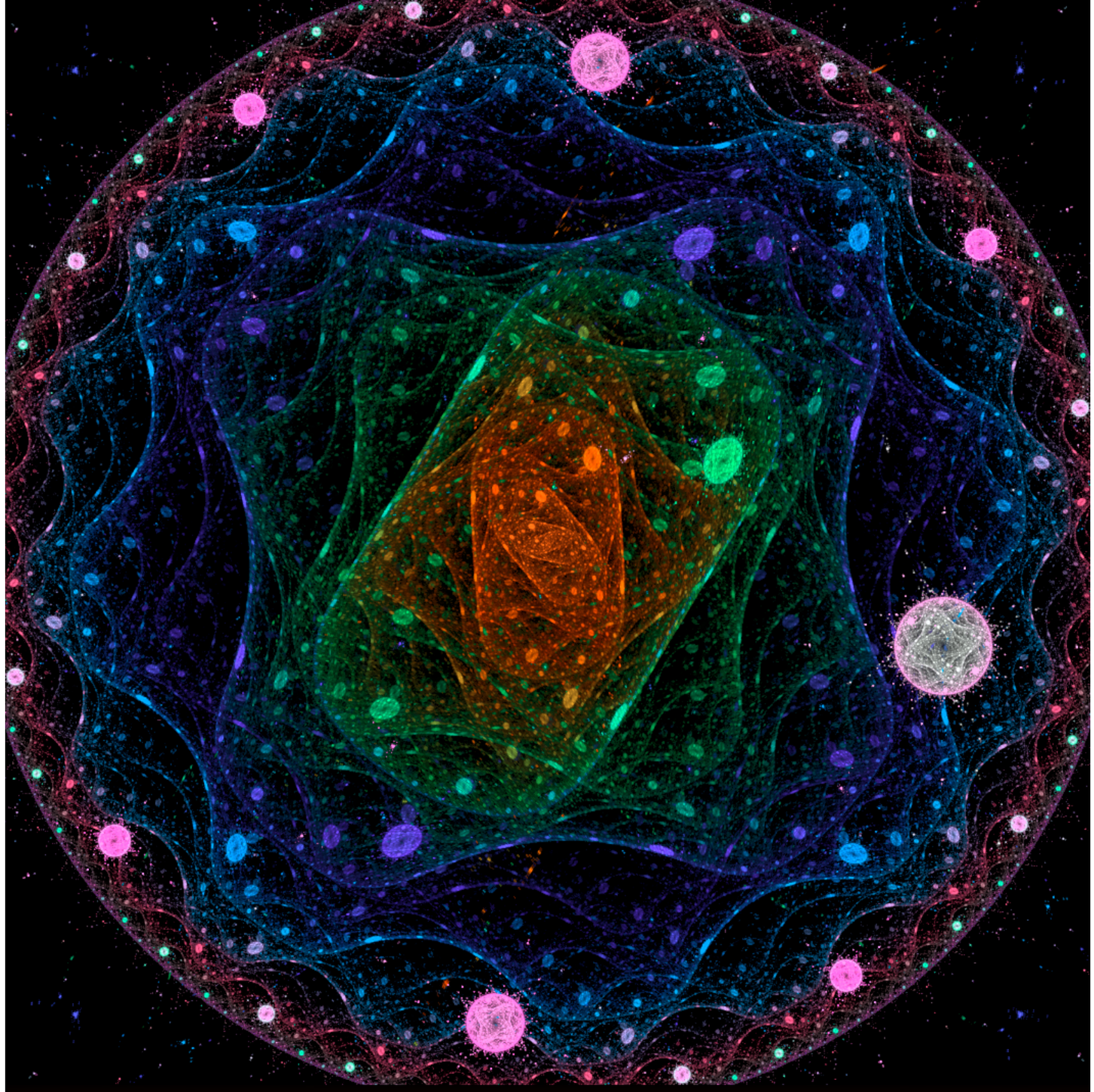


Secrets of Analysis

Generative OOA with UML, NLP, Literate Modeling and M++

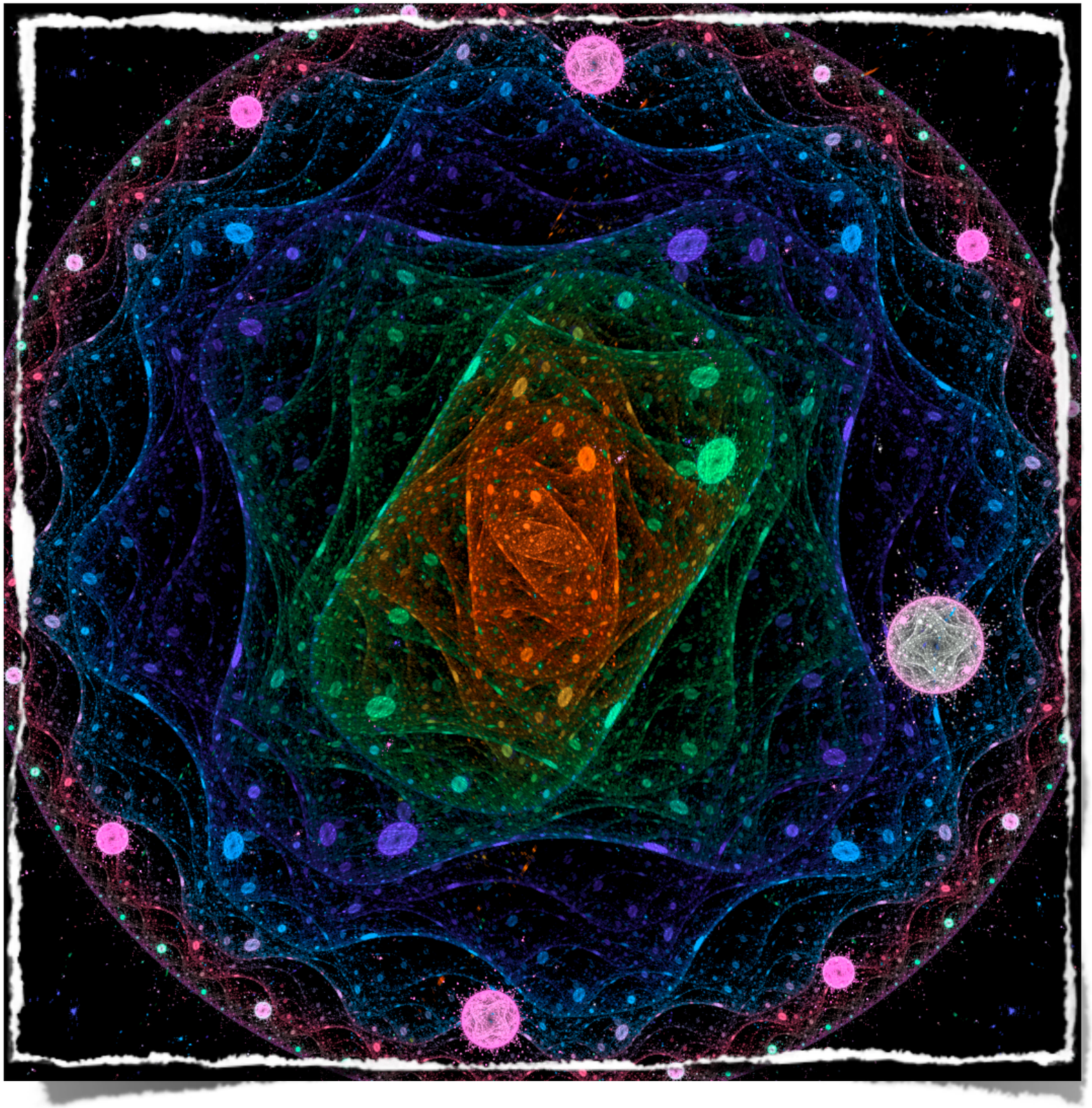
Jim Arlow and Ila Neustadt

Published by Mountain Way Limited

ISBN: 978-0-9572928-1-9

Cover and book design by Jim Arlow

Preface



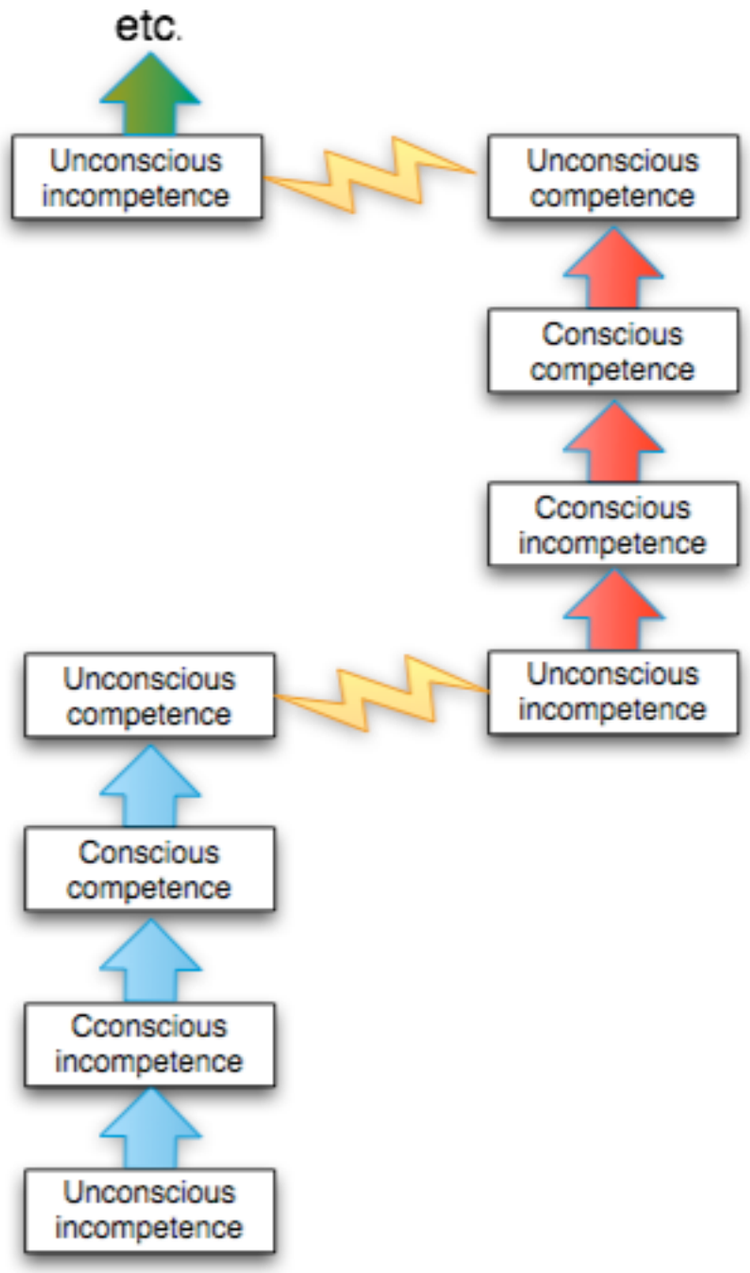
About this book

This is a book about Object Oriented Analysis (OOA). There are four levels of competence in OOA:

- Unconscious incompetence - you don't even know what OOA is.
- Conscious incompetence - you know what OOA is, but you don't know how to do it.
- Conscious competence - you know what OOA is and you can do it to some extent, but you still have to think about it.
- Unconscious competence - you are a true expert. You know what OOA is and you can do it without really thinking about it at all.

It's important for you to understand that by "incompetence" we simply mean "lack of competence", rather than "bad at". We're not using the term in a derogatory sense at all!

Figure Preface.1 The cycle of learning



Having achieved unconscious competence in any field, it's essential to remember that the world moves on. And it does so particularly quickly in the field of software engineering.

If you don't move with it, that unconscious competence can rapidly become unconscious incompetence as you ignore new innovations in your field. That's no problem, provided that at some point you *recognize* this and start the cycle of learning again.

We (Ila and Jim) are fascinated by unconscious competence! This is the realm of the true expert, of mastery in a particular field of endeavor. Usually this expertise is acquired over many years, often through hard experience.

But suppose we could model what an expert does, how they behave, what they think, how they approach their world

(their map of the world)? Surely then, we could teach this model to others so that they too can become experts. This is one of the key principles of Neuro Linguistic Programming (NLP) - what one person can do, so can another. This is exactly what we try to do in this book.

"What one fool can do, so can another"
Silvanus P. Thompson, Calculus Made Easy, 1910

Our original idea for this book was a simple volume of worked examples to be sold as a companion to our best-selling, "UML 2 and the Unified Process" [Arlow 1]. We've had lots of requests over the years for a really complete worked example.

However, as we began to write we became very aware that throughout the construction of the examples we were using particular thought processes and techniques that had been hard learned by us over many years of analysis and design. Nowhere (to our knowledge) had these thought processes and techniques been made explicit. In fact, until we had the leisure to work on these examples relatively free of time constraints, we had not even been consciously aware that we were using them. This is why we call them "secrets".

Other OOA books, such "UML 2 and the Unified Process", tell the novice *what* to do *when*, and are very valuable and relevant. In fact, they are an essential prerequisite for a book such as this.

However, they tend not to cover the *how* of OOA that experienced modelers apply often intuitively and unconsciously to get great results. After many years teaching OOA we observe again and again that beginners to analysis and design, even if told exactly what to do and when to do it, generally don't get the same quality of results as experienced analysts. Something is missing - the *how* that only the experts have. Thus a larger idea for this book was born.

We decided to model what we, as experienced analyst/designers, do and how we do it. Our goal is to impart this knowledge to you, the reader, in as efficient and entertaining a way as possible. We call the book "Secrets of Analysis" because the techniques we present here have been, until now, self-secret. That is to say no one has been consciously hiding them, it is just that the inner process of analysis, the how part, has never before been presented in a book such as this.

We proceeded towards our goal through a process of modeling. In particular, Jim, being trained in Neuro Linguistic Programming (NLP) and in Ericsonian hypnosis, was already quite familiar with modeling success strategies. As we created the UML models in this book, we also created informal conceptual models of how we were doing the analysis and design activities.

These models were a revelation to us! When we stopped to examine them, we found we were doing things that were crucial to achieving success, but

which we couldn't find written down anywhere. In fact, it has taken us many years to become sufficiently good modelers (in both the NLP and UML senses) to be able to model our own modeling strategies in a compelling way.

Although these models might be interesting from an academic perspective, it is really the insights and techniques (secrets) that we discovered that are the real value as far as you, the practitioner, is concerned. It is these secrets that we present to you in this volume.

Our approach to OOA is pretty unique, and so it's appropriate to give it a name so that we can refer to it easily and so that you can tell your boss what you are doing. It also helps to clearly distinguish this inner, *how* based approach to OOA from more conventional *what*, *when* approaches.

Generative Analysis

We call our approach Generative Analysis:

- It is *generative* because its focus is on generating information - how you capture information and how you transform information into new, more specific forms of information that take you ever-nearer to your goal.
- It is *analytical* not so much in the specific sense of OO Analysis but rather in the general sense in that it uncovers basic principles.

Through our studies in human communication we have learned many things that are directly applicable, and which we directly apply, to the process of OOA.

We have also learned a certain number of "spells" - ways to alter your perceived reality to help you achieve the specific goals that you want. Here, we use the term "spell" as a fun way to refer to mental programs that you can execute on Human Brain 1.0 (or compatible) to help you to achieve desirable results easily. They are software for your mind!

Some of these spells we find we use quite consistently in the process of OOA, and we thought that it would be interesting and useful to you if we were to present a few of them here. Do treat them as a bit of fun.

Because they work directly with your neurology through a process of metaphor, the spells can work whether you believe in them or not. However if you can approach them with curiosity and at least suspend *dis*belief, they will be even more effective. We provide follow-on references so that you can take your spell craft as far as you want it to go.

Why this book is different

In summary, this book is very different to other books on OOA because (as far as we know) it is the first to explicitly present the *how*, the inner processes you go through in order to get the results you need.

This is testable: we set out to answer the question "how do you do OO analysis"? Your success as an OO analyst is the measure of how well we have managed to answer that question for *you*.

About the Website

The website for this book is:

www.secretsofanalysis.com

There are some open source tools on the website, created by us:

- Semantic Highlighter - a way of looking at text from different dimensions of meaning.
- STAR modeler - a simple development environment for use cases, requirements and project glossaries.

We created these tools for purely didactic reasons - to help us to get our ideas across to you in a more concrete way, and to allow you to easily apply the techniques that we present. They can also help you to do the exercises in the book.

Our inspiration for these tools is BlueJ - an integrated environment specifically designed for teaching Java.

Some time ago, Jim gave a Java and XML course using BlueJ. He was blown away by the experience! He found that he could teach Java much more effectively using this environment than he had ever been able to before.

We wondered if it might be possible to create a simple tool set to teach some of the ideas we present here. Our research led to the Semantic Highlighter and the STAR modeler. We hope you find them interesting and useful, and we'd love to get some feedback from you!

You will also find hyperlinks to the following open source tools (not written by us):

- FreeMind - a mind mapping tool
- CMap - a concept mapping tool
- Compendium – a concept/dialogue mapping tool

When we get time, we also intend to make some of the key OLAS artifacts available. These will be in MagicDraw UML format.

Enjoy!

Who should read this book

The key prerequisite for our reader is that *you have some knowledge of UML and OO Analysis and Design (OOAD)*.

This is **not** a book from which to learn UML or OOAD! Our other book, “UML 2 and the Unified Process”, is the book for that.

Given the above prerequisite, readers may be:

- Novice OO analyst/designers who need to learn how to do it.

- Experienced OO analyst/designers who are interested in our secrets.
- Programmers who have some UML and OO experience and who wish to learn how to become better abstractionists.
- University students who are doing a course of OO analysis and design and who want to understand it more deeply.

Another prerequisite is that you are a creative and intelligent person who is open to new ideas and willing to change.

This book *will* change the way you think about things. If you don't want to change then you should not buy this book.

How to read this book

This book is in two parts:

- Part 1 (Chapters 1 to 11) – sets down the basic principles and techniques of Generative Analysis
- Part 2 (Chapter 12 onwards) – OLAS, a complete worked example

We prefer to write books that can be read in many different ways - for example there are at least five different ways you can read "UML 2 and the Unified Process". That's one of the reasons it has been so successful.

But because this book is based around a worked example, the narrative in Parts 1 and 2 is largely linear because it follows the development of the example.

You should start at the beginning of Part 1, and work your way through to the end of Part 2. We develop our ideas as you progress through these parts of the book, so it's difficult, at first reading, to skip around too much.

We have tried to pull out key ideas into figures and a detailed summary at the end of each chapter, so you could just read these if you are very pushed for time. You will certainly get something out of this approach, but not as much as if you read the whole text.

We decided to base this book around a worked example because we felt that we needed to present our ideas about OOA in the context of real, but simple, OOA activities. This allows us to make the presentation much more concrete, and we hope you will be able to see the immediate usefulness of the techniques we present. A different, technique-based structure for the book would have allowed you a more flexible approach to the text, but at the cost of loosing the context.

Miskatonic University – the Lovecraft connection

The examples in this book are set in Miskatonic University by the side of the Miskatonic river in Arkham, Massachusetts. Miskatonic is a fictional University, by a fictional river in a fictional town invented by the (real) author H. P. Lovecraft.

Lovecraft (1890-1937) was an author of weird fiction who lived in Providence, Rhode Island, USA. Over the years he developed a fictional world in which to explore his ideas. Many of his works are set in, or refer to, fictional New England towns such as Arkham, Kingsport and Innsmouth. His stories are usually found classified under horror, but Lovecraft always viewed himself as a teller of weird tales rather than as a horror writer.

Jim has enjoyed reading Lovecraft over the years, and when we needed a setting for the Orne Library Automated System (OLAS), the example project in this book, Lovecraft's Massachusetts seemed like a rich and viable environment.

We easily transferred our real experiences at Universities and businesses into this fictional ground and this allowed us to tell of these experiences and lessons learned, without compromising client confidentiality.

We apologize to any Lovecraft fans reading this book because we have taken some liberties with the mythos! In particular, the town of Innsmouth has been gifted with a new public library and certain Lovecraftian characters have been gathered from different times and places and engaged at Miskatonic University.

Lovecraft inspired and encouraged a circle of writers such as Ramsay Campbell, August Derleth, Clark Ashton Smith, Charles Stross, Neil Gaiman and others to play in the fictional world that he created and to flesh out its structure. We are honored to join them at play.

Why choose a University as the setting for our worked example? Firstly, a lot of our readers are likely to be students or to have been students in the not too distant past. The environment is therefore familiar to them. Secondly, a University provides the opportunity for many different systems that can be limited in scope to make the problems tractable in a book such as this. Our worked example, OLAS, although limited in scope, demonstrates real features that may be directly applied to many larger business systems.

OLAS

The OLAS example has been very carefully engineered to illustrate as many of our secrets as possible.

It appears, at first glance, to be a simple library management system. But the devil is, as they say, in

the details and, as you will see, OLAS unfolds to become a surprisingly rich problem.

We have used OLAS very successfully with with many of our commercial clients and with hundreds of students in different Universities over the course of many years, so OLAS has been thoroughly road tested! OLAS is purely for instructional purposes. If you are looking for a book of models for business systems, do check out our “Enterprise Patterns and MDA” [Arlow 2].

But is it science?

Neuro Linguistic Programming (NLP), which we discuss in Chapter 1, is one of the cornerstones of this book. As we present it, it is a powerful set of communication techniques.

Prior to publication, one of our reviewers raised the interesting question "Is NLP scientific?" It's an interesting question because, although it sounds like a reasonable question, it is in fact meaningless until qualified (see Chapter 7). What particular theory of science should provide the evaluation criteria? And what specific aspects of NLP should be considered?

If we assume that by "scientific", the reviewer means "backed up by an empirically verifiable theory", this strong definition excludes most of modern psychology, a lot of linguistics, many of the softer parts of computer science and even some physics (e.g. String Theory).

For example, just think about the "science" of software requirements engineering or software project management. Is a complex application such as Microsoft Word formally verifiable, or is our faith (or lack thereof) in its operation really a matter of pragmatics and experience?

If we assume this strong definition of "scientific", NLP is such a broad field that the argument can, in fact, go either way depending on which aspects of NLP you consider. As in many of the softer sciences, those research results that exist are often patchy, ambiguous and sometimes contradictory.

Doubtless, much of the modern self-help content of NLP is "unscientific" even by the weakest definitions of the term. But it's always worth remembering that "unscientific" doesn't mean "doesn't work".

Going back to basics by cutting through the NLP self-help hype, we are left with a powerful set of techniques for effective communication. It is these linguistic patterns and techniques that we focus on here.

We are pragmatic software engineers and ultimately, if something works for us, we use it. We'll worry about *why* it works once we've got the system in on time and on budget.

The "spells" we present are are definitely *not* scientific and that's why we call them spells. They are just a bit of fun that use some of the more effective "self-help" aspects of NLP. Think of them as

techniques and metaphors for manipulating your neurology for fun and profit. We encourage you to try them out!

We have clearly marked each spell so that any of our readers who prefer science to sensation can skip over them with no harm to either their world-view or to the rest of the text. Everyone else can enjoy them.

The appendix

The appendix, "Meta magic", introduces the idea of hypnosis and trance as valid tools for analysis. It is speculative, incomplete and (to some) controversial. The limited "yes/no" mind-set that likes things in neat little packages, complete and according to conventional wisdom, would perhaps prefer it if we hadn't included this appendix because it raises many more questions than it answers. However, "yes/no" is often insufficient for real world problems, few, if any, things in this world are ever really "complete", and if we always adhered to conventional wisdom, there could be no progress.

We have always felt that the best books both answer *and ask questions*. In this short appendix we want to share with you some really speculative ideas about OOA that might spark your imagination and creativity. We want to take you right to the edge of *our* map so that you might move beyond it, discovering, or perhaps, creating, new territory and extending the domain of OOA.

In the words of the old map-makers, "here be dragons".

Learning features of the book

The Chapter summaries in part 1 are one of the key learning features of this book. In these, we distill all of the important information from each Chapter into a series of headings, each with a brief description, to provide orientation, and then a sequence of short bullet points. We take this approach for four reasons:

- To provide a ready reference in which you can find all the key concepts from the Chapter presented very, very concisely. The focus in on information rather than on words.
- As a revision text. We're aware that this book is likely to be used in Universities, and so we have summarized all of the key points from each Chapter for revision. For a very efficient way to revise, read the Chapter introduction and then the summary, and if there is anything you don't understand, go back to the main text.
- As a source of presentation materials. Although we can provide training courses to go with this book, we are aware that many lecturers like to create their own materials from scratch, or just want to incorporate some, but not all, of our material into their own lectures. The Chapter summaries will get you off to a flying start with this.

- As a source of questions. Because the key points are pulled out and summarized, it's very easy to reword them as questions.

We used this approach in "UML 2 and the Unified Process" and it has proven to be extremely popular. We hope you find that it works well for you here!

Please note that in the OLAS Chapter summaries we do not reproduce the models, because that would be too repetitive. Instead we restrict ourselves to key learning points from the Chapter.

Secrets of analysis training course

We are working on a set of training materials to complement Secrets of Analysis. These are available for licensing and available free for Lecturers for use in Universities.

For details, contact us via our website:
www.secretsofanalysis.com

But you don't cover ...

Well, we can't cover everything! We've focused on the secrets that we have *personally* found most useful over the years - we're sure there are many others! Also, we have limited our scope to those techniques that we have actual experience in. If we have no direct experience with something, we feel we have little to say about it - so we omit it, or just reference it in passing.

Actually, an analysis text must touch on so many topics that completeness is impossible within the time and space constraints of a single book. For example, the topic of requirements engineering could very easily be expanded into several books, each at least as thick as this one.

