The Handoff Process

Bruce Powel Douglass
Chief Evangelist
IBM Rational

The Transition to Software Engineering

The Handoff process is presented as part of an end-to-end, integrated product development process taking product development from requirements capture through systems engineering, an iterative development design lifecycle and into final test and validation. An overview of the overall process is shown in . Its heritage as an elaborated "V-Process" is clear. The systems engineering work is done prior to the iterative analysis-design-implement-test cycle. After the handoff to software engineering, the iterative development cycle (IDC) carries the project forward. The IDC is an incremental approach, in which the system is developed as a series of builds of increasing capability and completeness. The system (referred to as the "prototype" even though it is fully production quality code) is constructed incrementally, with new builds adding new system-level capabilities to the increasingly complete system. The advantage of such an incremental approach are well documented in the literature, and include higher quality, lower cost of quality, and even reduced development time. In this paper, we will focus on how the handoff from systems engineering to software engineering occurs. In the progression of that discussion, we will touch on both the system engineering development side, model organization, and on the work done within the IDC. Nevertheless, our main focus in this paper will be to discuss how we do the transition from systems engineering to software engineering.

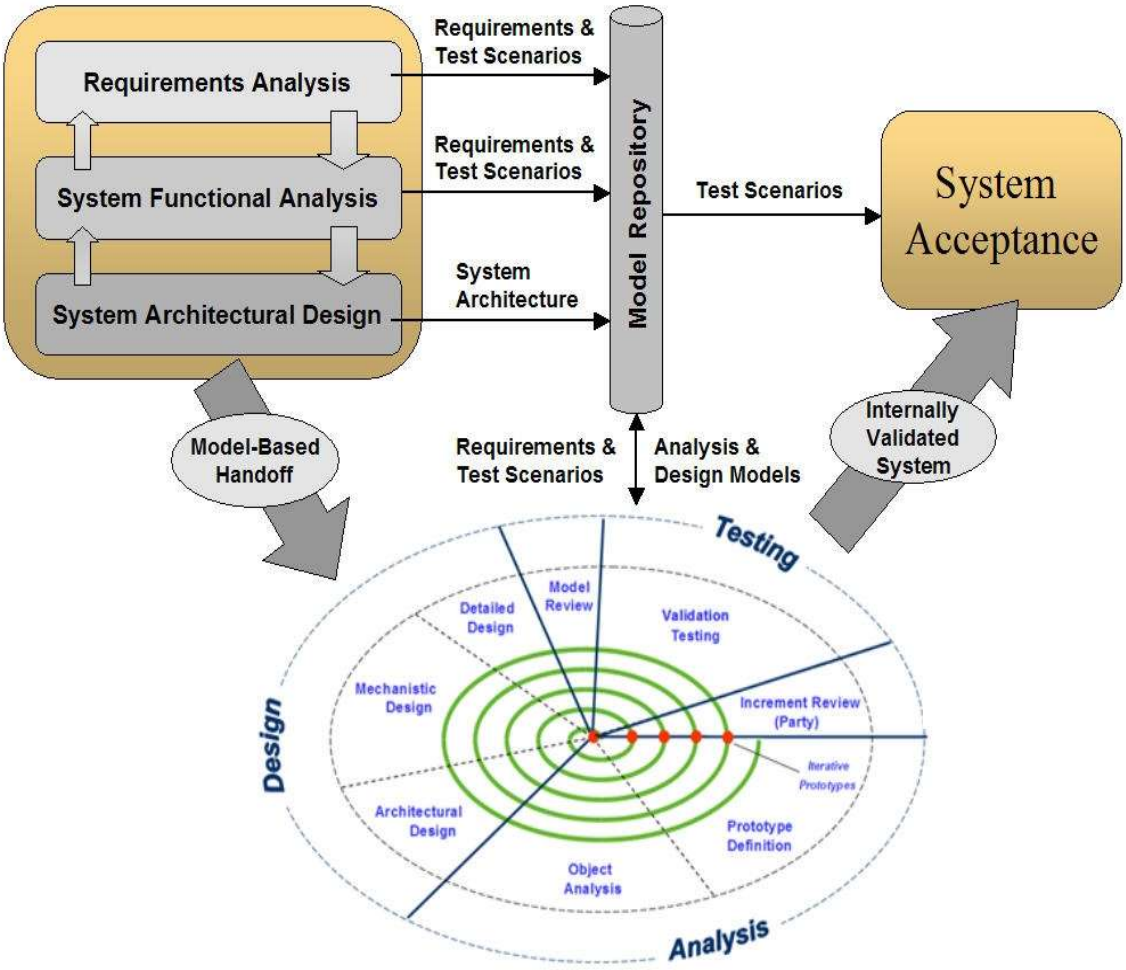

Figure 1: End-to-end Process Overview

Preparation: System Engineering Model Organization

The Systems Engineering Model

As discussed above, the overall process proceeds through the development and elaboration of UML/SysML/DoDAF models. It is important to organize the system engineering model to facilitate the construction of the right set of artifacts and to hand them off to the development engineers that will use and elaborate them. In small systems, model organization may not be a problem. However, for projects that have separate subsystem development teams, the model will be complex enough so that it will be very difficult to do an efficient hand off if this concern hasn't been addressed.

The model organization we recommend is to have separate areas (modeled as packages) in the systems engineering model for

- Requirements
- Architecture

- Subsystems

In systems of any significant complexity, these areas will be further subdivided to support both hierarchical layering of the model and separation of independent concerns. shows a package diagram showing the structure of an example model – in this case, an unmanned air vehicle.

