ROPES:
Rapid Object-Oriented Process for Embedded Systems

Bruce Powel Douglass
Chief Evangelist
I-Logix Inc.

This material is adapted from the author's upcoming book: *Doing Hard Time: Developing Real-Time Systems using UML, Objects, Frameworks, and Patterns* Reading, MA: Addison-Wesley, 1999

Introduction

Creating software is not truly an engineering activity in which the application of known rules against a problem statement repeatedly results in a highly reliable, cost-effective solution. The creation of good software is notoriously difficult and depends heavily on the skill of individual developers. The current state of the art sees software development as a technological craft rather than an engineering discipline. The creation of software is difficult primarily because software is essentially a real-world or imaginary-world model filled with complexity exceeding the capabilities of any one person to completely comprehend it at any one time.

The language and semantic models of software modeling are reasonably mature, although they are certainly still evolving. Nevertheless, the systems they model defy complete and formal description. The more complete the model, the more accurate and representative the model becomes, but also the more difficult to create and prove correct.

The developer is left with a problem. On one hand, there is a real-world system that must be created. On the other hand, there are semantic models and notations at the developer's disposal to capture the essential characteristics of the system being created and its informational and control context. How can the developer use these modeling elements to represent the system in a complete and accurate way that is understandable, portable, modifiable, and correct? What steps are necessary in this quest and in what order should they be applied?

The problem, as it stands, is similar to someone who knows a large number of French words and wishes to converse in French, but still does not know how to weave the words together into coherent sentences. The problem facing the developer, however, is not to just make sentences; it is to create a book of poetry in which the various phrases, refrains, and poems come together into an integrated whole. *How* to accomplish this task is ultimately answered by the development process.

A question commonly asked by managers is "Why bother with process?" Aside from keeping Scott Adams[1] dutifully employed, process allows us to

- produce systems of consistent quality
- reliably produce systems with complex behavioral requirements
- predict when systems will be complete
- predict system development cost
- identify milestones during development that enable mid-course corrections when necessary
- enable efficient team collaboration for moderate and large-scale systems

Terms and Concepts

A *methodology* consists of the following parts:

- a semantic framework
- a notational schema
- a set of sequenced work activities
- a set of deliverable work artifacts

The semantic framework and its notational schema together comprise the *modeling language*, such as the UML. The *development process* describes the activities that govern the use of the language elements and the set of design artifacts that result from the application of these elements in a defined sequence of activities.

Development Phases

The development process is divided up into large-scale activities, or phases, in an effort to simplify and clarify what needs to be done and when. These phases are as follows:

- Analysis
- Design
- Translation
- Testing

Analysis consists of identification of the essential characteristics of all possible correct solutions. *Design* adds the elements to the analysis that define one particular solution on the basis of the optimization of some criteria. *Translation* creates an executable, deployable realization of the design. *Testing* verifies that the translation is equivalent to the design and validates that the implementation meets all of the criteria for correctness identified in the analysis.

[1] Author of the Dilbert™ comic strip.

All phases of the development process work on a *model* of the system. This model must be an organized, internally-consistent set of abstractions that collaborate to achieve system description at a desired level of detail and maturity. It is important to understand that the analysis model, design model, translation (source code) model, and testing model are not different models which are somehow linked. They are (or at least ought to be) different *views* of the very same system model. Let me elaborate.

If you view the work artifacts of each phase as the results of capturing a different model, then you must permit those artifacts to vary independently. All software engineers have had the experience that the more abstract models can deviate from the source code. This is always a bad thing. An essential ingredient to the ROPES process is that all the artifacts of the model under development must always be of the very same underlying reality. Each phase of development is allowed to focus on different aspects of the same system, just as architects use different views (floor plans, electrical conduit plans, and plumbing plans, for example) to focus on different aspects of the same building.

The Analysis phase is further divided into the following subphases:

- Requirements analysis
- Systems analysis
- Object analysis.

Requirements analysis is the process of extracting the requirements from the customer and structuring them into a comprehensible form. *Systems analysis* builds more rigorously defined models and, based on the requirements,partitions system behavior into mechanical, electronic, and software components. It is used in the development of complex systems such as those in the aircraft, automotive, and factory automation industries. Many real-time domains may skip the systems analysis step because the system architecture is simple enough to not require it.

Both requirements and systems analysis are, in their very essence, functional descriptions of the system which rely heavily on behavioral and functional decomposition. The structural units of this decomposition are behaviors, functions, and activities. These structural elements are arranged into *system models*. The system model, along with the requirements model, form the *system specification*.

The third aspect of analysis is *object analysis*. This is a fundamentally different way to model the system under development. Object analysis consists of two subphases: *structural and behavioral object analysis*. Structural object analysis identifies the structural units of object decomposition which consist of classes and objects, their organizational units (packages, nodes and components), and the inherent relations among these elements. Behavioral object analysis defines essential dynamic behavioral models for the identified classes. Moving from systems analysis to object analysis requires a nontrivial translation step.

The design phase may likewise be divided into the following subphases:

- Architectural design
- Mechanistic design
- Detailed designs

Architectural design identifies the large-scale organizational pieces of the deployable software system. It consists of different views of the underlying semantic model. The *deployment view* organizes the elements of the object analysis into executable components which execute on various processor nodes. The *development view* organizes the non-executable artifacts (such as source code) into sections on which individuals work in order to enable team members to effectively manage their work and their collaboration with other team members. The *concurrency view* identifies concurrent time-based collaboration between the objects. The structural elements defined during architectural design constitute strategic design decisions that have wide-reaching impact on the software structure which are largely driven by the application of architectural design patterns, as discussed in [5] and [6].

An object itself is a small unit of decomposition. Objects collaborate in clusters to achieve larger scale purposes. A *collaboration* is a collection of objects collaborating or interacting together to achieve a common purpose, such as the realization of a use case. A *mechanism* is the reification of a collaboration (such as a design pattern). The process of "gluing" these mechanisms together from their object parts constitutes *mechanistic design*. Much of mechanistic design proceeds in similar fashion to architectural design by using mechanistic design patterns to "glue" objects together to facilitate collaboration.

Detailed design defines, structures and organizes the internals of individual classes. This often includes translation patterns for how model structures will be coded, the visibility and data typing of attributes, and the implementation of associations, aggregations and compositions in the selected programming language.

Design can proceed using either of two strategies. The most common is *elaboration*. Using this approach, design proceeds by refining and elaborating the analysis models until the model is implementable. In this approach, design information is added to the analysis model. The other approach, called *translation*, captures design information in a translator. The translator is then applied against the analysis model to produce the executable system. The translation approach commonly utilizes a context-specific framework and rules for translating analysis concepts into programming language statements automatically. Rhapsody™ is a UML-compliant design automation tool that works in exactly this way. The details of Rhapsody are discussed in Appendix B, and a demo copy of Rhapsody is provided on the CDROM accompanying this book.

Elaboration does not require as much tool support, but generally takes longer since the translator, once constructed, can be applied against many analysis models and must only be modified when the design decisions change. For real-time systems, the use of a real-time framework greatly facilitates construction since most of the common mechanisms

required for such systems are captured in the framework and used automatically by the translated analysis elements.

Is Automatically Generated Code a Reality?

A question I often hear is whether or not the technology for generating code automatically from UML object models is mature enough to use in real systems. Certainly, the elaborative approach is more common. However, automatically generated code can, and is, being used today in a variety of hard real-time and embedded systems. Most systems are between 60% and 90% “housekeeping’ and framework software. This software really varies little from system to system, but is typically written from scratch for each system. Some design automation tools not only provide this framework but also “glue” your object models into that framework. The Microsoft MFC library provides just such a framework for Windows programmers and greatly facilitates the development of Windows-based applications. Real-time frameworks do the same for embedded real-time systems today. Real-time frameworks provide a consistent way for handling event reception, finite state machine operation, control of concurrency, operating system abstraction, etc.

Ordering

Not only does a process consider the work activities and the products of that work, it must also define the order in which these activities are performed. The entire set of work activities organized into a sequence is called a *lifecycle*. Different lifecycle models are in use, with varying degrees of success.

Lifecycle models are important because, while the phases identify *what* must be done, the lifecycle model specifies *when* it must be done. Lifecycle modeling can be done both at the micro cycle level and the macro cycle level [1]. A micro cycle defines the ordering of activities within a portion of a macro cycle or within a single iteration of an iterative macro cycle.

Maturity

Deciding on these phases and the order of development-related activities is not enough. Each phase contains *activities* (things that developers do) and results in the form of *artifacts* (deliverable work products). These artifacts have different levels of *maturity*. The maturity of an artifact refers to both its *completeness* and its *quality*. The completeness of an artifact refers to how much of the intended scope of the artifact has been considered and taken into account within the artifact. An artifact’s quality is a comparison of the

specific artifact and the optimal qualities of artifacts of its kind, a process that is often subjective, leading to the statement that, like art, "I know bad software when I see it[2]."

Development Task Sequencing

Before we discuss the methods, procedures, and artifacts of the ROPES process, let's consider lifecycles in more detail. There are two primary approaches to sequencing development phases. The *waterfall lifecycle* is the most common. It orders the phases in a linear fashion. It has the advantage of simplicity and strong tool support. Other lifecycles are based on iterations of the phases. The advantage of *iterative lifecycles* is that they allow early risk reduction and better mid-course control over development projects.

Waterfall Lifecycle

The waterfall lifecycle approach is based on the serialization of development phases into a strict order. The artifacts produced within a phase must all meet a certain level of maturity before the start of the next phase is permitted. Thus, the waterfall lifecycle has the advantage of easy scheduling. It is, however, not without its problems. The primary difficulty is the problem of incompleteness – the artifacts of any phase cannot be complete until they are elaborated in subsequent phases and their problems identified. As a practical matter, experience has clearly shown that analysis cannot be complete until at least some design has been done, and design cannot be complete until some coding has been done, and coding cannot be complete until testing has been done. In the waterfall lifecycle, each of these phases depends on the artifacts from the previous phases being complete and accurate; but *this never happens*. So this model of the development process is inadequate. This is another way of saying that we *plan* one way but *do* another. Although this lifecycle facilitates planning, those plans are inaccurate, and we cannot, in general, tell where we are with any certainty. For example, if you admit that some of the analysis will have to be redone, and your team has scheduled 6 weeks for analysis and "finishes" in 5 weeks, are you late or not?

[2] One indication that a design or source code is "bad" is when the reviewer suddenly drops the source code on the floor, clutches his eyes and yells "I'm blind! I'm blind!" while running from the room. This is usually not a good sign.

