Bringing System Engineers and Software Engineers Together

by Jason Leonard
Principal Consultant
Rational Software Australia

For system development projects, the proportion of the budget allocated to software development has been continually increasing in response to demand for system flexibility and maintainability. As a result, many technical organizations started by mechanical and electrical engineers now find themselves awash with software engineers. As system engineers see it, software people have different ways of working: They talk about objects, responsibilities, and services, and generally seem to think software is the only part of the system that matters. Conversely, software engineers think their system engineering counterparts have a split personality. Although system engineers talk about formal techniques for requirements traceability, design for safety, and so on, when it comes to talking about softwareýwell, often they'd rather not *talk about software. The upshot is that software and system engineers don't communicate with each other very often, and even when they do, they are hampered by a lack of a common vocabulary.*

The natural result is that software and systems engineers often don't work very closely together, either. In systems development, this can mean that the system engineers make all the major decisions before passing the result "over the wall" to the software teams. Of course, this leads to the creation of systems that are far from optimal.[1] *I have witnessed system design reviews that were over in minutes, when it became clear that the system design did not take into account a software perspective, and the hardware did not come close to supporting the processing required by the software design.*

Obviously, the converse can also be true: If software engineers are given the upper hand in defining the architecture, then the system may not take *into account those issues that usually fall into the purview of the system engineer -- issues such as capacity, performance, safety, and reliability.*

For larger projects to succeed, a range of people providing expertise in a variety of disciplines must work together effectively. Below we will examine some of the key issues for larger projects, and how system engineering and software engineering techniques can complement each other in addressing these issues.

The System of Systems Approach

Many projects require the creation and/or integration of multiple interacting systems: In such instances the focus is not on engineering a monolithic system, but rather on engineering a system of systems. The subsystems have their own set of requirements, development teams, and test cases. Subsystem requirements are derived ("flow down") from overall system-level requirements, and interfaces between subsystems are specified through the use of Interface Control Documents (ICDs).

Project managers often control this documentation in a "high-ceremony" manner, in an attempt to minimize unauthorized modifications, and incomplete, incorrect, or ambiguous information. Subsystems are then developed largely in isolation (with the exception of reviews at formal project milestones) and thoroughly tested before integration into the larger system.

This approach has its virtues. Adopting a system-of-systems approach gives teams the ability to:

- **Subcontract**. Teams can subcontract the development of parts of the larger system.
- **Leverage COTS**. Teams can take advantage of Commercial-Off-The-Shelf (COTS) systems.
- **Develop and Release Independent Subsystems**. Individual subsystems can be sold and released independently.
- **Manage Complexity**. Subsystem developers have a smaller scope of work that focuses on a smaller set of requirements.
- **Do Low-Risk Parallel Development**. With individual subsystems that are independent from the larger system, parallel development is less risky.

Engineering a system of systems also has its drawbacks, however, and presents special challenges for the system engineer. All large projects consist of a large number of technologies, protocols, and people, but these projects are also likely to be dispersed across geographically distributed sites and different organizations, usually leading to a variety of processes and tools.

Often, subsystem teams work independently from one another, developing subsystems from different requirements documents, and using different tools and processes. There is clearly an opportunity for things to go wrong, such as:

- *Limited or delayed communications* between subsystem teams can lead to interface problems.
- *Some teams may have an incomplete understanding of the context* in which the subsystem should work, which leads to false assumptions and poor decisions.
- *Reuse tends to be minimal across subsystems*; this includes reuse of tools, process, and code.
- *Integration problems may increase rework*, often quite late in the project.

Depth and Correctness

Let us examine a little further the impact of using a system-of-systems approach.
We must accept that a design or requirements document created early in the project will have errors; indeed, sometimes these planning documents seem hopelessly naive later in the project. To counteract this problem, project staff attempt to drill down to low levels of detail, and have detailed inspections and reviews before handing over to the next person in the chain. Although these inspections often pick up trivial defects, they do not detect defects arising from complex algorithms or inherent complexity.[2]

In the context of a large system, this means that:

- Initial architectural partitioning of the system into subsystems may be nonoptimal.
- ICDs are likely to be incomplete and ambiguous.
- Quality cannot be assured from inspections alone.

The challenge is to validate the partitioning into subsystems as early as possible, and to ensure these subsystems will work together effectively.