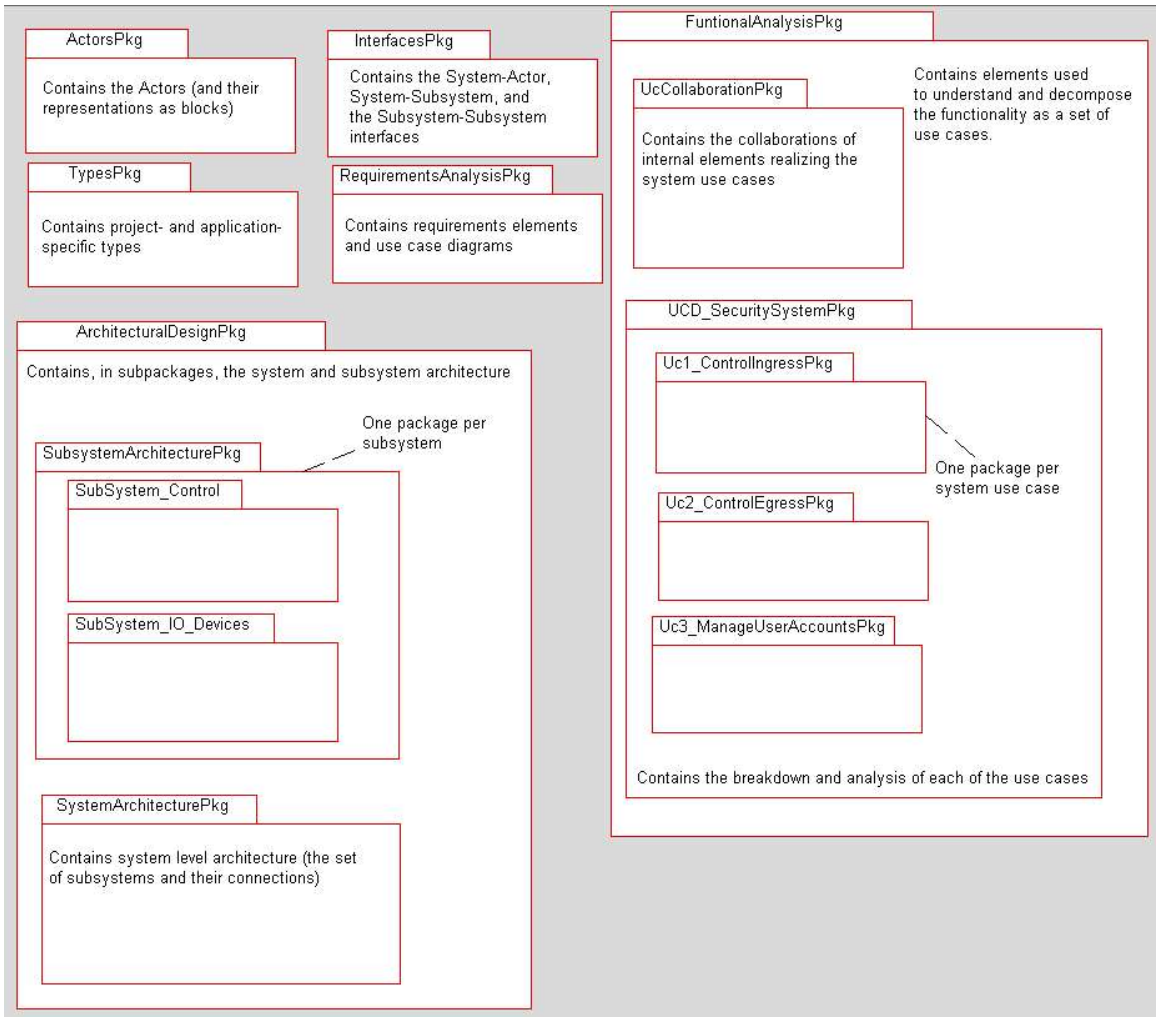

Figure 2: System Engineering Model Organization (Package View)

As you can see in (and in the browser view, shown in), the high level organization of the system engineering model consists of

- The Actors package
- The Interfaces package
- The Types package
- The System Requirements package
- The Functional Analysis (use case analysis) package
- The System Architecture package.

The functional analysis package has been further decomposed into nested packages for the Use Cases Package (which contains nested packages for each of the use case details) and Collaborations Package. The Use Case packages organize the operational and quality of service (QoS) requirements as system-level use cases. Each use case is stored in its own package and is represented on a class or structure diagram with blocks and has its own black- and white-box views. The Requirements package contains parametric requirements (such as "The aircraft shall be a dull metallic color") as well as other non-operational requirements.

The System Architecture package contains the physical architecture package defines the primary subsystems for the overall system. At this level of abstraction, they are not yet decomposed into the various engineering disciplines – that is done in the derivative subsystem model. The Subsystem Interfaces package holds the interfaces specified between the system, actors and subsystems.

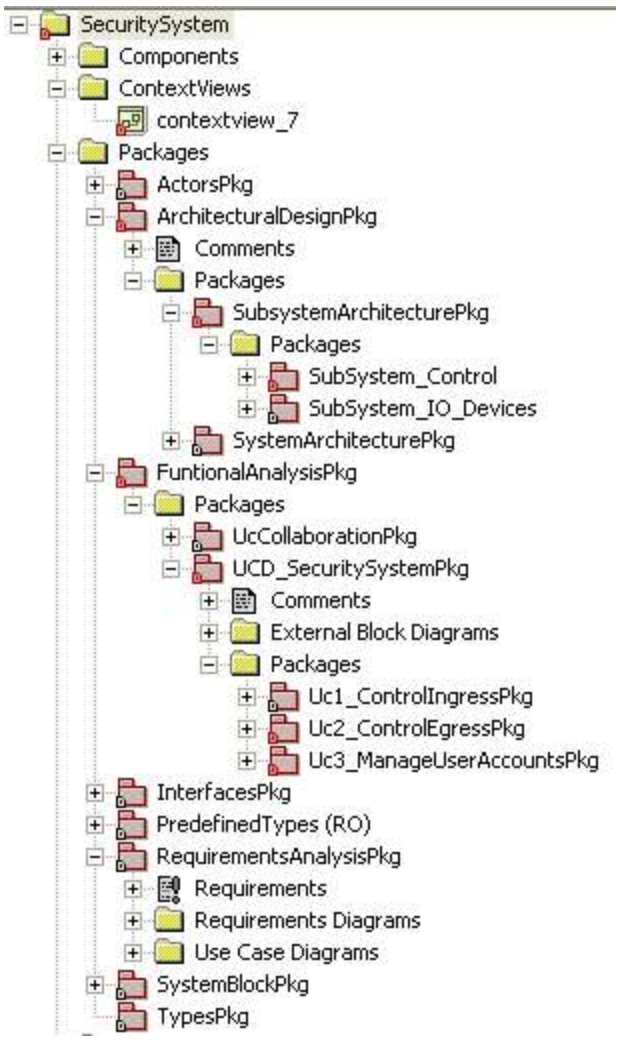

Figure 3: Systems Engineering Model Organization (Browser View)

The next figure, shows some of the subsystems in the example aircraft system, with their ports and interfaces. Since each subsystem is an encapsulated entity with well-specified points of interaction, the subsystem team responsible for developing the subsystem has it needs between the requirements for the subsystem (in the Subsystem Specifications package) and the interfaces among the subsystems and between the subsystems and the system (in the Subsystem Interfaces package).

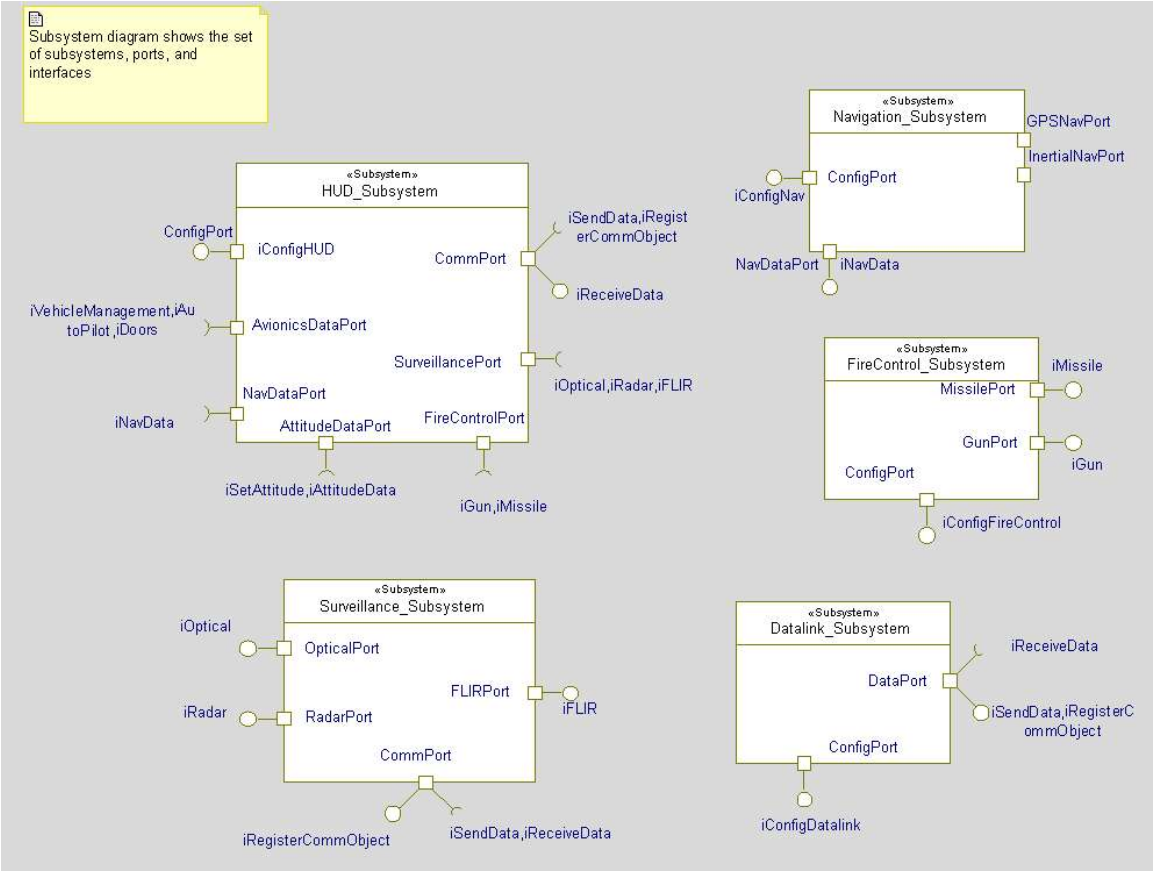

Figure 4: Subsystem Ports with Interfaces

 shows the detail of some of these interfaces in terms of the services that they provide to other subsystems or require from them.