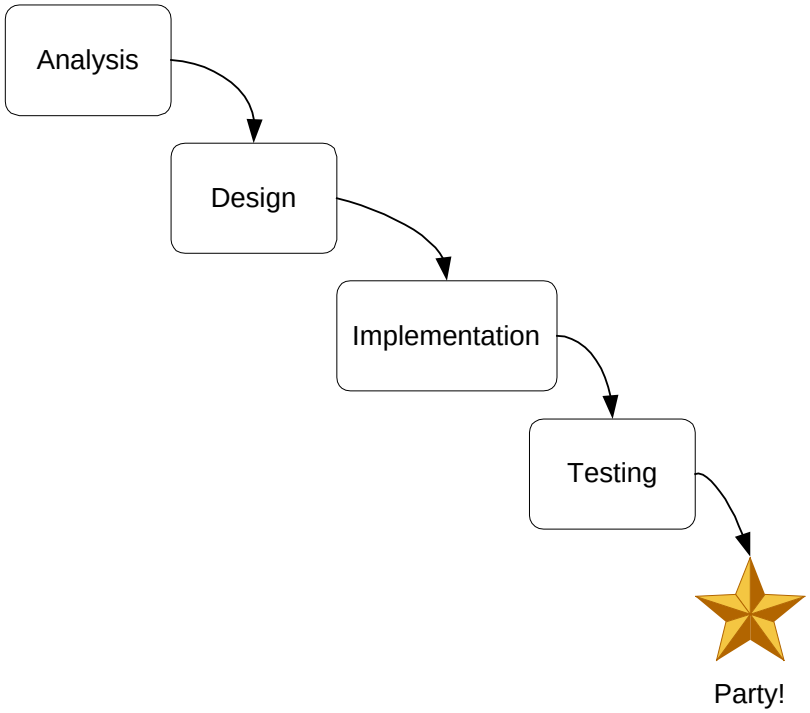

Figure 1: Waterfall Lifecycle

Iterative Lifecycles

Iterative lifecycles deal with the incompleteness problem by using a more representative (i.e. complex) model with the premise that each waterfall lifecycle model is not planned to execute only once, but possibly many times so that each "turn of the wheel" results in a *prototype*[3].. The iterative lifecycle allows for early testing of analysis models even though they are not complete. This is a powerful notion because it takes cognizance of the fact that the phases will not be complete the first time through and allows us to plan our projects accordingly. Iterative lifecycles are more flexible and can be tailored easily to the size and complexity of the system being developed. Additionally, by using reasonable criteria on the contents and ordering of the prototypes, we can assist project tracking with earlier feedback on high risk project issues.

This more accurate representation comes at a cost – iterative lifecycle projects are more difficult to plan and control, and they are not directly supported by project management tools. With the increased flexibility and accuracy afforded by these models, comes more complexity.

[3] Similar to the Hindu concept of reincarnation in which we come back in a higher form, but only if we've been good.

Barry Boheim published early work on iterative lifecycles[3] which he referred to as the *spiral lifecycle*. Managers often have concerns as to whether or not the spiral is convergent[4] and how to plan and manage projects along these lines. This topic will be dealt with in later in this paper.

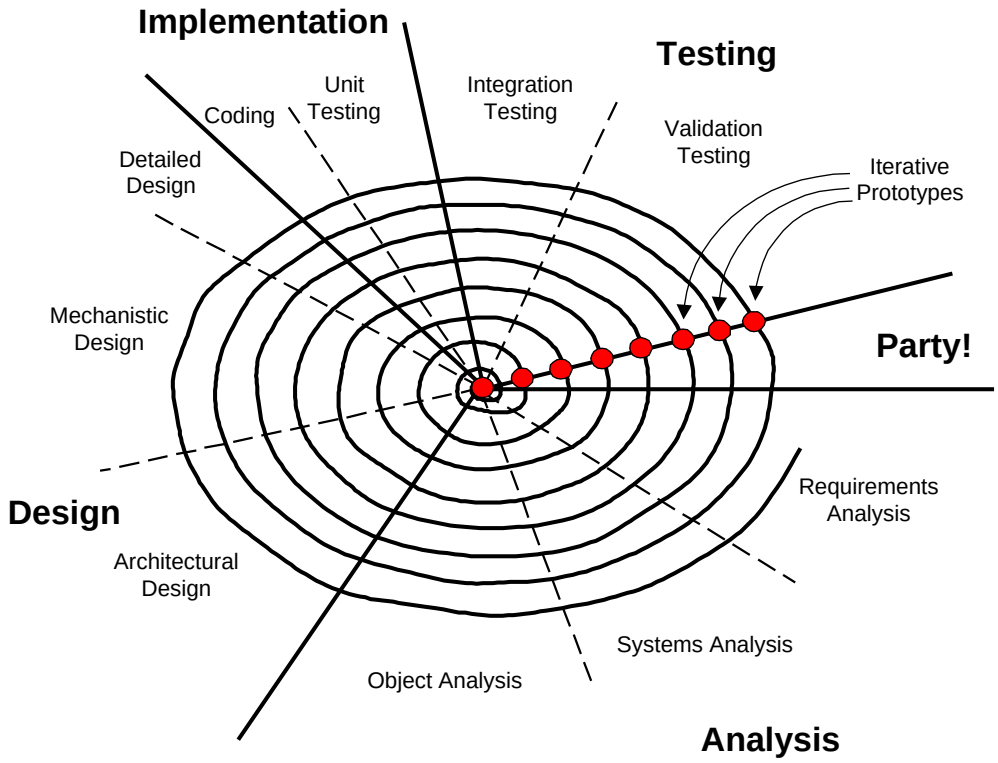

Figure 2: Iterative Lifecycle

Prototyping

A *prototype* is an instance of a system model. In the software context, they are almost always executable in some sense. Prototypes may be either *throw-away* or *iterative*. A throw-away prototype is one which will not ultimately show up in the final product. For example, it is common to use GUI-building tools to construct mock-ups of user interfaces to provide early exposure of a planned interface to a set of users. This visual prototype, which might be done in Visual Basic, will not be shipped in the final product, but helps developers understand their problem more completely and communicate with non-technical users.

[4] Hence its other name as the "software spiral of death."

Iterative prototypes are executable models of the system that ultimately will be shipped in their final form as the final product. This means that the quality of the elements (design and code) that make up an iterative prototype must be of higher quality than a throw-away prototype.

Iterative prototypes are constructed by modifying the previous prototype to correct uncovered defects and add new elements. In this way, prototypes become increasingly elaborate over time in much the same way that trees grow in spurts, indicated by their growth rings. At some point, the prototype meets the system validation criteria, resulting in a shippable prototype. The prototype continues to be elaborated even after release, constituting service packs, maintenance releases, and major upgrades.

Each prototype should have a *mission*, a statement of the primary purpose for the construction of the prototype. The mission might be to reduce some specific risk or set of risks, provide early integration of system architectural units, implement a use case, or provide an integrated set of functionality for alpha or beta testing. A common strategy is to implement use case prototypes in order of risk.

Many designs are organized into architectural layers based on their level of abstraction. The most concrete layers exist at the bottom and consist of abstractions related to the low-level manipulation of the underlying hardware. The more abstract layers exist above the more concrete layers, and realize their behavior through the invocation of lower-level services. At the very top layer are the abstractions that exist within the application domain. This is a marvelous way to decompose a set of abstractions, although rarely useful to organize the content and delivery of prototypes. If the prototypes are organized around use cases, then the implementation policy is *vertical*. By this is meant that the prototype typically will contain elements from most or all of the layers of abstraction. This approach is also called *incremental development*.

Figure 3 below shows the concept behind vertical prototyping. The model is structured as a set of horizontal layers, but the prototypes implement relevant portions from most or all of these layers.

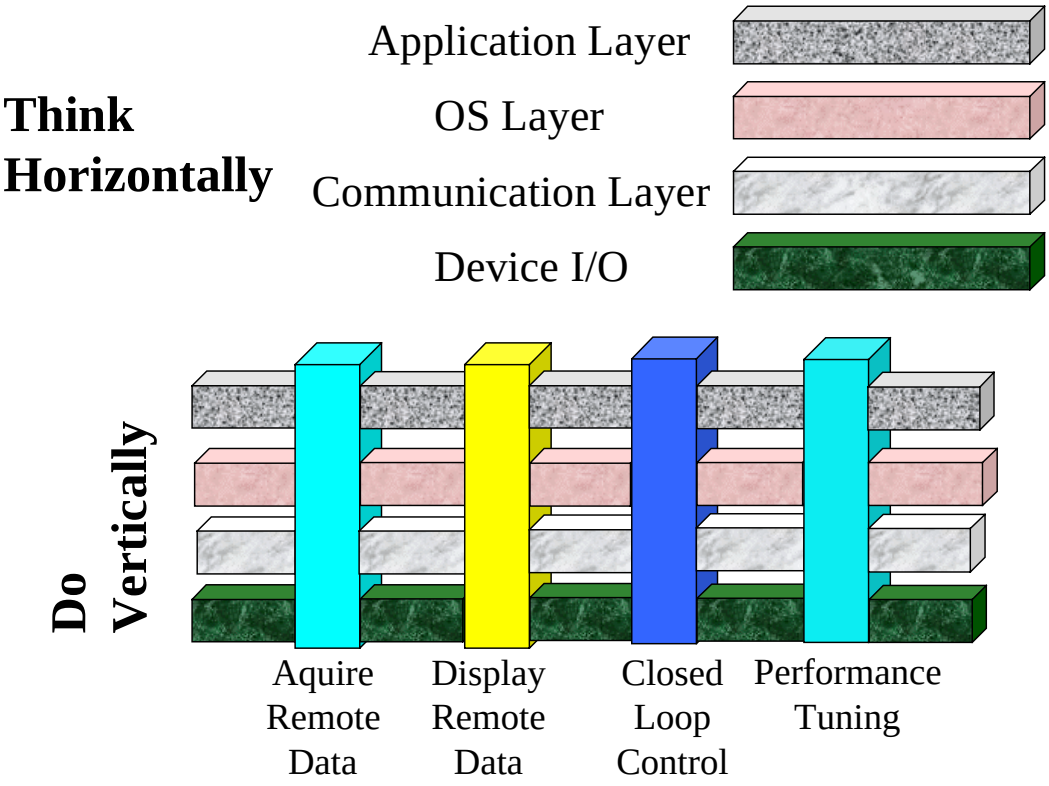

Figure 3: Vertical Prototyping

The technology that really enables the iterative prototyping process is *automatic translation of description models into executable models.* Executable models are crucial because the only things that you can really validate are things that you can execute. For example, an early prototype to realize a single use case might consist of a few dozen classes, five of which may non-trivial state machines. To manually create an implementation of a simple use case would certainly take, at minimum, several days, and might require up to several weeks to perfect. The developer would have to answer many low-level questions like "What is the implementation of a state? An event? How do I implement time-outs? How should I handle events that are not yet being handled? What about crossing thread boundaries?" All these questions must be answered, but they are ancillary to the work at hand – judging whether or not the collaboration of objects realizing the use case is, in fact, correct. With a translative approach, the code can be generated, compiled, and run within seconds.

Scheduling and Estimation

There is ample evidence in the software industry that we have not been tremendously successful in terms of predicting the resources, time, or budget required to develop

products[5]. PC Week reported in 1994 that 52% of all software projects in the US were at least 189% over-budget, and 31% of all projects were cancelled before completion. Other studies give similarly depressing statistics.