Architectural Convergence

The previous section may suggest that careful analysis of architecture is not worthwhile; after all, there will be problems, so why not just sketch out a solution and let the teams run from there?

In a perfect world, no self-respecting project manager, architect, or customer would allow a large project to proceed on such a rough basis. However, the commercial pressure to "see something working," combined with lack of certainty, has pushed more than one project along this perilous track. The symptoms of this situation often include:

- Subsystem teams are not focused on eventual integration.
- There is no clear documentation on the responsibilities of each

subsystem.

- There is no real authority vested in the architecture/system engineering team.
- Subsystem development commences in advance of an architecture description: This often occurs in the push to use a COTS system; the COTS system *becomes* the architecture, an end in itself

So, how do we effectively communicate the overarching system architecture to the subsystem teams and ensure they comply with it? The following section suggests some techniques that can help.

Solutions for System of Systems Issues

Any attempt at improving the way a system of systems project operates must proceed with three perspectives in mind:

- *People*: getting together people with the right skills and motivation.
- *Process*: providing guidelines to help people work effectively.
- *Tools*: automating error-prone, time-consuming, or simply tedious tasks.

Below we will survey techniques drawn from both the software and system engineering fields that encompass all three of these perspectives, aid in communicating system architecture to all project participants, and could potentially be used to get software and system engineers working together more effectively.

Integrated Product Teams

Some successful organizations address the issues we've discussed above by forming an IPT (Integrated Product Team), a group of people with skills in complementary areas. The IPT includes both system engineers and software architects, and also should include commercial and customer representatives if possible.

An effective IPT allows for "top-down" design, with all perspectives taken into account. This means that:

- Any software engineer versus system engineer design conflicts can be taken into account and resolved early.
- The design will remain focused on customer or commercial requirements (provided the IPT includes customer or commercial representatives).

Two important responsibilities fall onto the shoulders of the IPT: to generate and communicate the Vision of the system, and to provide the architectural foundation upon which to base further design.

"End-to-End" Requirements (Use Cases) and Testing

To ensure that subsystem teams understand the context that they are working in, they need requirements that document how the entire system should function. One commonly used technique to document this end-to-end functionality is known as the *use case*. A use case details a complete, end-to-end story of how the system is used from the perspective of an "actor": someone or something (another system) external to the system. A complete set of use cases is made for all the actors that will use the system.

Use cases are helpful in a system-engineering context for several reasons:

- **Use cases provide focus**. They help prevent development teams from losing sight of the real reason for the project (similar to the purpose of "Concept of Operations" [CONOPS] documents used in the defense industry, which present systems from a user's and/or buyer's point of view).
- **Use cases provide context**. By documenting scenarios within which the subsystem shall be used, they provide context for the subsystem developer and help prevent misunderstandings.
- **Use cases furnish details**. They help "flesh out" detail because of their methodical approach. I worked on a project that developed system requirements over a six-month period using more traditional techniques. Then we introduced use cases and discovered that these original requirements provided only 10 percent of the detail we needed.

Use cases, being end-to-end stories, can also provide the context for integration and system testing, helping to link the various subsystems together. Again, the system perspective is important; in certain cases, although each subsystem team meets its requirements, the system as a whole might still fail. End-to-end tests can help determine whether the potential for failure exists and then eliminate it.

Of course, use cases by themselves do not tell the whole story. They are not suitable for documenting other, typically nonfunctional, requirements such as capacity, usability, safety, or performance. These must still be documented elsewhere (the Rational Unified Process®[3] has a "supplementary" requirements document for this purpose) as standard "shall"-style requirements.

Iterative Development for Objective Progress Assessment

Testing at the end of the project is valuable to pick up defects in the implementation of the project before (hopefully!) they are passed on to the customer. The end of the project, however, is not an especially good time to find out that a major rework of the architecture is required, or that significant requirements were misunderstood.

Some organizations use a better approach: architecture-centric iterative development. The idea here is not to create "rapid prototypes," but rather to select a subset (not a subsystem) of the system for design, implementation, and test. The subset should be one that allows testers to *see* that architectural drivers such as complex functionality, system performance, throughput, capacity, or safety aspects have been satisfied and that progress has been made.

This focus on demonstrable progress is a key forcing function: The teams must converge *on a solution, and so requirements ambiguity, architectural mistakes, and integration issues are soon "flushed out."* This is a superior approach to inspections alone.