Because of of the sheer size of the analysis domain, our approach is to give you a very select set of tools that works extremely well for us, and that is complete enough to do the job. Doubtless, some of you will find your favorite tool missing from this set.

If there is a tool, secret, pattern or strategy that *you* find particularly useful, that you would like to share with others, and that we haven't covered here, we would love to hear from you!

Finally...

Ila and I sincerely hope you enjoy this book and find it useful and valuable. We have done our best to make the ideas presented herein as simple as possible for you.

We hope you will check out some of our other books and our training courses. If you have any feedback about this book, or any questions, we will always be delighted to hear from you!

Jim Arlow,
Isle of Wight, 2015

www.clearviewtraining.com

www.linkedin.com/in/jimarlow

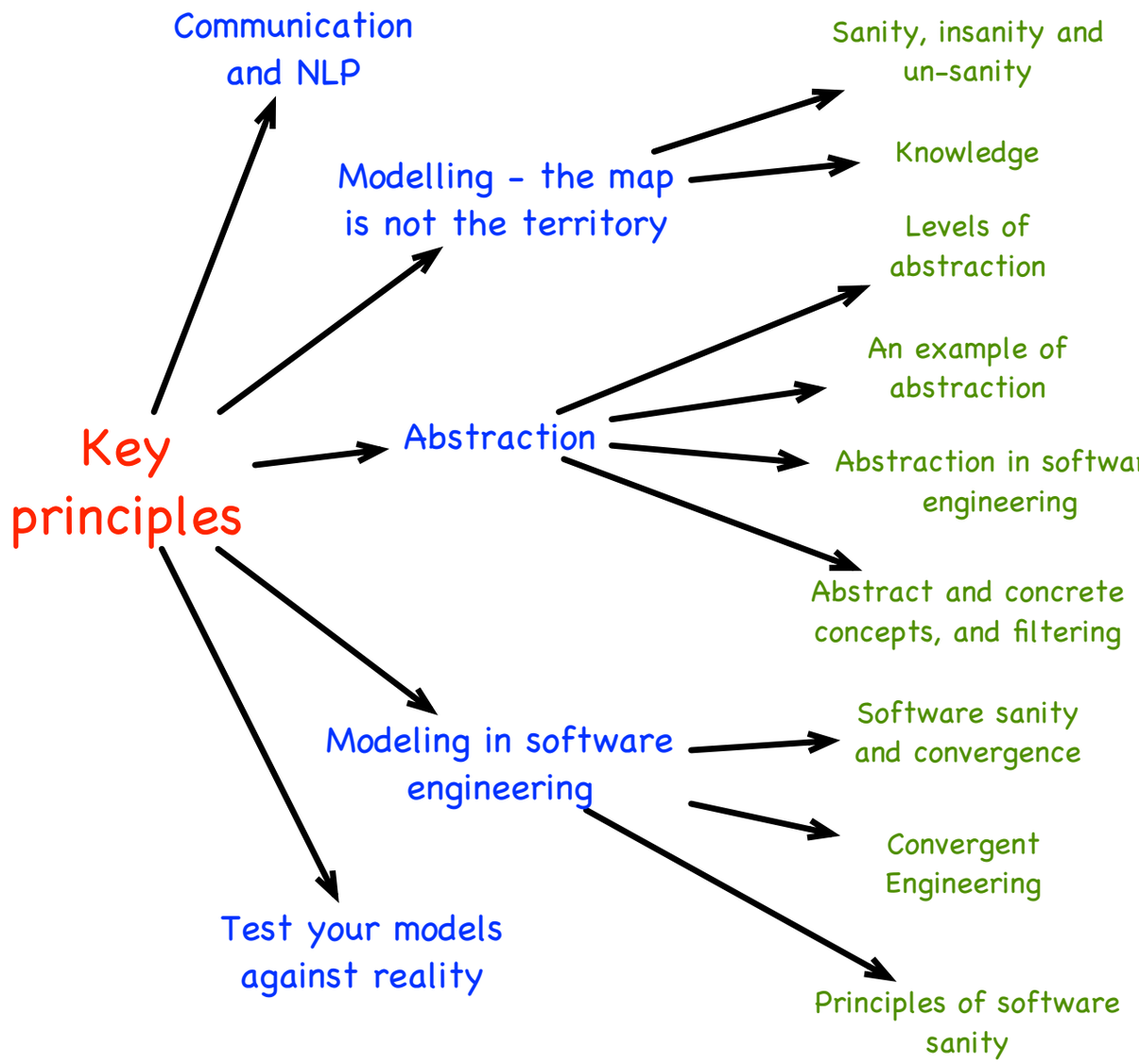
Jim and Ila

Chapter 1

Key Principles

In this chapter we are going to introduce some key principles. Firstly we will consider communication and NLP, then modeling – creating models of software systems. Next we will examine abstraction – the fundamental ability that drives all modeling. Finally we will introduce criteria for well-formed and useful models.

Figure 1.1 Learning objectives



Communication and NLP

Software engineering is, first and foremost, about communication.

One of the cornerstones of this book is Neuro Linguistic Programming (NLP) as a source of effective communication strategies for OO analysts.

This field was invented in the early 1970s by Dr. Richard Bandler, a mathematician, and Professor John Grinder, a linguist, at the University of California at Santa Cruz. They were working in collaboration with the social anthropologist, Gregory Bateson. So early NLP actually has quite solid academic credentials.

Neuro Linguistic Programming explores how the mind (*neuro*) is influenced by (*programming*), specific *linguistic* patterns.

In “The Structure of Magic” [Bandler 1], Bandler and Grinder analyze patterns of therapeutic language and present that analysis as the NLP Meta Model – a model of human communication.

In later work with the hypnotist Milton H. Ericson, Bandler presented the Milton Model – a model of hypnotic patterns of communication [Bandler 2].

Nowadays, you are more likely to be familiar with NLP as a "self-help" technique rather than as a communication science! To some extent, the field has been hijacked by self-help gurus. This is a shame, because NLP has important things to say about communication, which is after all, the basis of every software engineering activity.

From our particular perspective as analysts, NLP provides a rich set of communication strategies, the NLP Meta Model, that enables us get the accurate, high-quality information we need to do our jobs. This is primarily how we will be using NLP here.

In this book, we have extended the NLP Meta Model of communication into M++ which is adapted for OOA. This model gives you *specific* techniques for improving *any* communication, but is customized for analytic purposes. Our view is that excellence in communication, is an essential skill for every analyst.

Today, there are many different approaches to NLP. Our approach is based on the initial work by Bandler and Grinder. Being a hypnotist, Bandler regards trance as the underlying mechanism of NLP - and so do we. This is why trance, which is an unusual topic for a software engineering book, is discussed here in some detail in relation to effective communication. To be clear from the outset, by "trance", we simply mean your attention narrowing and focusing on a restricted set of (usually) internal events.

Modeling - the map is not the territory

The key concern of this book is modeling, and in particular, object-oriented modeling. In this section we set out exactly what we mean by modeling and define what we mean by good and bad models.

To model is to create a representation of something that captures certain features and ignores others. It is a process of abstraction.

We are all modelers, by virtue of how the human brain functions: we receive a limited amount of sensory data, and construct some internal representation of the world according to that data. This internal representation is shaped not just by sensory data, but also by purely "internal" factors such as our presuppositions about the world that arise from past experience.

Figure 1.2 The map is not the territory

We generally act and react to this internal representation as though it *is* the world even though it *is not* - it really is just a very personalized representation.

NLP has a lot to say about modeling. In NLP we say the map (your constructed, internal model of the world) is *not* the territory (the world itself).

This statement originally came from General Semantics [Korzybski 1] in which Alfred Korzybski stated four premises about maps:

1. The map is not the territory.

2. No map is a complete representation of its territory.

3. Maps can be mapped (meta-models of the world).

4. Every map at least says something about the map maker.

A good way to view these maps is as cognitive models of the world that are neither correct, nor incorrect - only more or less useful in a particular context.

Why do we say these maps are neither correct nor incorrect? This is simply because there is no universally accepted decision procedure to determine the correctness of a cognitive map. It is doubtful there ever will be.

Such a procedure, if it existed, would, by definition, give us access to absolute truth – the holy grail of philosophy.

Sanity, insanity and un-sanity

Clearly, we use these cognitive maps, these mental models to make our way (navigate) the real world. The usefulness of these maps for navigation is strongly related to how well they actually map on to the features of the real world. In other words, how isomorphic they are with the real world.

For example, if someone has a mental model of the world in which they can fly unaided, they will certainly be disappointed the first time they jump off a tall building. Such a model is dysfunctional due to it's poor mapping to the real world. It is not a useful model.

Korzybski introduces the term un-sane to refer to a dysfunctional mental model that has been adopted by someone who is considered to be clinically sane.

Compare the term *un-sane* to the term term *insane*. Insane specifically refers to a pathological state of an individual, and it is often also applied to the distorted maps arising from that state.

According to Korzybski, the distinction between un-sane and insane is a crucially important distinction. This is because each of us has the potential to generate an un-sane map now and then, but we are not therefore insane.

It is precisely because un-sane maps *don't* arise from some sort of pathology, that we have the possibility to recognize and correct them when we realize that they don't map on to the territory particularly well.

Knowledge

Based on this metaphor of maps and territories, a useful definition of knowledge from General Semantics is:

"Knowledge - the structural similarity between your nervous-system constructed maps and what you are mapping" [Kodish 1]

If the mapping is good, you can be said to know something about what you are mapping - the territory. If it is bad, the only thing you know about is the map itself!

This provides us with an operational definition of sanity:

Sane maps have a high degree of isomorphism with the territory, un-sane maps do not.

Un-sane maps are generally not useful.

Abstraction

The process of creating a model is a process of abstraction. You capture certain details of the world that you consider to be relevant, and you ignore others. We *have* to abstract, because the world is so dense in information that our limited cognitive capacities would otherwise be overwhelmed.

We are all born abstractionists by virtue of the way our senses operate (Figure 1.3).

Some event in space-time creates an energetic impact that triggers nerves in a sense organ. This electrochemical activity triggers further activity in our nervous systems that we experience as the higher-order abstractions of thinking and feeling. These higher-order abstractions can be further abstracted to be expressed verbally.

Ultimately, there is an "I" (not shown) that is aware of these things.

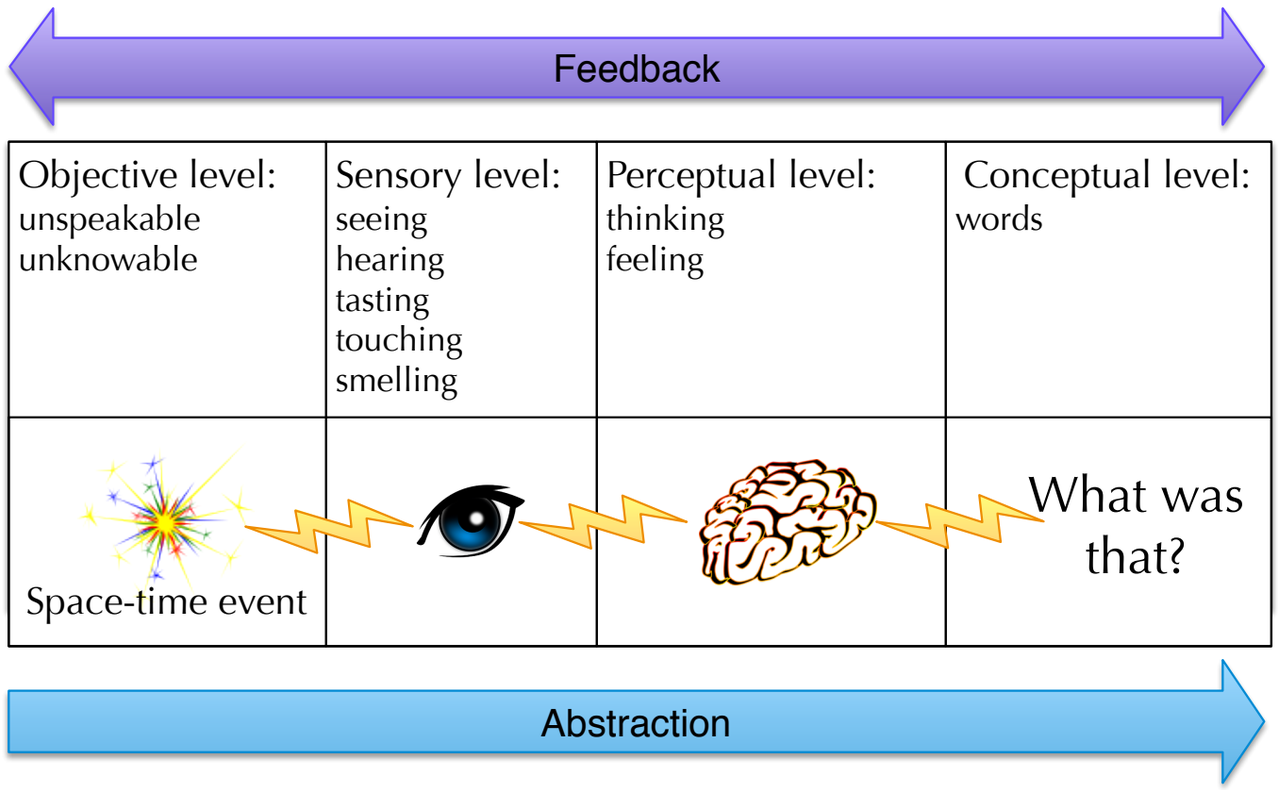
How energy is converted into the subjective experience experienced by the "I" (the qualia) is still one of the great unsolved problems of science. It is known as "the explanatory gap", or the "hard problem".

An even harder problem is that of the nature of the "I": what is it that experiences the the qualia? This problem is so hard, that some scientists try to sweep it under the covers by stating that the "I" is illusory. But if that is the case, what is it that experiences the illusion? The "I" remains unavoidable yet ungraspable.

As the space-time event is transformed by your nervous system, important information is conserved, and irrelevant information is discarded. This is precisely the process of abstraction.

For example, if you touch a hot cup of coffee, your

Figure 1.3 Abstraction in mental modeling



nervous system discards most of the details of the actual physical event. What you actually experience is an abstraction, the feeling of heat, rather than experiencing the specific thermal motions of the atoms in the cup. The feeling of heat is useful to you from a survival perspective whereas detailed knowledge of atomic motion is not.

This is also a process of enrichment - a stimulus triggers other stimuli that create a matrix of signs in which the original stimulus is enriched with meaning. Again, this process is not understood.

One of the most significant features that Figure 1.3 illustrates is that you are not merely a passive consumer of information. The senses are not mere windows "looking out" onto some objective reality. Rather the senses actually play an active part in *constructing* what you perceive as reality.

Furthermore, the abstractions we create from processing sensory information actually affect the process of abstraction itself in a very complex feedback loop.

This process of abstraction ultimately generates space-time events which interact with the objective level. For example:

- You see a splash of green and yellow.
- You interpret it as a flower.
- You recognize the flower as a dandelion.
- You remember that a dandelion is a weed.
- You tell yourself "weeds are bad".
- You pull up the weed.

Levels of abstraction

One of the most notable differences between human beings and other species is that we have a much more developed ability to abstract at the conceptual level.

Experiments with dogs [Pavlov 1] suggest that they can abstract to at most two or three levels. Pavlov explored this by ringing a bell (or using some other signal) just before his dogs were fed, and noting if they salivated. He found that they could easily associate the sound of the bell with food, and would salivate when the bell was rung even of no food was forthcoming. This amounts to one level of abstraction:

signal A->food one level of abstraction

So much is commonly known. It is not so commonly known that Pavlov also experimented with stacking signals thus:

signal A->B->food two levels of abstraction

signal A->B->C->food three levels of abstraction

He found that dogs could cope with at most two or three levels of abstraction, and then the chain of association was broken.

Unlike dogs and (we assume) other animals, human beings can apparently abstract to an arbitrary level. It is this ability to abstract that enables us to reason about the world, to reason about reasoning, to construct OO models and to create software systems. The human mind appears to be a self-modifying abstraction machine.

As we've already mentioned, a crucial aspect of this process of abstraction is the element of feedback. Our higher-order abstractions influence the lower-order processes of abstracting. Thus, words can change minds and can influence perceptions, feeling and thinking. In extreme cases of deep trance or mental disorder, words can even generate hallucinatory sensations leading to maps that are detached from the objective level (reality).

An example of abstraction

A truly great example of abstraction is the London Underground Map. Sadly, we can't show you a picture of it here, because it is copyright (and quite expensive to license!). However, you can Google it, or use this link to the copyright holders website.

The London Underground map was designed by Harry Beck in 1931 and has been so successful that his approach has been copied by metro systems all over the world. Harry Beck was an electrical draftsman, and he applied the metaphor of a circuit diagram to the layout of the London Underground, expressing it as a series of arcs and nodes.

This famous map bears only slight relationship to the actual geography of London.

Distances are compressed and expanded, stations are moved to align in straight lines. The Thames follows an imaginary path. The map contains no details of the actual street plan of London because none of these details are relevant for navigating the underground system - they have been abstracted away.

In this case, the map is clearly *not* the territory, and this is *precisely* why it is so useful in its intended context. It creates an abstraction of the London Tube system that we can easily understand and navigate. One that is abstract, precise and fit for purpose.

However, for a map, context is everything. The many tourists who try to use the map to take a walking tour of London are often left confused and lost.

Beck's process of abstraction may be summarized as:

1. Find a metaphor for the map.
2. Include only those details pertinent to an underground system.

Abstraction in software engineering

If you read ISO 10746-2, the Reference Model for Open Distributed Computing (RM-ODP), you find that abstraction is:

Abstraction - "the suppression of irrelevant detail". ISO 10746-2

That isn't a bad definition, but it's rather too abstract for us! We like to define abstraction more concretely as:

Abstraction - "the selection of important details within a particular context".

From an NLP/General Semantics perspective, we can define an abstraction as:

Abstraction - "a partially complete map that maintains sufficient similarity to the territory to be useful in a specific context".

It is very important to realize that a map is only useful within a specific context. For example, Harry Beck's London Underground Map is only useful within the context of navigating London by tube - it's useless for planning a walking tour of London, as many tourists have found out to their cost.

The "important details" are the details that make a difference within this context. In this case it is the stations, and the different lines connecting those stations.

The best maps are abstract, precise and fit for purpose, and the London Underground map is a perfect example of this.

It is *abstract* - all details not pertaining to the tracks and stations have been elided.

It is *precise* - within its domain of application, it is completely accurate. No lines or stations are omitted and everything is in an appropriate relationship to everything else.

It is *fit for purpose* - it is easy to read for humans, and tells us just what we need (and no more) to navigate the underground system.

Many beginning modelers confuse abstraction with "approximation" or "imprecision". But nothing could be further from the truth! To be of maximum utility, an abstract map needs maximum precision.

For good examples of very abstract UML models that are also very precise, take a look at the archetype patterns in "Enterprise Patterns and MDA" [Arlow 2].

In OO analysis and design you are trying to uncover sufficient information about stakeholder needs and their business domain so that you can design a software system that delivers business value.

For example, imagine that you are designing a banking system. You are deeply interested in representing the amount of money each customer has in their account, but not at all interested in their preferences for shoes or fear of spiders. This is the art of abstraction in software engineering - finding those details that make a difference within the business

context of your software system. How good you are at this task depends on:

- How well you can collect the information you need (we'll look at many ways to do this later in this book).
- How well you can analyze the information you collect (again, we'll give you tools to do this).
- How well you can apply the principle of convergence (see Section 1.5).

Abstract and concrete concepts, and filtering

Some concepts in the business domain refer to things that are quite concrete - you can go up to them and kick them. An example might be a card file in a library.

Other concepts are more abstract - you can't kick them. An example might be the date a book is due to be returned to the library, or a security profile.

You can often learn about the concrete concepts in the problem domain just by studying them. You may be able to examine them directly to see what their attributes and behaviors are.

The abstract concepts are more problematic. Typically, these only exist in someone's mental model, in someone's map of the world. To uncover these concepts you need to understand how to explore and interrogate this map. This is what we hope to teach you later in this book.

The key lesson you have to learn however is that, as we said above, the map is *not* the territory. A stakeholder will typically have a mental model of the business domain that is approximate. In other words it is the result of a filtering process that we discuss in more detail in Chapter 7. This filtering process is characterized by:

- **Distortion** - the map is not accurate, it has hallucinatory elements.
- **Deletion** – the map is not complete, information is missing.
- **Generalization** - specific details of the key concepts have been removed from the map and replaced by rules and beliefs.

People's models of the world are generally *just good enough* to get the job done. In other words they are *efficient*.

For example, most stakeholders typically have a view of the business domain that is centered on their role

in it. That's fine for them - it is an efficient approach that emphasizes their contribution. However, from the point of view of you, as an OO analyst, it is problematic because their point of view will be partial and may be very badly generalized and distorted.

So in order to create an accurate model of the business domain you need to be able to elicit accurate information from stakeholders who only have approximate maps.

You can do this by exploring their maps in very specific ways that we will discuss later in this book so that you can recognize and recover information that has been distorted, deleted, or generalized. This higher quality of information allows you to construct your own mental map of the domain that, through the process of analysis, you turn into models in UML. The mental map you construct has to be very precise because there is little room for imprecision in software engineering. We will present you with the techniques you need to do this in Chapter 7.

Collaborative maps

Up to now, we have spoken as though the OO engineering process is a solitary activity. However, you are virtually certain to be working in a team, and so your team will create a collaborative map of the problem domain that is spread across more than one individual. Therefore, another key skill is the ability to communicate your internal maps to each other in a clear, effective and unambiguous way.

In our view, being able to elicit and communicate accurate information is one of the keys to successful OOA so we'll spend a *lot* of time on communication strategies in this book.

Modeling in software engineering

Most of the models we create in software development are neither correct nor incorrect - they are only more or less useful in a particular context.

Clearly, there are cases, generally when we are modeling algorithms, where definite criteria exist that can be used to assess the correctness of a model. However, for many business systems, there are no such definite criteria.

We take it as axiomatic that the degree of usefulness (sanity) of these models depends on how well they map on to the "real world", which in this case is the problem domain.

In the next subsection, we set out a more precise notion of software sanity - the usefulness, or otherwise of a model of a software system (please note that we consider source code to be just another model). This notion of software sanity provides us with:

- Useful guidelines for creating models
- Useful guidelines for assessing models

- An understanding of why Convergent Engineering is the key to building good models

SOFTWARE SANITY AND CONVERGENCE

If we define "sanity" as the degree to which a map or model is useful in a particular context, then we can talk about sanity in software systems. And it turns out that this is a very productive thing to do.

There are two aspects to the sanity of a software system (Table 1.1):

- **Black-box sanity** - how well the external behavior of the software system corresponds to the users' needs. How useful it is to them, how easy and intuitive it is to use.
- **White-box sanity** - how well the internal structure of the software system maps on to the

Table 1.1 White-box/black-box sanity/un-sanity

	Sane	Un-sane
Black-box	• Does what you need it to do in a way that is intuitive and easy to understand. • "Feels right" because it seems to be using the same map of the world as you are. • Black box sanity is a necessary condition for software to interact successfully with human beings.	• Seems to be working according to an entirely different map to your own. • Behaves in a way that is un-intuitive and (to you) illogical, un-sane and hard to understand. • May or may not do what you need it to do. You may, or may not be able to get it to do what you need it to do, even if it can.
White-box	• Has an internal structure that is clearly isomorphic to the structure of the problem domain. • Because of this, its internal structure is understandable by domain experts. • White-box sanity is not a necessary condition for software to interact successfully with human beings.	• Has an internal structure that may be isomorphic to the structure of the problem domain, but this isomorphism is very complicated. • Is not understandable by domain experts.

mental maps of experts in the problem domain. How easy it is to understand.

We have all encountered sane and un-sane software systems, and we're sure you have your own favorite example of each.

White-box software sanity is a purely subjective value judgment that is useful only as long as humans are involved in developing software.

White-box sanity is a necessary condition for domain experts and software engineers to be able to understand the internal structure of a software system easily and effectively. But it's important to note that it is not a necessary condition for working software. Software systems may still work (and they often do!) even if they are white-box un-sane.

Try it now! *Software sanity*

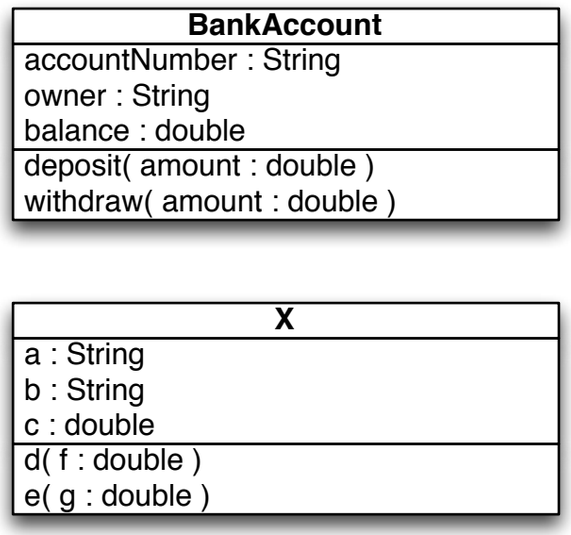
Think of a software system that you consider black-box un-sane. List the key features that make it so. Be as specific as you can.

Think of a software system that you consider black-box sane. List the key features that make it so. Be as specific as you can.

Repeat the exercises for white-box un-sane and white-box sane systems. Can you abstract any key principles for good software from your results?