There are a number of theories as to exactly why this is the case. A common theory is that software engineers are morons who can't schedule a trip to the bathroom. Understandably, software engineers often take the opposing view that managers are clueless pointy-haired individuals that wouldn't know an accurate estimate if it ran over them with its car. And, of course, everyone gets to blame marketing.

My own personal view is that everybody gets to share the blame, if any is to be had, and this poor performance is due to both technical and sociological issues.

The sociological issues include the following:

- the unwillingness of many managers to accept reasonable estimates
- an authoritative managerial style
- accuracy in estimation is often actively discouraged[6]
- non-stationary requirements
- unwillingness on the part of management to believe that software is *inherently* difficult
- a lack of understanding of the true cost of replacing personnel
- managers giving the "desired" answer[7]
- engineers providing the "desired" answer knowing full well its inaccuracy
- the use of schedules as a motivational tool in the belief that stress is a good motivator

Of these, the last is the least defensible and the most insulting to the engineering staff, and has the greatest long-term cost of the business.

The technical issues are follows:

- software involves invention; and estimation of invention is fundamentally difficult
- engineers are not trained in estimation techniques
- managers use waterfall rather than iterative lifecycles

Of the two, the technical problems are by far the more solvable.[8]

---

[5] This is, no doubt, the source of the phrase "There are lies, damn lies, statistics, fantasies, and software schedules."

[6] I once had a manager tell me in regards to an estimate "That's the wrong answer – do it again."

[7] Another manager once told me "You have 6 months to do this. How long will it take you?" Even *I* knew the answer to that one!

Advantages of Accurate Schedules

Despite their rarity, accurate schedules have a number of business advantages. Probably the main one is the selection of appropriate projects to pursue. I believe one of the reasons why the failure rate of software projects is so high is that many of these projects would not have been started had the true cost of the project been known at the start[9]. Projects which will not recoup their costs and return a profit generally ought not to be started in the first place. However, in the absence of accurate information as to the cost of projects, making good decisions about which projects to do is difficult. Accurate schedules can reduce the loss due to starting projects that should not have been begun in the first place.

Another advantage to accurate schedules is that it facilitates the planning of ancillary activities, such as

- Manufacturing
- Disk copying
- Documenting the software
- Preparing ad campaigns
- Hiring employees
- Gearing up for the next project

It is very expensive to gear up a manufacturing effort, which might include the purchase of expensive equipment and the hiring of manufacturing personnel, only to have manufacturing wait for a late project to complete.

Another benefit near to my own heart is the lowering of the very real "human cost" of software engineering. It is all too common for software engineers to work inhuman hours[10], often for months at a time[11], in order to achieve goals which were unrealistic in the first place. The whole notion of the software-engineer-as-hero becomes much less appealing when one actually has a life (and family).

Of course, many managers use "aggressive[12]" schedules. As Tom DeMarco and Timothy Lister [4] point out, using schedules as a motivational tool means that they cannot be used to accurately predict the time and cost of the project. Even from a strictly business financial aspect, this approach fails miserably in practice, not to mention the fact the

---

[8] You can lead a horse to water, but you can't make him write good software.
[9] I once had a manager tell me in response to an estimate, "That might be correct, but I can't report that to upper management. They'd cancel the project!" He seemed unconcerned that this might actually be the best thing from a business standpoint.
[10] I once worked 120 hours per week for three weeks to make a schedule date.
[11] On another project, I worked 90 hours per week for 6 months.
[12] As in "the probability of coming in at the scheduled time is less than the probability of all the molecules in the manager's head suddenly Brownian-motioning in the same direction causing his head to jump across the table."

premise underlying it is insulting to engineers.[13] Aggressive scheduling tends to burn the engineering staff out and they take their hard-won expertise and use it for the competition. The higher turnover resulting from a consistent application of this management practice is very expensive. It typically costs about $300,000 to replace a $70,000 engineer. Using schedules as planning tools instead of motivation avoids this problem.

Difficulties of Accurate Scheduling

The advantages of accurate schedules are clear. We are, nevertheless, left with the problem of how to obtain them. As stated above, the difficult problems are sociological in nature. I have consulted for a number of companies and have greatly improved their estimation and scheduling accuracy through the consistent application of a few key principles.

Estimates are always applied against estimable work units (EWUs). EWUs are small, atomic tasks typically no more than 80 hours in duration. The engineer estimating the work provides three estimates:

- Low/Optimistic (Engineers will beat it 20% of the time)
- Mean (Engineers will beat it 50% of the time)
- High/Pessimistic (Engineers will beat it 80% of the time)

Of the three, the most important is the 50% estimate. This estimate *is the one which the engineer will beat ½ of the time*. The central limit theorem of statistics states that if all of the estimates are truly 50% estimates, then overall, the project will come in on time. However, this estimate alone does not provide all necessary information. You would also like a measure of the perceived risk associated with the estimate. This is provided by the 20% and 80% estimates. The former is the time that the engineer will beat only 20% of the time, while the latter will be beat 80% of the time. The difference between these two estimates is a measure of the confidence the engineer has in the estimate. The more the engineer knows, the smaller that difference will be.

These estimates are then combined together to come up with the estimate actually used in

$$\frac{Low \; + \; 4 * Mean \; + \; High}{6} * EC$$

the schedule:

The EC factor is the “Estimator Confidence Factor.” This is based on the particular engineer’s accuracy history. An ideal estimator would have an EC value of 1.00. Typical EC values range from 1.5 to 5.0.

[13] “If I don’t beat them with a stick, they won’t perform!”

In order to improve their accuracy, engineers must track their estimation success. This success is then fed back into the EC factor. A sample from an estimation notebook is shown in Table 1 below.

Table 1: Sample From Estimation Notebook

Date	Task	Opt	Mean	High	Unadjusted Used	EC	Used	Actual	Dev.	% Diff
9/15/97	User Interface	21	40	80	43.5	1.75	76.1	57	17	0.425
9/17/97	Database	15	75	200	85.8	1.75	150.2	117	42	0.56
9/18/97	Database Conversion	30	38	42	37.3	1.75	65.3	60	32	0.842
9/20/97	User Manual	15	20	22	19.5	1.75	34.1	22	2	0.1

To construct a new EC value, use the formula below:

$$EC_{n+1} = \Sigma(\text{Deviations using } EC_n)/(\#\text{ Estimates}) + 1.00$$

For example, to construct a new EC value from the table above, you would compute the following:

$$EC_1 = (0.425 + 0.56 + 0.842 + 0.1)/4 + 1.00 = 1.48$$

In this example, the engineer went from an estimator confidence factor of 1.75 to 1.48 (a significant improvement). This EC value is then used to adjust the "Unadjusted Used" estimate to the actual "Used" estimate for insertion in the schedule. It is important to track estimation success in order to improve it. In order to improve a thing, it is necessary to track a thing.

A schedule is an ordered arrangement of EWUs. It is ordered to take into account inherent dependencies, level of risk, and the availability of personnel. The process of creating a schedule from a set of estimates is well covered in other texts and is not discussed here.

The ROPES Macro Cycle

The ROPES process is based on an iterative lifecycle which uses the standard UML metamodel for its semantic framework and notation. It encourages (but does not require) automatic code generation within a real-time framework from its models to facilitate rapid generation of prototypes. Although the elaborative approach can be used, the translative approach creates deployable prototypes much faster and with much less effort.

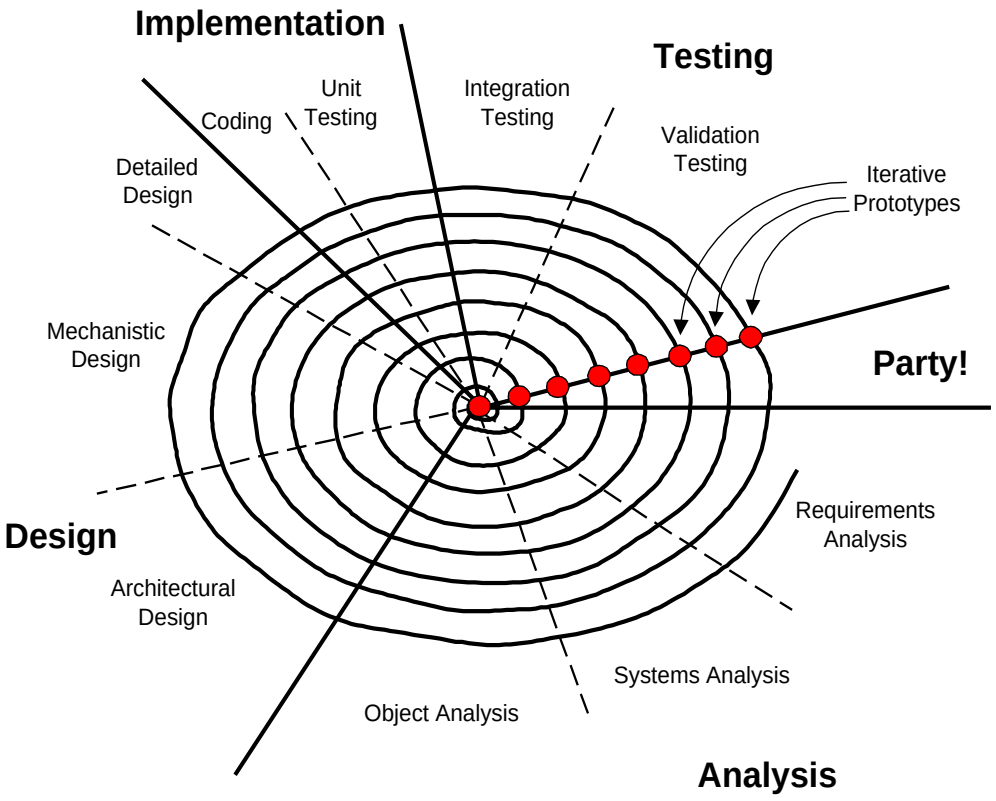

Figure 4: ROPES Process

The purpose of any process is to improve the product resulting from the process. In particular, the primary purposes of any development process are to do the following:

- increase the quality of the end product
- improve the repeatability and predictability of the development effort
- decrease the effort required to develop the end product at the required level of quality.

Other requirements may be levied on a development process such as "to aid in accountability, visibility, and regulatory approach." If your development process does not provide these benefits, then it is a bad process and should be replaced or revised.

To achieve its primary purposes, a development process consists of phases, activities, and artifacts. A phase is a set of activities that relate to a common development purpose which is usually performed at a consistent level of abstraction. Each phase contains *activities* (things that developers do) and *artifacts* (deliverable work products) which result from those activities. The primary artifacts of the ROPES process phases are shown in Figure 5 below.