In addition, parameters passed may be stated and detailed by specifying those parameters are primitive types or as structured types (called “classes”).

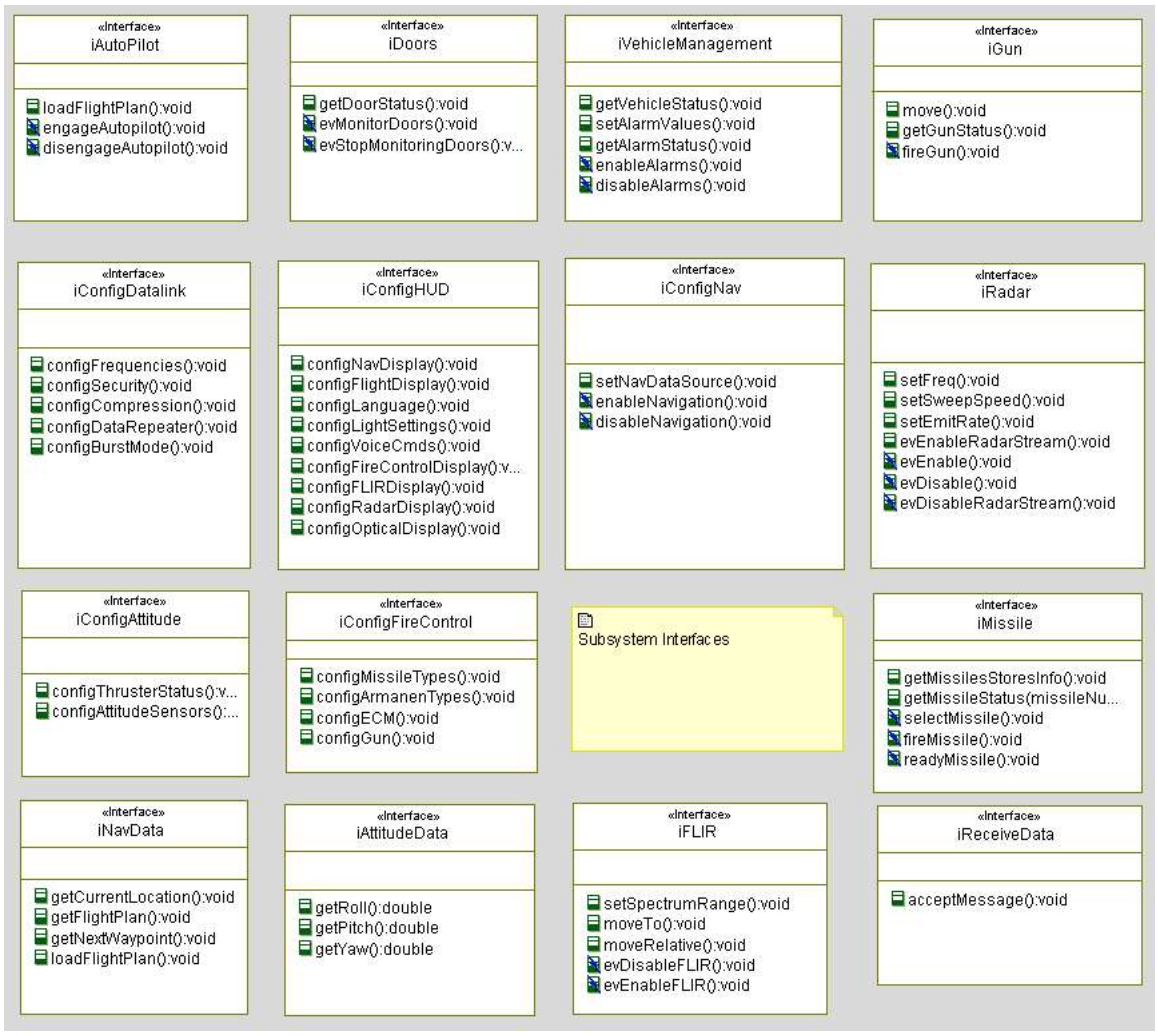

Figure 5: Subsystem Interfaces (detail)

The information shown in can also be shown in spreadsheet format – known as an N^2 diagram. shows the provided and required interfaces between the various subsystems. If desired, the details of the specific services within the interfaces as well.

		Provided Interfaces							
		Avionics Subsystem	Navigation Subsystem	Fire Control Subsystem	Attitude Control Subsystem	Thruster Management Subsystem	Surveillance Subsystem	HUD Subsystem	Datalink Subsystem
Required Interfaces	Avionics Subsystem		iConfigNav	iConfigFire Control	iConfigAttitude	iThruster		iConfigHUD	iConfig-Datalink
	Navigation Subsystem								
	Fire Control Subsystem								
	Attitude Control Subsystem								
	Thruster Management Subsystem								
	Surveillance Subsystem								
	HUD Subsystem	iDoors iAutopilot iVehicle-Management	iNavData	iGun iMissile	iSetAttitude iAttitudeData		iOptical iRadar iFLIR		
	Datalink Subsystem						iRegisterComm Object		iSendData iRegisterComm

									Object

Table 1: Subsystem N^2 Diagram

For the purpose of handoff and for general understanding, it is also useful to create a subsystem interface diagram per subsystem. The contents of this diagram are a single subsystem, its ports, and the detail of all its interfaces. Two such diagrams are shown below: a simple one is shows the interfaces for the Fire Control Subsystem and a more complex one for the HUD (Heads Up Display) Subsystem.