Later in the project, when the architectural drivers have been largely satisfied, the focus can shift to providing the most important functionality required by the customer. Again, iterations are used to demonstrate progress. Note that this is the opposite of the waterfall approach that many projects use: The focus from the beginning is functionality, and progress in the early stages of the project appears to be rapid. It is typically toward the end of the project, when the difficult issues such as safety and performance can no longer be avoided, that progress "mysteriously" slows.

When implementing iterative development, some system engineering projects face the challenge of different turnaround times for different types of development. For example, if a plastic case forms part of the final deliverable, then the injection molding die needs to be designed and manufactured. This process typically takes far longer than turnaround times within the software engineering sphere. The important thing is to ensure that feedback is obtained on how the system works as a whole; if the plastic case development process cannot keep pace with the software process, then choose a milestone (such as the end of an iteration) to bring the different development efforts together. Doing so may reveal, for example, that the software does not fit in the RAM available on the Single Board Computer (SBC) that was manufactured to execute it; or that a lack of ventilation in the plastic case results in high temperatures that cause the RAM on the SBC to fail occasionally, and consequently the software will exhibit unusual behavior.

Iterative development allows time for such problems to be discovered -- problems that would never be found during a paper-based design review. It also provides a natural opportunity to make process improvements and revisit plans.[4]

UML Design Methods and Notation

It is common for system engineers and software engineers to have very different approaches to solving design problems and communicating solutions. For example, object-oriented techniques and processes are used in the software world; software developers talk about services provided by objects (or components), whereas system engineers are more likely to refer to the functional blocks that make up the system design and the protocols by which they communicate. Typically, however, developers depict designs with diagrams that use the Unified Modeling Language (UML),[5] and this does offer a means for communicating the design to systems engineers.

The UML provides a number of different diagrams (e.g., class diagrams, sequence diagrams, and deployment diagrams) and views for modeling a system. This variety is essential for illustrating how the system architecture satisfies the different types of requirements and constraints placed upon it, such as functionality, capacity, performance, and reliability.

A common complaint from system engineers is that UML provides no way to depict structure. This problem has been addressed by proposals for UML 2.0,[6] and Rational Rose® RealTime has already implemented a solution that provides *structure diagrams*. So system of systems projects can now use UML quite effectively to communicate the architecture to both systems engineers and software engineers. Below, we will look at what each type of UML diagram is designed to do.

Structure Diagrams. A structure diagram, as the name suggests, shows a system engineer how a system is constructed as a set of subsystems, as well as the connections between these subsystems. In Figure 1, the "boxes" represent "logical" subsystems. The implementation of these logical subsystems might represent software components, bits of hardware, or a combination of the two. The important thing is that segmenting a large system into "boxes" with interconnecting cables/busses, and so on, requires far less visualization skill than that required to comprehend some of the more sophisticated diagrams used in object-oriented modeling. That said, the structure diagram is usually used to represent software blocks, which are later shown deployed on physical hardware nodes on a *deployment diagram* (see Figure 3). Note that *we use two perspectives: logical (represented in the structure diagram) and physical (represented in the deployment diagram)*. This is in contrast to many diagrams depicting system architecture that attempt to combine the two concepts, which results in a loss of clarity.