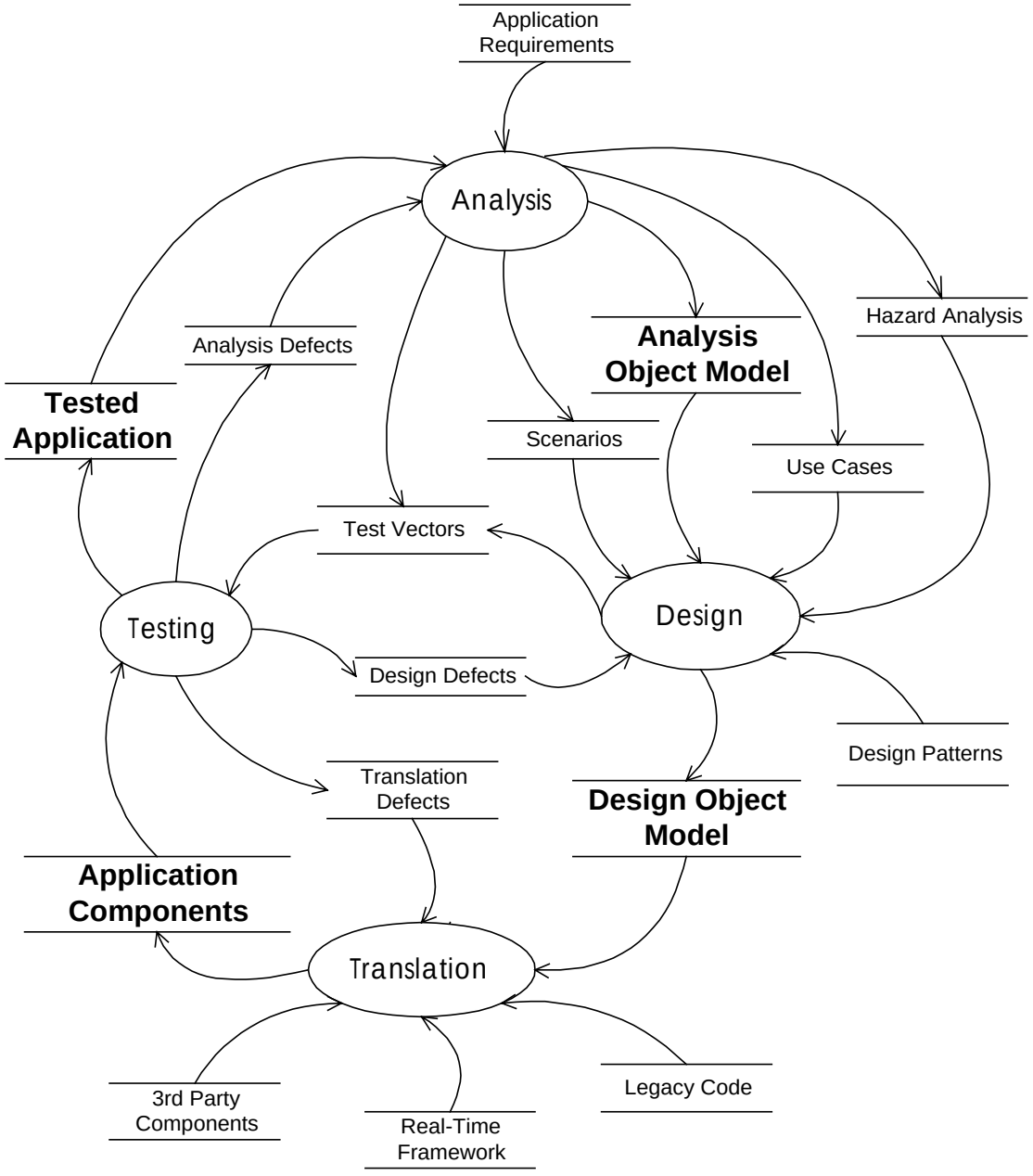

Figure 5: ROPES Process Artifacts

Figure 5 shows the different phases of a single iteration. In practice, these phases iterate over time to produce the development prototypes. The primary artifacts are the versions of the system model produced by the different phases. A prototype really consists of not only the executable thing, but also of the artifacts used to generate it.

Each prototype in the ROPES process is organized around system use cases. Once the set of use cases is identified and characterized, they are ordered to optimize the effort. The

optimal ordering is determined by a combination of the use case priority, risk, and commonality.

For example, suppose you lead a team that builds an embedded communications protocol which is modeled after the 7-layer ISO communications protocol standard. The layered architecture appears in Figure 6 below.

- ***Application Layer*:** Application services
- ***Presentation Layer*:** Data encode/decode
- ***Session Layer*:** Organized extended dialogs
- ***Transport Layer*:** Connection-oriented and Connectionless transportation
- ***Network Layer*:** packet routing
- ***Data Link Layer*:** Error free frame transmission, flow control
- ***Physical Layer*:** Encapsulate and abstract physical media characteristics

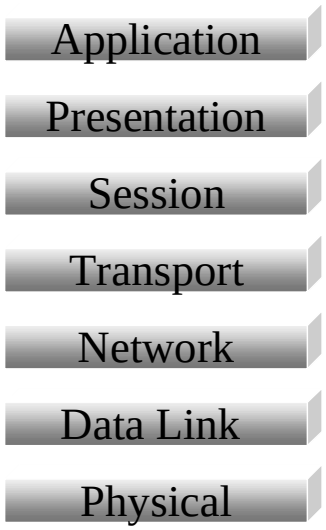

Figure 6: Communication Protocol Architecture

This protocol is to be implemented on several different processors using different RTOSs and compilers. The processor platforms include a heterogeneous set of 16-bit, 32-bit, and DSP processors. Additionally, some subsystems are to be implemented in C while others in C++. The team wants the same source code to compile and operate on all platforms, so this is identified as an early risk. You agree, in this case, to create an object model and implement the protocol using only the standard C that is also present in C++.

You want to reduce risk while providing the subsystems the ability to do at least some useful communication with others as early as possible. You then identify a series of prototypes which optimize both risk reduction and early availability as shown in Figure 7 below.

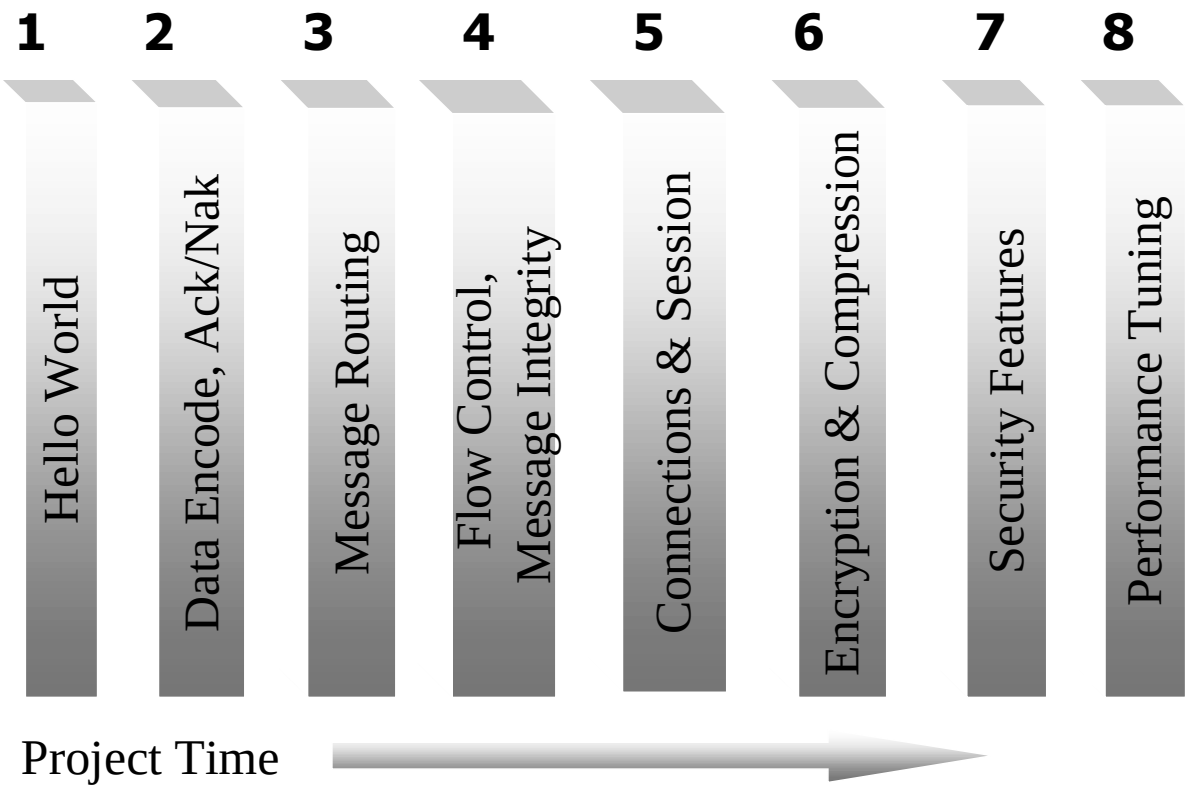

Figure 7: Prototypes for Communication Protocol Project

In my own experience, I have led just such a development project, and with great success. Interestingly, as a result of the "Hello World" prototype, which was compiled on every subsystem, we found that none of the ANSI C-compliant compilers were, in fact, ANSI C-compliant. By creating this prototype first, we were able to find the minimum set of language translation rules that worked on all subsystem compilers. At this point, only a small amount of code had to be rewritten. Had we waited until later, the rewrite would have been much more substantial and costly.

In the following sections, we will look at each phase of the ROPES process (the ROPES micro-cycle) and identify activities and deliverables. Each sub-phase is described in detail, including its *activities* (things you do during the sub-phase), the *metamodel elements used* (things you manipulate or define), and the *artifacts* (things produced). In some situations or business environments, some of the activities or artifacts of the sub-phases may be skipped. For example, if your system is not safety-critical, you won't spend the effort and time to produce a hazard analysis document. Pick the activities and artifacts that make the most sense for your system and work culture.

Analysis

As mentioned earlier, the analysis of a system identifies all characteristics of the system which are essential to correctness. That is, a system should be devoid of design characteristics that are free to vary. Figure 8 shows the subphases and deliverables for the ROPES analysis activity, including the generated and delivered artifacts.