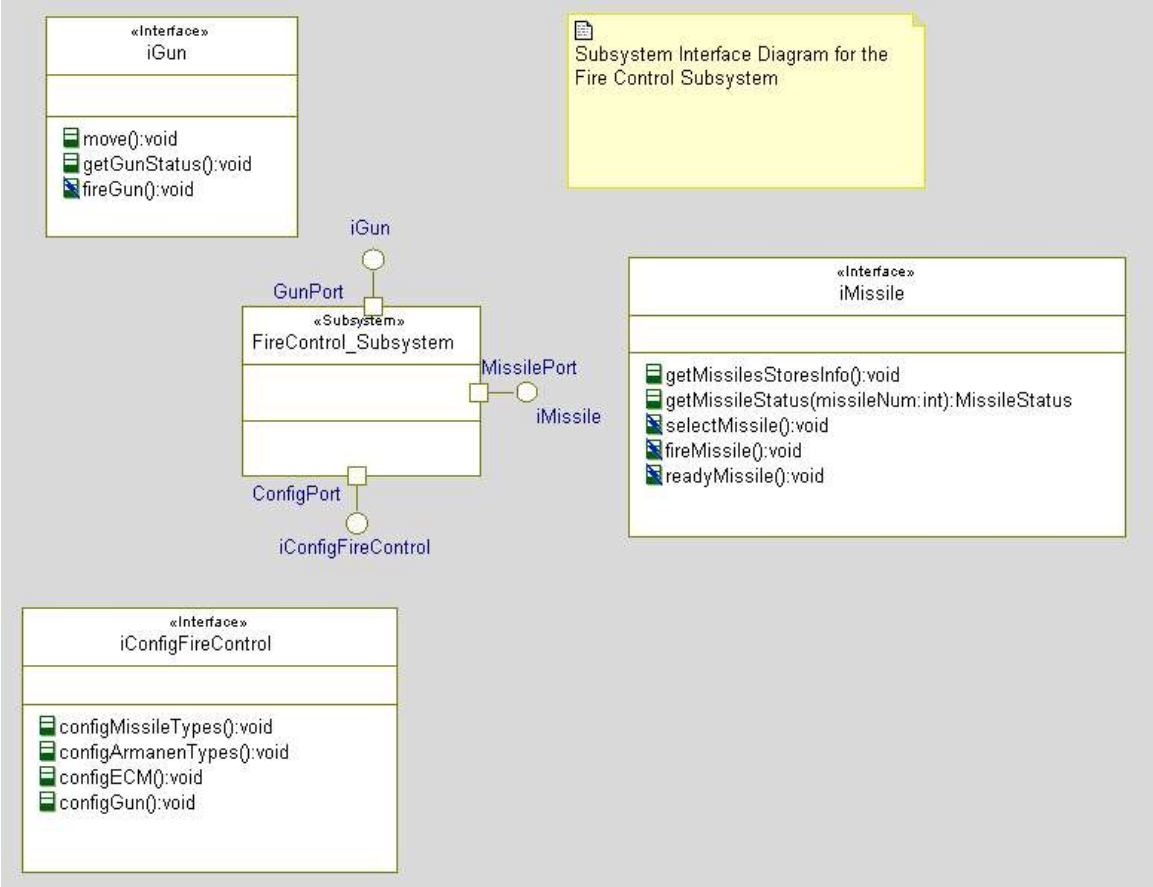

Figure 6: Fire Control Subsystem Interface Diagram

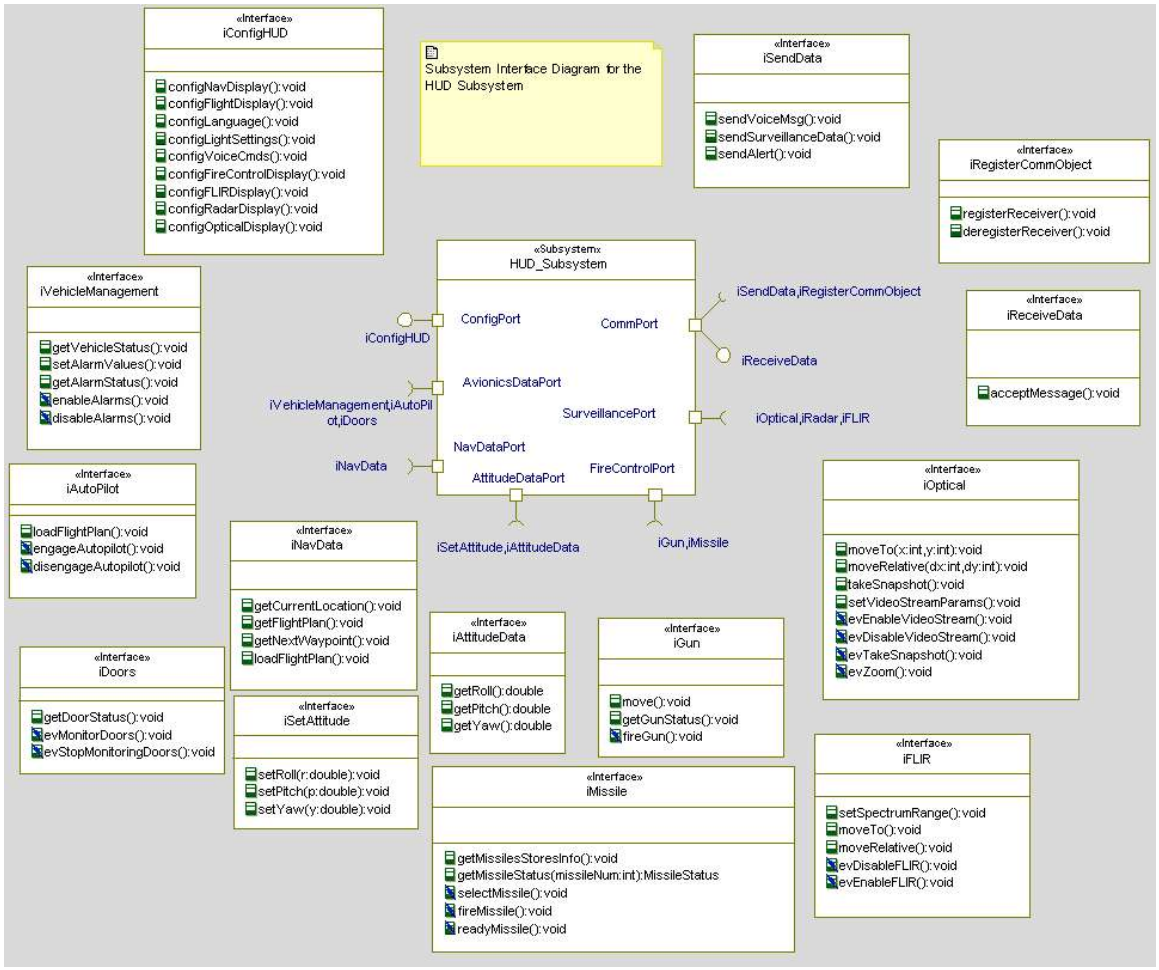

Figure 7: HUD Subsystem Interface Diagram

The mission for and is to show, at a glance, for the specific subsystem the set of interfaces it both provides and requires. This is, of course, a useful view for the team developing the subsystem in question but also for the teams that must interact with it.

Lastly, the Subsystem Specifications package is subdivided into one package per physical subsystem. The point of this organization is to simplify the hand-off to the subsystem team. shows the Subsystem Specifications package in more detail. We see that there is a nested package for each of the identified subsystems. Within the specific subsystem specification package, we see packages for the requirements, use cases, use case details and so on. As we shall soon see, the subsystem specification package is the primary handoff to the subsystem team.

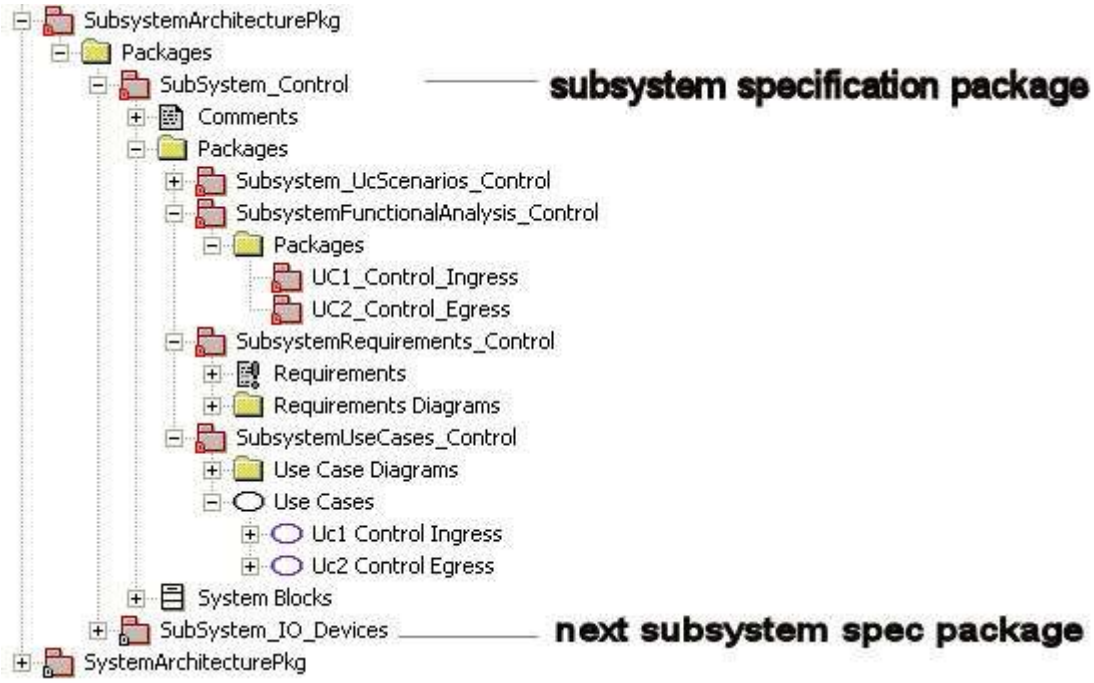

Figure 8: Subsystem Specifications Package

Preparation: Project Model Organization

The organization of the system engineering model is just one aspect of model organization. In a large-scale project, there is normally a set of related models:

- System Engineering Model
- Shared Model
- Subsystem Models (1 per subsystem)

You can see the basic organization of this set of models in . For smaller systems, it is possible to make each of these models into nothing more than a package in a single Rhapsody model, but it is almost always better to have them as separate models.

