveal themselves as you proceed through the analysis phase. The scale of the configurations can vary. However, the most important configurations relate to high-level organization. Aside from dealing with non-functional concerns, architectural patterns provide a blueprint for the cohesiveness of design patterns, limit the expression of these medium-scale solutions and provide an internal checkpoint for the delivery of capability given the analyzed requirements.

Some feel that design patterns and the original unified software development process is sufficient in uncovering the architecturally significant relationships. I advocate the search for significance as you zoom down in detail. This is usually achieved by discovery through the use-case realization [JAC99]. The level of abstraction provides sufficient high-level features to identify sub-system de-coupling that would satisfy non-functional requirements. However, this should be confirmed later with design pattern interactions at the interfaces to confirm the appropriateness of the partitioning. You may find greater sub-system efficiencies in adjusting the boundaries slightly. So while the primary search mode is based on a top-down approach, there should be an application of pragmatism when reconciling the partitioning with lower-level design.

A filed annotation for any boundary changes should be made, describing the reason for the change. Again the content of the annotation and the act of annotation is more important than the form which this takes.

### 3.4.2 Analysis patterns

Use cases and their derived analysis models often do not provide enough depth to address or represent features of the system, particularly with regard to data organisation. There is a large amount of information and business rules that is provided by the business and the users and this information can be unintentionally left out of the use case analysis. I use Martin Fowler's method for capturing these as analysis patterns [FOW97b].

The following excerpt describes the fairly complex organization and rules for expense approvals. In order to help capture the salient features, I use a slight modification of Fowler's pattern for hierarchies in order to depict the important concepts of approvals. As will be seen though, some requirements cannot be adequately captured without additional notation. The example shows the derivation from original captured description for approvals and the operation. The reworded description supplements the diagram and attempts to be more succinct in identifying the features. Remember that the aim is to model the structure and organization of information as understood by the user, not to determine the manner in which the approvals information structure will be implemented. Note that this model does not capture an entire process itself.

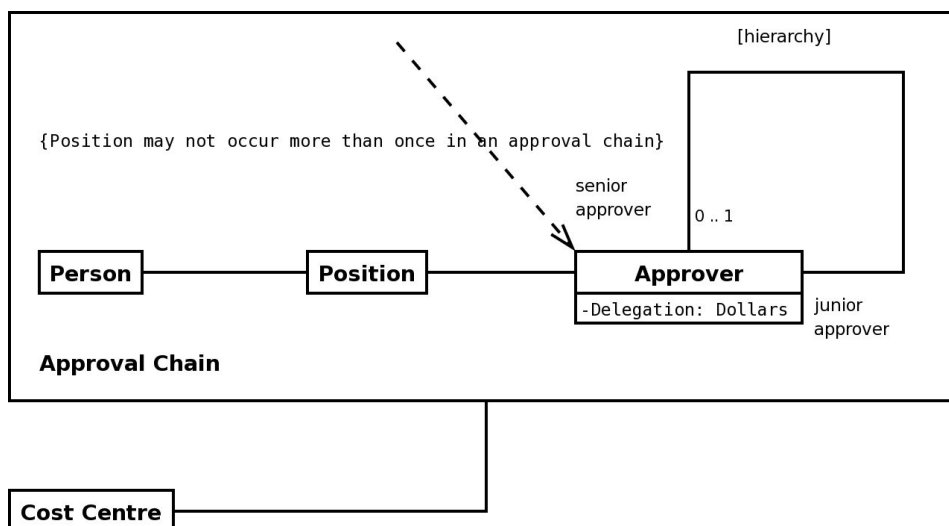
#### Approvals

Approvals for an expense, whether already incurred or pending receipt of goods, will follow an approval hierarchy as dictated by the Business. This hierarchy will be dependent on the dollar amount each position is delegated with authority to approve, arranging the hierarchy in an ascending order based on the delegation authority in dollars. Each approval chain relates to a specific cost centre. There is only one approval chain for each cost centre.

A person may have acting authority for a position, or in the event of a restructure or because of a promotion, may change position permanently.

The person requesting an expense approval may not participate in the approval of the expense. An expense must be approved by all approvers in the approval chain up to the approver whose delegation exceeds the value of the expense.

Any valid approver in the approval chain may be selected by the person requesting the approval as the starting point for the approval process. Approvals will always travel up the chain.