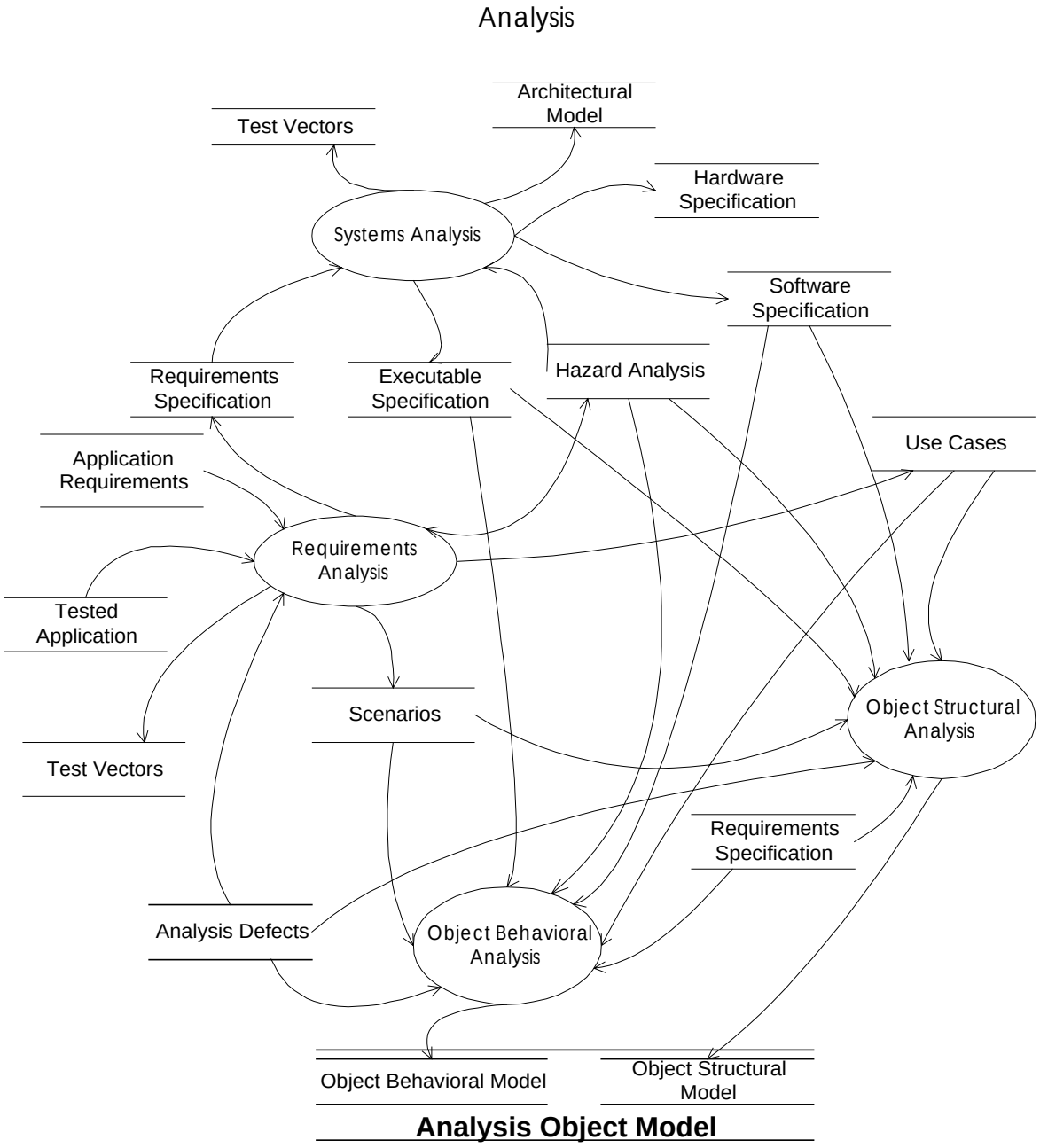

Figure 8: ROPES Analysis Model Artifacts

Requirements Analysis

Requirements analysis extracts requirements from the customer. The customer may be anyone who has the responsibility for defining what the system does. This might be a user, a member of the marketing staff, or a contractor. There are several barriers to the extraction of a complete, unambiguous, and correct set of requirements.

First, most customers understand the field use of the system, but tend not to think about it in a systematic manner. This results in requirements that are vague or incomplete. Even worse, they may specify requirements which are mutually exclusive, impossible, or too expensive to implement. The customer will forget to specify requirements which seem too "obvious" and will specify requirements which are really implementations.

Second, customers will specify what they think they need. However, this is often not the best solution to their problem. If you deliver a system which is what the customer requested but does not meet their needs, you will nevertheless shoulder the blame. It behooves you to figure out what the **real** requirements are.

It is important to note that requirements analysis is almost exclusively a functional view. It does not identify objects or classes. It comes as a disappointment to many beginners that after spending a significant effort identifying use cases and elaborating their behavior, not a single object is identified by the end of requirements analysis. That remains the task of object analysis, a later sub-phase.

Activities

The basic activities of requirements analysis are as follows:

- Identify the use cases and associated actors
- Decompose use cases with the following relations:
  - *generalization*
  - *uses (or includes)*
  - *extends*
- Identify and characterize external events that affect the system
- Define behavioral scenarios that capture system dynamic behavior.
- Identify required constraints:
  - required interfaces to other systems
  - performance constraints

The first tool is the use case. As mentioned in [6], a use case is a cohesive end-to-end, externally visible behavior of the system. They can often be identified by talking with the customer. In a medium sized system, you will expect to find anywhere from a few, up to a few dozen, use cases. Most systems consist of three kinds of use cases:

- Primary use cases are the most obvious and capture typical externally visible functionality.

- Secondary use cases are less common but still identify important piece of functionality.
- Sometimes it is possible to also identify safety-related use cases. Most of the time, however, safety and reliability issues are addressed within the previously mentioned types of use cases. These use cases exist when, in addition to normal functionality, the system also acts as a safety monitor or enabler for another system.

Remember, however, that a use case is a function that returns a visible result to an actor without revealing internal structure. This means that the identification of use cases and their relations *does not* indicate or imply an object structure. Use cases map to mechanisms and collaborations, not to objects. This is a stumbling block for many beginners who expect the set of use cases to automatically determine an object structure. There is no automatic way to bridge the gap from a functional black-box view of the system to an object-oriented white box view.

Once use cases are identified, scenarios can be examined in more detail. Remember, scenarios are instances of use cases. They walk a particular path through a use case. Requirements scenarios are a primary means for requirement extraction[14]. You will start with an obvious set of scenarios but quickly identify branching points where different decisions are made. Whenever a branching point appears in a scenario, it indicates the existence of another scenario (in which another decision was made). If the latter scenario is "interestingly different" then it should be added to the scenario set of the use case.

The identification of events relevant to the system and their properties is also done during requirements analysis. This analysis includes the messages and events that actors send to the system and the system response to those messages. The performance properties of each message at this level include the following:

- An Associated actor:
  - > Sender of message
  - > Receiver of response
- An Arrival pattern (periodic or episodic)
- An Arrival time for the message:
  - > Period and jitter for periodic message
  - > Minimum interarrival interval or burst length for episodic message
- The message response properties:
  - > Deadline for hard deadline message
  - > Average response time for soft deadline message
- Message state information:

[14] I like to visualize a scenario as the needle of a rather large syringe which I plunge into the brain of a marketer to suck out hidden requirements. I find such visualizations helpful during long meetings.

- > Preconditional invariants for the message
- > Protocol (acceptable message sequence)
- > Message data
- > Postconditional invariants for the response

This information is typically captured in a use case(s), object context and sequence diagrams, statecharts, and perhaps an External Event List.

Metamodel Entities Used

Two different kinds of modeling elements are used in requirements analysis: contextual and behavioral. It is important to remember that during this sub-phase, the system view is black-box, and only elements which are visible from an external perspective are specified. The contextual elements define the following:

- actors (objects that exist outside the scope of your system)
- the "system object"
- use cases
- use case relations
- external messages (including events)
- hazards

Use case relations may exist between use cases or a use case and an actor. Use cases capture typical and exceptional uses of the system which are visible to actors. Actors are objects outside use case context, and may be human users of the system, devices with which the system must interact, or legacy software systems. Actors associate with use cases. The association between an actor and a use case means that the actor can send or receive messages which enable participation in the scenarios of that use case.

Hazard identification is crucial for safety-critical or mission-critical systems, or for any system for which the cost of system failure is high. At a minimum, you must define the hazards and their associated risks. Many times, this description will also include safety measures if externally visible, although it is more common to define them later in design.

Behavioral elements include the following:

- constraints, such as
  - > performance requirements
  - > fault tolerance times
- statecharts, including
  - > states
  - > transitions
- scenarios
- message protocols in actor-system interactions

Behavioral elements define how the system behaves, but only from a black box perspective. It is relatively common to use statecharts to define the behavior of message protocols of external interfaces and the required behavior of use cases. Statecharts are *fully-constructive*; this means that a single statechart can define the complete behavior of its associated contextual element. Scenarios are only partially-constructive. It is impossible to fully define the complete behavior of a complex machine with a single scenario. Scenarios may be used to describe important paths through a state machine, or they may be applied when the behavior is not state-driven. It is common to define a few up to a few dozen scenarios per use case.

Artifacts

There are many different artifacts produced during this sub-phase. The ones listed here are the most common.

Table 2: Requirements Analysis Artifacts

Artifact	Representation	Basic?	Description
Requirements document	Text	Yes	A textual description of the system contents, interface to external actors, and externally visible behavior, including constraints and safety requirements.
Use cases	Use case diagrams	Yes	Identification of the major functional areas of the system and the interaction of actors with use cases.
	Statecharts	No	Some sets of use cases will be reactive, i.e. have state behavior. The full behavioral space of such use cases may be captured in statecharts.
	External Event List	No	A spreadsheet describing the properties of events received by the system or issued from the system. This includes properties such as period and jitter (for periodic events), minimum interarrival time and burst length (for episodic events), deadlines, event data, and so on.
	Context Diagram	No	This is an idiomatic use of a UML object diagram. It contains the system "object" and actors that interact with the system. The messages and events passing between actors and the system are identified.

Artifact	Representation	Basic?	Description
Use case Scenarios	Sequence diagrams	Yes	As paths through individual use cases, these represent scenarios of uses of the system. They show specific paths through a use case including messages sent between the use case and its associated actors.
	Timing diagrams	No	As another representation of scenarios, timing diagrams are also instances of use cases. Normally applied only to reactive use cases, they show state along the vertical axis and linear time along the horizontal axis.
Hazard Analysis		No	This is normally a spreadsheet format document which identifies the key hazards that the system must address and their properties, such as fault tolerance time, severity and probability of the hazard, and its computed risk.
Test Vectors	Textual	Yes	Specification of tests to validate system against requirements

For some systems it is appropriate to produce all the above artifacts, while for others only part are required. The set of normally required artifacts are identified as "basic" in the middle column of Table 2. The structure and contents of these artifacts are discussed in more detail in [5] and [6].

Systems Analysis

Systems analysis is an important phase in large complex embedded systems, such as aerospace and automotive applications. Systems analysis normally elaborates key algorithms and partitions the requirements into electronic, mechanical, and software components. Often, behavioral modeling tools such as Statemate[15] are used to construct executable models and explore system dynamics. This is especially true when the system displays non-trivial state or continuous behavior.

It should be noted that, like requirements analysis, systems analysis is still fundamentally a functional, and not an object, view. It does not imply, let alone identify, objects and classes. Systems analysis, therefore, will *not* normally result in a set of objects and classes. It is possible to perform system (and for that matter) hardware analysis using object model methods. However, systems analysts, as a group, seem reluctant to adopt this technology. In the meantime, it is best that we software types accept that and plan to bridge the gap between their functional view and our object view.

[15] Statemate™ is systems analysis tool for building and testing executable models of complex reactive systems. It is available from I-Logix. See www.ilogix.com.