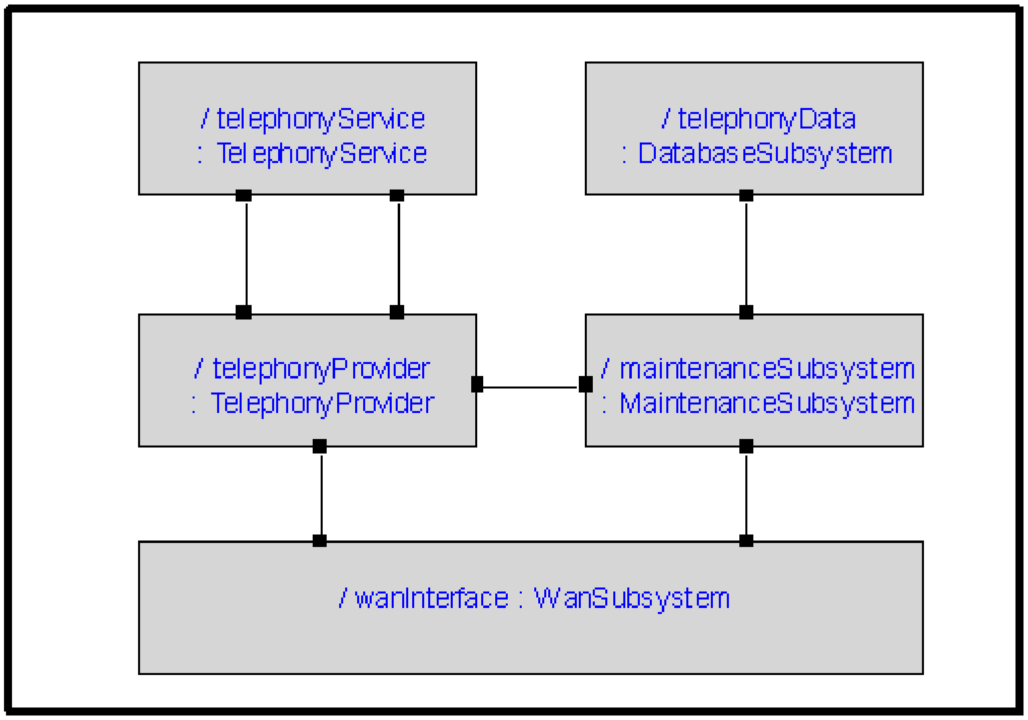

Figure 1: Structure Diagram

The structure diagram is hierarchical; that is, each subsystem may contain subsystems depicted on another, linked, structure diagram. This hierarchical approach allows the *system* engineer to efficiently document the architectural decomposition of the system, using the same notation as the *software* engineer.

In an object-oriented approach, these subsystems are determined based upon abstractions of the problem and solution space (e.g., Telephony Provider), rather than by progressively breaking down system-level requirements. This results in subsystems that are likely to be robust in the face of change to system requirements, and are more likely to be reusable.

One very interesting (albeit, slightly advanced) feature of the structure diagram is that dynamic structure modeling is supported. For example, in the diagram above, the Telephony Provider could be replaced during system execution by another subsystem that provides the same interface. In fact, the structure diagram depicts slots (or "roles") that compatible subsystems can fit into. This allows the engineer to define an architecture that doesn't need to be remodeled for each variation in how the system might be used or configured.

Sequence Diagrams. Extending this object-oriented system engineering approach one step further, we must also explain the sequencing of how messages are exchanged between subsystems. The UML defines a *sequence diagram* (see Figure 2) to document this information. The sequence diagram is quite semantically rich, in that it can document asynchronous communication, communication latency, processing constraints, and data.

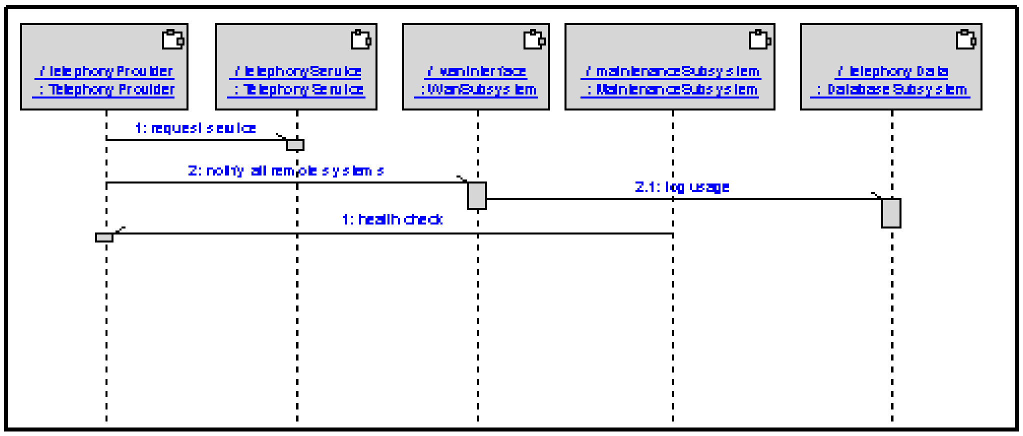

Figure 2: UML Sequence Diagram

The sequence diagram helps us to find responsibilities for the subsystems depicted in the structure diagram, and to define their interfaces. In doing so, we typically find extra subsystems; so we are also validating the original architectural breakdown. Sequence diagrams are typically discovered by examining the requirements, documented as use cases, so we need to establish traceability between the use cases and sequence diagrams. This traceability can be documented with simple naming conventions, or by using hierarchical packaging in a requirements management tool. Using a requirements management tool can help ensure that the team updates diagrams when use cases are modified.