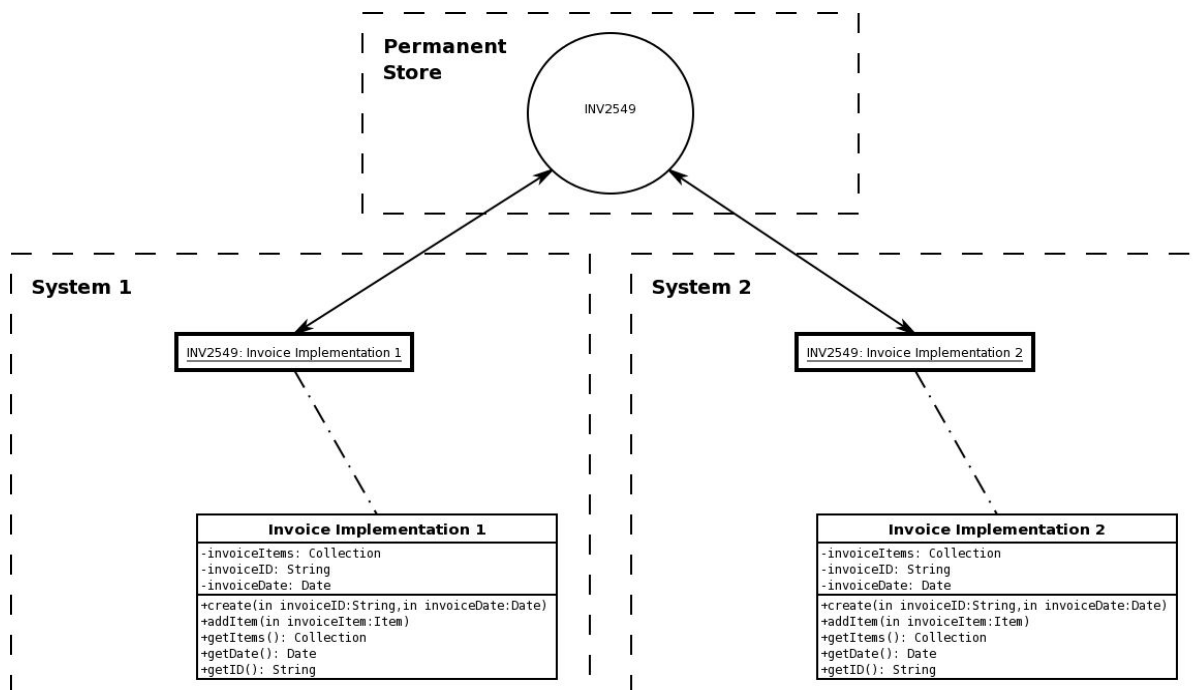
### 3.4.3 Scope of representation

Our models generally reflect real world objects and these real world objects have a long life. For example, an invoice for goods exists beyond the time it was created and used to request goods from a supplier. Our information systems create short term representations of these objects that last only as long as the system that generates the representation remains active. Even should we lose power to the information system, when power is restored the system should be able to recreate its representation of that real world object and the object should retain the same state and characteristics it possessed before power was lost. In order to maintain the state and characteristics of the object representation, we need long term storage for this information that will survive failures of the information system itself.

In object oriented systems, we use the object paradigm for several major reasons:

- Localize the areas of implementation changes and improve maintainability of the system implementation [JAC93]
- Encapsulate the data that represents the real world object, and limit changes to that data, according to rules and constraints that govern the real world characteristics of the object

The long term storage is a less faithful representation of the object and usually contains none of the rules and constraints. This creates a problem when there are multiple representations such as when two systems use the same data.



In the example, the invoice data is used by two different systems that use their own implementation of the invoice object. It is possible that the two implementations are the same. It is also possible that the two implementations are different. This can result in inconsistencies in the underlying data. It also breaks the localization and encapsulation benefits.

In order to retain the benefits of an object oriented approach, the object representation in this example must be able to span the environments of the two systems. Ideally, it should be separable from the systems that use the representation.

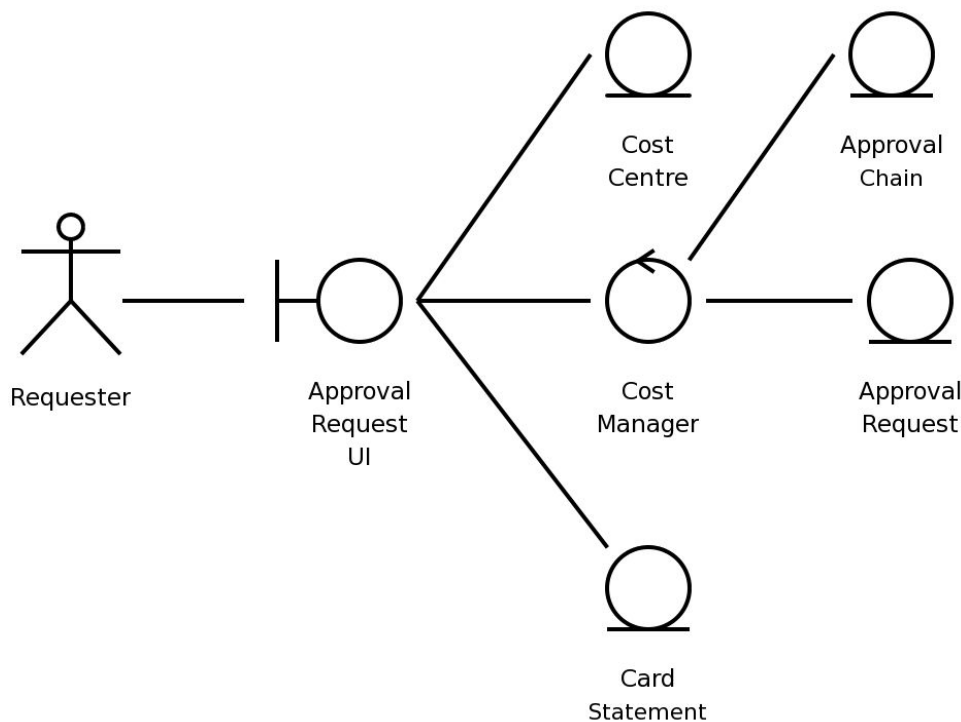
The scope of representation required of an object introduces an important design consideration. This consideration lends support to the Enterprise Java Bean (EJB) model as an EJB component can exist independently of the system that uses it. An EJB that represents a real world object retains the localization, encapsulation and maintenance properties we sought in the original drive for object oriented design. The single implementation also ensures data consistency, in accordance with one set of implemented rules.