To appreciate this point, consider the models in Figure 1.4. These models are functionally identical,

Figure 1.4 Three isomorphic models



and structurally isomorphic, but only one has a structure that you can intuitively understand. Only one of these models is white-box sane, but all the models could be made to work.

It's important to note that although this is a book about OO modeling, these principles of software sanity apply just as well to non-OO systems. Provided variables and functions are named appropriately, non-OO systems can be both black-box and white-box sane.

Some of the most useful software systems (e.g. those based on neural nets) are white-box un-sane by our definition. There is an isomorphism between their internal structure and the structure of the problem domain, but the mapping is complex, and is not generally meaningful or understandable by domain experts. This is one of the reasons that neural net systems can be hard to develop.

Convergent engineering

White-box sanity can be considered to be a restatement in terms of NLP and General Semantics of the principle of Convergent Engineering (that the structure of the software system should match the structure of the business [Taylor 1]). The principle of white-box sanity, or convergence, is the guiding force for all of the models that we create. We have found time and again, that the more convergent our models are, the saner they are, the more useful and effective they prove to be. In fact, over the years, convergence has become our primary concern because it works so well.

Our experience is that systems that are white-box sane *naturally* tend to be black-box sane. Likewise, systems that are white-box un-sane have a tendency to be black-box un-sane.

This appears to be because the external behavior of a software system is predicated on its internal structure - the more sane that internal structure, the more likely that the system will behave in an externally sane way.

There is also a key human factor here - software engineers who know about, care about and focus on white-box sane software will tend to apply the same criteria to the black-box aspects of the system.

Principles of software sanity

Now that we have a working definition of software sanity, we can set down its principles:

- The *key abstractions* in the problem domain are specified in models and realized in software. For example, in a banking system, classes BankAccount, Bank, Party, etc. appear in the problem domain, in the models and in the software
- The *internal structure* of the key abstractions in the problem domain are specified in models and realized in software. For example, in a banking system, the Bank abstractions might have attributes accountNumber, accountName, balance
- The *relationships* between the key abstractions in the problem domain are specified in models and realized in software, e.g. a Bank contains many BankAccounts
- The *behavior* of the key abstractions in the problem domain are specified in models and

realized in the software, e.g. the BankAccount class may provide a withdraw method

When we talk about the behavior of a key abstraction, this is shorthand for the business functions that the abstraction supports.

Obviously, the idea of a "bank account" is an abstract notion (you can't kick it) and so doesn't have any real-world behavior as such. In a manually operated business, this abstraction is realized as information recorded in some way and rules about how that information may be manipulated and this constitutes its implicit behavior.

However, when this abstraction is realized in a software system, it *must* be assigned explicit behavior to support all the business functions in which it participates. For example, let us suppose that the "bank account" abstraction must support the business functions of:

- Maintaining an account balance
- Withdrawing money from the account
- Depositing money into the account

These business functions imply structural and behavioral features of the abstraction that we must include when we realize it in a model as a BankAccount class. Thus, it needs an attribute to hold the balance, and operations to deposit and withdraw money from that balance.

The structural features are often fairly obvious - you analyze the key concept and you find that a "bank account" has parts - a balance, owner, name, etc. The behavioral features are not quite as obvious. One way to arrive at them is to imagine that you *are* the abstraction. What services do you need to provide to support the required business functions?

Anthropomorphizing abstractions helps you to realize them in OO models. We'll look at many more specific ways to find the structural and behavioral features of abstractions later in this book.

Try it now! *Anthromorphization*

Imagine you are a library system. What services would you offer to the librarians that use you? Make a list.

Test your models against reality

Every model is a theory about the world. The thing about theories (as opposed to hypotheses) is that they are testable. According to Korzybski,

---

"theories are the rational means for a rational being to be as rational as he possibly can" [Korzybski 1].

This brings us to a key guiding principle for all types of modeling:

Test your theories against reality at the earliest possible opportunity.

A key test of a model is to apply the principle of convergence (white-box sanity). How well does the structure of your model map on to the structure of the problem domain? Here are some simple techniques you can use to find out:

- Try presenting your model to domain experts and talking them through it. If the model is convergent (white-box sane) they should be able to understand it.
- Write a Literate Model for parts of your model. As we explain in Chapter 9, a Literate Model is part of your model embedded in an explanatory narrative that uses the terms defined in the model (e.g. "Every BankAccount has a balance."). Does the narrative make sense? Does it read well? Can domain experts and other stakeholders understand it?
- Create a project glossary for the key abstractions in the problem domain. Can you find these abstractions in your models? If not, why not? Do the abstractions have the same meaning in the project glossary and in your models?

Of course, the ultimate way to test any model is to execute it! This is what you do with your mental models of the world. You act as though they are true, and execute them in the real world. The feedback you get tells you how useful your models actually are.

If you have the behavioral flexibility to *change your models* according to the feedback you get from executing them then you can succeed at almost anything.

With tools such as xUML (www.kc.com) you can create executable UML models. We hope to see many more such tools in the years to come. They provide the ultimate in feedback.

Without tools such as these you can always realize the model (or part of the model) as software - this is what is known as a "behavioral prototype". It is a prototype specifically designed to test the behavior of a key part of the system. You may even be able to generate a significant amount of code from the model itself.

Creating a model is a bit like driving a car - you continually make small course corrections until you get to your desired destination. You can only know what corrections to make by knowing where you actually are at any point in time (rather than where you think you are), and where you are trying to get to. So the sooner you can test a model, the sooner you can take corrective action if that is necessary.

Perhaps the worst thing you can do is to defend a model that just isn't working very well. This is a bit like driving a car, going off the route, and yet continuing on because you are convinced your mental map is right despite what the world is telling you.

We've encountered this unfortunate habit quite a lot over our years in software engineering. It's completely understandable from a human perspective - you might have put a lot of time and effort into creating a model and thereby have become emotionally invested in it.

You find that the tendency to defend a broken model is common in circumstances in which there are no established criteria for assessing models. This is why we have spent some time in previous sections defining software sanity in terms of convergence. If you use our criteria, then you will always have a pragmatic benchmark against which to assess models.

In terms of the human factors in modeling, always remember that it's only a model. It has no intrinsic value outside of the purpose for which it was created. Although it does say something about you as a modeler, all it ever really says about you is how much you understand the problem domain and how well you understand UML. You can control and change both of those things. There is (or should be) no element of personal judgement at all and therefore no reason for overloading your models with emotional baggage.

Remember that the best modelers have developed the behavioral flexibility to change or abandon a broken model straight away if that's what the world is telling them to do!

Summary

In this chapter we've set the scene for introducing our secrets of analysis by looking at some of the basic underpinnings of OO analysis. In subsequent chapters, we will apply these principles and elaborate on them.

Here is a summary of the key points of this chapter.

Modeling

Modeling is to create a representation of something that captures certain features and ignores others. We are all modelers because we all create an internal representation of the world. Furthermore, and often problematically, we act as though this representation *is* the world. But *it is not*. We refer to these internal representations as maps:

- The map is not the territory.
- No map is a complete representation of its territory.
- Maps can be mapped (meta modeling).
- Every map at least says something about the map maker.

Maps are neither correct not incorrect – they are only more or less useful in a particular context. Generally, the usefulness of a map is determined by how isomorphic is it with the territory. This leads us to a useful definition:

Knowledge – "the structural similarity between your nervous system constructed maps and what you are mapping"

We can now distinguish between sane maps and un-sane maps:

Sane maps

- Highly isomorphic with the territory.
- Useful.

Un-sane maps

- Only slightly isomorphic with the territory
- Not useful

Bringing this into an OOA perspective, the usefulness of our models depends on how well they map on to the real world (the problem domain).

Abstraction

Abstraction can be a slippery concept. Here are some generally accepted definitions:

"The suppression of irrelevant detail" - ISO 10746-2

"The identification of important details within a particular context"

"A partially complete map that maintains sufficient similarity to the territory to be useful in a specific context"

We are all born abstractionists – the senses abstract from space-time events (the objective level) to:

- Sensory level – the five (or more) senses
- Perceptual level – feeling, thinking
- Conceptual level – words

This process of abstraction feeds back affecting itself at all levels and this means that we are not merely passive consumers of information. We transform and create it as well. Human beings have a very highly developed ability to abstract.

Abstraction, precision and fitness

The best maps are abstract, precise and fit for purpose - they capture those differences that make a difference within a specific context.

Although we might wish otherwise, a project stakeholder will have a map that is characterized by:

- Distortion - the map is not accurate, it is has hallucinatory elements
- Deletion - information is missing from the map
- Generalization - specific details of the key concepts have been removed from the map and replaced by rules and beliefs

People's models of the world are generally efficient – just good enough to get the job done!

In order to model successfully you must be able to:

- Elicit accurate information from stakeholders with approximate maps (handle deletion, distortion and generalization)
- Construct your own mental map that is abstract, precise and fit for purpose
- Be able to transform your mental map into formal UML models

Modeling is generally a team activity that involves creating a communal map. This depends on good communication!

Software sanity

Using this theory of maps and models, we can formulate some useful definitions of software sanity:

Black-box sanity:

- Does what you need it to.
- Is using a similar map to you.
- Intuitive.
- Easy to understand.
- Feels right.
- Is a necessary condition for excellent software.

White-box sanity:

- Structure is isomorphic to the problem domain.
- Structure is understandable by domain experts.
- Not a necessary condition for excellent software.

Black-box un-sanity:

- May or may not do what you need it to.
- Works according to an entirely different map than you.
- Un-intuitive.
- Illogical.
- Difficult to understand.

White-box un-sanity:

- Complicated isomorphism to the problem domain (if any at all)
- Not understandable by domain experts

Convergent Engineering

Stated simply:

The structure of the software system matches the structure of the problem domain.

Test your model against reality

Every model is a theory about the world - test your theories against reality at the earliest possible opportunity! Do this by applying the principle of convergence (white-box sanity):

- Present the model to domain experts and talk them through it.

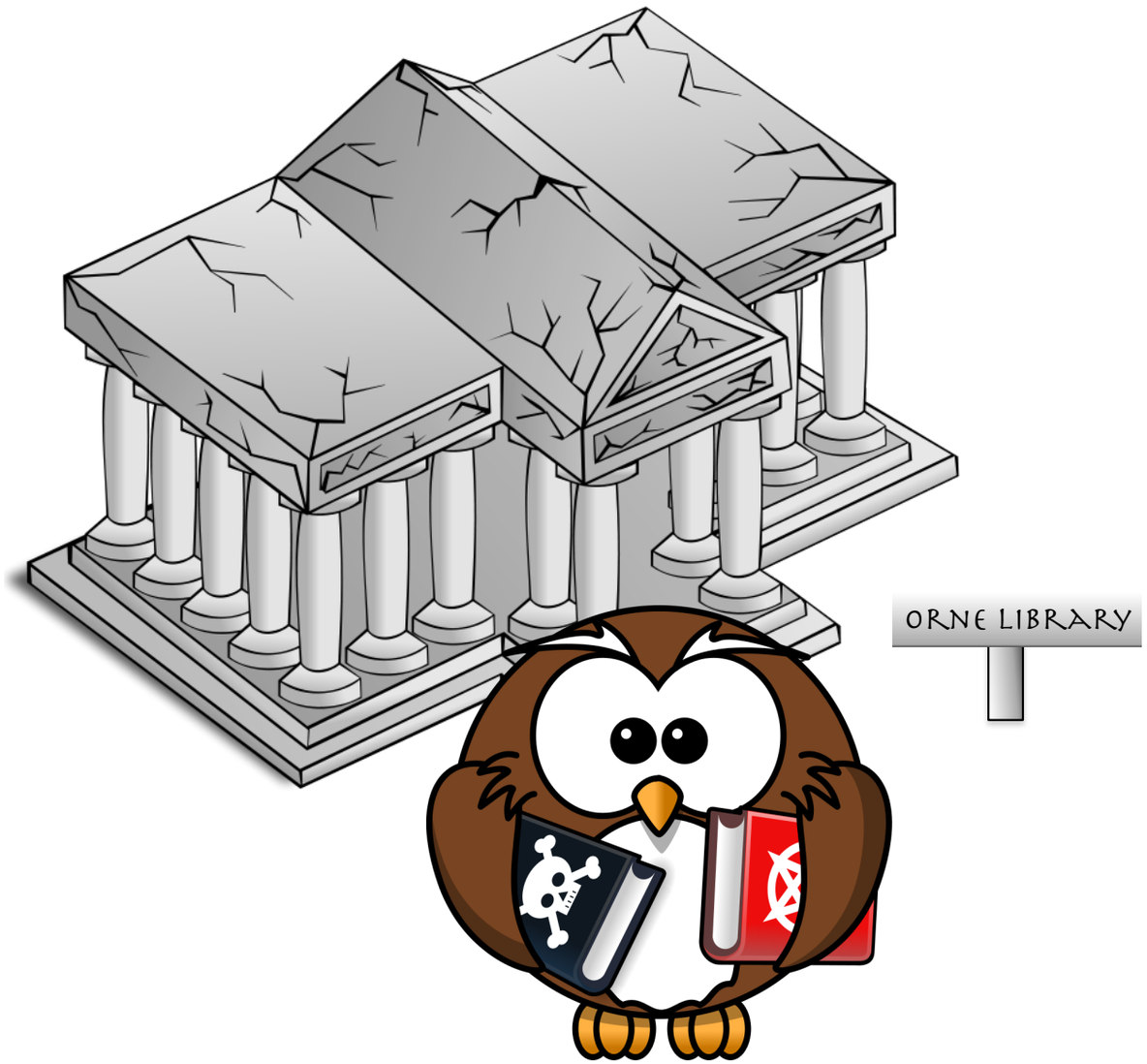
- Create a Literate Model. Does it read well? Can domain experts understand it?
- Create a project glossary. Key abstractions in the problem domain must occur in your models and have the same meaning.
- Execute your models.

Never defend a broken model. Emotional investment is pointless – it is only a model!

- Use software sanity and convergence as criteria for model assessment.
- Change or abandon a broken model as soon as you can!

Chapter 2

Launching the example project



This chapter sets the scene by describing the problem domain that we are going to use to illustrate our secrets of analysis. We are going to discuss three different, but related, topics: the problem domain, software engineering processes and the OLAS Inception phase.

OLAS - the problem domain

We will be taking you through the analysis of a software system for the Orne Library at Miskatonic University. This is the Orne Library Automated System (OLAS).

We have kept this worked example simple so that we can focus on the techniques rather than get caught up in details of the problem. Over the years, we have found that a library system makes a very good example for the following reasons:

- Many people already have a pretty good idea of how a library functions.
- The system is sufficiently simple to make a good example without getting lost in the business or technical details.
- It is rich enough to provide a satisfying problem and exercise all of the key OOA techniques.

OLAS will handle the main library functions of catalog management and loan management. It will also have to apply security (the library restricts access to its restricted collection), and allow communication with the Innsmouth Public Library.

Software engineering processes

When you decide to build a software system, you could just sit down and write the code (and in the early days of software development much great software was developed in just this way) , *or* you can apply a software engineering process.

A software engineering process describes how you are going to go about developing a piece of software. It describes the who, what, when, where and how of building software. We like to think of a software engineering process as being a way of turning stakeholder needs into software. Any reasonably complex project involving a team of developers needs a software engineering process to specify:

- Who – the roles needed in the software development project.
- What – the artifacts to be delivered.
- When – the project plan for when specific software engineering activities have to occur.
- Where - is it a distributed project?
- How – the actual activities performed in the software engineering process.

To understand why you need a software engineering process (and why sometimes you don't), we like to compare constructing software to real, bricks and mortar construction.

For example, if Ila and I want to build a small shed in our garden, we can go to the local DIY store, buy some wood, and knock something together. The result is probably quite serviceable, and we've achieved it with little or no planning and few

organizational overheads. This is the Nike® method – “just do it”®. This method works for very small software development projects that have clear requirements, involve very few people (less than five), and that only take a few days or so to develop by one or two programmers.

The Nike® method is quite widely used. A friend of ours who worked for the traders on the floor of the London stock exchange used to use the Nike® method extensively. A trader would come to him with an idea for a small utility to help them work with a particular financial instrument, and he would be expected to deliver that as quickly as possible.

On the other hand, if we want to build an extension to our house, this is an altogether different class of problem. We need an architect to draw up the plans, a builder to build the extension, a project manager to oversee the whole process, and we need to get the plans approved by our local Council so that the project can go ahead in the first place. This is similar to a small software development project that is building a medium size software application for a business. There will be many people involved, and many activities to plan and coordinate. The software needs to be architected, and there needs to be some process in place to manage the complexity. The Nike® method just won't work here – it can't handle the complexity, and would lead to chaos and, ultimately, failure.

Suppose now that we wished to build a skyscraper, or even a whole city. Again, these are completely different classes of problem, and they require very different processes. Similarly, building the software system for a battleship or for Heathrow Terminal 5 requires very different processes.

As you can see from the above construction analogy, the software engineering process needs to be matched to the project on hand. A simple, short project, usually needs a simple, informal, "low ceremony", software engineering process. If it's really, really simple, you may even be able to use the Nike® method, which is no process at all! However, a more complex project, which might be distributed over long periods of time and multiple geographical locations, requires a more formal, "high ceremony" process so that it can be managed, coordinated and controlled effectively.

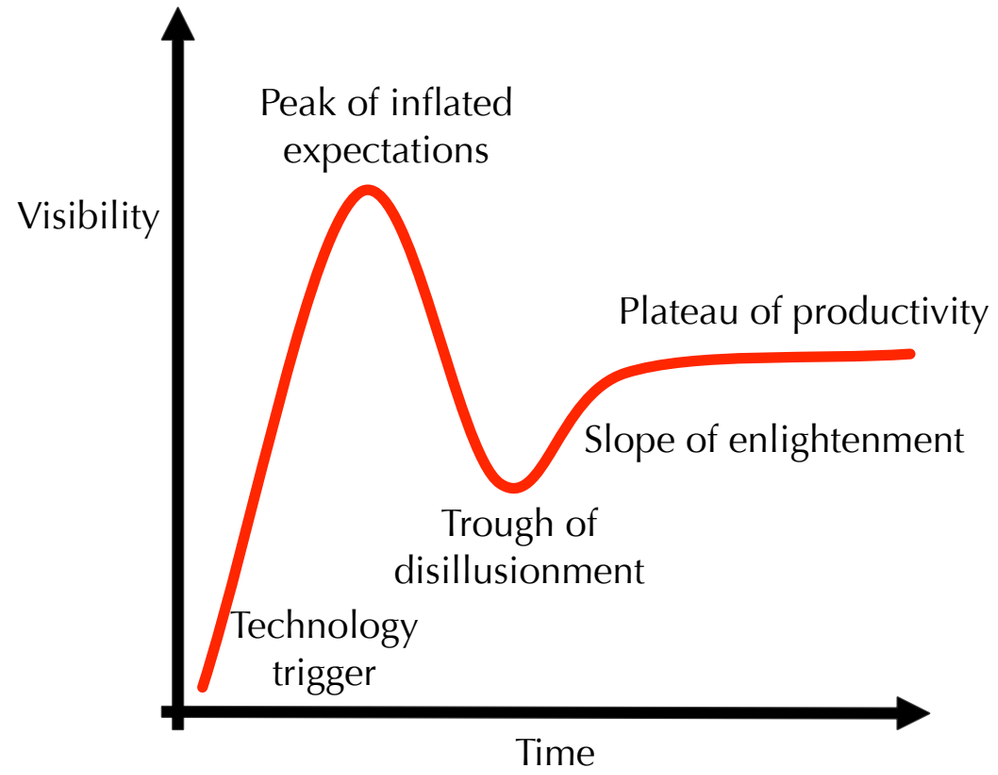
We know of many software development projects that got into trouble by adopting a software engineering process that just didn't fit the requirements of the project. We observe that this is often due to “experts” who inappropriately push their favorite software engineering process as a sliver bullet.

As we go to press Agile “methods” such as XP (eXtreme Programming), Scrum and DSDM (Dynamic Systems Development Method) are in many cases being mis-sold as “the one true way”. However, one thing we have learned over the last fifty years of

software engineering is that there is simply no "one true way" for software engineering processes.

As we descend into the trough of disillusionment of the Agile hype cycle, more and more companies are learning or relearning this unfortunate lesson.

Figure 2.1 Gartner Hype Cycle



If you are not tailoring your software development processes to the specific needs of of specific projects, then you are probably doing something wrong.

In the next section we will briefly look at the Unified Process - a software engineering process *framework* that you can customize to create a software engineering process that's just right for your project.

The Unified Process (UP)

The Unified Process, or UP, is a software engineering process *framework* described in [Jacobson 1]. It is a set of process, role and artifact templates that you can use to construct many different software engineering processes, each suited to a particular project.

To use the UP:

- Determine the process requirements for your project (based on size and complexity).
- Tailor the UP accordingly.

For example, if you are building a fairly simple system such as OLAS, you need a fairly lightweight process. On the other hand, if you are building a very complex system, a more complex process will almost certainly be required.

Tailoring the UP in this way is really a project management issue and is out of scope for this, a book about analysis. However, you will see the process we use for OLAS unfold as we go along.

There are several flavors of UP currently available:

- Unified Process – described in the book [Jacobson 1].
- IBM Rational Unified Process – the UP greatly extended and made more complete. It is a commercial product that comes with artifact templates and is usually packaged with tools and consultancy.
- OpenUP – an open source version of the UP from the Eclipse Project.
- Agile UP - an agile instantiation of the UP.
- Enterprise UP - an extension to RUP that includes Enterprise disciplines such as Enterprise Business Modeling.

We will be using "vanilla" UP after Jacobson.

As we will be using UP for OLAS, we will take a more detailed look at it over the next few subsections. We will only give a summary of the UP here – just enough to work on OLAS. You can find out more details in [Arlow 1] and [Jacobson 1].

Characteristics of a UP project

According to Jacobson, a UP project is:

- Architecture centric – you are focused on the architectural structure of the application (rather than on ad hoc coding) and you are concerned with how the application breaks down into components and what relationships exist between those components.
- Requirements driven – the development of the application is determined by what the users need in order to deliver business benefit as quickly as possible.
- Risk driven - there will be different degrees of risk involved in developing different parts of the application at different points in time. Risks are identified, quantified (e.g. low, medium, severe) and specific plans are made to mitigate them. It's always best to tackle severe risks as early in the project life cycle as you can because this leaves time in the project schedule to take corrective action if needed. Leaving risks to the end of the project is highly dangerous and will typically lead to cost and time overruns and, in the worst cases, project failure.

Requirements and risk often act as opposing forces in a UP project. Requirements may indicate that a certain piece of functionality needs to be developed at a particular point in time, but risk mitigation may indicate that a completely different piece of functionality should be addressed first. It is the role of the project manager and system architect to balance these opposing forces.

Iterative and incremental development

All flavors of UP are iterative and incremental: the project is developed in stages (iterations) where each iteration delivers a useful increment of functionality.

The experience of the last 50 or so years of software development (and perhaps the whole of human history) has taught us that we're really just not very good at dealing with big, complex problems. Iterative and incremental development breaks a big problem down into a sequence of smaller, simpler problems, that we can solve relatively easily.

Each iteration is run as a mini project with all of the activities associated with a normal project. These include:

- Planning.
- Analysis and design.
- Construction.
- Integration and test.
- An internal or external release of an executable architectural baseline.

The amount of time you spend in each of these activities depends on the nature of the particular project. For example, if your project requires an Agile UP variant, you may spend less time in Planning and Analysis and Design and more time in Construction and Integration and Test.

At the end of each iteration, a stable executable architectural baseline is produced that serves as the basis for the next iteration. The baseline is:

- Executable – it is an actual piece of software that can be executed to deliver business benefit.
- Architectural – it embodies the architectural principles and style of the final application. It is not just some ad-hoc solution hacked together to meet a deadline.
- Baseline – it serves as a stable platform on which to add functionality in subsequent iterations. Each new baseline builds on the one before it. A baseline can only be changed through a formal change management process.

Each architectural baseline delivers a useful increment of functionality and, in the best case scenario, this can be released to the user community so that they can begin to see business benefit early in the project life cycle and give important feedback to the developers.