The process is recursive; subsystem development teams might also use sequence diagrams to document the internals of the subsystem, based upon the higher-level sequence diagram. Again, traceability between the diagrams is required to ensure consistency.

Note that the subsystem team will need to utilize not only the description of the data exchanged and the ordering of messages, but also the timing constraints, typically shown as notes on the diagram. The timing constraints also flow down from higher-level diagrams. For example, the top-level sequence diagram may indicate that a message must be processed by subsystem "S" within 400 milliseconds. A sequence diagram that depicts the collaboration of sub-subsystems within S must show how this requirement is met. Note that a tool (e.g., RapidRMA from TriPacific Inc.) could be used for verification if real-time deadlines are a major concern.

Some other important considerations for sequence diagrams are the following:

- *Sequence diagrams depict interactions between elements at the same level of the architectural hierarchy*. This prevents the diagrams from becoming large and unwieldy, and therefore makes them easier to understand. We should link each sequence diagram to exactly one structure diagram, and limit the elements in the sequence diagram to those subsystems shown in the associated structure diagram. This link also allows us to pinpoint "illegal" messages -- that is, messages shown in a sequence diagram for which there is no connector in the structure diagram. For example, in the sequence diagram shown in Figure 2, message 2.1 is illegal because there is no connector between the WAN Subsystem and Database Subsystem in the structure diagram.
- *Subsystems are designed with flexibility in mind so they can be adapted to changing requirements*. A focus on encapsulation of implementation details can assist with this, but before committing to a subsystem interface specification, it is important to validate that the interface indeed caters to a variety of uses. This is a key reason for using sequence diagrams prior to resorting to detailed subsystem design or implementation: Sequence diagrams allow teams to focus on the interfaces without getting caught up in the details and to explore and document many scenarios in a short time. The need to depict multiple scenarios means that one structure diagram will be linked to multiple sequence diagrams.
- *Teams should attempt to identify common elements across subsystems to avoid creating similar subsystems multiple times*. Sometimes referred to as the "stovepipe" problem[7] this issue is discussed later in this article.

Use-Case Flow-Down. The use-case flow-down[8] is another approach that can be used by itself or as an adjunct to detailed subsystem requirements. Simply put, a *use-case flow-down identifies use cases not only for the system as a whole, but also for the subsystems*. It is important to understand how the subsystem use cases are identified: An examination of how the subsystems collaborate to realize system-level use cases allows the engineer to discover subsystem-level use cases, rather than simply allocating requirements. As we noted earlier, a system design that is based solely on a requirements hierarchy is unlikely to adapt smoothly to a modification of those requirements.

- *It results in a level of detail that makes the system structure clear and comprehensible*. Inexperienced authors of use cases often have difficulty deciding on the level of detail to include. The use-case flow-down approach leads to clearer decisions on how much detail is appropriate at each level.
- *Obscure use cases are more easily discoverable*. Use-case flow-downs are an extra mechanism to ensure that all uses of the subsystem are found. For example, this approach may help to uncover use cases for subsystem maintenance, subsystem configuration, or subsystem security.
- *Subsystem uses are consolidated into one convenient document*. The possible uses of each subsystem are detailed in one document. If you choose instead to use a purely diagram-driven approach, then subsystem developers must know about each sequence diagram in which the subsystem is used. Of course, this is not a major issue if these diagrams are kept in a tool (such as Rational Rose® RealTime) with reasonable searching and reporting capabilities.
- *Use-case information provides an important basis for user documentation*. If the subsystems are intended for reuse, or if they are to be sold or deployed individually, then documentation from a user perspective is very important. Use cases can easily serve as a basis for user manuals.

There is no such thing as a free lunch, though, so we should also understand the costs that are associated with use-case flow-down. It requires:

- *An investment of time and effort*. Mainly, you have to write the subsystem use cases! A project that wants to use a lean process may resist this effort without a strong understanding of the associated benefits. For small subsystems, simply defining the interface and generating diagrams depicting how the interface should be used is often sufficient.
- *Maintaining consistency*. Establishing and maintaining the consistency of system and subsystem use cases may be an issue: Each step in a system use case may need to be traced to steps in multiple subsystem use cases.