This environmental independence comes at a price as there are complexities in delivering the connectivity to achieve this. However, a single view of enterprise-wide data when built early will avoid potentially more difficult work later in re-engineering and integrating multiple systems to an inserted middle-tier. The architect must make an assessment of the scope of representation, both currently and during the operational lifetime of the system, and determine the appropriate structural course.

Such an approach to assessing the scope of representation for business objects should also be extended to the processes or sub-systems that manipulate, transform or manufacture such business objects.

### 3.4.4 Use-case realizations

The tendency has been to transform the use-case directly to the use-case realization. However, I have found that Fowler's analysis patterns are useful in uncovering entities and depth of meaning. In our example, the approval request process does not elaborate on the manner in which the approval process operates. The hierarchical organization of the of approvals was documented with the analysis pattern. We can also make reference to this in a meaningful way when we devise the use-case realization. I show this in the following diagram for the request process.



## 4 Pitfalls

### 4.1 Partitioning a system

Partitioning can be artificial. Sometimes this is out of necessity when a sub-system or system cannot be satisfactorily split, yet the existing functional block is too large for adequate maintenance. The use of structural patterns at the architectural level can help partition the system into manageable blocks of functionality. Sometimes this can introduce inefficiencies in system operation. The system architect needs to make a value judgement on maintainability over efficiency.

The OSI seven layer model is an example of a logical partitioning. It separates the responsibilities definitively. However, the strict separation introduces problems in operation. At gigabit speeds, no single processor can process the protocol. It introduces complexity in the coupling and intercommunication.

There are also times when the architect must introduce artificial boundaries to the domain or the system that affect the system partitioning. These most often occur when digital systems must model a continuous domain. The artificial partitioning introduces difficulties in sub-system communication or collaboration but cannot be avoided. An example of such problems is a Supervisory Control And Data Acquisition (SCADA) system for vehicular traffic regulation. Such a system might be required to manage the traffic light sequencing for a city. The large two-dimensional topology is partitioned into districts for efficiency in management. However, such partitioning is arbitrary and introduces issues of synchronizing sequencing of lights across the district boundaries.

Such are the problems that may be encountered. The skill and the experience of the architect is needed to make value judgements in these cases. Different approaches to modelling the system can help reveal better solutions. The study of solutions in natural systems can also be of benefit. An experienced architect makes use of many tools and analogies. The catalogue of patterns being built for the various software development domains can also help. But I would recommend that the scope of the search for solutions should extend beyond the field of software development.

## Bibliography

- BLA90: Blanchard, Benjamin S.; Fabrycky, Wolter J., Systems Engineering And Analysis, 1990, Prentice Hall, ISBN 13-880758-2
- FOW03: Fowler, Martin, "The New Methodology", 2003, <<http://www.martinfowler.com/articles/newMethodology.html>> (20 Nov. 2003)
- LAR03: Larman, Craig; Basili, Victor R., "Iterative and Incremental Development: A Brief History", IEEE Computer, vol. 36, no. 6, 2003, pp. 47-55
- DEB98: De Bono, Simplicity, 1998, Viking, ISBN 0-670-88155-4
- DEM87: DeMarco, Tom; Lister, Timothy, Peopleware: Productive Projects and Teams, 1987, Dorset House, ISBN 0-932633-05-6
- DEM78: DeMarco, Tom, Structured Analysis and System Specification, 1978, Yourdon Press, ISBN 0-13-854380-1
- FOW97a: Fowler, Martin; Scott, Kendall, UML Distilled: Applying the Standard Object Modeling Language, 1997, Addison-Wesley, ISBN 0-201-32563-2
- JAC93: Jacobson, Ivar; Christerson, Magnus; Jonsson, Patrik; Övergaard, Gunnar, Object-Oriented Software Engineering, 1993, Addison-Wesley, ISBN 0-201-54435-0
- BUS96: Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael, Pattern-Oriented Software Architecture: A System of Patterns, 1996, John Wiley and Sons, ISBN 0-471-95869-7
- JAC99: Jacobson, Ivar; Booch, Grady; Rumbaugh, James, The Unified Software Development Process, 1999, Addison-Wesley, ISBN 0-201-57169-2
- FOW97b: Fowler, Martin, Analysis Patterns: Reusable object models, 1997, Addison-Wesley, ISBN 0-201-89542-0