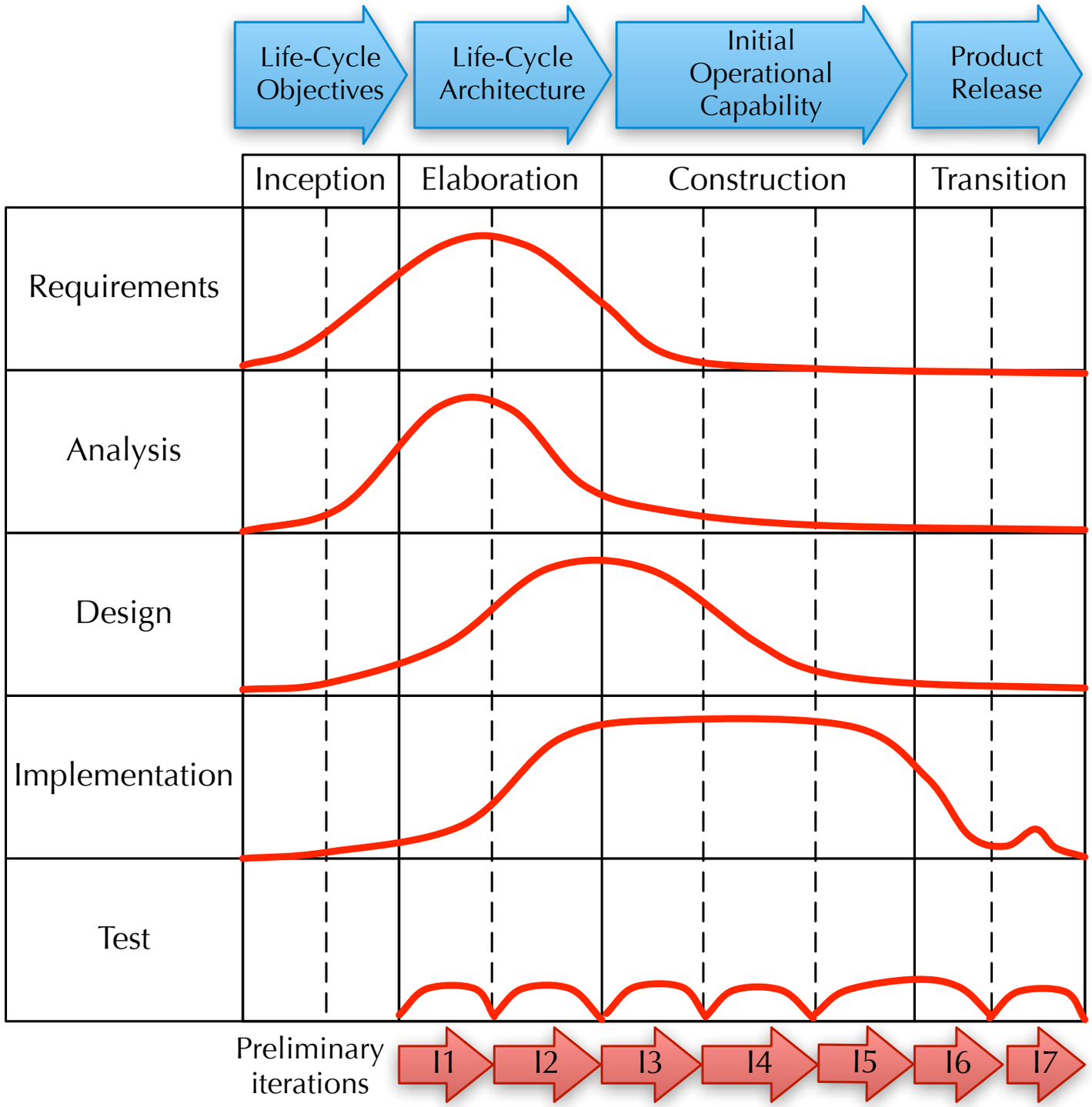
Unfortunately, this doesn't happen very often for purely logistical reasons – imagine the nightmare logistics of rolling out a partial check-in system to the 500 or so check-in agents at Heathrow Terminal 5! Instead, it is often released to subset of the user community, or (worst case) just within the project. However, there *must* be a release of some sort to terminate the iteration.

UP structure

The UP has a well-defined structure comprising:

- **Iterations** – a mini project with all of the activities associated with a normal project.
- Core workflows – definitions of activities such as requirements capture or analysis. These are:
  - Requirements – capturing what the system should do.
  - Analysis – refining and structuring the requirements.
  - Design – realizing the requirements in the system architecture.
  - Implementation – building the software.
  - Test – verifying that the implementation works as desired.
- **Phases** – groups of iterations that constitute identifiable stages in the project. They each end with a major milestone. The phases are:
  - Inception – Life-cycle

Figure 2.2 UP structure



Objectives milestone.

- Elaboration – Life-cycle Architecture milestone.
- Construction – Initial Operational Capability milestone.
- Transition – Product Release milestone.

Because UP is a process framework, *all* of these features are subject to customization to meet the needs of a particular project.

The structure of the UP is shown in Figure 2.2. This is a rather complex figure, but it summarizes the UP very, very well. It's worth taking some time to study and understand it, because it's fair to say that if you understand this figure, you understand a *lot* about UP.

There are five horizontal swimlanes each representing a particular workflow. These workflows are Requirements, Analysis, Design, Implementation and Test. Remember them with the acronym RADIT.

RADIT *- **R**equirements, **A**nalysis, **D**esign **I**mplementation, **T**est*

Within each workflow is a curve that indicates the relative amount of work being performed in that workflow as a function of time. It's important to understand that this curve is only an estimation of the amount of work for a typical UP project of about 18 months duration. It represents the expected case for many projects.

Vertical swimlanes divide the figure into phases, and each phase cuts across all five workflows.

By looking down each phase you can see an indication of the relative amounts of work being done in each workflow in that phase.

For example, looking down the Inception phase, you can see that in Inception there is a lot of requirements gathering, some analysis, a very small amount of design and implementation and no test.

Each phase ends with a milestone that specifies the conditions of satisfaction for completing the phase.

Along the bottom of the figure you can see some iterations (I1..I7). Each phase typically requires at least one iteration, but there may be many iterations in a particular phase depending on the project size.

Each iteration typically executes the five core UP workflows, requirements, analysis, design, implementation and test, to some degree.

Notice that the iterations in the Inception phase are called "preliminary iterations". This is because Inception occurs *prior* to the formal launch of the project and Inception iterations do *not* create executable architectural baselines. There may be some implementation activity in Inception to create proof of concept prototypes, but these are generally

discarded – they are not "architectural" because the system architecture has yet to be finalized.

Each phase ends with a major milestone. Each milestone is a set of preconditions for closing down the phase and moving on to the next. This is very different to many other software engineering processes that close phases when particular deliverables have been produced. Such processes can encourage projects to turn into machines for producing documentation rather than software! UP focuses on software, and documentation is produced only if it delivers real benefit.

In the next few sections we will give you an overview of the UP workflows and phases.

UP WORKFLOWS

UP specifies five core workflows that occur in *every* project and in *every* iteration to some degree.

Obviously, there may be many other workflows. For example a project planning workflow for project managers and workflows specific to a particular project. However these are not specified in UP.

The five core workflows are RADIT:

- **Requirements** – capturing what the system should do.
- **Analysis** – refining and structuring the requirements.
- **Design** – realizing the requirements in the system architecture.
- **Implementation** – building the software.
- **Test** – verifying that the implementation works as desired.

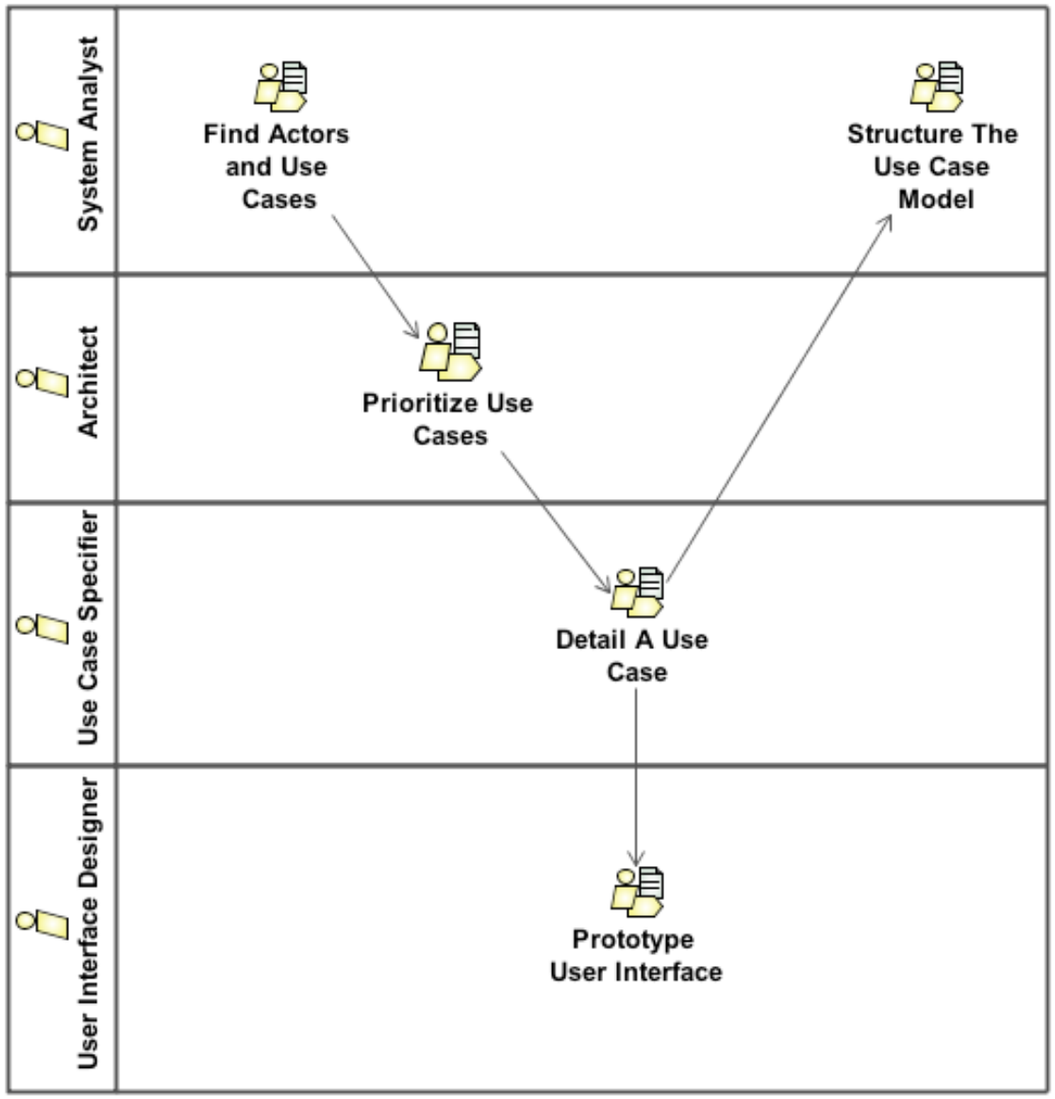
For each workflow, the UP describes the roles (e.g. System Analyst) involved in the workflow, the activities (e.g. Find Actors And Use Cases) that those roles are likely to perform and the artifacts that they are likely to deliver.

An important point to note is that a role is *not* an individual – a role may be adopted by zero or more individuals over the course of a project.

Figure 2.3 shows a UML activity diagram that shows the Requirements workflow as described in [Jacobson 1]. UML is an extensible modeling language and this Figure uses the SPEM 2 (Software & Systems Process Engineering Metamodel) profile to provide a visual modeling language for software engineering processes.

Note that the visual syntax and terminology in [Jacobson 1] and [Arlow 1] is slightly different to that in Figure 2.3. This is simply because these books were written prior to SPEM. However, the concepts are much the same and you will have no problem understanding the corresponding diagrams in these texts.

Figure 2.3 Requirements workflow



The UP gives *guidance*, but is not *prescriptive*, as to what roles project members can play, and what activities those roles need to be assigned at a particular stage in the project lifecycle.

In this book, our focus will be on the Elaboration phase and the Requirements and Analysis workflows.

UP phases

We will now summarize each UP phase in terms of its focus, goals and milestone:

- Figure 2.4 the inception phase.
- Figure 2.5 the elaboration phase.
- Figure 2.6 the construction phase.
- Figure 2.7 the transition phase.

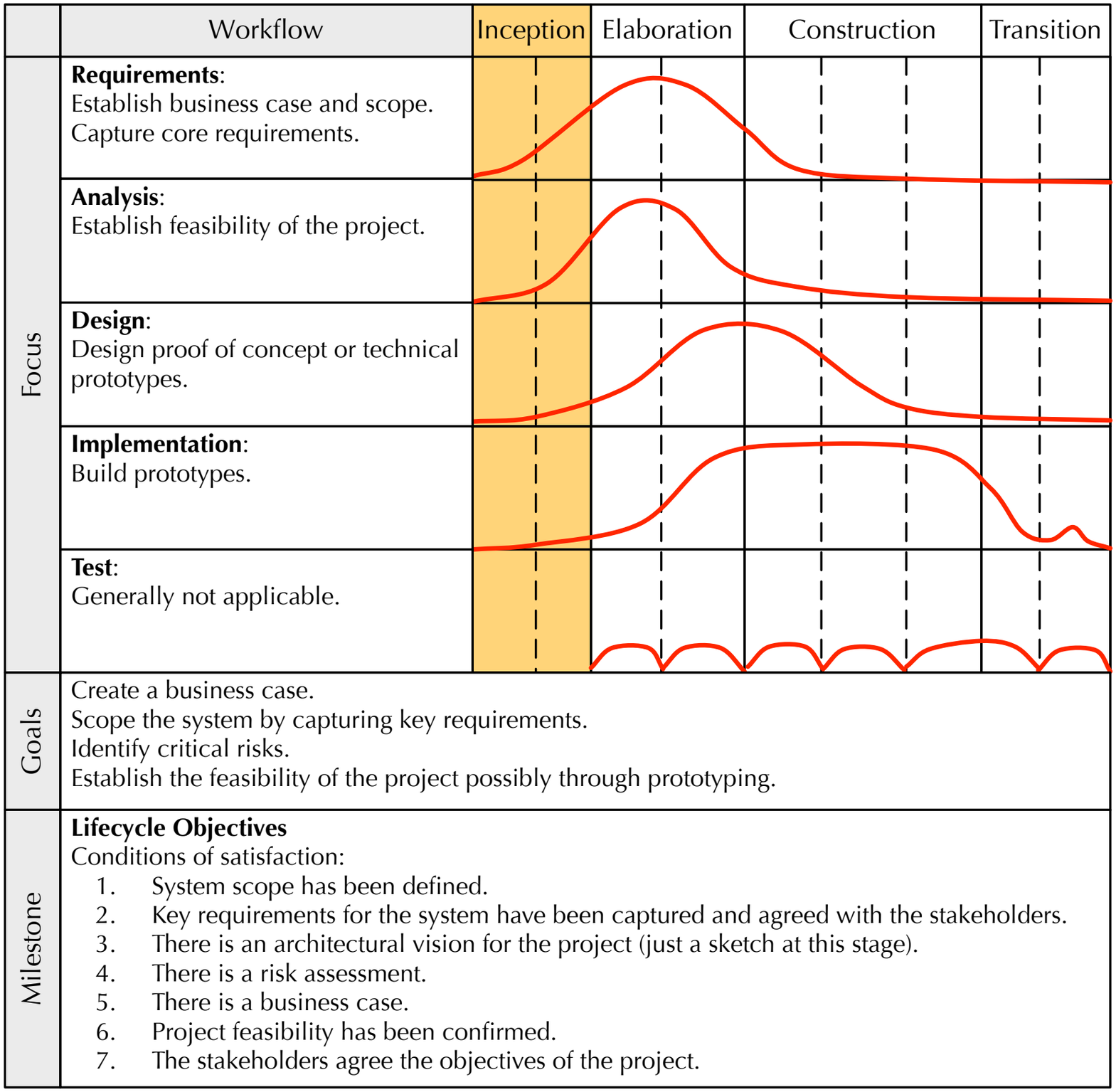
These figures give you simple maps that tell you your focus, goals and milestone for each of the UP phases. Clearly the figures are just a summary, and both [Arlow 1] and [Jacobson 1] go into much more detail about what happens in each phase and in each core workflow.

The inception phase

This phase is about getting the project off the ground.

The major activities in Inception are establishing what the software system needs to do, and the feasibility of that.

Figure 2.4 The inception phase



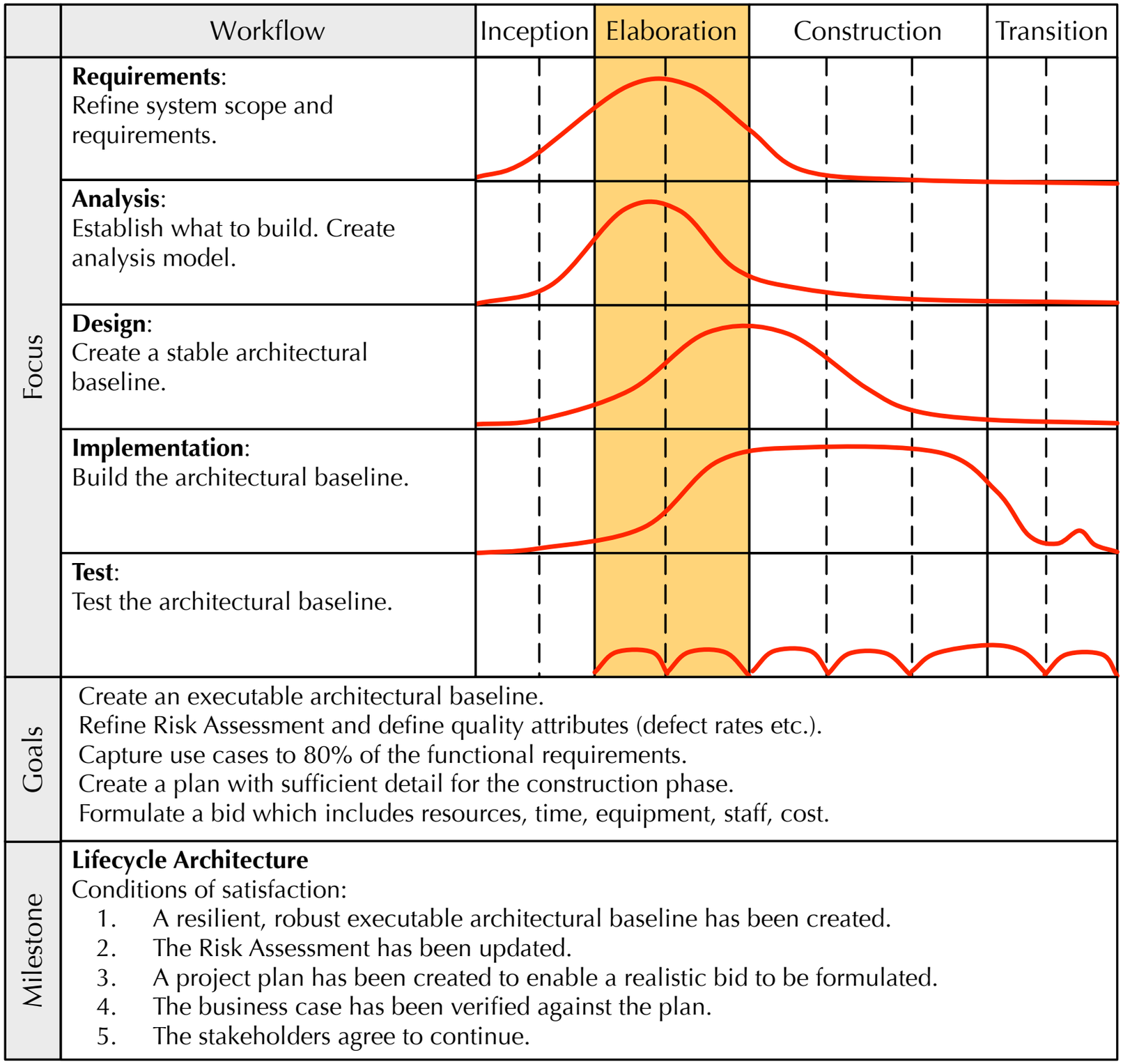
The elaboration phase

This phase is where the main work in requirements and analysis is done, and it will therefore be the focus of this book.

Elaboration is critical to project success, because the decisions you make in this phase will have a big impact on all subsequent phases.

Because UP is iterative, you will always have the opportunity to revisit and reassess these decisions if necessary, but this will have a negative impact on project time and resources.

Figure 2.5 The elaboration phase



The construction phase

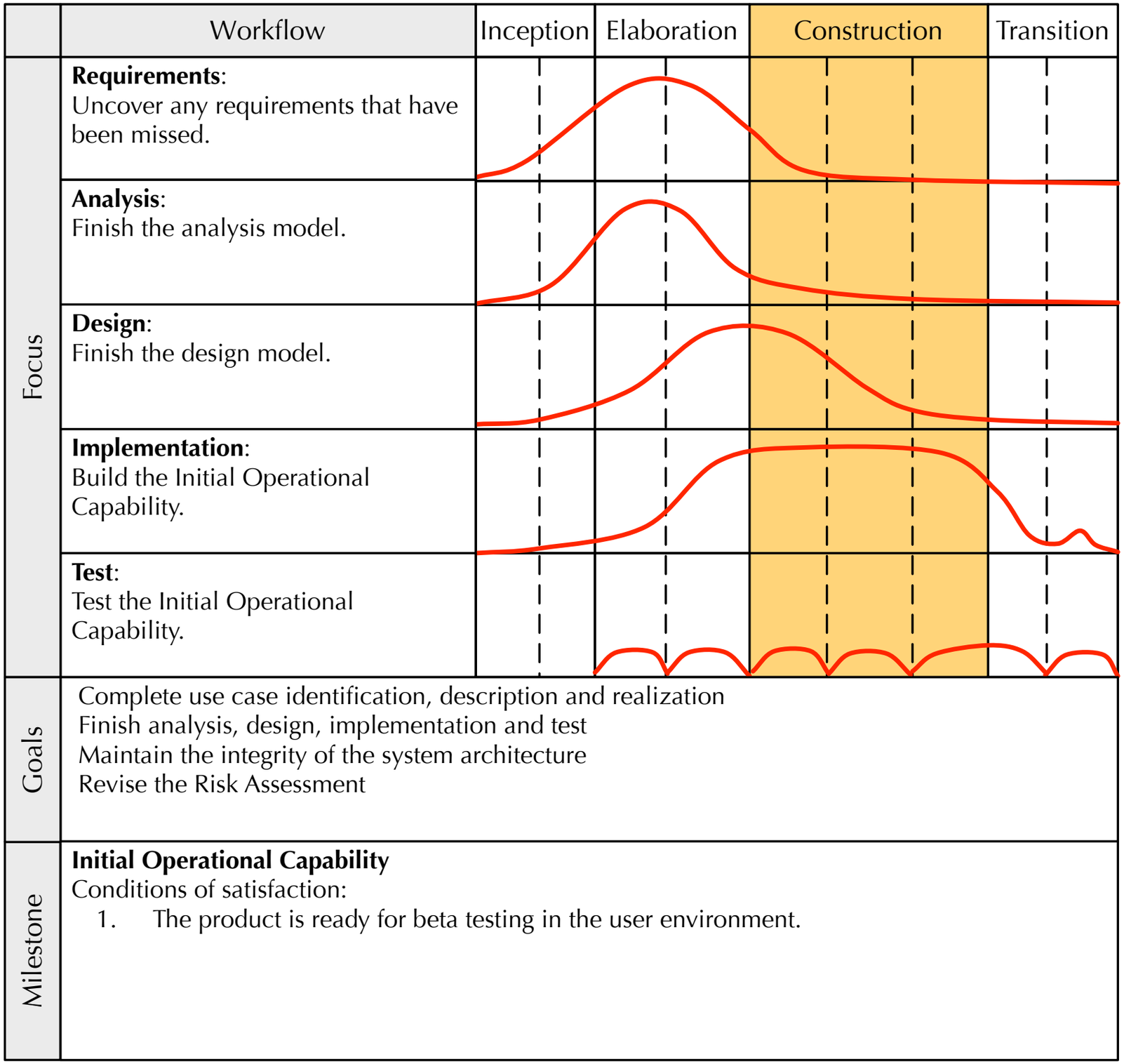
This phase is about building software.

The requirements and architecture should by now be stable enough so that you can transform them into code with relatively little risk of having to extensively rework that code later.

If either the requirements or architecture are still in flux you should not transition to Construction as you will only waste time building something that you will have to rebuild later.

But are requirements and architecture really necessary for this phase? Can't we just continually refactor? Sadly, there is no evidence to suggest that constant refactoring can generate a viable architecture. In fact the reverse seems to be the case. Constant refactoring is also very expensive.

Figure 2.6 The construction phase

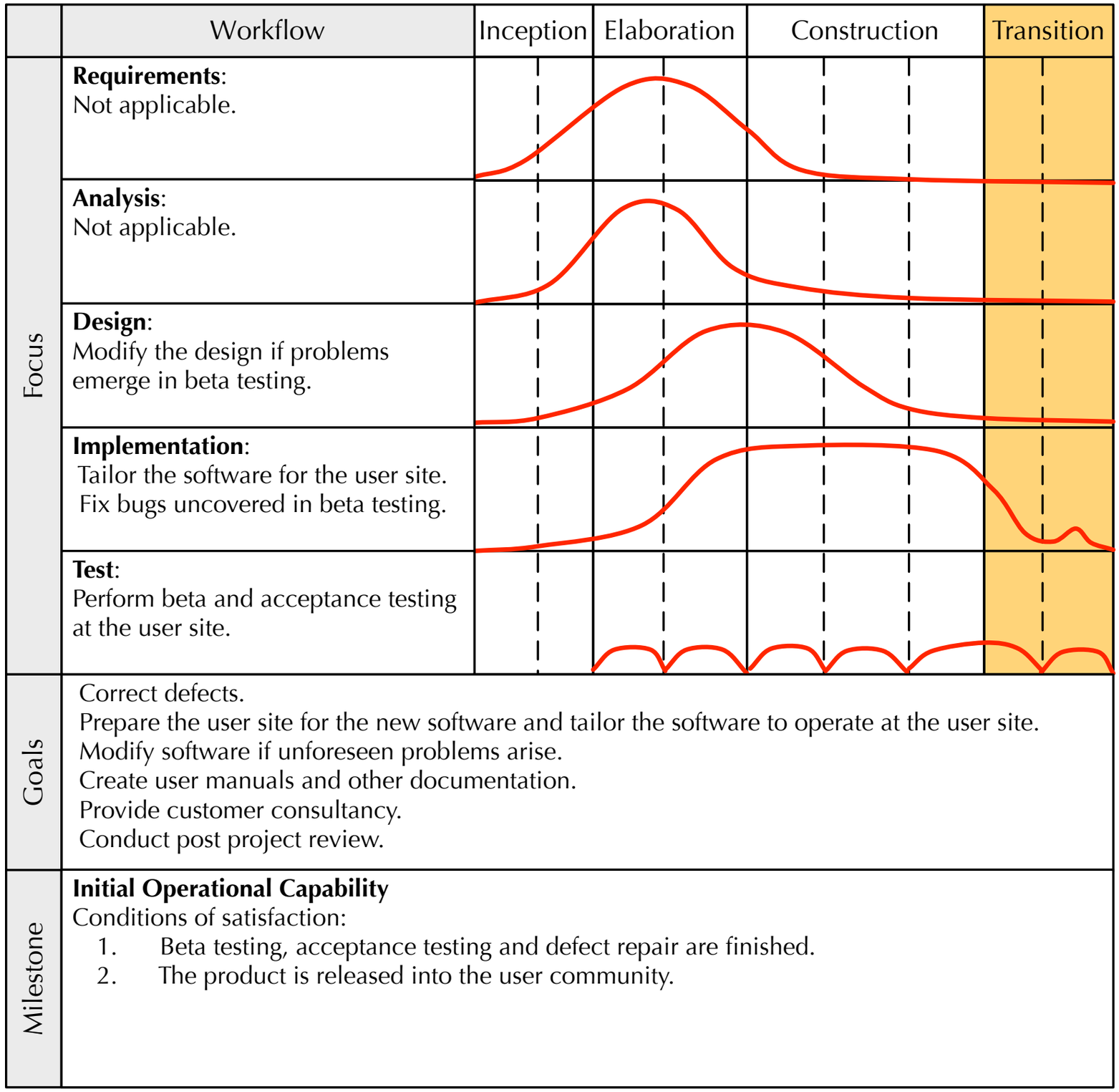


The transition phase

This phase is about delivering working software to the user site. Its focus is therefore very much on implementation (customizations for particular users or sites) and testing.