Activities

The primary activities of system analysis are to

- identify large-scale organizational units for complex systems
- build and analyze complex behavioral specifications for the organizational units
- partition system-level functionality into the three engineering disciplines of
  - > software
  - > electronics
  - > mechanics
- Test the behavior with executable models

The large-scale organization is often time-specified in the systems analysis sub-phase. The result is a functionally-decomposed architecture containing black box nodes which contain behavioral elements called *components*. These components are then analyzed in detail and hardware/software trade-offs are typically made. The interfaces among the components must be defined, at least at a high level. This allows the engineers of different disciplines to go off and work on their respective pieces.

The analysis of behavioral components is done with finite state machines, continuous control systems, or a combination of the two. For reactive systems, it means constructing complex statecharts of the component behavior. This results in finite state machines with potentially hundreds or thousands of states. For continuous control systems, linear or nonlinear PID control loops are the most common way to capture the desired behavior.

In complex systems, it is crucial to execute the models to ensure that they are correct, unambiguous, and complete. Errors in specification during this phase are the most costly to correct later because the errors are often only discovered after the system is completely implemented. Correcting errors in systems analysis requires substantial rewrites of major portions of the system under development. Early proofs of correctness (such as formal proofs or functional and performance testing) can greatly reduce this risk at minimal cost. It is important that any tests designed during systems analysis be constructed in such a way that they may also be applied later to the developed system. This not only saves work in producing tests, it helps ensure that the delivered system has the correct behavior.

Metamodel Entities Used

The metamodel elements used in systems analysis are both structural and behavioral. Structural components are used to decompose system functionality into functional blocks (nodes on deployment diagrams). These are typically what is meant by the systems analysis term "subsystem[16]." The UML *component* can be used to represent unelaborated boxes.

[16] A subsystem, to a systems analyst, is a box that contains hardware and software, and meets some cohesive functional purpose. This is somewhat different than, although related to, the UML use of the term. A UML subsystem is a subclass of two metamodel entities,

When the hardware pieces are identified, they become UML «processor» nodes on which software-only components reside.

The UML does not define a way to specify continuous mathematics. Continuous algorithms are best specified using equations or pseudocode. UML activity diagrams can be used as a type of concurrent flowchart to represent algorithms as well. Performance constraints are normally elaborated at this sub-phase. The performance constraints are applied at the component level, meaning that the end-to-end performance of behaviors defined for the components must be specified.

Artifacts

Artifacts resulting from system analysis are outlined in the table below.

Table 3: Systems Analysis Artifacts

Artifact	Representation	Description
Architectural Model	Deployment diagrams	Identifies hardware boxes that contain executable components.
	Component diagrams	Identifies functional boxes, potentially residing on nodes, which may be composed of a combination of hardware and software. Some components may be specialized to interact with «device» nodes (hardware that does not execute software of interest) or may be software-only components.
Executable Specification	Statecharts	Finite state machines defined at the component level.
	Mathematical Models	Mathematical descriptions of continuous functionality, such as PID control loops.
	Activity diagrams	UML diagrams that roughly correspond to concurrent flowcharts. These have some of the properties of Petri nets as well and can model complex algorithms.
	Sequence diagrams	Paths of interest (such as typical and exceptional) through the statecharts.
Software Specification	Text	Detailed requirements of allocated responsibilities and behaviors required of the software
Hardware Specification	Text	Detailed requirements of allocated responsibilities and behaviors required of the hardware platform
Test Vectors	Sequence diagrams	Scenarios used to test that the implemented system meets systems analysis requirements.

Package and Classifier. Thus subsystems can group model elements (since it is a type of package) and may also have attributes, associations, and may be specialized itself.

Artifact	Representation	Description
	System test plan	A plan for how the system will be tested to ensure that it correctly implements system analysis requirements.

Object Analysis

Previous sub-phases have defined the required behavior of the system. These requirements must be met by the object structure identified in this phase. Object analysis is the sub-phase in which the essential objects and classes are identified and their important properties captured. It is important to note that this is the first point at which objects and classes appear. It is also important that only the objects and classes that are *essential to all possibly correct solutions* are captured[17]. That is why the analysis object model is often referred to as the "logical" or "essential" model.

Object analysis consists of two different subphases: structural and behavioral object analysis. In practice, these subphases are most often done concurrently. For example, exploration of the important scenarios (behavior) leads to the identification of additional objects within a collaboration (structure) that realizes a use case.

Activities

The primary activities of object analysis are to

- apply object identification strategies to uncover essential objects of the system
- abstract the objects to identify classes
- uncover how the classes and objects relate to each other
- construct mechanisms of object collaboration that meet the use case behavioral requirements
- define the essential behavior of reactive objects
- identify the essential operations and attributes of the objects
- test the identified mechanisms with scenarios
- decompose end-to-end performance constraints into performance constraints on class operations

Metamodel Entities Used

The modeling elements used are the standard ones found in class and object diagrams: classes, objects, relations, and so on. In addition, mechanisms are instances of

[17] It is difficult to imagine, for example, an accounting system object model which did not include the class *account*, or a fire-control system which did not include the classes *target* or *weapon*, at least in some form.

collaborations. UML collaborations are namespaces that contain interactions of classifiers (such as classes).

It is common to divide up the classes into different areas of subject matter called *domains*. A domain is an area of concern in which classes and objects are related. Typical domains include Device I/O, User Interface, Alarm Management, Persistence (long-term storage). A Device I/O domain might contain classes such as A/D Converter, Hardware Register, Port, and so on. A User Interface domain might contain classes such as Window, scrollbar, button, font, bitmap, and icon. Each system will also have one or more specific application domains, such as spacecraft management, avionics, electrocardiography, or reactor management.

Generalization taxonomies are usually held within a single domain. Naturally, collaborations must span across domain boundaries so classes may associate across domains. Domains are commonly represented as packages on class diagrams. Part of the usefulness of domains comes from the fact that because they are well encapsulated and self-contained, analysis may proceed on domains independently.

In order to divide up the analysis work among team members, development components are usually identified. Development components represent non-deployable organizations of model elements (artifacts), such as documents and files. These differ from deployable components such as data tables, pluggable components, libraries (e.g. dlls) and executable programs. The identification of deployment components allows team members to divide up the work. This work is, of course, normally done on top of a configuration management layer to facilitate the efficient collaboration of team members.

Finite state machines continue to be used to capture reactive behavior, but during this sub-phase they are applied to classes only; if you identify a statechart during object analysis, then there must be a class to which the statechart belongs[18]. This means that statecharts identified in use cases and components from earlier sub-phases of analysis must be partitioned among the classes identified during object analysis. This usually results in some re-orthogonalization of the state behavior and care must be taken to show that in all cases the resulting set of class state machines is isomorphic with the use case and component state machines.

Message sequence, collaboration, and timing diagrams are used to specify the dynamic collaboration of objects within mechanisms. Early in analysis, most people prefer to use sequence diagrams, but later, as the object structure stabilizes, some prefer to move to collaboration diagrams. When timing is important, timing diagrams can clearly represent operation or state behavior over time.

[18] Less common, a statechart may be added to a collaboration. Usually, however, such a statechart has already been defined for the use case realized by that component.

Artifacts

Table 4: Object Analysis Artifacts

Artifact	Representation	Description
Object Structural Model	Class and object diagrams	Identifies key abstractions and their logical organization in packages and mechanisms.
	Domain diagrams	An idiomatic usage of a class diagram consisting primarily of packages organized around subject matters of interest and their dependency relations.
	Component diagrams	Identifies development components, such as files and documents, to enable team members to collaborate by sharing work units.
Object Behavioral Model	Statecharts	Finite state machines defined at the class level.
	Activity diagrams	UML diagrams that roughly correspond to concurrent flowcharts. These have some of the properties of Petri nets as well and can model complex algorithms.
	Sequence diagrams	Paths of interest (such as typical and exceptional) through the collaborations of identified classes.
	Collaboration diagrams	Same as sequence diagrams, but organized visually similar to object diagrams.
	Timing diagrams	Specifies the timing of operations and state transitions in collaborations of identified classes.

Design

While analysis identifies the logical or essential model of a system, design defines a single particular solution that is in some sense "optimal". For example, design will identify things like

- Which objects are active (concurrency model)
- Application task scheduling policies
- Organization of classes and objects within deployable components
- Interprocessor communication media and protocols
- Distribution of software components to nodes (esp. if systems analysis step was skipped)
- Relation implementation strategies (How should associations be implemented – pointers? references? TCP/IP sockets?)
- Implementation patterns for state machines
  - > Management of multi-valued roles (i.e. 1-* associations)
  - > Error-handling policies

> Memory-management policies

Three are currently three subphases of design: architectural, mechanistic, and detailed. Architectural design defines the strategic design decisions that affect most or all of the software components, such as the concurrency model and the distribution of components across processor nodes. Mechanistic design elaborates individual collaborations by adding "glue" objects to bind the mechanism together and optimize its functionality. Such objects include containers, iterators, and smart pointers. Detailed design defines the internal structure and behavior of individual classes. This includes internal data structuring and algorithm details.

The work artifacts of the ROPES design process are shown in Figure 9 below. The primary deliverable artifact is the design model.

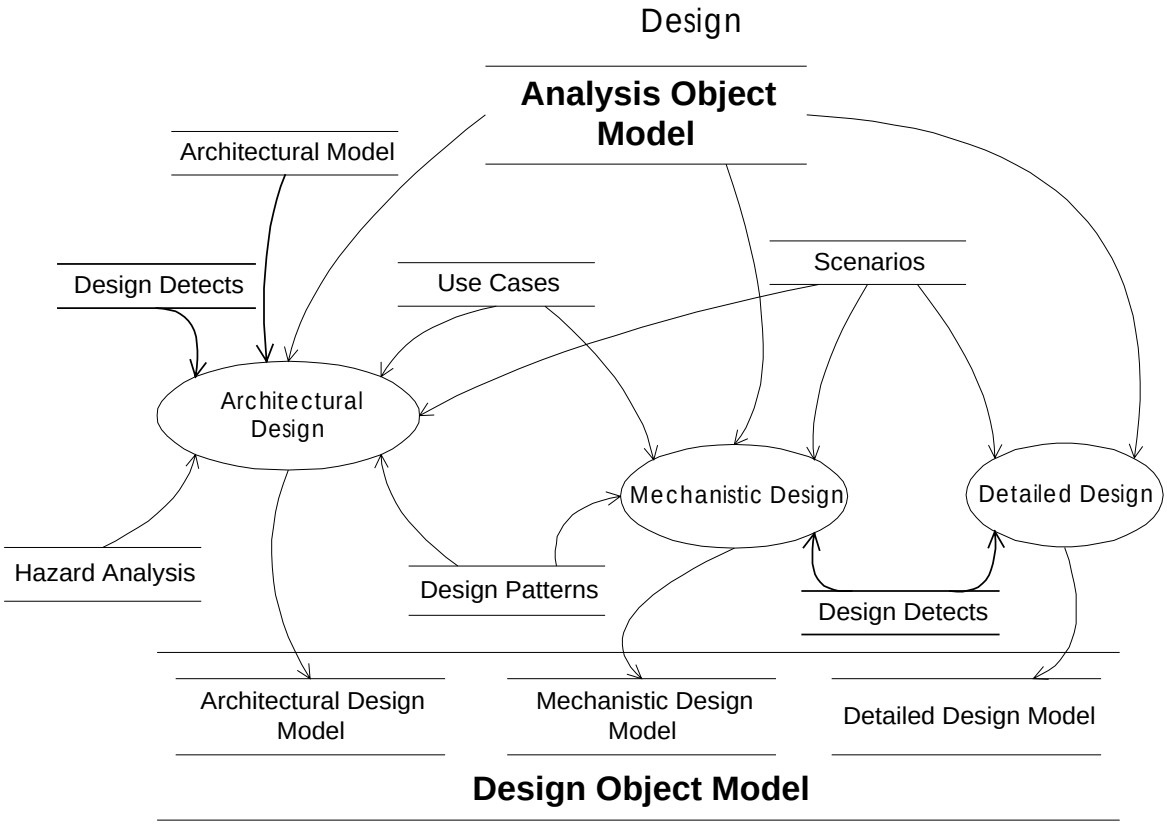

Figure 9: ROPES Design Model Artifacts

Much of design consists of the application of design patterns to the logical object model. These patterns may be large, medium, or small scale, mapping to architectural, mechanistic, or detailed design. Of course, to be correct, both the design model and the analysis model are different views of the same underlying system model, albeit different levels of abstraction. It is obviously important that the design must not be inconsistent with the analysis. Moving from the more abstract analysis model to the design model may be done in ROPES using either the elaborative or translative approach.

