On the ROPES

Object-oriented design is usually part of an overall development process. Here’s a new development process called ROPES (Rapid Object-Oriented Process for Embedded Systems) that may work for you.

I’ve taught UML for real-time and embedded systems for a number of years, in many venues, and in quite a number of places around the planet. I’ve always found it interesting that most of the questions I get during these courses are questions such as:

- How do I use UML effectively?
- What artifacts should I create?
- When should I use statecharts?
- How do I move from a structured approach to an object-oriented one?

In other words, questions about the process in which I use the UML rather than questions on the UML itself.

Why process?
The basic reason why we, as software developers in general and embedded developers in particular, should be concerned about and use a good process is that the purposes of process are to:

- Provide a project template to guide workers through the development and delivery of a product
- Improve product quality in terms of:
  - Number of defects
  - Severity of defects
  - Reusability
  - Stability and maintainability
The basic advantage of the spiral lifecycle is that the system is tested—early and often—so that fundamental flaws can be caught early when there is less rework to do.

- Improve project predictability in terms of:
  - Effort
  - Calendar time
- Communicate project information appropriate to different stakeholders

If you have a process that doesn’t achieve these goals, then you have a bad process and should think about changing it for the better. These goals can be achieved with a good process or they can be inhibited by a bad process.

So what’s a process? In ROPES, we define a process to be a sequenced set of activities performed by a collaborating set of workers resulting in a coherent set of project artifacts, one of which is the desired system.

A process consists of worker roles, the “hats” worn by workers while doing various project activities. Each activity results in one or more artifacts being produced. For example, most processes have requirements capture (activity) somewhere early on before design occurs. This is performed by someone (worker) acting as a requirements specifier (worker role), and might result in a requirements specification, a set of use cases, and elaborating sequence diagrams (artifacts). A complete process will have more steps than this, of course. The result is shown in Figure 1.

For a process to be scalable, I mean that it can be used effectively on small one-person projects as well as on huge, distributed team projects with hundreds of developers. This is quite a challenge to achieve within a single process. ROPES does this by defining a set of core activities, worker roles, and artifacts, as well as many optional activities, worker roles, and artifacts. The most common mixtures of these are defined as a set of ROPES profiles, such as the Small-Team Quick Turnaround Profile, the Large Scale Safety-Critical Profile, and so on.

**Multi-level lifecycles**

Even though the waterfall lifecycle has been pretty resoundingly denounced over the last 20 years, it is still by far the most common way of scheduling and managing projects. The reason is that it is easy to plan and think about. But no project actually follows such a lifecycle, which leads to any number of problems in the development process. The basic difficulty with the waterfall lifecycle is that defects introduced early in the process are not identified or fixed until late in the process. By far, the most expensive defects are specification and architectural defects. The reason that these defects are so expensive is that their scope is far reaching and because many other system aspects end up depending on them. In order to fix such defects, it is necessary to also fix all the aspects of the system that depend on those flaws. This is inherent in the waterfall lifecycle because testing comes at the end.

To reduce or remove the problems associated with the simplistic waterfall lifecycle, the spiral or iterative lifecycle has become popular. The basic advantage of the spiral lifecycle is that the system is tested—early and often—so that fundamental flaws can be caught early when there is less rework to do. This is done by breaking up the development project into a set of smaller projects, and scheduling them so that one such subproject builds upon and uses those that came before, and provides a building block for those that will come after. This is the “spiral.” Each subproject is more limited in scope, is produced with much greater ease, and has a much more targeted focus than the entire system. The result of each subproject or spiral is an iterative prototype—a functional, high-quality system that is not as complete (or perhaps not done in as high fidelity) as the complete system. Nevertheless, the prototype does correctly implement some portion of the requirements and/or reduce some set of risks.

The ROPES process can be conceptualized as occurring simultaneously in three different scales or time frames. The macrocycle process occurs over the course of many months to years and guides the overall development from concept to final delivery. The ROPES macro process has four primary, but overlapping, phases. Each macrophase actually contains multiple microcycles, as we will see shortly, and the result of each microcycle is the production of an iterative prototype.

The microcycle is more limited than the macrophase—usually completing within four to six weeks. It is focused around the production and delivery of a prototype with limited functionality. This is most commonly focused around one or a small number of use cases, but may also include specific risk-reduction activities.