If you are performing incremental delivery, then there will an element of transition in all the other phases. The transition phase itself will shift focus to moving the project into an on-going maintenance mode.

Figure 2.7 The transition phase



The OLAS inception phase

As we have seen, the inception phase of a UP project is about getting the project off the ground.

You can think of Inception as being the phase in which the business develops the *will* to undertake the project. This will is usually expressed as some sort of business case that outlines the goals of the project and the business benefits that the project is expected to realize.

Inception may be very informal and often happens before the project has been formally launched. As such, although it is clearly part of the lifecycle of the software system, it is often prior to the lifecycle of the actual software development project.

As an OO analyst, you may or may not be involved in this phase. Often, it just involves management, key stakeholders and perhaps a system architect.

If you are involved in Inception at all, you will be only doing pretty much exactly what you do in Elaboration but in *much* less detail.

In Inception, an analyst may well be employed by a project sponsor to develop initial ideas about what the system should do and how feasible that is. You will be collecting evidence that they can use to create a business case that will launch the project.

As this tends to be a very informal activity, you are more likely to be creating documents and informal diagrams than UML models!

In OLAS, we were not involved in the Inception phase at all. We were given a single deliverable, the OLAS Vision Statement, that was the single output of that phase. We'll look at this deliverable next.

The OLAS Vision Statement

The OLAS Vision Statement (Figure 2.8) is the document that "kicks off" the OLAS project.

The Vision Statement presents a high level view of what the system is going to be about. It is only a vision, and so it may present options that might or might not be realized.

We will use this document in exercises throughout the next few chapters, so read it now to familiarize yourself with the system. We suggest you bookmark this page so that you can easily refer to the Vision Statement as you read the rest of the book.

The Vision Statement had metadata attached that we separated out into Figure 2.9. We take a very particular approach to document metadata as we discuss in the next subsection.

Figure 2.8 OLAS Vision Statement

OLAS Vision Statement

The Orne Library of Miskatonic University holds, as well as conventional reference materials, many rare and valuable texts in the restricted collection. These are in increasing demand from scholars at the University and worldwide. Since its inception, the Library has been using a manual index card catalog, and a manual ticketing system to manage loans. However, with the increasing number of students at the University and with the increasing demands for access to the restricted collection, we feel that the time has come to automate both the catalog and the ticketing system.

After preliminary discussion with the University IT department, we have come to the conclusion that placing the library catalogs on a computer system would allow scholars to access the collection more effectively, and would also take some of the workload off the library staff who are under increasing pressure. As part of this project, the manual ticketing system will also be automated. This will allow the librarians to manage the day-to-day loans more effectively, and to track overdue loans more easily. This is especially important for the restricted collection as over the years, it has been particularly prone to attrition attributable partly to outright theft and partly to a series of unusual and unfortunate events that have affected some of the borrowers. The librarians have recently implemented a vetting system for access to the restricted collection that allows them to restrict access to the catalog to trusted parties only. This has greatly reduced the number of losses and vetting should be incorporated into the automated system.

Another driver for automation is that the Library needs to be able to exchange catalog information about the restricted collection with Innsmouth Public Library, which also has a collection of rare and valuable books. Innsmouth has already automated its restricted collection catalog and the Miskatonic system must be capable of exchanging catalog information with the Innsmouth system. Scholars should be able to access both Miskatonic and Innsmouth catalogs provided they have permission to do so.

Our IT department has proposed that the library catalogs will be made accessible to scholars through a web browser interface.

Figure 2.9 OLAS Vision Statement Metadata

Author: Dr. Henry Armitage, Head Librarian

Date: 31 October 2005-10-21

Office: 616 Orne ext. 666

Reviewed by:

Warren Rice, Director of IT, Office: 333 Altwood, ext. 766

When we received the Vision Statement, the first thing we asked Dr. Armitage was why he didn't want an off-the-shelf system. Many such systems exist.

His answer was that the Orne Library has very specific requirements for its catalog and the University IT department had decided that a bespoke system was the best way to address those requirements.

Another point to consider is whether OLAS should be an Online Public Access Catalog (OPAC). If you do a web search on OPAC, you will find that most University catalogs are OPACs.

However, Dr. Armitage made it very clear to us that OLAS was *not* public access and that there were no plans to make it so. The Orne Library has some very sensitive materials, and has only ever shared its library catalog with Innsmouth Public Library.

No chrome

When we present documents such as the OLAS Vision Statement to you in this book, we will strip them of all "chrome".

By chrome, we mean such things as corporate branding, and metadata such as version control information, extra fields such as, "Who should read this document", "Executive summary", and so on.

We call this "chrome" because it reminds us of the chrome decorations on old American cars. It doesn't really add anything to the function or capabilities of the car, but it makes it look good. This is, of course, only an amusing analogy, because, unlike car chrome, document chrome *can* have a useful function. However, as our focus here is on accessing the essential information content of documents, we can safely ignore it.

When you are creating documents for your project, we recommend that you keep the amount of chrome to the necessary minimum. Otherwise, it acts as a barrier to entry to the document and as "noise" that obscures the documents' meaning.

Some corporates seem to confuse chrome with quality and we have seen style guides for corporate documents that emphasize chrome to the exclusion of more useful things such as information content and clear writing style. Perhaps this is because compliance to chrome is easily measurable by using

a simple checklist, whilst assessment of information content and writing style is not.

We often come across documents several pages long that only contain a few paragraphs of useful information! We also notice that sometimes a lot of chrome is purposefully used to disguise low or poor quality content.

We recognize the need for a certain amount of chrome, but generally feel that less is more.

In principle, chrome should be *separated* from the information content of the document as metadata that is viewable on demand. Many word processing applications provide you with document properties for such metadata, but this is usually a very crude facility and in our experience, isn't widely used.

Content management systems such as Plone can separate document metadata from content as well as provide access, security and distribution mechanisms. These are the *best* places to store project documents.

Summary

In this chapter we have explored how we're going to address the example problem, OLAS, using UP, and seen the output of the Inception phase of OLAS – the OLAS Vision Statement.

Software engineering processes

A software engineering process describes how you are going to go about developing a piece of software. It describes:

- Who – the roles needed in the software development project.
- What – the artifacts to be delivered.
- When – the project plan for when specific software engineering activities have to occur.
- Where - is it a distributed project?
- How – the actual activities performed in the software engineering process.

The non-process is the Nike® process – "just do it"®. Remember that you must match the software engineering process to the project.

The Unified Process (UP)

The Unified Process is a software engineering process framework that needs to be customized for your project. You determine the process requirements

for your project (based on size and complexity) and tailor the UP accordingly.

There are several flavors of UP.

UP projects are architecture centric, requirements driven and risk driven.

Iterative and incremental development

Put simply: Break a large complex project down into small simple ones (iterations). Each iteration is a mini project including:

- Planning.
- Analysis and design.
- Integration and test.
- An internal or external release of an executable architectural baseline.

An executable architectural baseline is:

- Executable – an actual piece of software that can be executed to deliver business benefit.
- Architectural – embodies the architectural principles and style of the final application. It is not just some ad-hoc solution hacked together to meet a deadline.
- Baseline – serves as a stable platform on which to add functionality in subsequent iterations. Each new baseline builds on the one before it. A baseline can only be changed through a formal change management process.

Iterations

A mini project (see previous section).

Core workflows

These are definitions of activities such as requirements capture or analysis. There are 5 core workflows:

- Requirements – capturing what the system should do.
- Analysis – refining and structuring the requirements.
- Design – realizing the requirements in the system architecture.
- Implementation – building the software.
- Test – verifying that the implementation works as desired.

Remember these with the acronym RADIT.

The workflows are organized into phases.

Phases

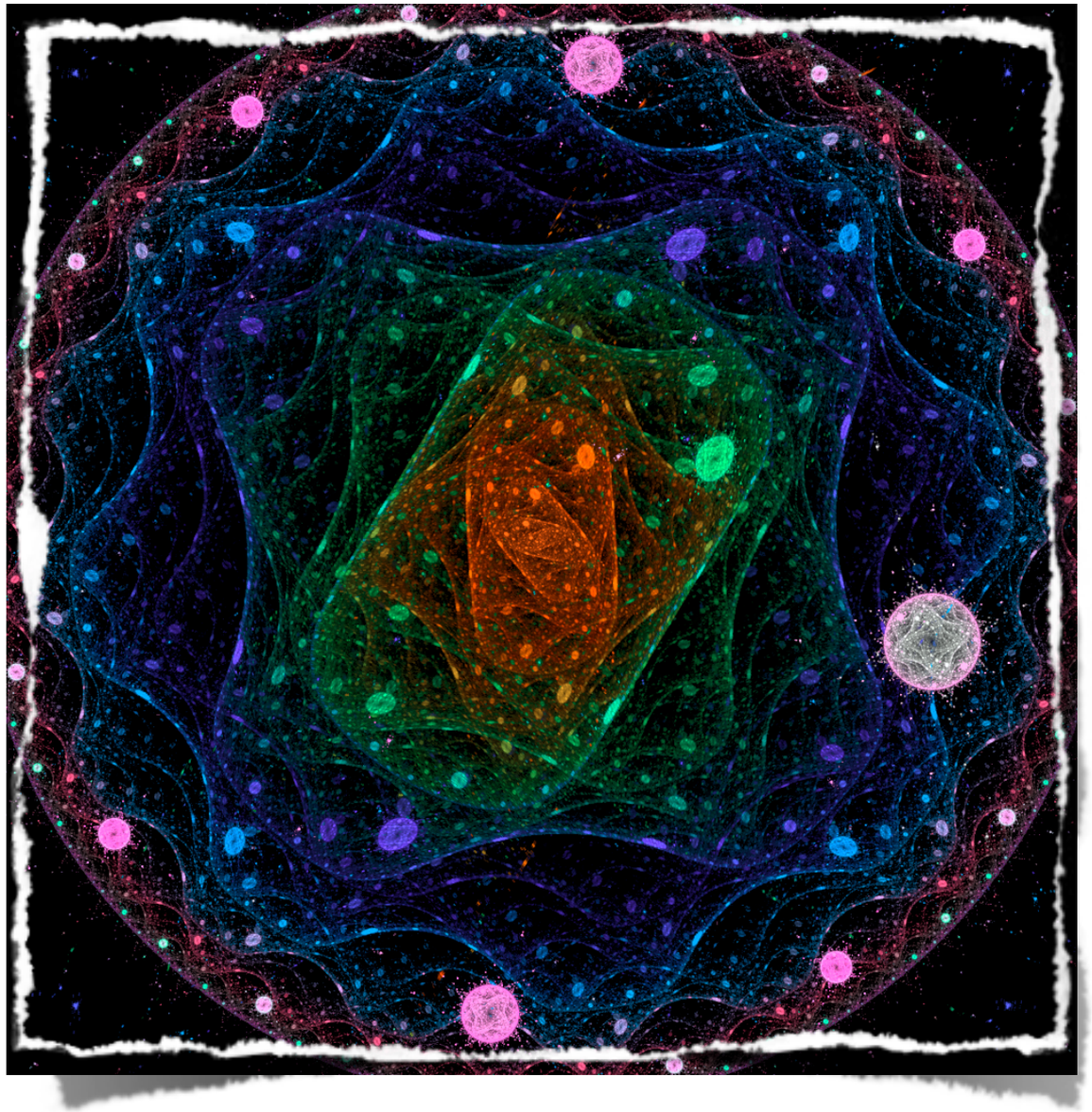
These are groups of iterations that constitute identifiable stages in the project. They each end with a major milestone. The phases are summarized here:

- Figure 2.4 the inception phase.

- Figure 2.5 the elaboration phase.
- Figure 2.6 the construction phase.
- Figure 2.7 the transition phase.

Chapter 3

Generative Analysis



In this chapter we introduce a technique that we call "Generative Analysis" (GA). GA is a new approach to object oriented analysis that tries to fill in the bits that other approaches tend to leave out. Specifically, how you can deal with unstructured, informal information in analysis and how you can elicit high quality information from information sources that are subject to deletion, distortion and generalization.

Generative Analysis

Generative Analysis is an approach to OOA predicated on two facts:

1. Most of the information you get in analysis (especially the early stages) is informal and unstructured.
2. Most of the sources for the information you need to perform analysis effectively are subject to the forces of deletion, distortion and generalization.

So the goals of Generative Analysis are to teach you how to deal with these real-world human issues of software engineering and to provide you with the new analysis tools you need to do the job.

In effect, GA is learning to approach all information in an effective, structured, and intelligent manner. You need to know what types of information you have to deal with, and how to deal with each specific type.

GA recommends a general set of best of breed tools that can be used with any analysis technique you happen to be using, be it use case modeling, class modeling, activity modeling or other forms of modeling.

In this chapter we introduce those tools that we use in GA to start working with informal, unstructured information. We will introduce other tools, in particular the meta-language M++, in subsequent chapters.

It's worth noting that GA is also very applicable to non-OO modeling. Furthermore, we have found that it has wide application in business in general whenever informal, unstructured information needs to be handled in an intelligent way. These more general applications of OOA are outside the scope for this particular book.

Why "Generative Analysis"?

Initially, we were loathe to introduce another named technique! After all, the field of computing is already littered with often meaningless buzz words, as the various pundits jostle for "head space" in an increasingly idea-saturated domain. In many ways we would rather have presented an anonymous set of techniques ("secrets") and left it at that. In fact, that was our original plan.

However, naming something has power. In particular, naming a process objectifies it and turns it into a "thing" that you can easily refer to. Because you can refer to it, you can consider it and talk about it effectively and concisely. You can more easily define and scope a named thing with appropriate degrees of precision. All of this is much more difficult if you have no definite way of referring to something.

This was driven home to us when one of our friends asked us what our next book was about and we had

great difficulty answering him! Turning this on its head, we also imagined asking one of our students or employees what they were learning from this book and realized that they would have similar difficulty.

After careful consideration, we decided that the approach in this book, the collection of techniques and tools, are sufficiently cohesive and sufficiently new (at least in computer science) that we can justify naming the process Generative Analysis.

We chose the term "Generative Analysis" because:

- The technique is "generative" in that it seeks to generate high-quality, structured, formal information from low-quality, unstructured, informal information. It does this by first capturing informal information, then transforming it through specific analysis techniques so that it is sufficiently formal to (eventually) turn into a software system.
- It is "analytical" because the focus is on analyzing information, and, through this analysis, transforming it. In GA we focus on the requirements and analysis stage of the project life cycle, so strictly speaking, it might have been called Generative Requirements and Analysis. But that is too much of a mouthful! Instead, we chose the term "analysis", in the broader sense of "analytical", and subsumed software requirements and analysis into that.

Of course, *all* analysis should be generative. By using the term GA we have a good opportunity to highlight and stress this important, but often overlooked, fact.

Where analysis begins

Analysis generally begins with the gathering and analysis of unstructured information.

In fact, you can view the whole software engineering process as a process of transformation in which informal, unstructured information is transformed by analysis and design into information that is sufficiently structured and formal that it can be executed by a computational system.

When you look at methodologies such as the UP, you see that these methodologies give you many excellent ways of structuring information. But they have relatively little to say about how you analyze informal, unstructured information.

For example, exactly how do you get from something as informal and unstructured as a user interview, to something structured such as a use case or class diagram? This is one of the missing pieces that analysts generally acquire only through hard experience.

GA addresses informal, unstructured information head on. In fact, it assumes the worst - that most of the information you acquire in your analysis activities is:

- Partial - essential information is deleted.

- Inaccurate - essential information is distorted.
- Generalized - essential information is modified by rules and beliefs about the world which may or may not be true.

This is merely a restatement of the three filters that Noam Chomsky says characterize all human communication - deletion, distortion and generalization – that we discuss in detail in Chapter 7.

Based on the assumptions of informal, unstructured information and deletion, distortion and generalization, GA provides you with the following tools:

- Specific ways to capture unstructured, informal information (communication techniques).
- Specific ways to organize unstructured, informal information (mapping and modeling techniques).
- Specific techniques to recover deleted, distorted and generalized information (a meta-language called M++ for disambiguating natural language).
- A set of transformations to generate high quality information from low quality information (M++, mapping and modeling).

The last bullet is the "generative" part of GA - information of low quality and low value is transformed by specific analysis techniques to information of high quality and high value. You perform GA recursively on information until it has the quality you need to move on to some more formal approach such as class modeling.

In this book, we present GA quite formally to make it teachable and learnable. However, the true essence of GA about learning an attitude and a set of internalized approaches to information, rather than specific formal techniques. Techniques come and go and can be applied as needed, but learning this new approach to information is the heart of GA.

Capturing informal, unstructured information

Clearly, there are very many ways of capturing informal, unstructured information.

GA begins by making you consider how you currently approach that task. Oddly enough, few people seem to be particularly aware of, or concerned about, how they capture informal, unstructured information.

For most of us it's just habitual - perhaps we learned to take notes and make informal diagrams in school, and that's about as good as it gets. But this process is often the beginning of the whole analysis activity, so it bears closer examination.

Try it now! *Capturing informal, unstructured information*

Take a moment and write down at least five ways you capture informal, unstructured information in your

day to day work. Here are some questions to consider:

1. How effective is each of those ways (1 = not at all, 5 = completely effective)?

2. Think of several different specific contexts in which you need to capture informal, unstructured information (e.g., brainstorm, stakeholder interview). Which techniques are most suitable for which contexts?

3. Do you feel you are in control of the information you capture (1 = not at all, 5 = completely)?

We have observed that the best analysts are very good at working with unstructured, informal information, whilst the worst don't even recognize it as a process.

GA introduces four key techniques that we will discuss in this chapter to help you begin to work with informal, unstructured information:

- Mind mapping
- Concept mapping.
- Dialogue mapping.
- Structured Writing .

This is not an exhaustive set of techniques, and, of course, you must use whatever else works for you.

We highlight these specific techniques partly because of their proven effectiveness over the years that we have used them, and partly because each of them teaches you something quite particular about working with information effectively. By learning these simple techniques you can change your approach to information to make you a more effective analyst.

In Chapter 2, we talked a little about mental or cognitive maps of the world. The mapping techniques we provide here give you specific ways to get versions of those cognitive maps down on paper. Later on in the book, we look at ways of exploring these maps and transforming them into precise UML models.

These mapping techniques can be particularly effective in workshops.

You can start with a mind map to scope the workshop and get down any ideas in a brainstorm.

You can then use more formal concept mapping to explore particular areas in detail - to do what Jim calls “covert OO analysis”.

If you have a difficult problem domain in which the stakeholders express many different and perhaps contradictory points of view, then you can use dialogue mapping to capture those points of view and perhaps to reach some kind of resolution.

These informal and semi-formal mapping techniques provide a powerful and pragmatic prelude to the more formal processes of OO analysis.

Whilst we will present these mapping techniques in the context of OOA, you may notice that they are completely general, and you may find many other uses for them that we don't discuss here. These mapping techniques are flexible, creative, and very effective for *any* type of analysis, so have fun and experiment!

Mind mapping

This is a very simple but elegant technique invented by Tony Buzan in the 1960s. You can find out more about it in [Margulies 1].

We like mind mapping because it mirrors to some extent the associative way that the human mind and memory works. We also like it because it is a flexible technique that we can customize according to our needs.

We use mind mapping a lot in stakeholder interviews and in workshops, to help us understand the problem domain and to capture and generate ideas.

Mind maps are easily converted to presentations by converting the most important ideas to slides and the sub-ideas to bullet points, and many mind mapping tools will automate this process for you. We often prefer to present them to others in this more structured way.

As we've mentioned, the brain works by association - by relating concepts to each other. In fact, you can think of the brain as a very complex and powerful pattern-matching machine.

In mind mapping you start in the center of a blank page with your main idea. If you can, you can sketch a picture of it because this will stimulate your imagination more than a description. Branches radiate out from the central idea. Each branch represents an idea associated with the main idea. These main branches can themselves branch into sub branches and so on as your mind pattern-matches and uncovers more and more associations.

As you can see, this creates a hierarchy of ideas rooted on the central idea.

Represent each idea as text, a picture or both. Use colors to highlight ideas. Use lines to connect related ideas. Use your imagination to make the mind map as vibrant and alive as you can. Look in [Margulies 1] for more inspiration.

There are very few mind mapping rules:

- Try to use just one keyword per idea. If you need to write more text, draw a "call-out" box, or write in the margin of the page and cross reference that to an idea.
- Use pictures where you can.
- Mind mapping is a brainstorming technique so just get the ideas down on paper - you can analyze them later.

The one word per idea rule is a very important constraint when you are first learning to create mind maps.

In conventional note taking, you generally scribble as much down as you can without thinking about it too much. The idea is that you will go back and understand it later. However, often this doesn't happen - as any student will tell you!

Mind mapping is completely different. In mind mapping you understand the information as you create the map. One of the key ways in which you do this is by summarizing the information using keywords. Another way you generate understanding is by creating connections between the keywords. Both of these things *force* you to think about the information as you write it down. This is why mind mapping is such a powerful technique.

When you first begin to mind-map, you might find the one word rule difficult to follow. This is because you are not used to summarizing information on the fly. However, if you persevere you will learn a useful and empowering skill.

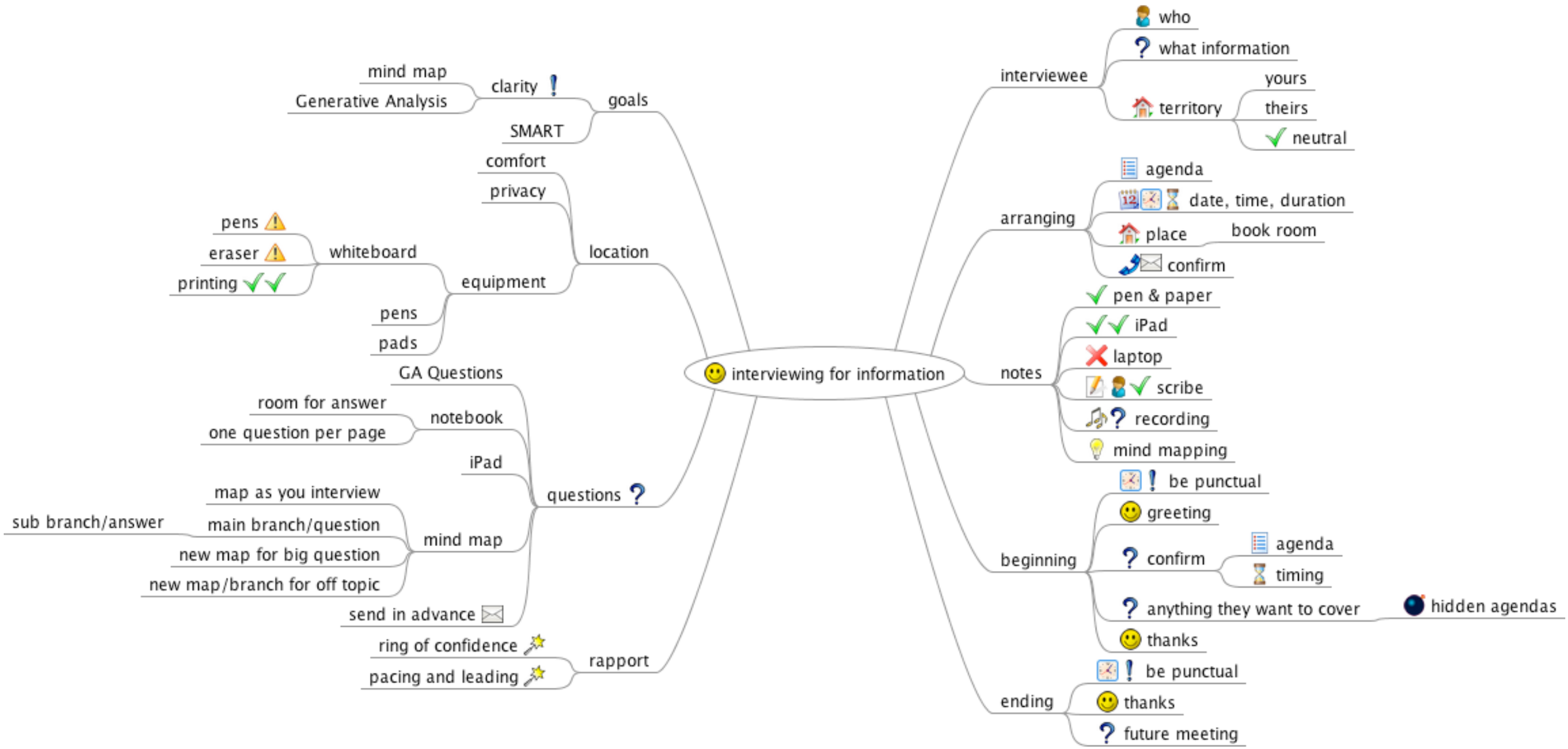
If there is something that you need to record that requires more than a single keyword, then you can draw a thought bubble or speech bubble connected to a keyword, and write the text in that. Try to avoid doing this at first.