The translative approach has much to recommend it. In a translative macro cycle, a translator is built which embodies the design decisions and patterns to be used. This translator is then applied against the object model to produce an executable system. The advantage of a translator is that once built, turning around a prototype from an analysis model is a matter of seconds or minutes, rather than days or weeks. The translator normally has two parts: a real-time framework (see [5] for a description) and a code generator. The real-time framework works much like a GUI framework (such as MFC or Motif). Base classes provide attributes and behavior common to such elements in the domain. A real-time framework will provide classes such as timers, threads, semaphores, state machines, states, events, and a set of operating system abstractions. The code generator generates code from your modeled classes, relations, and statecharts, that in turn use these framework classes (by subclassing or associating with them). The generated code is then compiled and executed.

Tools such as Rhapsody™ provide translators and frameworks out-of-the-box that embody a default set of design decisions. These decisions can be used without change, particularly for early prototypes, but even in final deployable systems as well. However, the user may go in and modify the translator or the framework if desired. This may be done by

- subclassing the classes provided by framework, specializing the behavior as necessary
- Replacing parts of the framework
- setting code generation configuration properties of the translator

In addition, because translating tools provide generation of code and a real-time framework, they can insert animation instrumentation that allows them to monitor the execution of the generated program and graphically depict that behavior using analysis and design level abstractions rather than source code. In the case of Rhapsody, this instrumentation communicates back to the tool as the generated application system runs, allowing it to graphically depict the execution of the model using UML constructs. Thus, statecharts animate, showing the acceptance of events and the transition within state machines; sequence diagrams animate, showing the messages sent among objects; the object browser animates, allowing you to view the current value of attributes and associations – all as the application executes. This provides a model-level debugger that can graphically show the execution of the system using the same diagram views used to create it, complete with breakpoints, event insertions, scripting, and so on. This debugging environment can be used while the generated system executes either on the user's host machine or on remote embedded target hardware.

The more traditional method for designing object-oriented software uses a different approach. The analysis model is taken and elaborated with design detail. This process still works by applying design patterns to the analysis model, but it is done by manually adding the classes to the analysis diagrams and manually typing in code which implements the modeled semantic constructs.

One of the common questions I am asked about the elaborative approach is whether the design model should be maintained as a distinct entity from the analysis model. There are two schools of thought about this. The first school holds that the analysis model is a "higher-level", more abstract view and should be maintained separately. The second school feels that it is important that the design and analysis models always coincide, and the best way to achieve that is to add the design elements to the analysis model. There are advantages to both approaches. However, my strong preference is the latter. I have seen far too many problems arise from the deviation of the analysis and design models to suggest any other approach. On the other hand, there are cases where the same analysis model will be used in multiple designs. If this is true, then it might be best to live with the potential for dual-maintenance of the analysis and design models. Care must be taken to ensure their continuing consistency.

Architectural Design

Architectural design is concerned with the large-scale strategic decisions that affect most or all software in the system.

Activities

The primary activities in architectural design are the

- Identification and characterization of threads
- Definition of software components and their distribution
- Application of architectural design patterns for
  - > global error handling
  - > safety processing
  - > fault tolerance

Some (even most) of the architecture may be dictated by the systems analysis. However, there is still usually plenty of work left to define the concurrency and reliability/safety model even if systems analysis is performed (small engineering projects may skip the systems analysis step altogether).

Metamodel Entities Used

Since the design model is an elaboration of the analysis model, it consists of the same set of elements. The basic elements are collaborations, classes, objects, and their relations. Of particular concern in architectural design are the nodes, components, and active classes. Nodes and components capture the physical deployment architecture. Active objects are objects which are the root of threads. They form the basis of the UML concurrency model. Finally, protocol classes are normally added to manage communications in a multiprocessor environment.

Artifacts

Artifacts from the Architectural design phase are shown in Table 5 below.

Table 5: Architectural Design Artifacts

Artifact	Representation	Description
Architectural Design Model	Class and object diagrams	Updated to include architectural design patterns
	Component diagrams	Identify development components, such as files and documents, to enable team members to collaborate by sharing work units.
	Statecharts	Finite state machines defined at the class level.
	Activity diagrams	UML diagrams that roughly correspond to concurrent flowcharts. These have some of the properties of Petri nets as well and can model complex algorithms.
	Sequence diagrams	Paths of interest (such as typical and exceptional) through the collaborations of identified classes.
	Collaboration diagrams	Same as sequence diagrams, but visually organized similar to the object diagrams.
	Timing diagrams	Specifies the timing of operations and state transitions in collaborations of identified classes.

Mechanistic Design

Mechanistic design is the “medium” level of design. The scope of design elements in this sub-phase is normally from a few up to a dozen objects. Similar to architectural design, most of the work in mechanistic design proceeds by the application of design patterns to the analysis models.

Activities

In mechanistic design, the details of collaborations of objects are refined by adding additional objects. For example, if a controller must manage many pending messages, the Container-Iterator pattern can be applied. This results in the insertion of a container (such as a FIFO queue to hold the messages) and iterators which allow the manipulation of that container. The container and iterator classes serve as “glue” to facilitate the execution of the collaboration.

Metamodel Entities Used

The metamodel elements used in mechanistic design are no different than those used in object analysis. Most of the emphasis is at the level of the class and object.

Artifacts

Artifacts from the Mechanistic design phase are shown in Table 6 below.

Table 6: Mechanistic Design Artifacts

Artifact	Representation	Description
Mechanistic Design Model	Class and object diagrams	Updated to include mechanistic design patterns
	Component diagrams	Identify development components, such as files and documents, to enable team members to collaborate by sharing work units.
	Sequence diagrams	Paths of interest (such as typical and exceptional) through the collaborations of identified classes.
	Collaboration diagrams	Same as sequence diagrams, but visually organized similar to the object diagrams.
	Timing diagrams	Specifies the timing of operations and state transitions in collaborations of identified classes.

Detailed Design

Detailed design is the lowest level of design. It is concerned with the definition of the internal structure and behavior of individual classes.

Activities

Most classes in a typical system are simple enough not to require much detailed design. However, even there, the implementation of associations, aggregations, and compositions must be defined. Additionally, the pre- and post-conditional invariants of the operations, the exception-handling model of the class, and the precise types and valid ranges of attributes must be specified. For some small set of classes, complex algorithms must be clarified.

Artifacts

Artifacts from the Detailed design phase are shown in Table 7 below.

Table 7: Detailed Design Artifacts

Artifact	Representation	Description
Detailed Design Model	Object model	Define the structure and valid values for attributes, and the decomposition of behaviors into a set of operations within the class.
	Statecharts	Finite state machines defined at the class level.

Artifact	Representation	Description
	Activity diagram	Definition of algorithmic behavior in terms of the operations invoked, their sequence, and the branching decisions.
	Pseudocode	Definition of algorithmic behavior

Translation

The translation phase turns the UML model into source code, and, via the compiler, into an executable system as depicted in Figure 10 below.

Activities

In a translative development micro cycle, translation happens more or less automatically. In the elaborative approach, the developer must map the UML model elements to programming language elements. If an object-oriented language is used, this process is fairly rote because all the important decisions have already been made during analysis and design. If a non-object-oriented language is used, then the programming effort is more "creative." In this case, it is common to write a translation style guide. This guide defines the translation rules to be used for the programmer to implement the UML model in the target language, whether it is C, Pascal, or assembly language.

The ROPES process also includes unit testing in this phase. Unit testing is a set of white-box tests that ensure that the unit under test is, in fact, internally correct and meets the design. This normally consists of Unit Test Plan and Unit Test Procedures documents, and culminates in a Unit Test Results document. The Unit Test Plan is normally organized around the basic unit of the class. The document is normally written at the package or component level. Each class within the package is represented in a separate chapter. Within a class' chapter, the set of tests for each operation are identified. The Unit Test Procedures document is organized around identical lines, but includes much detail on the actual execution of each test, including the operation of external equipment and test fixtures. In my teams, we don't even review source code until it passes its unit tests. Properly run, unit level testing greatly increases quality and speeds the development process by identifying errors early and with minimal impact.

Artifacts

Artifacts from the Translation phase are shown in Figure 10 and listed in Table 8 below.