The nanocycle is the most limited scope of all—on the order of 30 minutes to a single day. In the nanocycle, ideas are tried-modeled/executed/fixed at the rate of several iterations per day. Figure 2 shows the standard ROPES lifecycle model.

For a number of reasons, a true spiral model may not be palatable or practical for some organizations, often because of customer milestone requirements (required adherence to DoD 2167A comes to mind) or because the business climate is “uncomfortable” with the notion of a spiral model. For these organizations,
The semi-spiral lifecycle gets its name from the fact that the requirements analysis and systems engineering are not part of a large overall spiral.

The ROPES process provides an alternative lifecycle, called the semi-spiral lifecycle, which is shown in Figure 3.

The semi-spiral lifecycle gets its name from the fact that the requirements analysis and subsystem model do proceed in a spiral fashion, with the multidisciplinary subsystems being integrated together frequently. This alternative approach is often more palatable to organizations accustomed to waterfall approach for very large-scale systems. It provides some, but not all of the benefits of a true spiral development lifecycle. The development and iteration of the design of the system does occur in the semi-spiral lifecycle, but it is not very resilient in the presence of unknown or changing requirements and great care must be taken to ensure that the high level systems engineering model is solid before allowing the subsystem teams to begin working.

The spiral microcycle

Figure 4 shows the standard ROPES microcycle. The primary phases of the spiral are shown in Table 1.

The ROPES spiral is different from most other spiral processes in a couple of ways. First, note the systems engineering subphase in analysis. This subphase is not always required, but should be present for either complex software systems or when there is hardware-software codesign. Systems engineering identifies a high-level subsystem or component architecture and decomposes the system-level use cases to subsystem-level use cases that map to individual subsystems. These subsystems are then further decomposed into the various disciplines of software, electronic, mechanical, and possibly even chemical engineering.

The party phase is where initial project planning takes place, as well as
The requirements subphase identifies requirements, preferring to capture them as use cases, scenarios, statecharts, and constraints.

The ongoing process improvement and project redirection. An initial set of prototypes is identified, although their details aren’t fully defined until some requirements analysis has taken place. As a rough rule of thumb, the prototypes should be scheduled no less than every three weeks and no more than every six weeks apart. The initial planning also must construct software development plans, configuration management plans, reuse plans, and so on so that the team knows the procedures by which they will accomplish their work.

The requirements subphase identifies requirements, preferring to capture them as use cases, scenarios, statecharts, and constraints. In the full spiral lifecycle, it is common during the first iteration to get “the lay of the land” of the requirements—that is, identify all or most of the use cases. But in each spiral, only a few (one to three is very common) use cases are explored in any great depth. In the semi-spiral model, the requirements and system engineering subphases are removed from the spiral, but object analysis does occur during each spiral. There may also be some risk reduction activities and goals that are incorporated as part of the prototypes as well, such as the exploration of performance of a specific compiler or middleware component.

The systems engineering subphase then identifies subsystems and/or components of the overall system (not necessary for simple systems) and breaks down the system use cases into sub-use cases using <includes> and <extends> use case relations. Each of these subsystem-level use cases maps onto a single subsystem. The system use case is then realized by the set of subsystems collaborating together. Each subsystem is specified in terms of its own subsystem level use cases (derived from the system level use cases) and its interface specification. The adequacy of the subsystem decomposition is ascertained by the
application of a simple rule, which applies recursively at every descending level of abstraction:

The structural model at level \( n \) is adequate if and only if all of the scenarios defined at level \( n-1 \) can be realized using the new, more detailed set of abstractions.

This means, simply, that a subsystem structural model is adequate if all of the system level scenarios (defined using the actors and the system) can be realized using the newly identified subsystems. These refined scenarios then show the same set of scenarios as described at the previous level of abstraction, but also how the subsystems interact in addition to the system and the actors.

The next step in systems engineering is to break the subsystems down into single-discipline components (software, mechanical, electronic, and chemical) and understand how these single-discipline components interact. This is commonly called the “hardware-software decomposition.”

In terms of the software, the next subphase, object analysis, identifies the objects that are essential to the problem being solved. For example, if you’re building a guidance and navigation system or subsystem, concepts such as waypoint, vectors, positions, and so on are essential and would be expected to appear in any acceptable solution. This analysis is done one use case at a time. That is, each use case is realized by a set of collaborating objects. The identification of the objects is most often done a use case at a time. Object analysis also reifies these objects into classes, identifies the structural relations among them (associations of various kinds and generalizations), and defines the collaborative and individual behavior.