Once you have been mind mapping for a while, you will find that you can summarize even complex information naturally and easily. At that point, you can relax the one word rule.

Figure 3.1 shows a mind map we created that captures our ideas on interviewing for information. Neither of us is particularly artistic, so we created this with the excellent FreeMind (www.freemind.com), an open source mind mapping tool.

Mind mapping tools are great for when you need to include your mind maps in printed materials such as this book. They are also excellent for creating a mind map on a data projector in a meeting.

Figure 3.1 Interviewing for information mind map



Mind mapping process

"Mapping Inner Space" [Margulies 1], is the best book on mind mapping we have read. It describes a very effective six-step process for getting the most out of mind mapping (Figure 3.2).

Try it now! *Mind mapping - just do it*

The best way to learn to mind-map is the Nike® method - just do it®. The key skills are learning to summarize and create associations on the fly. You will rapidly develop your own graphical style. The more you practice, the better you get.

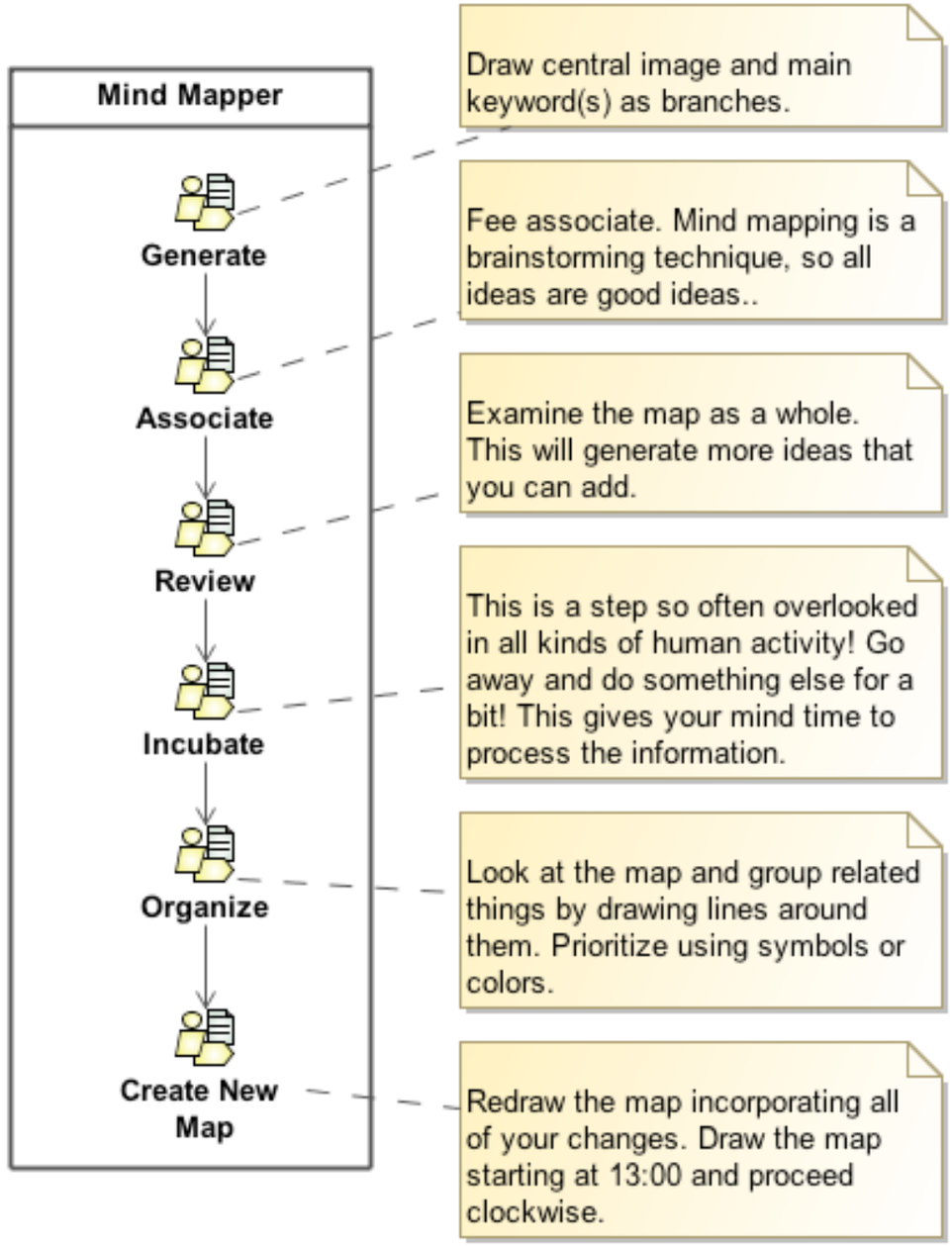
Create a mind map summarizing the information in this chapter up to this point. Do this from memory. Compare it to the mind map at the beginning of this chapter. It's interesting to see how much you have remembered.

Applying mind mapping

Mind maps are useful for:

- Capturing information.
- Generating ideas.
- Uncovering relationships between ideas.
- Memorizing information.
- Remembering information.
- Understanding information.

Figure 3.2 Mind mapping process



As we've said memory works by associating one thing with another. The more associations you create between an idea and other ideas, the easier you will find it to remember.

This is the spell or “meta program” behind all memory enhancement techniques - creating associations [Lorayne 1]. The more sensory modalities (sight, sound, vision, smell, taste, touch) you can involve in an association, the stronger it gets. Because mind maps cause you to create associations, they have the highly desirable side-effect of helping to anchor things in your memory.

Because mind maps work in an associative way, one idea triggers another, and that triggers another and so on. This is a generative process - ideas generate ideas. We like generative processes because they move us forward to higher levels of meaning. This associative cascade can also help you to recover information from your memory.

In OO software engineering, mind mapping can be particularly useful in lots of ways. For example, if you have been speaking to a stakeholder but did not have an opportunity to take notes at the time, create a mind map as soon as you can after the meeting. This will help you to recover most of the details of the conversation. Using mind maps in this way has been a great help to us on many occasions and you may be pleasantly surprised by how effective it is!

Try it now! *Mind mapping mind mapping in your project*

Create a mind map with the central topic "mind maps in my project". How many uses for mind maps can you come up with?

Concept mapping

This approach was invented in the 1970s by Joseph Novak at Cornell University. It is a more formal forerunner of mind mapping.

In concept mapping, concepts are connected by linking words to form propositions. What does this mean?

Concept: “an abstract or general idea inferred or derived from specific instances” [WordNet 1]

From the point of view of concept mapping, a concept is a thing (object) or happening (event) derived from our knowledge about the world. We can consider a concept to be a landmark on one of our cognitive maps.

Linking words: These specify relationships between concepts. They link concepts together to form propositions.

Linking words create the topography of our cognitive map by relating the landmarks.

Proposition: unit of meaning.

Propositions are statements that affirm or deny something about the domain. These are meaningful relationships that we see in the map.

Each concept map is created to explore a specific focus question. This question determines the topic and scope for the map.

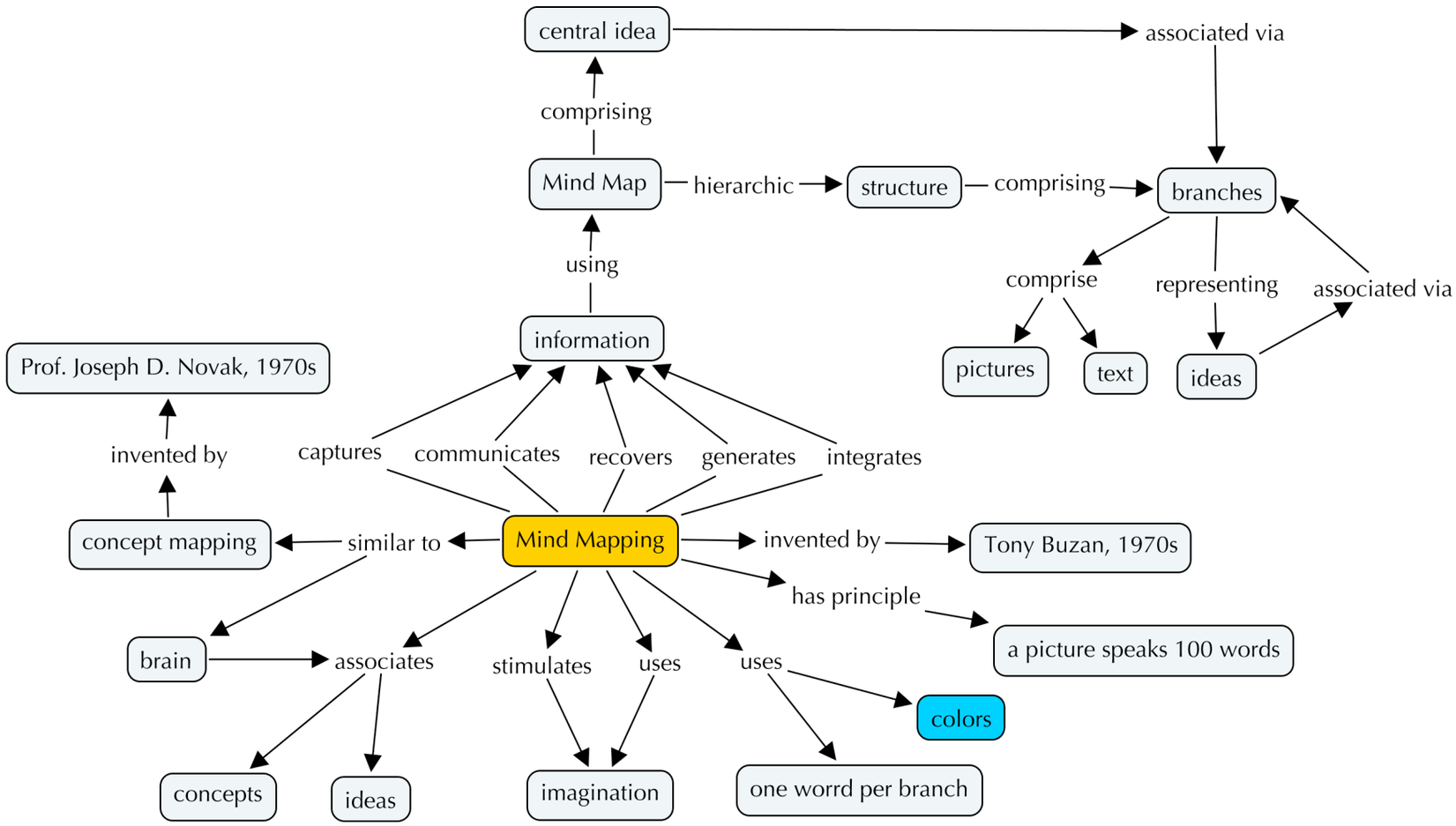
Both mind mapping and concept mapping tend to be hierarchical in structure - one central idea branches out into many sub-branches. However, concept maps are *not* strictly hierarchical because each concept can be connected to zero or more other concepts by a directed line. The concepts can form networks. The line connecting concepts expresses the relationship between them and may be labeled appropriately.

Figure 3.3 shows a concept map for mind mapping.

This concept map was created using the excellent CMap. This is a set of tools that allows the creation and sharing of concept maps.

We like concept maps because they represent a half-way house between the very informal mind maps and formal UML models. They also capture very definite propositions about the domain in a way that mind maps typically don't.

Figure 3.3 Concept map for mind mapping



Applying concept mapping

Concepts maps can be used in much the same way as mind maps. However, whilst mind maps are useful for getting the big picture and generating ideas, concept maps are useful for focusing in on specific details. The more formal nature of concept maps means that they are more useful when you are trying to home in on detailed information and they are less useful for creativity.

Concept maps work very well in a brainstorm situation when you are trying to find out how part of the problem domain works in some detail.

As you will see in the next section, concept maps are also very useful for what we call covert OO analysis!

Covert OO analysis

The structure of a concept map is very similar to the structure of an OO model. You will find that a good concept map for a problem domain maps, in a fairly obvious way, onto a first-cut analysis-level class diagram.

This is one of the big advantages of concept mapping - it is a covert approach to OO analysis that is easy to understand and is acceptable to virtually all stakeholders!

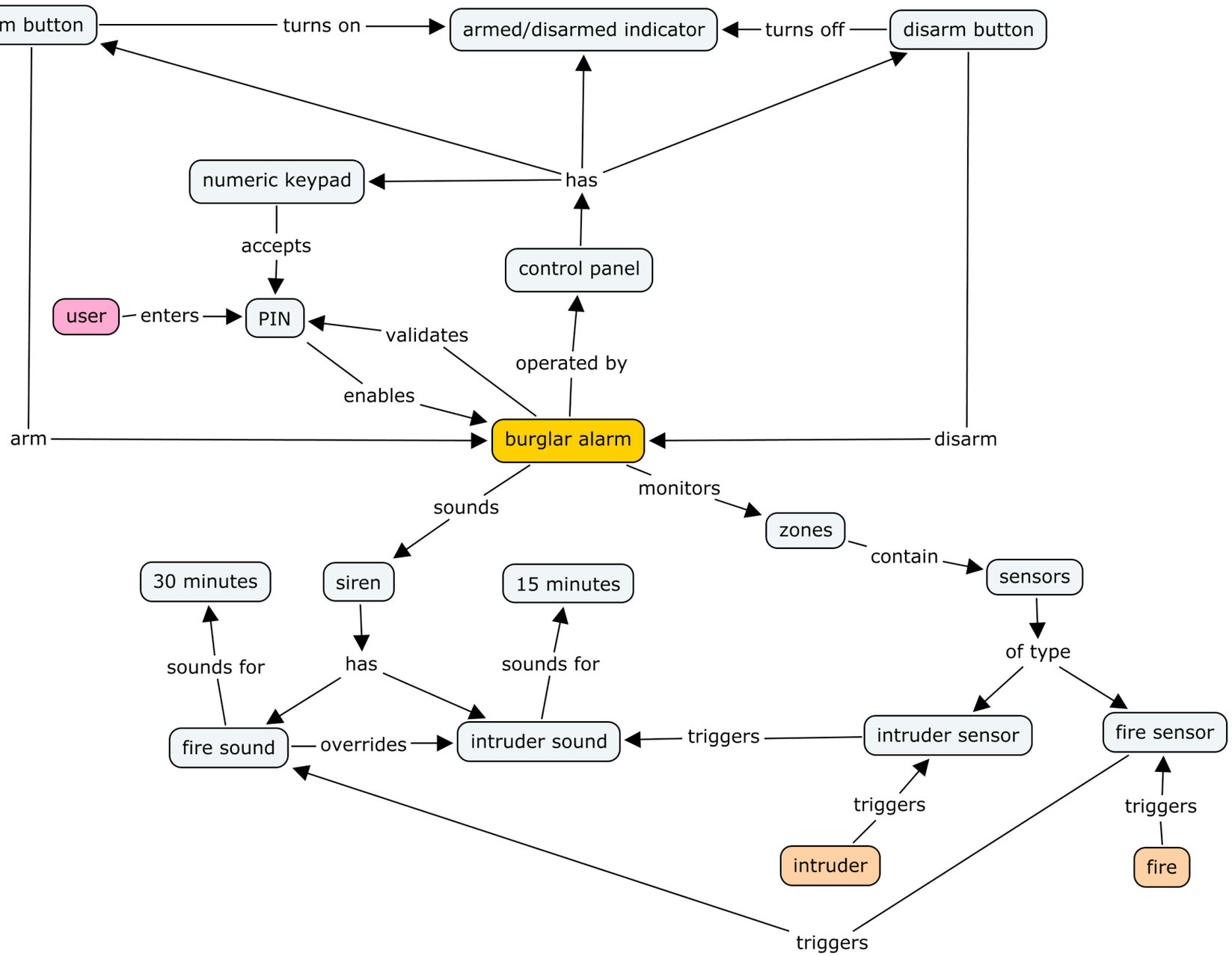
It's easiest to understand this by example: Look at the concept map in Figure 3.5. This is for the simple burglar alarm system that we use as an example in "UML 2 and the Unified Process" [Arlow 1].

The burglar alarm system has a control panel (Figure 3.4) that monitors zones. Each zone contains one or more sensors. There are two types of sensor, fire and intruder.

Figure 3.4 Simple burglar alarm system control panel

We can create a concept map with the focus question “what does the simple burglar alarm system do?” as shown in Figure 3.5.

Figure 3.5 What does the simple burglar alarm system do?



One of the really neat things that CMap lets you do, is extract the propositions from the concept map as text. You can see the result of this in Table 3.1.

What can we say about these propositions from an OOA perspective?

Firstly, we can notice that the propositions capture specific *requirements* for the burglar alarm system.

For example, the first proposition, "arm button->arm->burglar alarm", may be rephrased as a requirement: "The arm button *shall* arm the burglar alarm". Here we are expressing requirements as simple *shall* statements as described in [Arlow 1].

Secondly we notice that there are propositions about the *structure* of the burglar alarm system. These structural propositions are ontological statements about what things exist and how they are related to each other. The structural propositions are marked as S in the table.

Finally, we notice that there are also *behavioral* propositions. These are propositions about how the things in the system behave. Behavioral propositions

Table 3.1 Propositions about the burglar alarm

concept	linking word	concept	structural	behavioral
arm button	arm	burglar alarm		Y
	turns on	armed/disarmed indicator		Y
burglar alarm	monitors	zones	Y	Y
	validates	PIN		Y
	sounds	siren	Y	Y
	operated by	control panel	Y	Y
control panel	has	arm button	Y	Y
	has	armed/disarmed indicator	Y	
	has	numeric keypad	Y	
	has	disarm button	Y	
disarm button	disarm	burglar alarm		Y
	turns off	armed/disarmed indicator	Y	
fire	triggers	fire sensor		Y
fire sensor	triggers	fire sound		Y
fire sound	overrides	intruder sound		Y
	sounds for	30 minutes		Y
intruder	triggers	intruder sensor		Y
intruder sensor	triggers	intruder sound		Y
intruder sound	sounds for	15 minutes		Y
numeric keypad	accepts	PIN		Y
PIN	enables	burglar alarm		Y
sensors	of type	intruder sensor	Y	
	of type	fire sensor	Y	
siren	has	fire sound		Y
	has	intruder sound		Y
user	enters	PIN		Y
zones	contain	sensors		Y

can describe responsibilities of the system as a whole, and of its parts. These behavioral propositions are marked as B in the table.

Structural propositions

Given these structural propositions, we can easily construct a first-cut analysis class diagram that satisfies them (Figure 3.6).

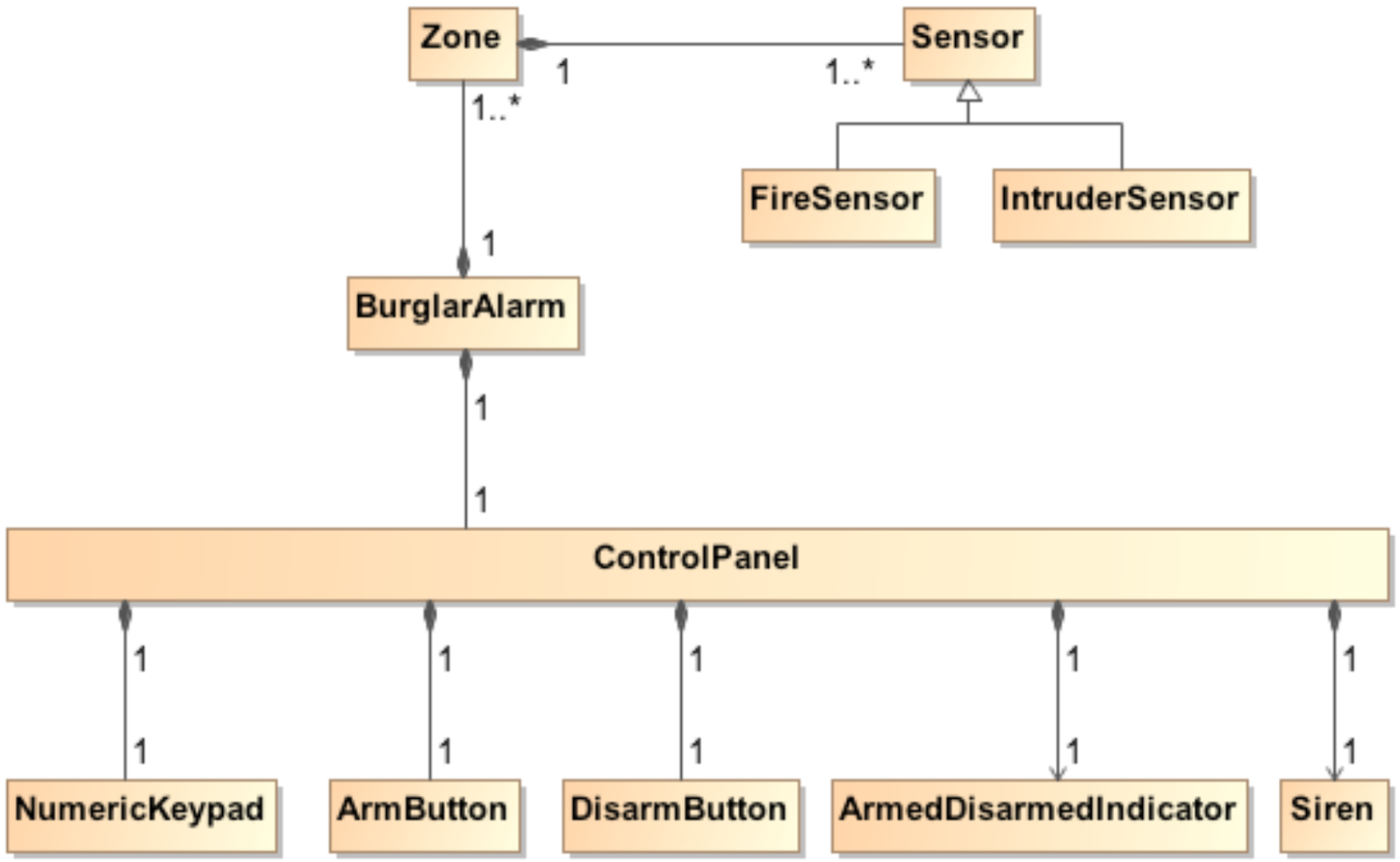
Because this is an analysis class diagram, we are not at all concerned about how we would actually implement the system, we are *only* concerned with creating a logical structure that satisfies the propositions, and gives us a way to reason about the software we will eventually produce. Figure 3.6 is a very plausible model that we have arrived at directly from the concept map with very little effort.

In fact, we can say very little more about the structure of the system without introducing implementation details.

For example, if the keypad is a separate physical unit that communicates via I2C, then it makes perfect sense to have a NumericKeypad class to provide an interface to that.

However, if the BurglarAlarm is all one physical unit, then a separate interface class would seem to be unnecessary, and the structural feature,

Figure 3.6 First cut class diagram



NumericKeypad could be reduced to a method, getPIN() on the BurglarAlarm class itself.

Behavioral propositions

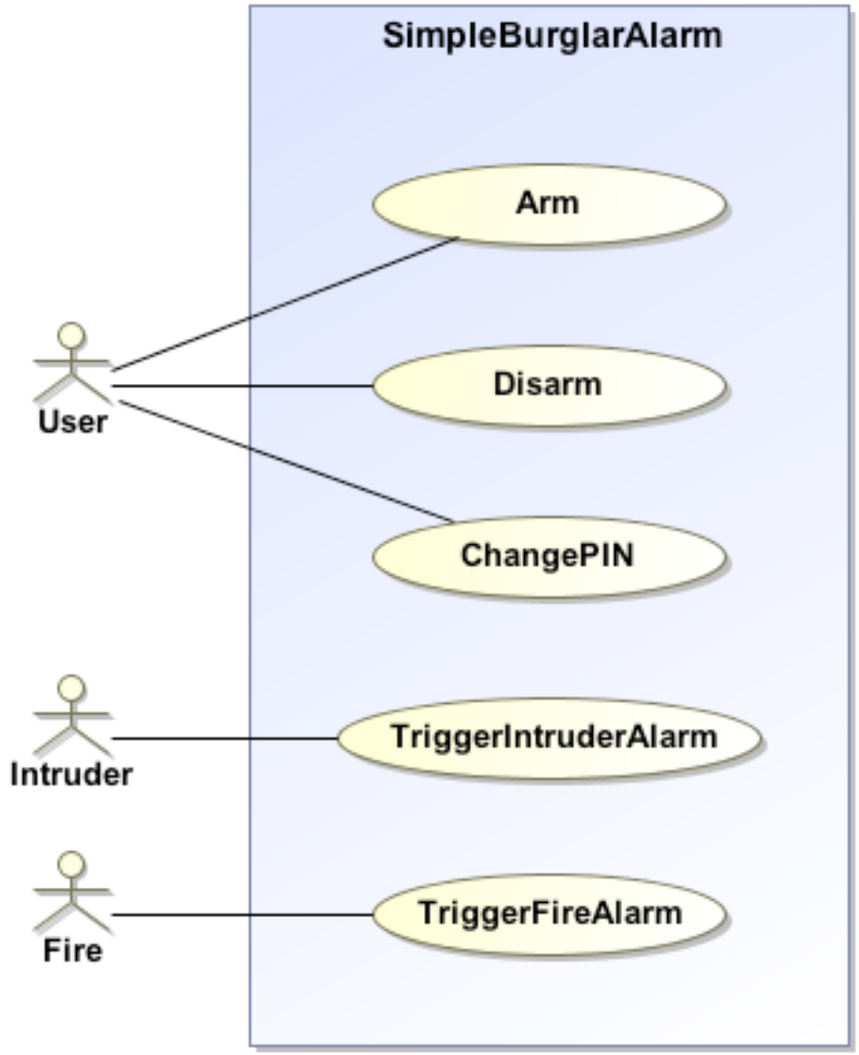
The behavioral propositions from the concept map indicate use cases, object interactions, object life cycles and class operations. Figure 3.7 shows a first-cut use case model for this system derived from the behavioral propositions.