Figure 10: ROPES Translation Model Artifacts

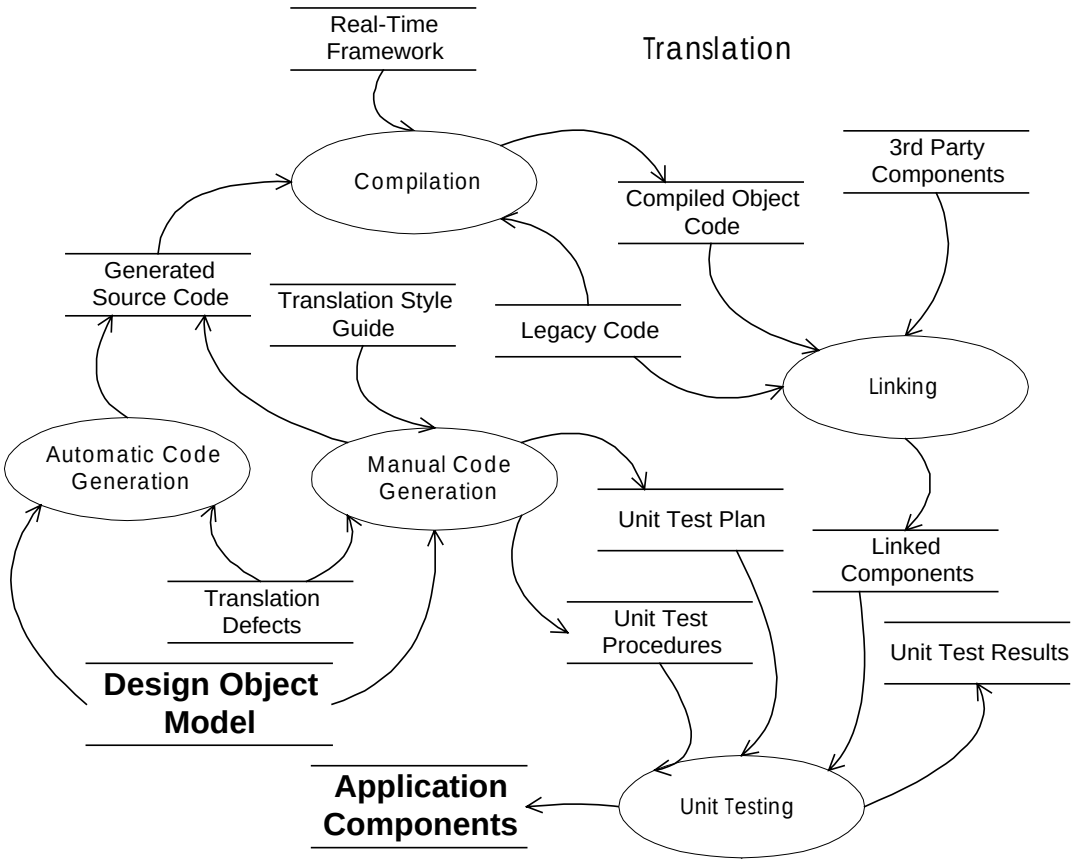

Table 8: Translation Artifacts

Artifact	Representation	Description
Generated Source Code	Textual source code	Programming language statements in the target language
Translation Style Guide	Textual document	A set of rules for converting the UML models into source code statements.
Compiled object code	Compiler-dependent output format	Compiled executable code
Legacy code	Source code Compiled object code	Existing source or compiled object code providing services used by the application
Real-time framework	Source code	Framework classes from which application classes are specialized
3rd party components	Binary object code	Libraries and components to be used by the application such as container and math libraries
Unit test plan	Text, organized by object and operation	The internal white-box testing documentation for the classes describing the breadth and depth of testing
Unit test procedures	Text, organized by object and operation	Detailed, step-by-step instructions for unit test execution, including pass/fail critieria

Artifact	Representation	Description
Unit test results	Textual document	Test execution information, including tester, date, unit under test and its version, and pass or fail for each test
Linked components	Executable components or applications	Components compiled and linked, but not yet tested
Application components	Executable components or applications	Components after they've been tested

Testing

The testing phase in the ROPES process includes both integration and validation testing. Testing applies a set of test vectors to the application that have observable results. The test vectors are based primarily upon the scenarios identified in the requirements and object analysis phase. The resulting artifacts are a tested application and revealed defects, as shown in Figure 10 below.

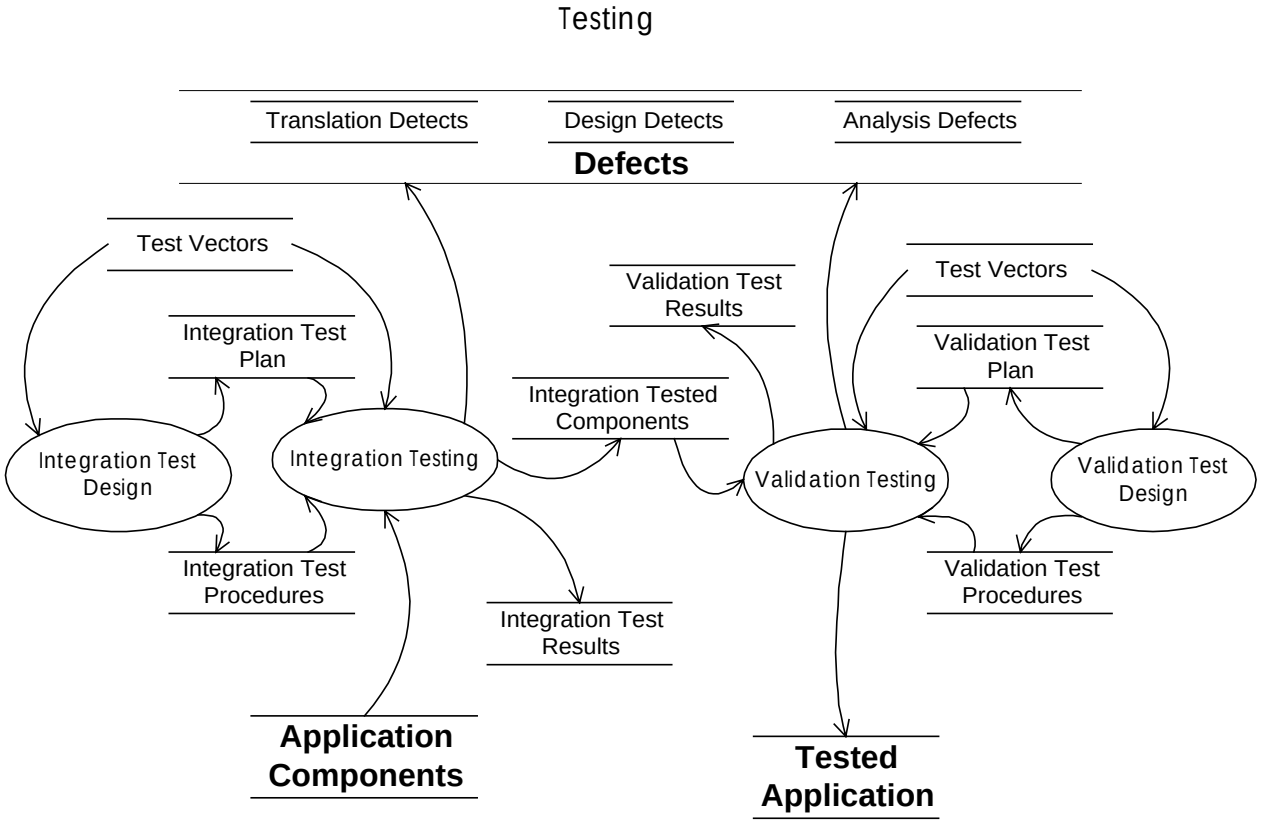

Figure 10: ROPES Testing Model Artifacts

Activities

All tests should be executed against a test plan and in strict accordance with a test procedures document. In integration testing, the system is constructed by adding a component at a time. At each point where a new component is added, the interfaces created by adding that component are tested. Validation tests are defined by a test team and represent the set of requirements and system analysis done early on. Validation tests are fundamentally black box. The only exceptions are the tests of the safety aspects. These tests are white box because it is usually necessary to go inside the system and break something to ensure that the safety action is correctly executed.

Artifacts

Artifacts from the Testing phase are shown below in Table 9.

Table 9: Testing Artifacts

Artifact	Representation	Description
Integration test plan	Textual document	Identifies and defines the order of addition of the system components, and the set of tests to be executed to test introduced interactions among the components.
Integration test Procedures	Textual document	A detailed description of how to execute each test, including clear and unambiguous pass/fail criteria.
Integration test Results	Textual document	The results from the execution of the integration test plan including the tester, name and revision of the components being integrated, date of test, and pass or fail for each test
Integration tested components	Executable component	Tested component
Validation test plan	Textual document	Identifies and defines the set of tests required to show the system is correct. These tests are black box and map to the requirements from the requirements and systems analysis.
Validation test procedures	Textual document	A detailed description of how to execute each test, including clear and unambiguous pass/fail criteria.
Validation test results	Textual document	The results from the execution of the validation test plan including the tester, name and revision of the application being tested, date of test, and pass or fail for each test

Artifact	Representation	Description
Tested Application	Executable application	Validated executable application

Summary

In this paper, we have looked at a number of issues surrounding the process of software development. Early on, we looked at the process for the estimation of software work. As an industry, we are dismal failures at predicting how long software development will take and how much it will cost. There are both sociological and technical reasons for this. If the sociological issues can be addressed, the technical solutions provided in this section can greatly improve estimation accuracy. I've received many reports from companies for whom I have consulted that their ability to accurately predict software projects has been enhanced significantly as a result of the application of these techniques.

The remainder of the paper was spent on the ROPES process model. This model divides software development into 4 primary phases: analysis, design, translation, and testing. Analysis and design are further divided into sub-phases. Analysis is divided into requirements, systems, and object analysis phases. Design is broken up into architectural, mechanistic, and detailed designs. Each of these sub-phases has a set of deliverable artifacts, which constitute the work products for the sub-phase.

The phases identified in the ROPES model are organized into an iterative lifecycle. Each prototype typically implements one or more use cases, organized by risk (greater risk first). This allows the early exploration of risks and minimizes the number of model aspects that must be modified due to those risks. In order to explore those risks, executable prototypes must be developed, since only executable things can be tested. Thus, the key enabling technology which makes this process rapid and efficient is the automatic translation of models into executable code. This reduces the time necessary to do complete iterations from weeks or months to hours or days. ROPES supports both elaborative and translative development micro cycles, but leans towards the latter.

About I-Logix

I-Logix Inc. is a leading provider of application development tools and methodologies that automate the development of real-time embedded systems. The escalating power and declining prices of microprocessors have fueled a dramatic increase in the functionality and complexity of embedded systems—a trend which is driving developers to seek ways of automating the traditionally manual process of designing and developing their software. I-Logix, with its award-winning products, is exploiting this paradigm shift.

I-Logix technology supports the entire design flow, from concept to code, through an iterative approach that links every facet of the design process, including behavior validation and automatic "production quality" code generation. I-Logix solutions enable

users to accelerate new product development, increase design competitiveness, and generate quantifiable time and cost savings. I-Logix is headquartered in Andover, Massachusetts, with sales and support locations throughout North America, Europe, and the Far East. I-Logix can be found on the Internet at http://www.ilogix.com

References

[1] Booch, Grady *Object Solutions: Managing the Object-Oriented Project*; Reading, MA. Addison-Wesley, 1996

[2] Kant, Immanuel, Guyer, Paul (Translator), and Wood, Allen (Translator) *Critique of Pure Reason*; Cambridge, MA; Cambridge University Press, 1998

[3] Boheim, Barry *Software Economics*, Englewood Cliffs, NJ; Prentice Hall, 1981

[4] DeMarco and Lister *Peopleware: Productive Projects and Teams* New York, New York, Dorset House Publishing Company, 1987

[5] Douglass, Bruce Powel *Doing Hard Time: Developing Real-Time Systems using UML, Objects, Frameworks, and Patterns* Reading, MA: Addison-Wesley, 1999

[6] Douglass, Bruce Powel *Real-Time UML: Developing Efficient Objects for Embedded Systems* Reading, MA: Addison-Wesley, 1998