Some of the models are imported while others are referenced. By “imported”, we mean that you add the model “by copy” – the new model adds (and manages) its own copy of the original model or package from the source model. This is shown with the «import» dependency. By “referenced” we mean that the model or package from the model is added to the model but in such a way that the user model cannot change the referenced elements and if the original changes, the user model is updated with those changes. This is shown with the «usage» dependency.

The system engineering model (SE Model) structured is described above. The other two kinds of models are the shared model and the set of subsystem models.

The subsystem models each begin by importing the relevant subsystem specification package from the SE model and then adding, by reference, the shared packages for system

architecture and interfaces from the SE model. The import is used because the specifications shouldn't change in the Hybrid-Spiral variant of the overall process, the one that defines an explicit handoff to software (and so, models that are the subject of this paper).

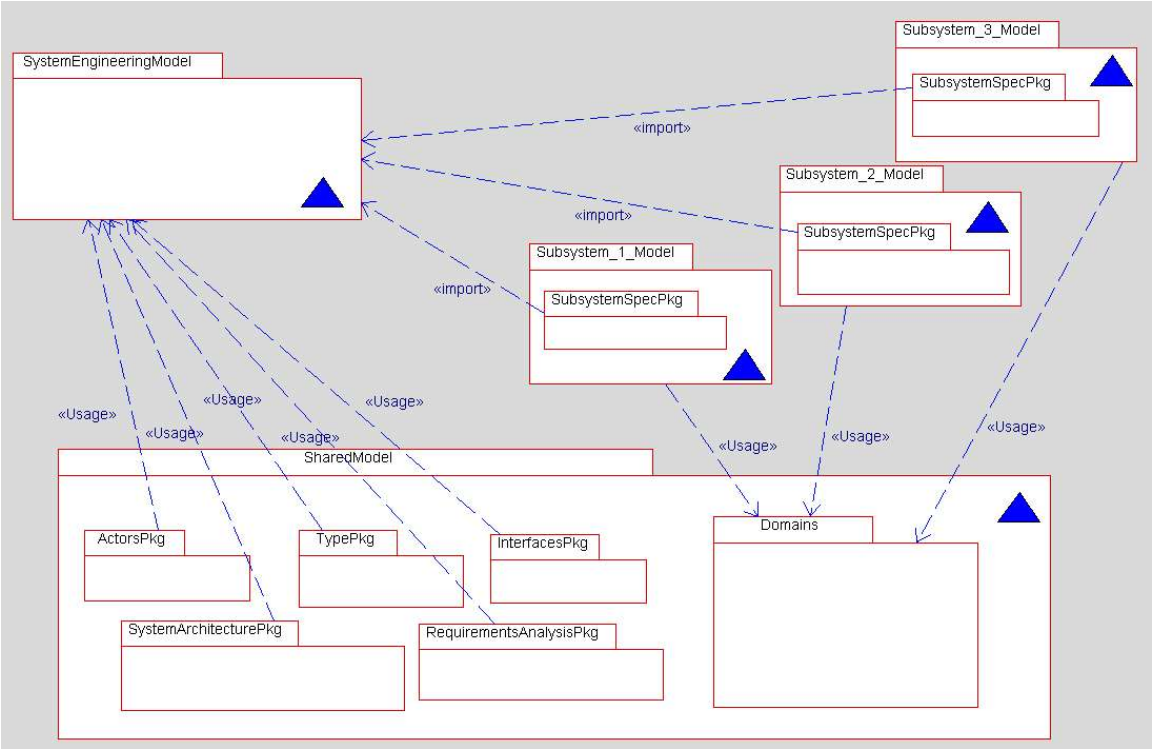

Figure 9: Recommend Set of Models for a Project

From there, the subsystem models are elaborated as their detailed analysis and design progresses. The detailed subsystem model organization will be discussed in the section entitled .

The last kind of model is the Shared Model. The purpose of this model is to hold things that are shared among multiple subsystem teams. This model starts out life by importing the system architecture (which includes the interfaces for the system and subsystem interfaces) and the requirements (important for traceability and allocation). Later, as software progresses, the Domains package is added. This package is *not* of concern to system engineers, but is crucial to the software engineers. Inside the Domains package are nested packages that hold reusable elements (software classes and types).

Note: Software Engineers Only – Domains in the Shared Model[1]
Before we leave this topic, a brief discussion of domains is in order. A *domain* is defined to be a coherent subject matter with its own vocabulary and defining concepts. User

[1] In general, systems engineers need not be concerned about the content or organization of the domains, since they are software-specific and added after the handoff to the subsystems occurs. However, the discussion of domains is added here because the subsystem teams do need to understand the organization and semantic content of the shared model, including the domains.

Interface is such a coherent subject matter and may contain classes such as Window, ScrollBar, Icon, Cursor, and so on. These are stored in the shared model but can be reused in any or all subsystems.

The classes within the user models may either be specific to that subsystem – in which case they are defined and used solely within that scope – or they may be shared across multiple teams and models. Shared classes are placed within the domain defining those elements of that subject area.

For example, suppose that the aircraft being constructed uses datagrams from a common bus architecture specification, such as Ethernet or 1553. It would be inefficient to have each subsystem team develop their own classes to represent the datagrams and formats, so we would put such classes and types within a Comm_Domain package so that they can be shared and reused among the teams. A sample domain structure is shown in .