Clearly, we can (and must) elaborate the behavior much further than this with sequence diagrams, activity diagrams, state machines and timing diagrams. All the information for at least a first attempt at these is available in behavioral propositions of the concept map.

Try it now! *Concept map for OLAS*

Create a concept map for OLAS using the OLAS vision statement as input.

Figure 3.7 First cut use case model



Compendium

Compendium is a mapping methodology and tool developed by the Compendium Institute. The essence of Compendium is that it is a way of structuring complex and heterogeneous pieces of information as specific types of node on a map. The nodes can be hyperlinked to other maps and other media.

We only give a brief introduction to the method here, and you can find much more information at the Compendium Institute.

Let's let Compendium speak for itself!

Figure 3.8 summarizes what Compendium is about. You can see from the figure that it is an ideal tool for working with a variety of unstructured, informal information.

Compendium provides two different view of your map. The map view (Figure 3.8) and also a list view, that simply lists the nodes and their attributes.

Compendium is a very flexible tool, because you can load different stencils to customize it for different types of modeling. However, we will only use the standard stencil, which provides the node types summarized in Figure 3.9.

Figure 3.8 What is Compendium?

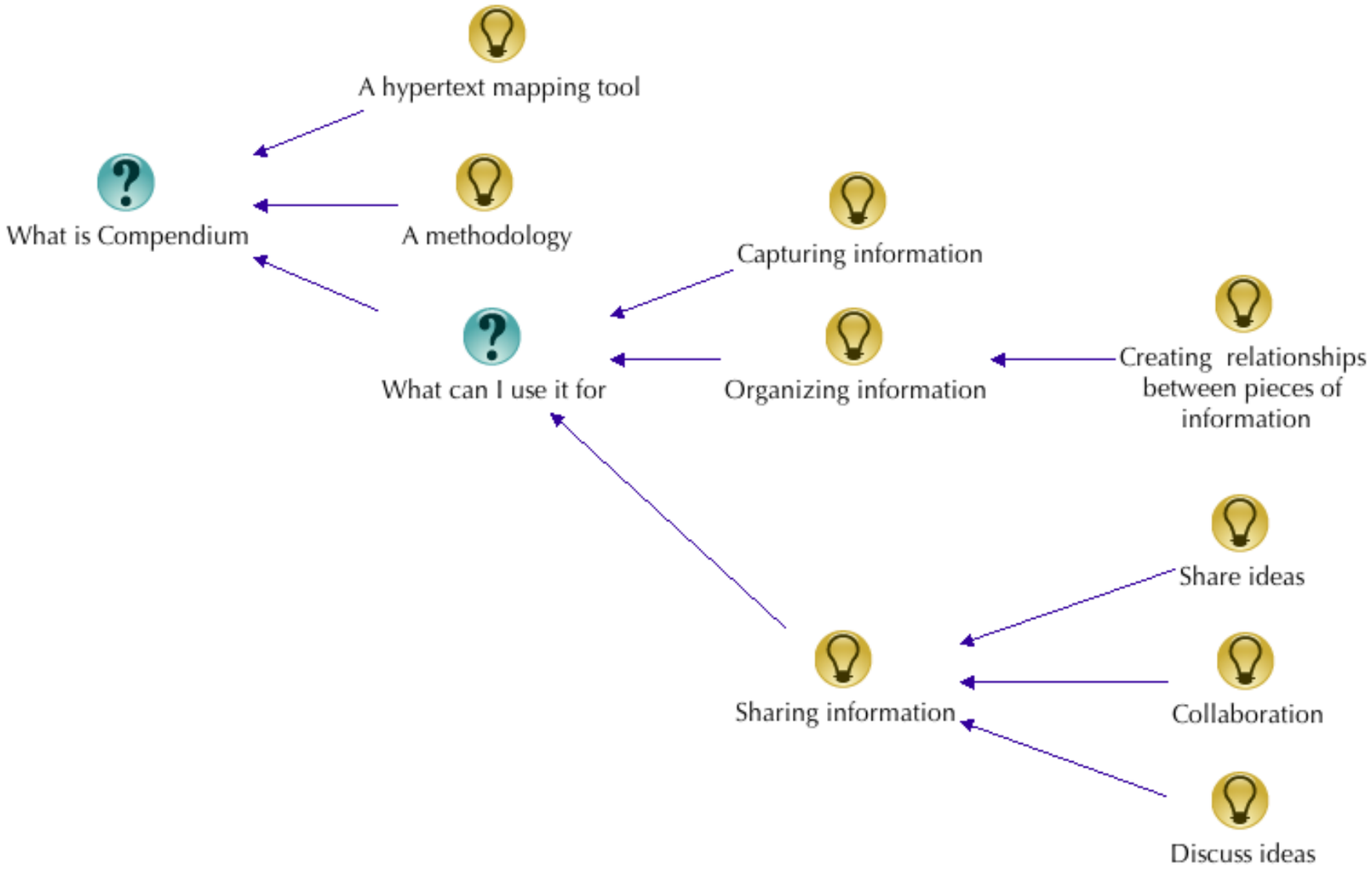
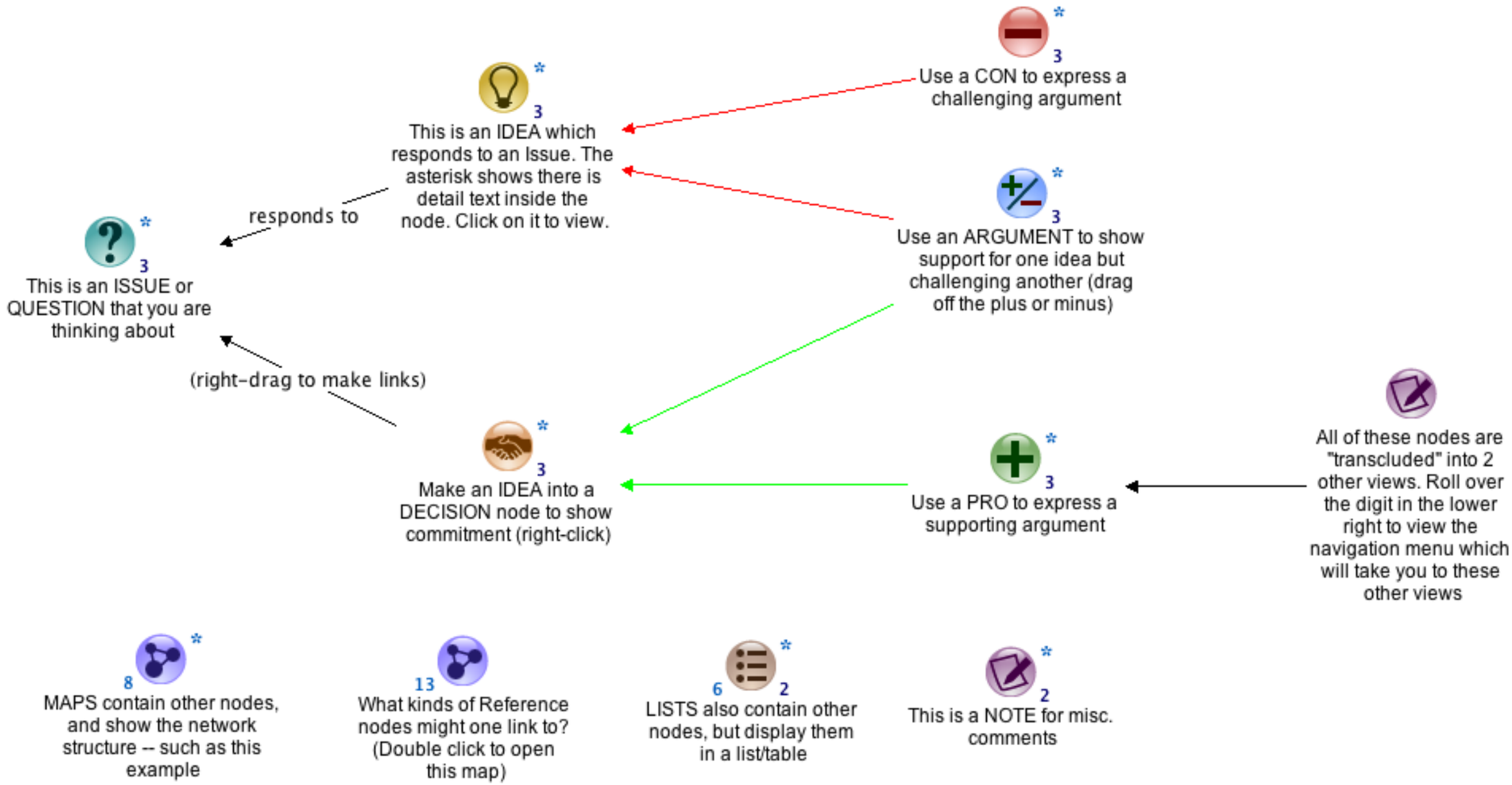


Figure 3.9 Compendium node types



You can see that you usually start a Compendium modeling session with a Question. This generates Ideas. You can show support for an Idea with a Pro, or challenge and Idea with a Con. You can also introduce an Argument which supports one Idea, but challenges another. Maps allow you to link to other Compendium maps. Notes allow you to annotate the map with comments.

One of the things that makes Compendium a very powerful mapping tool, is that there are three types of link between Nodes:

- Associative – a link between different nodes in the *same* view. This is just like the concept mapping we have already seen.
- Transclusive – the *same* node can occur in many *different* views, and a transclusive link is just a link between all of those occurrences. Each node has a transclusion indicator (digit in the lower right hand corner of the node) that allows you to track all of its occurrences.
- Categorical - a link between different nodes because they are in the same category. Nodes can be given one or more tags to define their categories.

If you use this structure, the Compendium tool creates structured concept maps that cause you to address and think about the problem in a particular way that is defined by the node types. However, as we have mentioned, you can also create your own node types (stencils) to create your own methodology.

In Generative Analysis, we use Compendium for Dialogue Mapping.

Dialogue Mapping

We can only give a short introduction to dialogue mapping here – just enough to enable you to use it effectively in meetings. We refer you to [Conklin 1] for more details.

Dialogue mapping is a facilitation technique for meetings that is based around Compendium. It is particularly useful for dealing with difficult problems that have a dimension of social complexity.

Conklin calls these difficult, socially complex problems "wicked problems" as opposed to "tame problems". This is a useful distinction. Table 3.2 compares the characteristics of wicked and tame problems.

If you've any experience with software engineering, you will immediately recognize that many software engineering problems can be classed as wicked problems! Typically, problems of requirements and analysis are likely to be wicked and technical problems are likely to be more tame.

For example, in software engineering a common wicked problem is the case where there are many project stakeholders, all with different, and perhaps

contradictory, points of view that have to be resolved into a definitive vision statement for the system.

You often find this kind of situation in the Inception phase of a project when everyone is trying to figure out precisely what the software system should do.

At this point, there can be lots of different potential requirements. There can also be a lot of politics and power plays that may have a big impact on the requirements for the system even though they may otherwise have little or nothing to do with the actual problem domain.

We call this the political domain of the system – it is the arena in which politics and power play themselves out, with a subsequent impact for better or (often) for worse on the system. Dialogue mapping is particularly good for disentangling the political domain from the domain of the problem itself. It can

Table 3.2 Wicked and tame problems

Wicked problem	Tame problem
You don't understand the problem until you have solved it (if ever).	Has a well defined, stable problem statement.
Wicked problems have no stopping rule. There is no ultimate solution. The problem is considered solved when: • You run out of resources and have to stop. • You have reached a solution that is good enough. Economics Nobel Laureate Herb Simon calls this *satisficing*.	Has a definite stopping point.
There is no right or wrong solution – only better or worse solutions.	There are criteria for objectively evaluating solutions as right or wrong.
They are unique in their specifics.	Belongs to a class of problems all with similar solutions.
Their solutions are one-shot. You can't try out many different solutions to see which is best.	You can try out many different solutions.
Their solutions are not givens – they must be arrived at through creatively sorting through a host of possibilities and generating new possibilities.	There is a limited set of possible solutions from which to choose.

also help you integrate the two maps, by helping you to achieve a consensus that balances their imperatives.

There are three things you need for dialogue mapping:

1. A shared display – for this you need the Compendium tool and a data projector.
2. A facilitator who knows the dialogue mapping technique.
3. An agreed notation – this is supplied by the Compendium method as previously described.

The easiest way to understand dialogue mapping is to look at a simple example.

Dialogue mapping by example

The main goal of dialogue mapping is to build a *shared understanding of the problem*, be it wicked or tame. A shared understanding will often lead to a solution provided all parties want a solution!

Jim set up a meeting with the librarians to discuss what OLAS should do. His goals for this meeting were to:

1. Understand, from the librarians' perspective, what OLAS needs to do (the problem domain).
2. Surface any concerns about the OLAS project and (if possible) to address those concerns (the political domain).

Figure 3.10 shows the mind map Jim used to plan the meeting.

Figure 3.10 "What should OLAS do?" dialogue mapping session plan

Jim used dialogue mapping as a facilitation technique, proceeding as follows:

Step 1: Introduce the central question "What should OLAS do".

Step 2: Build a Compendium concept map by listening to each participant and capturing their comments as a node that is related to other nodes.

The key skills in dialogue mapping are listening and summarizing. According to Conklin, experienced dialogue mappers go through a four-stage process when capturing information (Table 3.3)

It's important to understand that each iteration of this four stage process is about what *one person* says. It is *not* about what the group is saying.

This is in direct contrast to other meeting techniques that strive to find a group statement arising from a group consensus. In dialogue mapping, the focus is on achieving a shared understanding of the issues

Table 3.3 Dialogue mapping process

Step 1: Listen
• Pick one person and focus on what they are saying. • If there are many people speaking, that's OK - choose *one* and focus on that person.
Step 2: Guess
• Guess what they are trying to say and come up with a concise, short summary of what you think they are getting at (one line).
Step 3: Write
• Create a node of the appropriate type.
Step 4: Validate
• Check with the speaker that you have accurately captured what they wanted to say. • If not, help them clarify their statement and modify the map accordingly.

and you do this by capturing *individual* contributions and exposing them to the group for discussion.

You may arrive at a group consensus about something or you may not – the important thing is that you achieve a shared understanding of the problem or issues.

This shared understanding of the problem naturally leads towards a consensus, *provided* the participants actually desire or are constrained to solve the problem.

Patterns and anti-patterns in meetings

As well as the listening cycle, there are several strategies that dialogue mapping facilitators can use to deal with common meeting gambits in the political domain.

Unfortunately, this isn't an exhaustive list – it just covers some common cases. We have described these as patterns or anti-patterns. Patterns have a positive impact on the meeting and anti-patterns have a negative impact. Each pattern and anti-pattern has a name, description and solution.

Anti-pattern: All speak at once

Description

The meeting is chaotic – many of the participants are speaking at once as each jockeys for control, power and imagined territory. It may also be an attempt by a group of hostile participants to simply derail or not engage in the meeting.

Solution

Focus on one person, even when many are speaking. This tends to cut through chaotic power plays that can all too easily consume the bulk of a meeting. Participants realize that they won't get something on the map unless you are listening to them. And if it's not on the map, they have not made a contribution. Focusing on one person is also a powerful way to ensure that all stakeholders eventually get a chance to participate.

Anti-pattern: Shout the loudest

Description

A dominant personality tries to assert their point of view, power, control and/or territory simply shouting louder than anyone else.

Solution

Ensure that the person you are listening to gets their input added to the map – not the person who is shouting the loudest. After a while, this becomes obvious to the shouters who generally abandon their strategy when they realize it is failing to get their contribution recorded.

Anti-pattern: Domination through repetition

Description

A participant attempts to enforce their perspective on the others by simply repeating it until everyone else has lost the will to live and gives in to it.

Solution

Once you have captured something on the map to the satisfaction of the speaker, that issue has been exposed to the group and all participants may comment on it. Should the original speaker repeatedly return to the issue, just say "So you mean this", and indicate the issue on the map. If they have anything to add, just add it. After a short while, this makes any attempt at domination through repetition obvious and increasingly ridiculous.

Anti-pattern: Private agenda

Description

One of the participants tries to hijack the meeting by introducing a topic that they wish to discuss. This may also be a cynical attempt to derail the meeting for political reasons.

Solution

If a speaker goes off topic to explore a private agenda, create a new map, capture the essence of the new agenda and resolve to explore the new map in detail in a separate meeting. The private agenda is now exposed to the group. Should the participant try to hijack the meeting to the new agenda (domination through repetition), just say "So you mean this" (indicating the new map).

Pattern: Making a case for something

Description

A participant wishes to make a case for an idea.

Solution

Capture it as an option with pros.

Pattern: Making a case against something

Description

A participant wishes to make a case against an idea.

Solution

Capture it as an option with cons.

Anti-pattern: Challenging the basis of the discussion

Description

This is also known as "grenade throwing". A participant tries to undermine the meeting with a "Why are we looking at this?" or more specifically, "Why are we looking at this when we should be looking at <new topic>?" (which often introduces a private agenda). Although we've categorized it as an anti-pattern, the grenade thrower may have positive intent and may actually make a positive contribution to the meeting by identifying a flaw in the underlying presuppositions.

Solution

Create a question node for the "why" part of the challenge and a separate map for the “new topic” part of the challenge. Capture the essence of the new topic quickly in the new map with one or two nodes and resolve to look at it in a future meeting. It's important to separate the "why" part of the question, which appears on the current map, from the “new topic” part, which must appear on a different map. This allows you to deal with each separately.

Anti-pattern: But I don't own this map

Description

You have a non-participant who has no sense of ownership for the map.

Solution

Ensure that everyone participates, even if you have to specifically ask someone to make a contribution. Ensure that everyone gets something on the map. This will help to achieve a sense of shared ownership.

Anti-pattern: The rambling man [Dilbert 1]

Description

A participant (who is often a senior member of staff) starts rambling about something so wildly off-topic, for example their last holiday, or their recent UFO abduction experience, that you can't even open a new map for it. This may be simple stupidity, but it is more usually a power-play in which the participant is demonstrating their complete control of the meeting.

Solution

Ignore them. Encourage someone else to contribute to the map. As the participant realizes they have actually lost control, because the map is developing without their input, they will usually abandon this strategy and return to topic.

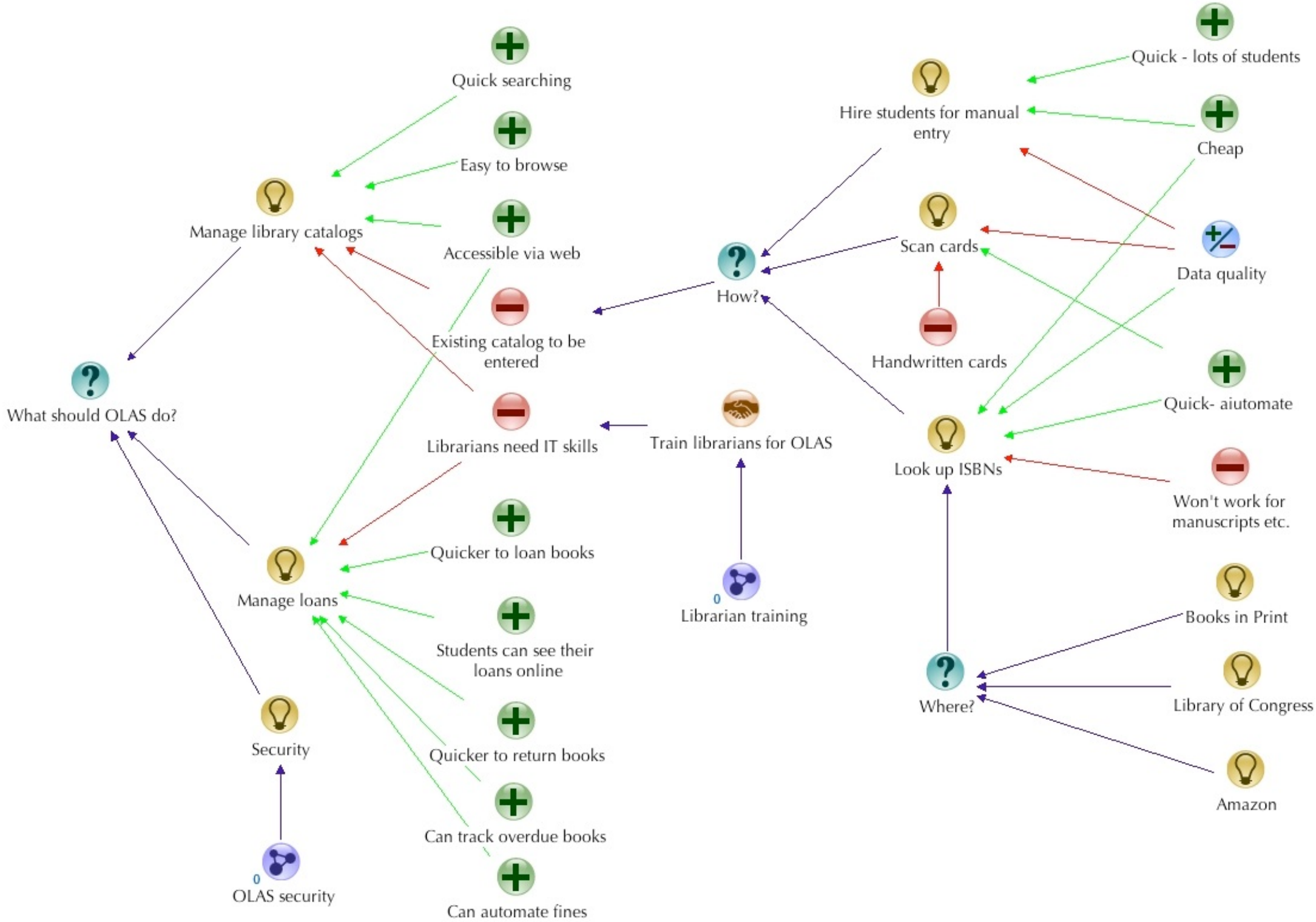
Dialogue map for “What should OLAS do?”

Figure 3.11 shows the dialogue map created in the "What should OLAS do?" meeting. Interestingly, this map doesn't really say that much about what OLAS should do. The librarians had major concerns about training and catalog entry, and the meeting rapidly began to centre around these issues. That's OK – one of the goals for the meeting was to expose these issues.

The most emotive issue was about IT skills for the librarians. The analyst agreed (the node "Train librarians for OLAS") that OLAS training was needed and opened up a new map (the node "Librarian training") to discuss requirements for that training in detail in a future meeting. This allowed him to defer discussion of this topic and move on.

You can see from the map that another major branch of the meeting began with the con "Existing catalog to be entered". Although this issue was not resolved in the meeting, the librarians generated many useful

Figure 3.11 Dialogue mapping "What should OLAS do?"



ideas. The next step would be to copy this branch of the map into its own map and arrange to explore it further in a follow-on meeting. Dialogue mapping is a generative process in which issues and ideas spawn new maps that may require further dialogue mapping or other types of analysis.

All the participants considered this meeting to be a success because:

- Each participant contributed something so there was a sense of shared ownership of the map.
- The participants had created a shared understanding of some (but not all) of the issues relating to OLAS.
- The librarians aired their concerns about training and catalog entry and felt that they were heard and "on the table" for future discussion. Useful ideas were generated for resolving some of the catalog entry issues but Jim recognized that a follow-on dialogue mapping session was required.

Decisions were generated to:

- Train the librarians.
- Discuss requirements for this training in more detail.

Thanks to dialogue mapping what might have been a chaotic and perhaps emotive meeting went fairly smoothly, and in a relatively short period of time, the analyst managed to achieve some useful results.

We'll look at more strategies for meetings in Chapter 8.

Structured writing

As well as creating visual maps of information, as an OO analyst you will also need to capture and present information as text - this is what we look at in this section. The technique we use is called Structured Writing.

We have purposefully written this section in the Structured Writing style. As you will see, Structured Writing makes heavy use of tables and lists. Whilst this can definitely make information easier to access, if over used, it can have a negative effect on readability. This is because the chunking of information breaks the reader's flow. Also, lists and tables can be *very* condensed forms of information, and can lead to fatigue. As with all techniques, common sense needs to be applied!

Text can obviously be a very effective way of communicating information. However, it can also be very ineffective, and it is important to understand why. We list the key factors that determine the effectiveness, or not, of textual communication in Table 3.4.