If a team uses both use-case flow-down and a sequence/structure-diagram-driven approach, then these approaches must also stay consistent with each other.

- *More time for validation*. Use cases, typically documented as text, are subject to the ambiguity inherent in natural language. Although tools such as Rational® QualityArchitect can automate validation of UML diagrams, there are no such tools for the written word (of course, use cases may be supplemented by diagrams).

Deployment Diagrams. Subsystems eventually need to be assigned to the hardware on which they will execute. Note that sequence diagrams can help the system engineer decide which software subsystems should be colocated to minimize network traffic; the deployment diagram shown in Figure 3 illustrates this.

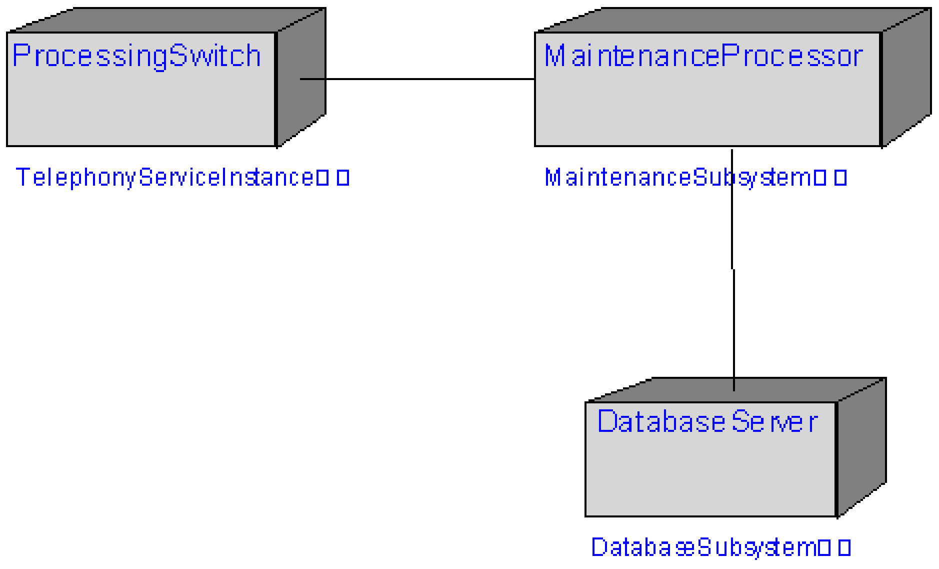

Figure 3: UML Deployment Diagram with Assigned Subsystems

The assignment of subsystems to hardware is very clear in Figure 3. However, sometimes such an approach is a little too simple: Subsystems may map to multiple executable components, and so may also execute across multiple processing units. We may want to postpone making a decision, for example, in the analysis phase; we really don't know how much processing power is required, and we may not know where redundancy is required. Nevertheless, we do want to be able to convey that (for example) one subsystem will execute on a hand-held processing device, and that messages to a collaborating subsystem will be passed over some kind of network connection (again, it would be premature to specify the kind of connection in the analysis stage). The concept of "localities," as defined in the Rational Unified Process for System Engineering can be of assistance here.[9] A locality is an analysis tool, defining the characteristics (e.g. cost, reliability, performance, size, etc.) of the processing hardware, without tying the engineer to a particular implementation.

We can also enhance the diagram with UML *tagged values*. Tagged values provide extra information about a modeling element. In this case, useful tagged values (from RUP) include:

- *Quality*: reliability, availability, performance, capacity, and so on
- *Management*: cost, technical risk

The connections between the deployment nodes may also have tagged values, such as:

- *Throughput*: data rate, supported protocols
- *Management*: cost, technical risk

State Diagrams. Finally, in an event-driven system it is useful to document the states that subsystems pass through, and relate incoming messages to these states. *State diagrams* provide a mechanism that is both readily understandable and yet formal enough to allow for automated construction of software subsystems and simulation of hardware subsystems. Like the structure diagram, the UML state diagram is hierarchical, allowing substates to deal with detail (see Figure 4).