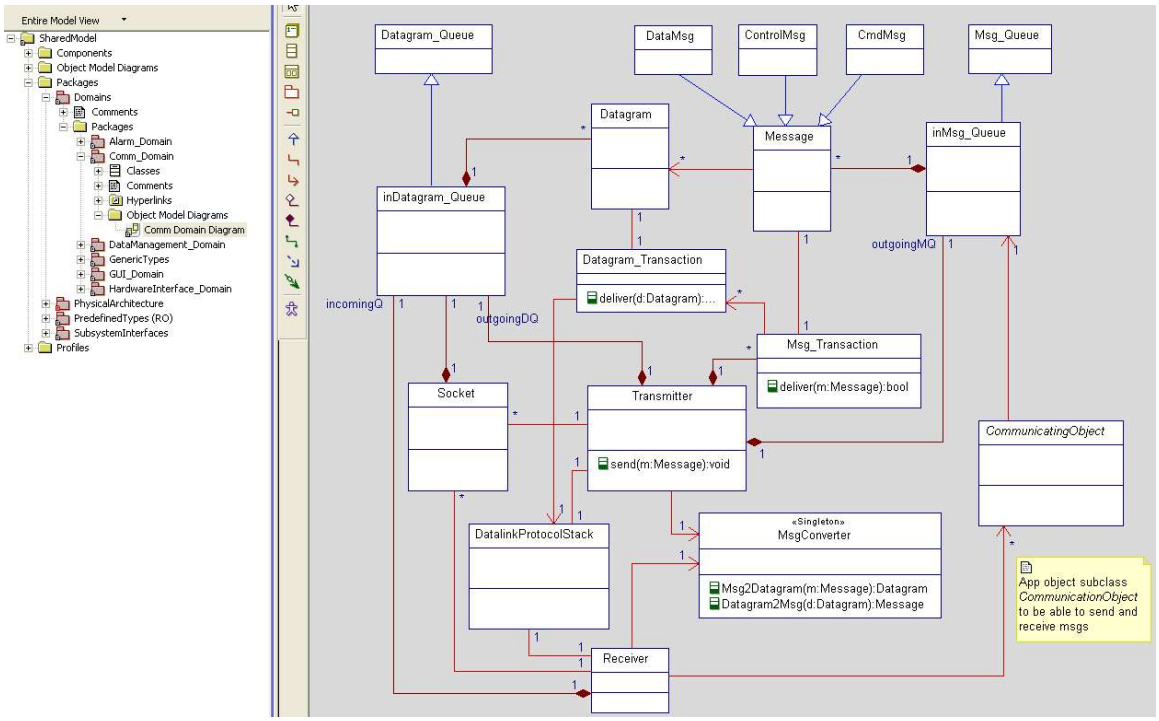

Figure 10: Domains

On the left-hand side of the figure we see that several domains are defined:

- Alarming
- Communications
- Data Management
- Generic Types
- GUI
- Hardware Interfaces

Each of these contains several classes. A *domain diagram* - showing the contents of the Communications domain is shown on the right-hand side of the figure. The elements of

this domain may be used in the subsystem team models in combination with their own subsystem-specific elements. It is common to subclass the domain classes when they must be extended or specialized for the subsystem team usage. A typical usage is shown in . This collaboration structure is taken from the Fire Control Subsystem.

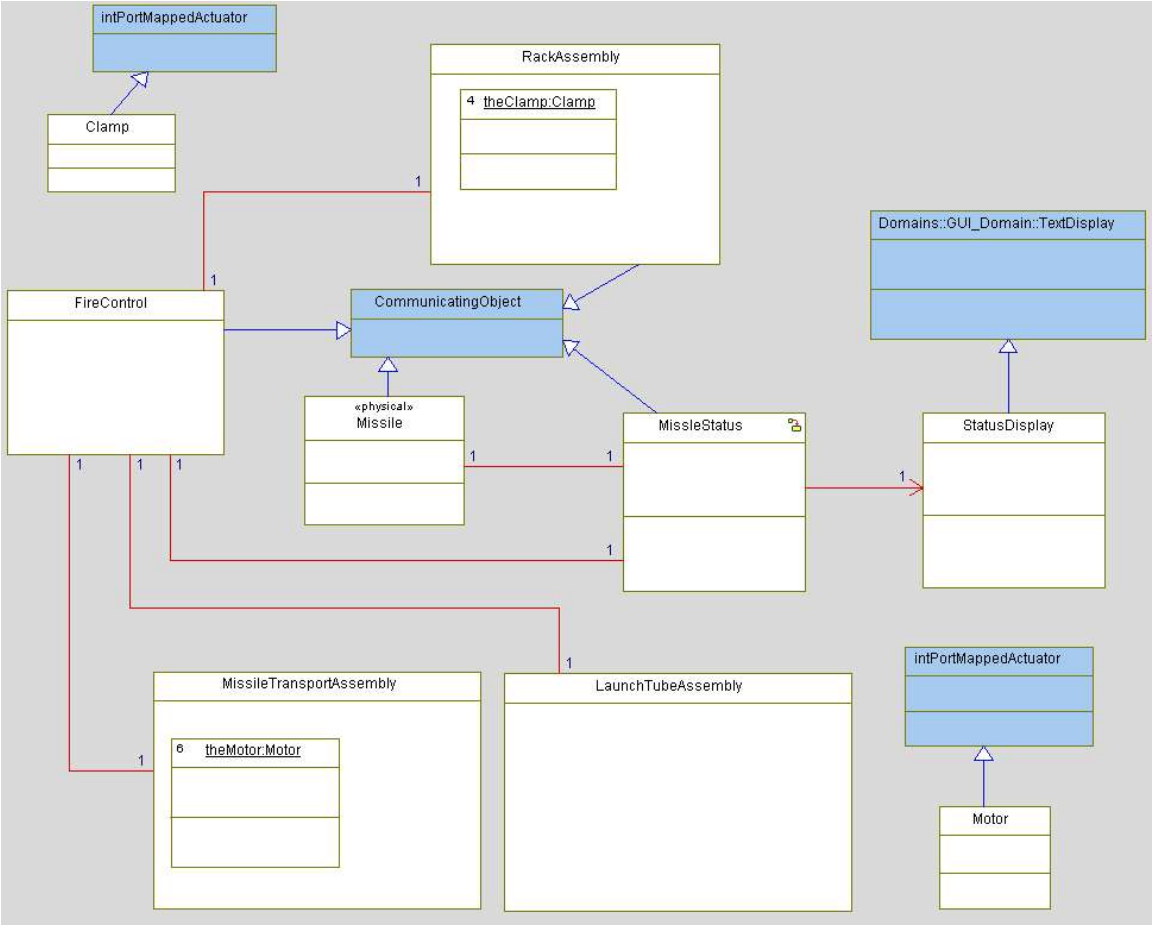

Figure 11: Use of Domain Classes

The colored elements are defined in the domains. We can see that the FireControl, Missile, RackAssembly, and MissileStatus classes subclass CommunicatingObject from the Comm_Domain so they inherit the ability to send and receive messages. The StatusDisplay is a specialized TextDisplay from the GUI_Domain so it knows how to display text. The MissileTransportAssembly contains a set of six motors; the class for this is based on the IntPortMappedActuator, so that its interface is defined in terms of an integer value written to or read from a specific port address. In this way, common definitions of elements from the various kinds of domains can be efficiently reused among various teams.

Handoff to Subsystem Teams

In the previous sections, we recommend a way of organizing the SE model and the set of models constituting the overall project. One of the primary goals of the recommended organizational schema is to facilitate the hand off of portions of the SE model to downstream engineering concerns, particularly the subsystem teams. The structuring of the Subsystem model is shown in .

As mentioned, a subsystem team needs three pieces of information to do an effective job at designing their portion of the overall system:

- The requirements for the subsystem
- The interfaces to which they must adhere
- The architecture into which the subsystem must fit

These areas are clearly delineated out in the systems engineering model structure outlined in the previous sections.

Capability Allocation: from System Use Cases to Subsystem Use Cases (recommended)

There are three primary ways to allocate requirements down to the subsystem level. The first is just to pass operational contracts, specified on sequence diagrams, only. The second pass operational contracts, and then group these into use cases at the subsystem level (bottom-up approach). The third, which we will discuss here, is to decompose the system use cases into subsystem use cases (top-down approach). Since this section only applies to the latter workflow, this section need not be read by engineers doing either of the first two approaches.

The primary focus of systems engineering is to precisely specify the requirements and define a system architecture that supports those requirements. The handoff to the various subsystem teams consists of the requirements specific to that subsystem and to the architecture into which that subsystem must fit. Thus, one of the steps that must precede the handoff is the identification of the specific requirements of each and every subsystem.

Subsystem requirements arise from the interaction of two sources – the system requirements and the system architecture. The system requirements are, for the most part, organized around use cases – large-scale capabilities of the system. More detailed views (e.g., requirements diagrams, sequence diagrams, statecharts, and activity diagrams) are used to fill out the detail of what is required. Thus, use cases are not individual requirements, but rather coherent clusters of requirements around a common capability. We will use the same concept to organize the requirements for each of the subsystems as well.