An alternative approach is to specify the objects a domain at a time. In ROPES, a domain is an area of subject matter with a common vocabulary. Typical domains in real-time and embedded systems are user interface, device I/O, alarm management, and bus communications. Every semantic class in the system falls into a single domain; in fact, all generalization hierarchies fall within a single domain as well. Domains are modeled using UML packages and may be internally decomposed into subpackages to allow them to be more easily managed. Domains organize what ROPES refers to as the logical model—the types and classes of a system, that is, things that exist at design time. This is distinct from the physical model, which, as we shall see later, organizes the objects, subsystems, and components—things that exist at run-time.

If analysis defines the essential properties of a system, design picks a single particular solution. Design is all about optimization against system and project quality of service requirements. Requirements come in two flavors: functional “capability” requirements and quality of service (QoS) requirements. QoS requirements define how well a capability is to be achieved. Designs have to optimize all the important QoS requirements of a system simultaneously in accordance with their respective importance in the particular system or project. In some cases, worst case performance and high safety may be crucial—such as in the development of a controller for a nuclear power plant or an avionics flight control computer system. In other projects, time to market and reusability may be more important. A good design optimizes each important
QoS property of the system in accordance to its relative importance to the system or project.

**Subphases of design**
The ROPES process divides design into three subphases. Architectural design decisions have broad system-wide scope. Architectural design is broken into five important sub-models:

- Subsystem or component model
- Concurrency model
- Distribution model
- Safety and reliability model
- Deployment model

Although these are each called “models,” they are really views or perspectives of the single system model, concentrating on some particular aspect. All of these aspects must work together harmoniously for the system to achieve its overall purpose.

The subsystem or component model has already been touched upon. This defines the large-scale pieces of the system. Each of these pieces will contain (that is, have a composition relationship with) smaller objects and will organize and orchestrate their behavior. At the bottom we have primitive or “semantic” objects that do the real work of the system. The classes that define the semantic objects are all defined in the domains. The classes that define the subsystem model are not considered “semantic” because they exist solely to organize the run-time

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Party</td>
<td>1. Initial project planning, scheduling, resource allocation, concepts, and development strategies, including development plans, configuration management plans, the set of prototypes to be produced, and so on. 2. Ongoing process improvement and realignment activities to ensure the process is working well, risks are being reduced, the architecture is scaling, and so on.</td>
</tr>
<tr>
<td>Analysis</td>
<td>Identification of all aspects of the product that are required for correctness.</td>
</tr>
<tr>
<td>Design</td>
<td>Selection of a specific solution that optimizes all required quality of service of the system, such as worst case performance, average case performance, reusability, safety, reliability, and so on.</td>
</tr>
<tr>
<td>Translation</td>
<td>Generation of machine-executable code, either via automation or manual generation, and the unit-test of those components.</td>
</tr>
<tr>
<td>Test</td>
<td>Ensuring that the prototype is constructed properly of its various components and subsystems (integration) and achieves its specific set of requirements (validation).</td>
</tr>
</tbody>
</table>
objects to meet some functional objective, and many such organizations are possible. The advantage of separating the logical and physical architectures in this way is that one can change how the system is deployed (for example, going from one to several processors) without altering the logical model.

The concurrency model defines the task threads and the semantic objects they will contain, as well as the policies for selecting tasks to run, to resolve mutual exclusion problems, and so on. The ROPES process defines about a dozen strategies for identifying task threads. The typical way the concurrency model is derived is to apply one to three task identification strategies to identify the task threads, and then add an “active” object for each. The semantic objects that execute within the task frame of that active object are then added by means of composition relationships.

The distribution model defines how objects will communicate and collaborate over communication media. This is often done using middleware such as CORBA, COM/DCOM, or a publish-subscribe mechanism. Distribution can be managed as asymmetric (objects are dedicated to a single processor), symmetric (objects can be dynamically loaded to any processor depending on current processor loading), or semi-symmetric (object loading maps are computed at boot time).

The safety and reliability model defines how faults will be identified, localized, and handled and the policies surrounding both reliability and the safety of the system. Many people find the difference between safety and reliability subtle, but the distinction is crucially important for a great many systems. Safety refers to the freedom from accidents or losses, while reliability refers to the probability that the system will continue to function. A handgun is a very reliable piece of equipment, but not very safe. On the other hand, my 1972 Plymouth station wagon is extraordinarily safe—I can’t get it out of the garage! It is safe, but not reliable. In general, when a system has a fail-safe state (a condition known to be always safe), safety and reliability are opposing concerns. When a system has no fail-safe state, then improving one requires improving the other.