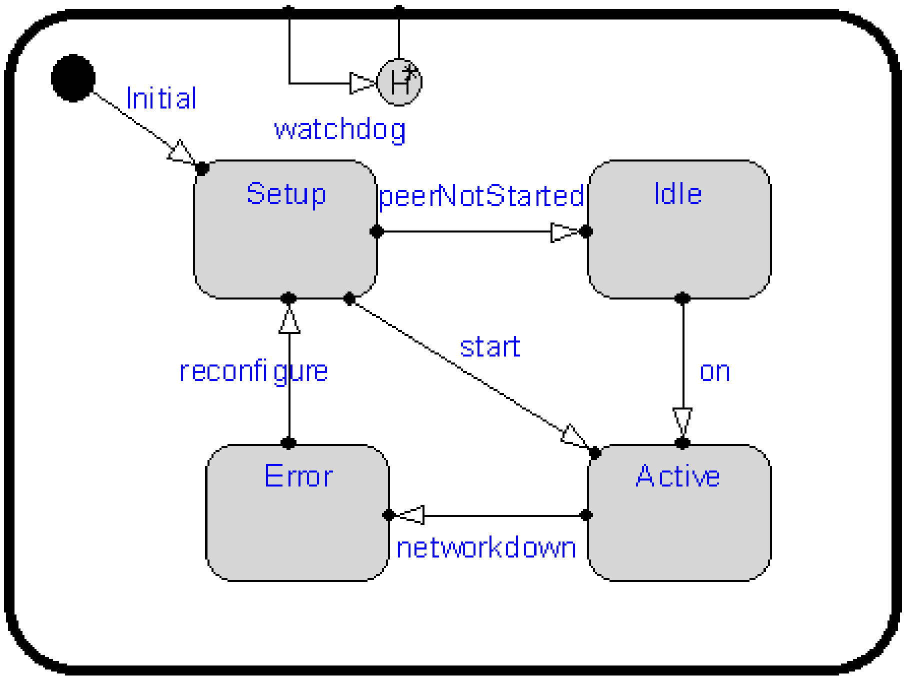

Figure 4: UML State Diagram

Benefits of the UML Approach

Using UML 2.0 diagrams to communicate design intent improves productivity and quality in a number of ways:

- *Diagrams form part of the specification*. The diagrams can provide a large part of the specification for the software teams working on subsystems. They document the interfaces, ordering of messages, and surrounding context (e.g., distributed communication).

- *Interface control documents can contain executable features*. A tool (e.g., Rational Rose® RealTime) can generate an executable software representation of the system to validate or demonstrate the operational concept (perhaps "stubbing out" subsystems that are not yet available). In effect, we are getting to the point of creating an *executable* ICD. Making requirements executable leaves no room for ambiguity.
- *Layered frameworks simplify major issues such as concurrency and distribution*. A tool can take advantage of layered frameworks to simplify major issues such as concurrency and distributed computing. For example:
  - *Concurrency*: If the subsystems are marked as executing on the same hardware unit, the challenge might be to execute the subsystems concurrently while avoiding issues such as race conditions. The tool could generate code that multitasks the subsystems and automatically queues incoming asynchronous messages as required.
  - *Distribution*: Alternatively, if subsystems are marked as operating on separate hardware units, the generated software representation might automatically route messages exchanged between these subsystems over the network.
  - *Schedulability*: A tool (e.g., TriPacific Software Inc.'s RapidRMA for Rational Rose® RealTime) can perform rate monotonic analysis to check the schedulability of the model, given estimates of processing time. This means that the system engineer can ensure that real-time constraints will be met.

However, the most important benefit of using UML for system engineering purposes in software-intensive systems is that it improves communication between people with different skill sets.

Dealing with the Stovepipe Problem

Unfortunately, the real world intrudes whenever we logically decompose a system into collaborating subsystems. That is, we often cannot drive the partitioning of the system purely from architectural concerns; we must also take into account external concerns such as:

- Security of intellectual property.
- Fixed contractual boundaries between organizations.
- Need to use existing or COTS systems.
- Staff skill sets.

When different teams or organizations are responsible for different subsystems, interface misunderstandings can arise and even result in contractual arguments. One way to avoid such problems[10] is to separate interface components from the implementation of the subsystems, and have only one team design and implement these components. A related concept is the Interface Control Working Group (ICWG),[11] a group of people representing the development teams impacted by the interface.

Even if you try these useful techniques, however, keep in mind two basic premises:

- If iterative development incorporates system-wide testing and demonstration, then interfaces will be validated early.
- If you use a common *visual* notation such as UML to define interfaces and the exchange of information across them, then you will increase joint understanding and reduce ambiguity.