The advantages to using use cases to organize subsystem requirements are three-fold. First, some organizational principle is needed to structure the requirements into sets that are usable by the subsystem development team. Use cases are not only well-understood, but they also have strong support within the UML and its supporting tools. Second, it is possible to create a mapping from the system use cases with dependencies and hyperlinks

to provide good traceability from system requirements into subsystem requirements and design. Tool such as the Requirements Gateway provide excellent support for tracing derived requirements and design realizations from system requirements. Lastly, use cases are the fundamental basis for the development cycle that is to following the systems engineering activity. Specifically, the system is development as a set of incremental constructions (called “builds” or, more commonly, “prototypes”). Each prototype realizes and is validated against a set of requirements. Use cases fit the need for specifying the prototype missions extraordinarily well and provide a basis for project management as well as design.

As part of the system engineering handoff, a requirements model for each subsystem must be developed by the system engineering team (see for the organization of this handoff subsystem structure). The subsystem package within the SE model is organized around the subsystem use cases this package is loaded into the subsystem model as the starting point for the subsystem development. The subsystem model is not normally broken down into the various engineering disciplines by the system engineers. However, once handed off, the subsystems team – along with other stakeholders – will specify the decomposition of that subsystem into the different engineering disciplines.

The basic workflow for doing the systems-to-subsystems use case decomposition is shown in . The point is that if we take each system level use case in term and look at the operational contracts that result from the decomposition of its black-box system-level sequence diagrams into the white-box subsystem-level sequence diagrams.

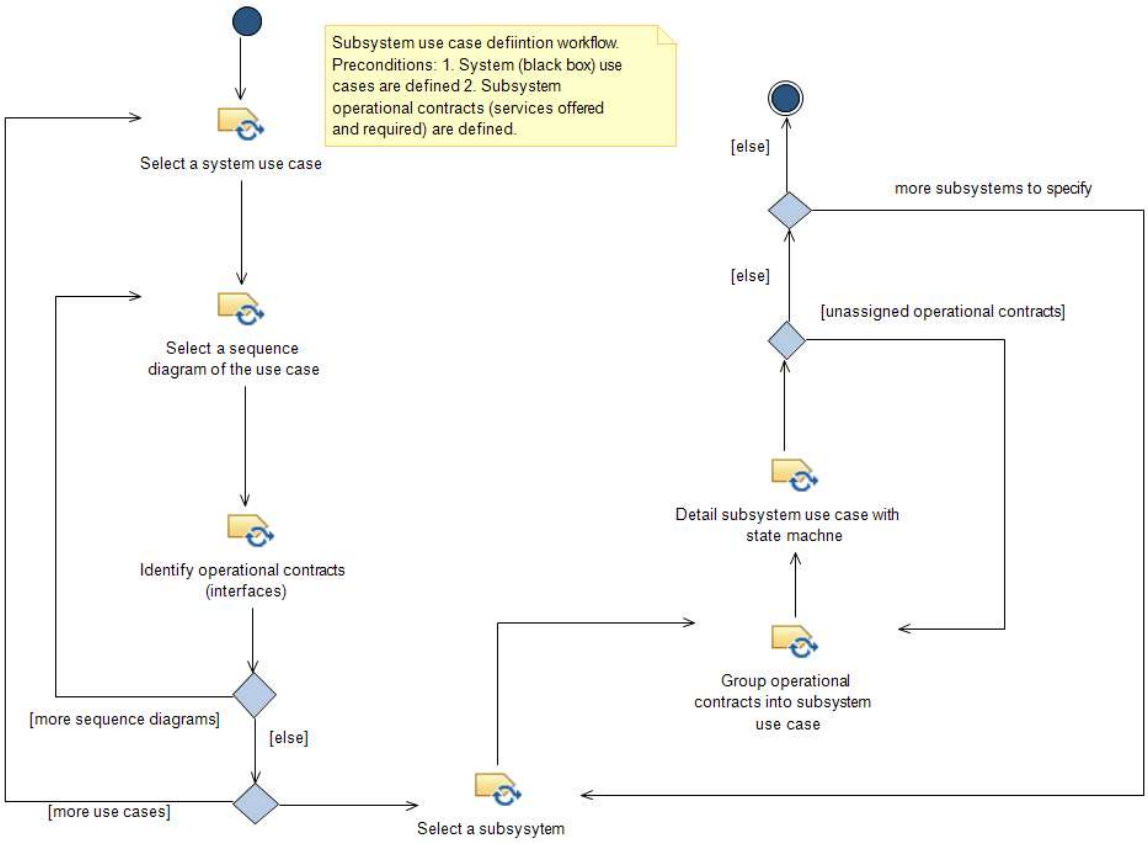

Figure 12: Subsystem Use Case Definition Workflow

Once this workflow has been completed, each subsystem has a set of use cases, elaborated into operational contracts (services offered or required by the subsystem) and details such as subsystem black-box state behavior or activity diagrams specifying the required behavior of the subsystem in detail. This completes the picture of the requirements of the subsystem and the analysis, design, implementation, and validation of the subsystem can proceed. This set of use cases for each of the subsystems, derived from the system use cases and the subsystem architecture, constitutes the hand off from system engineering to the IDC.

Organizing the Subsystem Model

Different handoff artifacts are kept in different models. For the subsystem-specific information is passed off to each subsystem team as a separate model. Each subsystem begins life with an empty project and then loads “by value” (i.e. makes its own copy of) its subsystem specification. This is appropriate because only that team will elaborate that specific subsystem into a realized design.

The other parts must be shared among the subsystem teams and so are stored in a separate model that is shared among the subsystem teams. This shared model contains the system architecture and interfaces. This shared model will (as the development progresses) also include reusable design pieces, organized within packages known as “domains”. The

shared model is loaded “by reference” into the subsystem models rather than by value since changes made to this model need to propagate to all the users of that model.

In both cases, this loading is done via the “add to model” feature of Rhapsody. Once the handoff is complete, there is a shared (and referenced) model of the architecture and the interfaces, and each subsystem has created its own project from its specification in the systems engineering model.

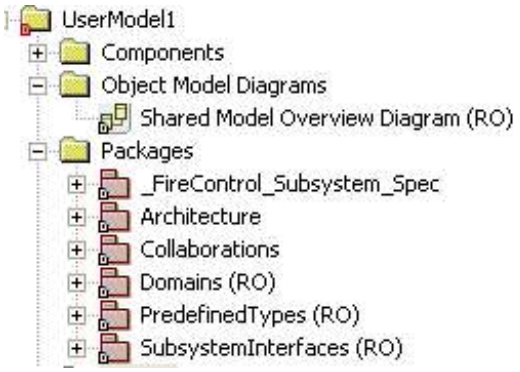

Figure 13: A Sample Subsystem Model

shows the basic structure of the subsystem model. The FireControlSubsystem_Spec package was added by copy from the Subsystem Specifications::FireControl Subsystem package. The Subsystem Interfaces package and Domains packages (and the Shared Model Overview diagram) were loaded by reference from the Shared Model. The other parts – the architecture and collaborations packages are added by the subsystem team to further elaborate this specific subsystem model.

The Architecture package contains architectural-level classes within the subsystem model and classes which represent the architectural units for the electronic and mechanical aspects of the subsystem. The details of the interaction between these disciplines is shown on a “deployment model” diagram, which may be either a UML deployment diagram or (as is recommend in SysML) a class diagram. An example of such a diagram is shown in . In this case, the structural elements are colored coded (as well as stereotyped) to indicate which discipline is responsible for their design.