Finally, the deployment model maps the other models to the underlying physical hardware. The mapping can be static, as in the case of asymmetric multiprocessing strategies, or dynamic, as is the case with symmetric multiprocessing strategies.

Those five aspects of architectural design all occur within the architectural design subphase, although not necessarily all within the same spiral or prototype. The next subphase of design is called mechanistic design. It focuses on optimizing individual collaborations and is, therefore, much more local in scope than architectural design. This is where the common design patterns are applied, such as the container pattern or façade pattern.

Both architectural and mechanistic design proceed largely, although not exclusively, through the application of design patterns. Buschmann et al. provide a number of useful architectural design patterns while the GoF (“gang of four”) patterns book is the standard text for mechanistic design patterns.

Detailed design focuses on the internals of individual objects. Usually only 5% to 10% of the classes in a system require special care—either because they are algorithmically or structurally rich. Most classes are fairly obvious, but a small percentage normally requires extra work.

**Translation and testing**

The translation phase takes the design model and produces something that the computer can execute. This can be done either by automatic code generation from the design model, by manual hand-coding, or a combination of the two. Legacy and third-party software are incorporated during this phase as well. In the ROPES process, unit testing at the individual class level is performed before objects and components are permitted for use in team builds. In my experience, this is a tremendous timesaver over the more common approach of throwing the system together and then trying to discover why it doesn’t work.

The testing phase has two subphases. The first is called integration and test. This subphase takes the unit-tested components from the translation phase and combines them together to construct the prototype. In doing so, it tests all the interfaces among those components to ensure that the system adheres to the subsystem and component model design. This means testing not only the signature of the interfaces (the easy part) but also the pre- and post-conditions, exceptions that get thrown across the interface, and so on. Integration and test is done according to a preset plan called the integration test plan. This plan can be written as soon as the subsystem and component structure of the prototype is known—that is, the following systems engineering subphase.

The last portion of the testing phase is validation. This tests the constructed prototype against its mission. The mission of a prototype is the set of use cases and requirements to be implemented in the prototype and any additional risk reduction or exploratory requirements. Validation testing is done against a validation test plan, which can be written immediately after the requirements of the prototype are defined, that is, after the requirements analysis subphase.

**Summary**

Although we have only touched the surface of the ROPES process in this article, we have seen that it specifies how to perform a number of activities relevant to real-time and embedded systems. We specifically showed, for example, how to identify functional and quality of service requirements, how to identify subsystems, and even
more important, how to link these two things together.

There’s much more to ROPES. For example, we didn’t touch on how design patterns are applied, the importance of executable models in effectively constructing spiral models, how to turn requirements scenarios into test vectors, or how to estimate cost and time for the spirals and prototypes.

The ROPES process is highly scaleable. It’s suitable for single-person projects all the way through projects with hundreds of developers. Scaleability is largely done through selecting which artifacts need to be generated. In a small project, communication with a small team may require only a few artifacts and less detail, because the team members are all co-located and the system is simple. For large systems, more ritual and detail is necessary because there will be more diversity in the team members, the cost of failure is higher (relative to the cost of producing the artifacts), the teams may not be co-located and the system is very complex. In such systems, more rigor is required to ensure that the much larger number of requirements are properly rendered in the analysis and design, and to communicate the exponentially greater number of details that arise in a large complex system.

The ROPES process has evolved out of my experience in constructing many kinds of systems over the last 20-odd years. Ideas that didn’t work well were discarded, and successful ideas were more strongly supported. Currently, ROPES is a well-understood and successful approach to building systems of varying complexities. It is being used successfully on many projects. It can help yours too.

Bruce Powel Douglass has 20 years of experience designing safety-critical real-time applications in a variety of hard real-time environments. He has designed and taught courses in object orientation, real time, and safety-critical systems development. He is the author of several books including Real-Time UML: Efficient Objects for Embedded Systems (Addison-Wesley, 1998) and Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns (Addison-Wesley, 1999). He is currently employed as the Chief Evangelist at I-Logix, and you can e-mail him at bpd@ilogix.com.

References
1. Actors are objects outside the scope of the system that interact with the system.