Making the Change

For most organizations, adopting the techniques we have suggested above will not represent a big change at all. Many of these techniques have been advocated and practiced within the system engineering community for a number of years. Indeed, some of them, along with supporting tools, are now available in a COTS format. For instance, Rational provides a framework and a set of tools to better enable communications amongst team members -- hence the word *unified* frequently appears in Rational product names and marketing literature.

Let us summarize the key recommendations:

- Create use cases to document detailed system-level requirements.
- Use visual models in preference to text-based designs.
- Adopt UML as the common notation for these visual models.
- Use iterations as the basis for feedback on progress, and to force convergence.
- Use architecture-centric practices to combat the stovepipe problem.
- Use Integrated Product Teams for larger multidisciplinary projects.
- Create Interface Control Working Groups to minimize the impact of changes to important interfaces.

It is important to remember that any process change requires a change in mindset as well as time to be absorbed. Therefore, we recommend that you introduce such changes incrementally. It is unwise to (a) introduce many changes at once, or (b) introduce the changes across an entire organization at once. Either approach can result in failure. Instead, it is best to lay out a plan that introduces change in phases; this plan should incorporate significant amounts of training and mentoring, and should explain how staff will be motivated to make the change.

Acknowledgments

I would like to acknowledge Murray Cantor's contributions to the shaping of this article, and also David Brown and Catherine Southwood for their reviews. Thanks to Mike Taylor and Ross Beattie for their insight and help.

References

Philippe Kruchten, "Common Misconceptions about Software Architecture." *The Rational Edge*, April 2001: http:www.therationaledge.com/content/apr_01/m_misconceptions_pk.html

Walker Royce, *Software Project Management: A Unified Framework*. Addison-Wesley, 1998.

Rational Software, *The Rational Unified Process*, v2002. See http://www.rational.com/products/rup/index.jsp

Philippe Kruchten, "From Waterfall to Iterative Development." *The Rational Edge*, December 2000: http://www.therationaledge.com/content/dec_00/m_iterative.html

Grady Booch et al., *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

James Rumbaugh et al, *The Unified Modeling Language Reference Guide*. Addison-Wesley, 1999.

U2 Partners. *UML 2.0 Superstructure Proposal*, November 2001. Available at: http://www.u2 partners.org/artifacts/presentations/U2P_Superstructure_Proposal_011115R5.zip

Dean Leffingwell and Don Widrig, *Managing Software Requirements: A Unified Approach*. Addison-Wesley, 2000.

Rational Software, *The Rational Unified Process for System Engineering*, v1.0. See http://www.rational.com/services/rup-se-deployment.jsp

Defense Systems Management College, *System Engineering Fundamentals*. Defense Systems Management College Press, 1999.

Notes

1 Philippe Kruchten, "Common Misconceptions about Software Architecture." *The Rational Edge*, April 2001: http://www.therationaledge.com/content/apr_01/m_misconceptions_pk.html.

2 Walker Royce, *Software Project Management: A Unified Framework*. Addison-Wesley, 1998.

3 Rational Software, *The Rational Unified Process*, v2002. See http://www.rational.com/products/rup/index.jsp

4 For more information on the strengths and pitfalls of iterative development, see Philippe Kruchten, *From Waterfall to Iterative Development* in the December 2000 issue of *The Rational Edge*: http://www.therationaledge.com/content/dec_00/m_iterative.html

5 See Grady Booch et al., *The Unified Modeling Language User Guide* (Addison-Wesley, 1999); and James Rumbaugh et al., *The Unified Modeling Language Reference Guide* (Addison-Wesley, 1999).

[6] U2 Partners, *UML 2.0 Superstructure Proposal* http://www.u2-partners.org/artifacts/presentations/U2P_Superstructure_Proposal_011115R5.zip November 2001.

[7] See Dean Leffingwell and Don Widrig, *Managing Software Requirements: A Unified Approach*. Addison-Wesley, 2000.

[8] Rational Software. *The Rational Unified Process for System Engineering*. v1.0. See http://www.rational.com/services/rup-se-deployment.jsp

[9] Rational Software, *The Rational Unified Process for System Engineering*. v1.0. See http://www.rational.com/services/rup-se-deployment.jsp

[10] Leffingwell and Widrig, *Op.Cit*.

[11] Defense Systems Management College, *System Engineering Fundamentals*. Defense Systems Management College Press, 1999.