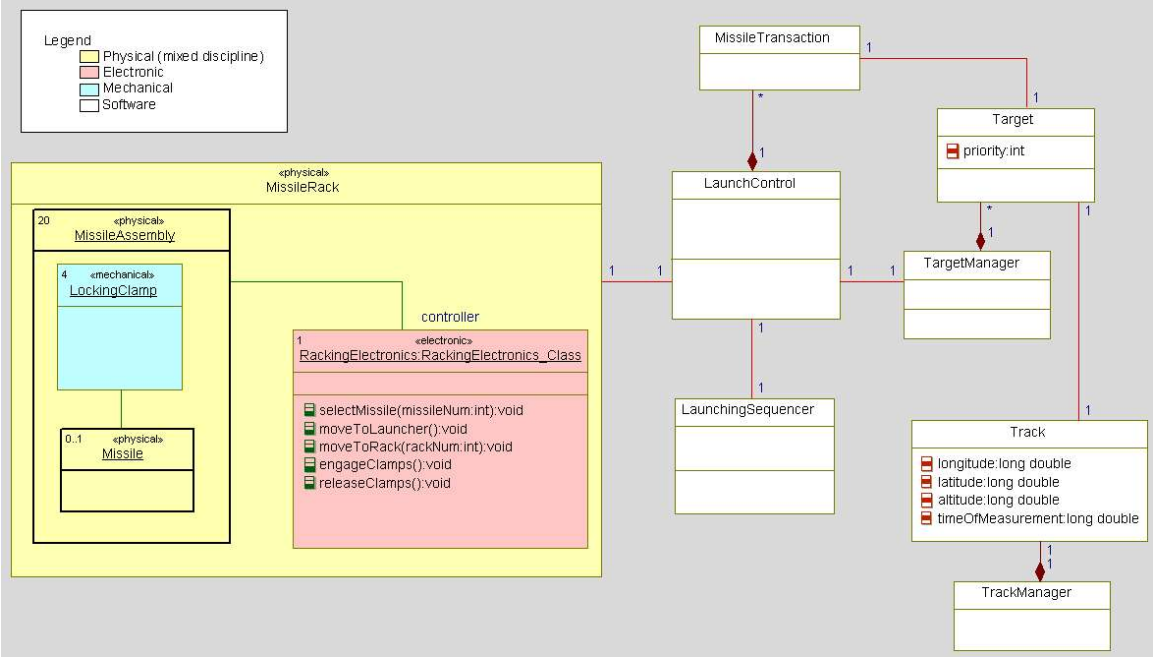

Figure 14: Subsystem Deployment as a Class Diagram

Of course, more detail than this is necessary for the engineering disciplines to work together. It is crucial to specify the properties of the interfaces. From the software point of view, only the software-electronics interfaces are useful. These may be memory, port, or interrupt mapped. We recommend adding this information to the interfaces in the form of *tags* (also known as "tagged values") so that they can be assigned values. For example, see . In this figure, we show a set of operations (services) defined on the electronics part called "RackingElectronics" and the tags associated with the service. In this case, the service is port-mapped to port addresses 0010:AF14 through 0010:AF17, holding a value which is a 32-bit integer. These tags hold the requirements that both the electronics and software engineers will design against. If, for some reason, the interface details change, these tags may be easily updated.

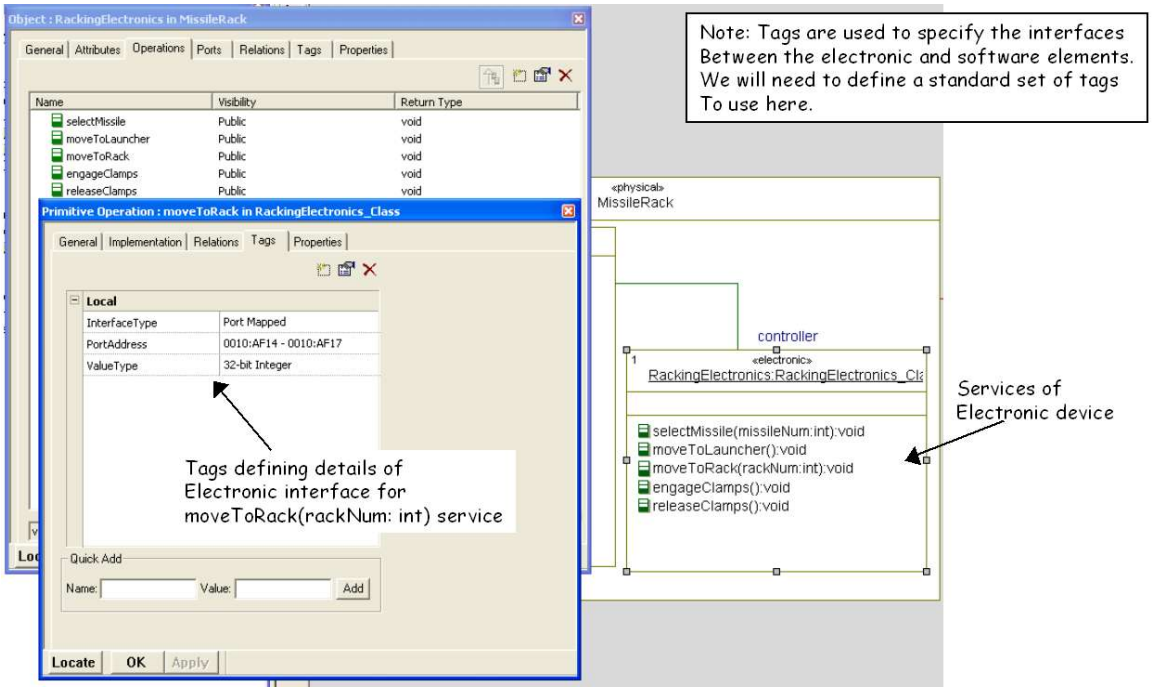

Figure 15: Deployment Interfaces

The Collaborations package in is intended to hold the collaborations realizing the subsystem use cases. This package is subdivided into nested packages for each collaboration realizing a use case.

The remaining two packages (the PredefinedTypes package is automatically included by Rhapsody) are imported *by reference* from the shared model.

This basic model structure takes the hand off from system engineering and allows straightforward elaboration into a design model as the project progresses. Let us now consider how that is accomplished.

Handoff to Software Subsystem Teams

At this point, the overall project set of models has been organized as has been the subsystem model (for each of the subsystems, working independently). Within the subsystem team, the hand-off to the various disciplines must be performed.

For non-software disciplines (e.g. mechanical engineering, electronic engineering, chemical engineering, etc), the hand off is easy. The original handoff remains untouched and serves as a specification for the functionality and quality of service for the portions done by that engineering discipline. That is, we don't expect mechanical or electronic engineering to continue in UML, but the specification of requirements *is* specified in UML. Further, the detailed interfaces among the disciplines *are* specified in UML as well.

Only the software is expected to continue with model elaboration. The elaboration proceeds with objects analysis and into software design.

Conversion of SysML/DoDAF Models to UML

If the SE model is constructed in pure UML, then the transition, as we have seen, is little more than creating a usable model organization structure, and allocating models and packages to be handed off to the subsystem teams. There are some details associated with that, but they are relatively straightforward. If the model is constructed in a Domain Specific Language (DSL), such as SysML or DoDAF, then there is an additional step to convert the model to a "plain vanilla" UML model.

This turns out to be simpler than one might expect. The SysML DSL is a UML profile – meaning that while some custom notations and lightweight extensions might be made, everything done in SysML is really done in UML underneath. All that is necessary is to expose the UML underpinnings so the software work can continue. As far as DoDAF, there are DoDAF tools that are not UML representations – such as those that rely on IDEF – but if you're using Rhapsody or Tau, then the DoDAF representation is as a UML profile. Again, all that remains is to import them without the profile, and their underlying UML representation will be exposed.

About the Author

Dr. Bruce Powel Douglass has over 30 years 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 an advisory board member for the Embedded Systems Conference, UML World Conference, and Software Development magazine. He is a former co-chair for the Real-Time Analysis and Design Working Group in the OMG standards organization. Bruce has written 15 books on software development including "Real-Time Agility" released in the spring of 2009 and "Design Patterns for Embedded C" due out in the fall of 2010. He is the Chief Evangelist at IBM Rational, a leading systems and software design automation tool vendor. He can be reached at Bruce.Douglass@US.IBM.com.