Structured writing is a way to address the issues raised in the table. It is based on research into how

Table 3.4 Key factors influencing the effectiveness of textual communication

Effective textual communication	Ineffective textual communication
Structure	
• Information is logically ordered. • Related information is chunked together. • There is a clear logical basis for the flow of information.	• Information is arranged randomly. • Unrelated information is chunked together. • There is no logical basis for the flow of information.
Volume	
• Information broken into small chunks.	• Information in large chunks.
Medium	
• Clear layout. • Appropriate fonts. • Different types of information are distinguished typographically. • Graphics are used where they add value.	• Messy layout. • Inappropriate fonts. • No typographical distinction between different types of information. • Graphics used in an ad-hoc manner.
Purpose	
• Information is logically ordered. • Related information is chunked together. • There is a clear logical basis for the flow of information.	• Confused.
Effect on reader	
• Interested in content. • Finds content easy to understand. • Finds content easy to learn. • Motivated to read.	• Not interested in content. • Finds content difficult to understand. • Finds content difficult to learn. • Not motivated to read.

human beings best organize and communicate information.

Although you might not know about Structured Writing, you have almost certainly already encountered aspects of it in corporate or educational writing. This is because Structured Writing has been one of the main influences on technical writing since 1965, when it was invented by Robert E. Horn, a psychologist at Columbia University. In 1982, it was commercialized as Information Mapping™ which is a registered trademark of Information Mapping Inc.

In this section we will discuss the non-commercialized version - Structured Writing.

The goals of Structured Writing are:

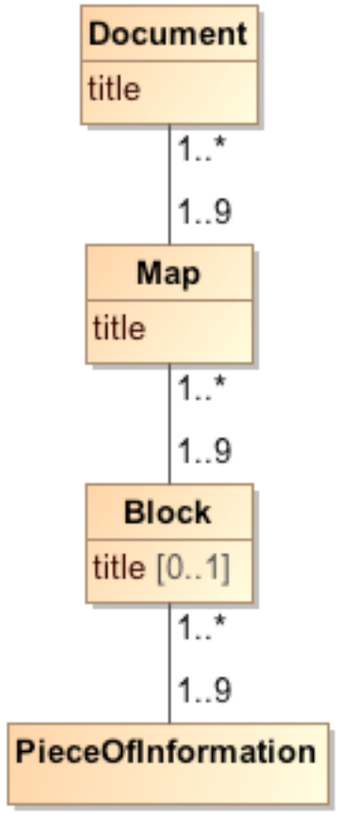
1. Break information into a basic set of elements called blocks.
2. Understand the different types of block.
3. Present each type of block in the best possible way so that readers can access the information quickly and effectively.

All of our technical writing is influenced by Structured Writing, and it works very well for us and for our readers. For example, our book "Enterprise Patterns for MDA" [Arlow 2] has been called by one reviewer, "a real page-turner", despite content that is usually considered to be dry as dust.

In the next few subsections, we will look at the specific details of Structured Writing:

- Structured documents.
- Principles for structuring information.
- Seven type of information.

Figure 3.12 Canonical structured document



STRUCTURED DOCUMENTS

A simple canonical form for a structured document is shown in Figure 3.12.

It comprises the following components:

- Document - a collection of up to about nine maps. These maps all have some relationship to each other to justify their inclusion in the same document. Each document has an explanatory title.
- Map - a collection of up to about nine blocks all about the same topic. Each map has an explanatory title. This would correspond to a chapter in a book, or a section in a technical report. Maps may have maps within them - for example, a book such as this has chapters and sections.

- Block - a chunk of information that is organized around a single topic and has a clear purpose. It is composed of several pieces of information e.g. sentences, formulae or figures. It has an explanatory title. You try to keep the number of pieces of information in a block to about nine pieces or less. The block corresponds to a paragraph.

Maps and blocks are considered to be reusable, and may be combined in many different ways. This is indicated in the model by a multiplicity of 1..*.

The 1..9 multiplicity comes directly from the famous paper "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information" [Miller 1]. This says that we can hold about seven plus or mins two chunks of information in short term memory.

The most controversial aspect of Structured Writing is that each block *must* have an explanatory title. The idea is that this makes blocks reusable, allowing blocks to be referred to and combined in different ways. However, for our purposes, this is unnecessary and unworkable - for example, imagine if every paragraph in this book had an explanatory title!

We use the *principle* of the block as a more refined type of paragraph *without* always giving it a descriptive title. This gives us a weak form of Structured Writing that is more suitable for our purposes in OOA.

Principles for structuring information

Structured Writing has several principles that are designed to make information easy to access, understand and remember. Different authors use slightly different sets of principles. We use the following nine:

1. **Relevance**:

Only include information relevant to the topic. This is the process of abstraction.

2. **Audience**:

- Who are the audiences for your document?
- What should they get out of your document?
- What do you want them to do with the information they get?
- How much of the document are they supposed to read?
- Can you ensure that different audiences can easily identify and access those parts relevant to them?

3. **Purpose**:

What is your purpose in communicating? Are you trying to:

- Inform - impart useful information.
- Persuade - convince them of a particular point of view.

- Move - cause them to act in some way.
- Guide - show them how to perform a particular activity.

4. **Chunking**:

You must chunk information into small easy-to-understand units:

- Short sentences with a simple structure.
- Short paragraphs with a simple structure. Try to keep to one topic per paragraph.
- Simple diagrams.
- Present information according to its type (see next section).

5. **Hierarchy**:

You must organize chunks into a hierarchy of meaning so that large chunks contain smaller chunks with a finer level of detail.

6. **Labeling**:

- Each chunk of information is given a descriptive title that clearly indicates its meaning.
- We relax this rule for blocks/paragraphs.

7. **Consistency**:

- Structure - titles, headings, sub-headings, etc.
- Presentation - layout, fonts, etc.
- Terminology - your project glossary will help with this (see Chapter 7)!

8. **Integrated graphics**:

Always use graphics embedded in the text to enhance its meaning. Never separate graphics into a separate section of the document!

9. **Accessible detail**:

- Enable readers to read to different levels of detail.
- Use chunking and labeling to separate and highlight the main points from the supporting points.

These nine principles give you a useful and practical set of guidelines for effective business and technical writing. You can see that they are all about understanding *what* you are writing about (relevance), *who* you are writing for (audience), *why* you are writing (purpose) and *how* you make that as easy for your readers as possible (chunking, hierarchy, labeling, consistency, integrated graphics, accessible detail).

10 Cs of business writing

As well as these Structured Writing principles, there are also the 10 Cs of business writing to consider. Some of these overlap with the Structured Writing principles:

1. Considerate - focus on the reader. Make the document as easy as possible for them.

2. Concise - state important points as concisely as you can. Use bulleted lists or highlight them so that they can be easily accessed.
3. Correct - ensure you know your facts. For example, it's a well known fact that 72.593% of all statistics are made up on the spot! Give references to support your facts.
4. Courteous - it costs nothing to be polite and you will gain enormously.
5. Clear - express yourself in ordinary language.
6. Concrete - make things as concrete as you can. Give concrete examples where you can.
7. Coherent - organize your ideas and make them easy to follow and understand.
8. Conversational - write as though you are talking to your reader.
9. Complete - make sure you have said everything you need to say.
10. Credible - be honest so that your reader develops trust in you. Avoid "weasel words" that create FUD. Be fair and balanced.

The 11th hidden C is that this is all just **C**ommon sense!

Seven types of information

One of the key concerns of Structured Writing is to represent information as simply, clearly and efficiently as possible. In order to do this, Structured Writing defines seven types of information and shows how to present each of these types most effectively.

Each type of information, and its preferred mode of representation is illustrated in Table 3.5.

Action tables are a very useful way of describing a procedure. We won't say any more about them here, because you will see lots of examples later in the book when we talk about use cases.

When you have a sequence of concepts, principles or facts to present, consider using a bulleted list to:

- Highlight the information.
- Keep related information together.
- Make information easy to access.
- Increase comprehension.

Compare the above with the following:

"A sequence of concepts, principles or facts may be presented as a bulleted list which has the advantages of highlighting the information and keeping related information together whilst also making it easy to access the information and increasing comprehension of the information."

We know which representation we prefer!

Table 3.5 Seven types of information

Type	Semantics	Presentation
Procedure	Instructions for doing something.	Action tables - concise, declarative, sequentially numbered instructions with one instruction per line. Each instruction begins with an action word (e.g., our use case specifications are written as action tables).
Process description	Explanations of how something is done.	Action tables. Diagrams to map the process (e.g., UML activity diagrams).
Structure	How something is composed – the relationships between its parts.	Text. Diagrams (formal or informal) e.g. UML class diagrams.
Concept	A general idea derived from specific instances.	One or more concrete examples. The basis of the conceptualization.
Principle	Rules about something.	Text (possibly highlighted with some typographical convention).
Fact	Objective information about something.	Text.
Classification	Types of things.	Text and diagrams to indicate the relationship of the classes and instances.

Structured writing example

In this section we take the OLAS Vision Statement and rewrite it using Structured Writing.

OLAS Vision Statement

Introduction

This document describes the initial vision for the Orne Library Automated System (OLAS).

OLAS is a computerized system that shall replace:

- The card catalog.
- The manual ticketing system.

OLAS shall at least provide the same functionality as these existing systems.

OLAS Stakeholders

- Librarians.
- University IT Department.

The existing system

The Orne Library has two collections of books:

1. The ordinary collection - ordinary reference materials.
2. The restricted collection - rare and valuable texts.

There are two manual systems in place that manage the library catalogs and loans:

1. Card catalog - consists of an index card catalog with a card for each title in the library.
2. Manual ticketing system - consists of a collection of tickets used to track borrowed books.

Security - access to the catalogs

- Borrowers may access the ordinary collection catalog without restriction.
- Borrowers may access the restricted collection catalog only if they have been vetted by the Librarians.

Reasons we need OLAS

Demand:

- Increasing demand for access to the catalogs has increased the librarians workload.
- Increased number of students has increased the librarians workload.

Exchange of information:

- The library must be able to exchange information about the restricted collection catalog electronically with Innsmouth Public Library who have already automated their library systems
- Access - borrowers should be able to access the Innsmouth Public Library catalog, if they have permission.

Efficiency:

- Borrowers can access the catalogs more efficiently using a web browser interface rather than having to go to the card file.
- Librarians can track overdue loans more easily.
- Librarians can manage loans more efficiently.

Table 3.4 shows some interesting statistics that compare the original OLAS Vision Statement to the Structured Writing version.

With Structured Writing there has been about a 23% *decrease* in the number of words. This isn't bad considering that the OLAS Vision Statement was already pretty good compared to many we have seen. However, there was a massive 520% increase in the number of paragraphs. This reflects the increased structure in the Structured Writing version.

Table 3.6 Regular vs. Structured Writing

	Original	Structured	% Change
Words	336	260	-23%
Paragraphs	5	31	520%

When you read the new version, it's surprising how clear and concise the OLAS Vision Statement can be made! Information that was previously buried in a flow of text now stands out. It takes much less time and effort to understand the rewritten OLAS Vision Statement.

Try it now! *Assessing information content*

Many of us feel that we are being overloaded by information, when perhaps we are merely being overloaded by words.

Another cause for the feeling of overload might be the sheer difficulty in finding the information you need.

Pick a technical book off your bookshelf at random. Open it to a page at random. Rewrite the page using Structured Writing. How concise can you make it without loosing important information? How much useful information did the page actually contain? You can measure this by the ratio of the number of words in your structured document to the number of words in the original text.

Complexity vs. profundity?

Before we leave this discussion of Structured Writing, we'd like to highlight the fact that many people are programmed to unconsciously interpret "difficult to understand" as "deep" or "profound", especially if the information comes from an "authoritative source".

In our opinion, far too much business and academic writing relies on precisely this misinterpretation.

This programming seems to happen at a very early age. Perhaps you failed to understand something as quickly as expected and were told that it was "your fault". This negative programming weakens your ability to distinguish between true complexity and poor communication, and it makes the world seem much more complex than it actually is. The techniques we present in this book are an antidote to this and will enable you to assess text (and other communications) intelligently.

You should realize that the best communicators can make even difficult subjects seem easy, and the worst communicators can make even easy subjects seem difficult.

For example, Jim routinely deals with two teachers, Andy and Zach. If Jim goes to Andy with a problem, it invariably turns out to be more difficult than Jim thought. But if Jim goes to Zach with a problem it invariably turns out to be easier than Jim thought!

Perhaps there are no difficult subjects, only inadequate communications? And even if this isn't true, it's certainly a very empowering and useful belief for any communicator to hold.

Try it now! *Assessing information difficulty*

Find a book or paper that you think is really difficult to understand. Take a page at random, and test the text against the principles of structured writing presented above.

Estimate how much of the "difficult to understand" is due to the inherent difficulty of the subject matter, and how much is simply down to poor writing.

Try rewriting the text using Structured Writing. How does this change how you feel about it?

Take a look at this quote from a book I (Jim) happen to be reading as I write this chapter:

"The importance of dealing explicitly with epistemological issues in a theory of HCI, namely, with issues related to how knowledge is gained, analyzed, tested, and used or rejected lies in our need to discriminate the validity, the reach, and the applicability of HCI knowledge coming from such widely different areas as computer science, psychology, sociology, anthropology, linguistics, semiotics, design, and engineering among others." [de Souza 1]

Now that seems pretty profound, doesn't it? In fact, it's a strongly hypnotic communication as we discuss later.

Let's ignore the information content for the moment and look critically at the structure of the communication. This single sentence contains:

- 63 words (!).
- Two words that aren't in common usage, epistemological and semiotics.
- One acronym, HCI.
- Three lists.
- One explicit logical operator, or.
- Implicit connecting logic that joins the lists together.

It's a long, complex sentence that is poorly structured for ease of comprehension, so naturally, readers will find it difficult to comprehend! Some may misinterpret this as profundity.

The sentence is a good example of what we call "drive-by writing". This is a writing idiom in which the author sprays the unsuspecting reader with long, comma-delimited lists of information whilst cruising quickly past on wheels of largely incomprehensible prose. In this particular case, the author shows no mercy and reloads twice!

Drive-by writing is an idiom that is often used to create an entirely bogus sense of depth and profundity.

Using the principles of Structured Writing, we can restructure this sentence as follows:

"Sources of HCI knowledge include:

- *Computer science.*
- *Psychology.*
- *Sociology.*
- *Anthropology.*
- *Linguistics.*
- *Semiotics.*
- *Design.*
- *Engineering.*

This theory of HCI raises important questions about how HCI knowledge is:

- *Gained.*
- *Analyzed.*
- *Tested.*
- *Used (or not).*

These questions allow us to assess it according to:

- *Validity.*
- *Reach.*
- *Applicability."*

But now it doesn't seem quite as profound. In fact, it seems rather obvious, even though it is still semantically ill-formed in several very important ways. This "sense of the obvious" is exactly and precisely what you are looking for in a good communication.

But, you might argue, the structured version is longer. It is certainly longer on paper, but in terms of text the structured version has 46 words, whilst the original version has 63. This is a reduction of about 27%, which is quite a lot. Now imagine that you could make all your business documents 27% shorter! In many cases, the reduction would actually be *much* greater than 27%.

Notice how we have chunked up the information in the sentence as:

- Possible sources of HCI knowledge.
- Knowledge related issues in this theory of HCI.
- Assessment criteria for HCI knowledge.

Each of these chunks is clearly delimited and easy to find and access. The connecting sentences establish relationships between these chunks thereby generating the meaning of the communication.

There are many other semantic issues with this communication that stand out now that it has been

expressed in structured writing. For example, what, specifically, does it mean to “test” knowledge, what is the “reach” of knowledge, how is that assessed? What does “among others” in the original mean? Does it mean:

- The list is in principle not enumerable – there are infinite possibilities.
- The list is in practice not enumerable – there are finite possibilities, but too many to list here.
- The list is in practice and in principle enumerable – but I decided not to.
- I have no idea if the list is enumerable or not, and have only listed those items I could think of.

Why hasn't the author finished the list? We just don't know.

The “drive-by” idiom allows the author to simply race over important issues, moving quickly on before we can notice them. We are left with a sense of perplexity that is all to easily confused with a sense of profundity. A better approach would be to address each issue by short explanatory notes after each bullet as needed.

There are many other semantic faults with the original sentence (and with the structured version). We recommend that you reanalyze it after you have learned M++ in Chapter 7.

Summary

This chapter has introduced the idea of Generative Analysis and some specific GA tools and techniques.

What is GA?

A new approach to OOA that tries to fill in the bits that other methods tend to leave out. In particular:

- Dealing with unstructured, informal information (most of the information you get is like this).
- Eliciting high quality information from information sources that are subject to:
  - Deletion.
  - Distortion.
  - Generalization.

The goals of GA are to teach you how to deal with these real-world human issues of software engineering and to provide you with suitable new analysis tools.

Analysis begins with informal, unstructured information

GA assumes that information is:

- Partial - essential information is deleted.
- Inaccurate - essential information is distorted.

- Generalized - essential information is modified by rules and beliefs about the world which may or may not be true.

To cope with this situation, GA provides you with:

- Specific ways to capture and organize unstructured, informal information
- Specific techniques to recover deleted, distorted and generalized information.
- A set of transformations to generate high quality information from low quality information - this is the "generative" bit.

Mind mapping

Mind maps work in an associative way similar to the human brain. You brainstorm to capture the ideas, creating associations as you go, and you analyze the information later. To do this you have to understand the information as you summarize it.

- Start at centre with main idea (use a picture if possible)
- Branches denote associated ideas

Try to use the one keyword per idea rule in order to learn the art of summarizing on the fly.

Use a simple six step process to create a mind map:

1. Generate.
2. Associate.
3. Review.
4. Incubate.
5. Organize.
6. New map.

In analysis, mind mapping can be used for:

- Capturing information.
- Generating ideas.
- Uncovering relationships.
- Memorizing information.
- Remembering information.
- Understanding information.

Concept mapping

Concept maps allow you to analyze information into concepts and propositions about those concepts. The concept map has the following elements:

- Focus question – determines the topic and scope of the map.
- Concepts – things or events derived from our knowledge about the world.
- Linking words – link concepts to form propositions.
- Propositions – units of meaning that affirm or deny something.

In analysis, concept maps can be used for "covert OOAD", because they are structurally similar to the first cut analysis class diagram:

- Concepts -> classes.
- Propositions:
  - Structural propositions -> relationships.
  - Behavioral propositions -> operations.

To turn concept maps into requirements, rephrase propositions as “shall” statements.

Compendium method

Compendium is a concept mapping methodology and tool developed by the Compendium Institute. Figure 3.8 summarizes what it can be used for.

Compendium has a simple structure comprising views, nodes and relationships between nodes.

Views:

- Map – a concept map comprising nodes, relationships and views
- List – nodes in a sortable column

Nodes: see Figure 3.9.

Relationships between nodes:

- Associative – a link between different nodes in the *same* view.
- Transclusive – links between the same node in the *different* views in which it appears. Each node has a transclusion indicator that allows you to track all of its occurrences.
- Categorical - link between different nodes because they are in the same category. Categories are defined by one or more node tags.

Dialogue mapping

A facilitation technique for meetings that is based around Compendium. It is particularly useful for dealing with "wicked" problems that have a dimension of social complexity. Wicked vs. tame problems are described in Table 3.2

Dialogue mapping occurs in a facilitated meeting, and you need the following resources:

- A shared display comprising Compendium (the tool) and a video projector.
- A facilitator who knows the dialogue mapping technique.
- An agreed notation as supplied by Compendium (the method).

The key facilitator skills for dialogue mapping are listening and summarizing. The dialogue mapping facilitation process is:

1. Listen.
2. Guess.

3. Write.
4. Validate.

Anti-patterns for meetings:

- All speak at once
- Shout the loudest
- Domination through repetition
- Private agenda
- Challenging the basis of the discussion
- But I don't own this map
- The rambling man

Patterns for meetings:

- Making a case for something
- Making a case against something

Structured writing

This is about effective communication in text. You must consider:

- Structure.
- Volume.
- Medium.
- Purpose.
- Effect on reader.

Goals of structured writing:

- Break information into elements called blocks.
- Understand the different types of block.
- Present each type of block in the best possible way.
- Organize blocks into maps.
- Organize maps into documents.

Structured documents

These are documents created according to the principles of Structured Writing and have the following structure:

- Document.
- 1..9 maps.
- 1..9 blocks.
- 1..9 pieces of information.

Maps and blocks should be reusable and you create them in your document by consistent use of chapters, sections and headings.

Here are the key principles for structured writing:

- Relevance
- Purpose
- Chunking
- Hierarchy

- Labeling
- Consistency
- Integrated graphics
- Accessible detail

Structured writing categorizes information into seven types, each of which is best presented in a particular way:

1. Procedure - action tables. These must be concise, declarative, sequentially numbered with one instruction per line
2. Process description - Action tables and diagrams
3. Structure – text and formal or informal diagrams
4. Concept - concrete examples and description of the basis of the conceptualization
5. Principle - text
6. Fact - text
7. Classification - text and diagrams

General principles of business writing

All business writing should conform to the 10 Cs:

1. Considerate
2. Concise
3. Correct
4. Courteous
5. Clear
6. Concrete
7. Coherent
8. Conversational
9. Complete
10. Credible

Complexity vs. profundity

It's a fact that poor communication is often confused with profundity. The best communicators make even difficult subjects seem easy whilst the worst communicators make even easy subjects seem difficult.

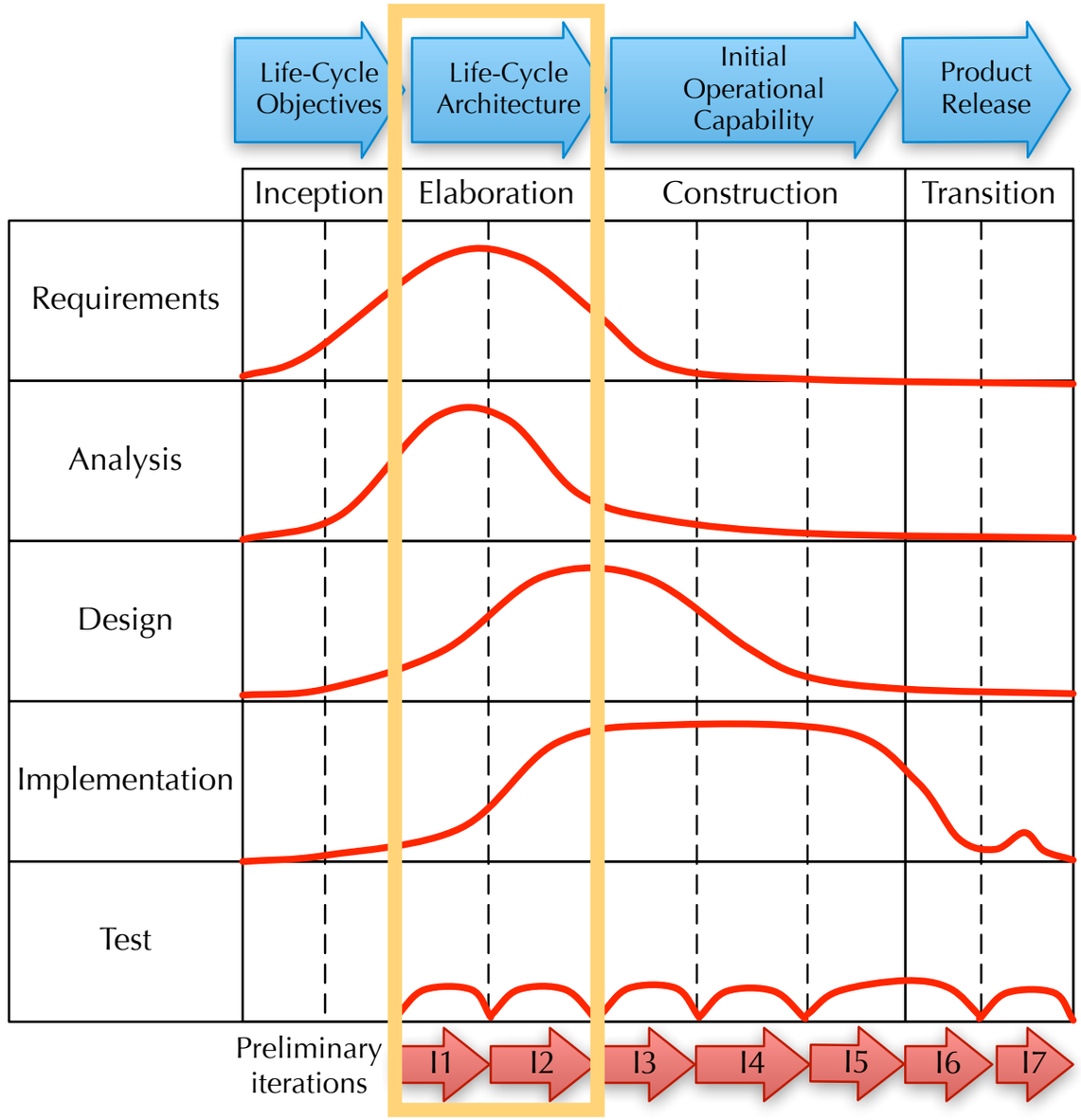
We adopt the principle that there are no difficult subjects, only inadequate communications.

Use structured writing, mind mapping, concept mapping and M++ (see later) to:

- Cut through authority and read the text as written (i.e., what it actually says).
- Distinguish between poor communication and inherent complexity.
- Make even difficult subjects easier.

Chapter 4

OLAS Elaboration



In this chapter we begin work on the OLAS Elaboration phase. We summarized Elaboration in chapter 2 and you might want to refer this chapter now to remind yourself of the goals, focus and milestone of Elaboration from the point of view of the OO analyst.