**Model-Driven Software Development Modeling Guidelines**

**With**

**UML and Rhapsody**

V 1.0

September 20, 2021

Prepared by

Dr. Bruce Powel Douglass, Ph.D.

A Priori Systems

[www.bruce-douglass.com](http://www.bruce-douglass.com)

Table of Contents

Contents

[Overview 4](#_Toc83287901)

[Tooling and Languages 4](#_Toc83287902)

[Definitions and Acronyms 4](#_Toc83287903)

[Syntactic Standards 6](#_Toc83287904)

[General Modeling Standards 6](#_Toc83287905)

[Model Organization 7](#_Toc83287906)

[Federating Rhapsody models 7](#_Toc83287907)

[Specify Design Classes, Objects, and Types in the Software Models 11](#_Toc83287908)

[Naming Conventions 11](#_Toc83287909)

[Basic Naming convention 11](#_Toc83287910)

[General Diagramming Conventions 14](#_Toc83287911)

[Every diagram shall have a singular mission 14](#_Toc83287912)

[Other diagramming conventions 15](#_Toc83287913)

[Object Model Diagrams (OMDs) 17](#_Toc83287914)

[Content 17](#_Toc83287915)

[Classes and Objects 18](#_Toc83287916)

[Interfaces 18](#_Toc83287917)

[Interfaces and Ports 18](#_Toc83287918)

[Relations 19](#_Toc83287919)

[Structure Diagrams 21](#_Toc83287920)

[Use Case Diagrams 21](#_Toc83287921)

[Use Cases 21](#_Toc83287922)

[Actors 23](#_Toc83287923)

[Relations 23](#_Toc83287924)

[Requirements Diagrams 23](#_Toc83287925)

[Relations 24](#_Toc83287926)

[Requirements Tables 24](#_Toc83287927)

[Matrices 24](#_Toc83287928)

[Sequence Diagrams 25](#_Toc83287929)

[Activity Diagrams 27](#_Toc83287930)

[Flow Charts 28](#_Toc83287931)

[State Diagrams 28](#_Toc83287932)

[States 28](#_Toc83287933)

[Actions 32](#_Toc83287934)

[Guards 33](#_Toc83287935)

[Submachines 33](#_Toc83287936)

[Semantic Standards: Using SysML 34](#_Toc83287937)

[Model Overview Diagram 34](#_Toc83287938)

[Modeling Software Architecture Context 35](#_Toc83287939)

[Modeling Systems Architecture Behavior 39](#_Toc83287940)

[Modeling Data Schema 39](#_Toc83287941)

[Defining Units 40](#_Toc83287942)

[Defining Named Constants 41](#_Toc83287943)

[Relating to ASPICE and ISO 26262 41](#_Toc83287944)

[Generating Reports 42](#_Toc83287945)

[References 43](#_Toc83287946)

# Overview

This document is to provide a set of principles and specific guidance for the development of Model-Driven Development (MDD) software in UML with the Rhapsody tool.

## Tooling and Languages

The Rhapsody tool shall be used to develop the software. Other tools, such as Simulink, may be used to develop software, but their use falls outside the guidance provided by this Guideline. The version of Rhapsody, at the time of this writing, is 9.0, although it is anticipated that the tool will be upgraded to later versions over time.

The software architecture and design will be specified in the UML Language. The action language addressed in this Guidelines is the C programming language. This has a number of specific impacts in the detailed use of the tool that are reflected in these guidelines.

## Definitions and Acronyms

BDD – Block Definition Diagram; a structural diagram type defined in the SysML, representing blocks and types, their structural properties, and their relations.

Class Diagram –a structural diagram type defined in the UML, representing classes and types, their structural properties, and their relations.

Composition Architecture – a model of the internal structure of a System in terms of its large-scale classes or blocks, their properties, and their relations.

Connected Architecture – a model of the internal structure of a System in terms of instances or parts instantiated from elements of its Composition Architecture

Connected Context – a model of the system environment in terms of instances or parts instantiated from the System and its related Actors.

Context Architecture – a model of how the system fits into its environment, in terms of the actors to which it connects and the relations among those elements.

Data Schema – a model representing information, its important properties and its relations. Data schemas are represented using BDDs in SysML and Class Diagrams or OMDs in UML.

Forced Closure Rule – The set of actions executing when a transition is taken completes when the state is entered; this includes exit, transition, and entry actions.

IBD – Internal Block Diagram; a structural diagram type defined in the SysML, representing instances (aka Parts) and their relations.

Instance – a run-time occurrence of a block or class. Also known as a *part* or *object.*

Magic number – a literal numeric value that appears in a design without documentation, justification, or rationale.

MBSE – Model Based Systems Engineering; the use of models as the primary work products to capture, represent, and analyze systems engineering data. The primary language used for MBSE is SysML.

MDD – Model-Driven Development; the use of models to capture and represent software architecture and design.

Model Overview Diagram – a diagram used to explain the purpose, scope, and organization of a model

OMD – Object Model Diagram; a Rhapsody-specific structural diagram which is basically a UML class diagram

Package Diagram – a diagram whose primary purpose is to show model organization via packages, their relations, and their content.

Parametric diagram – a SysML diagram that specializes an Internal Block Diagram. This diagram is used to show the relation between values and constraint properties.

Part – see Instance.

Rhapsody – a powerful UML and SysML modeling tool.

Scenario – a particular sequence of interactions between a set of elements, often used to model interactions during use case analysis

Sequence diagram – an interaction diagram in UML and SysML used to model scenarios and test cases.

Structure Diagram – a kind of structural diagram that shows instances, their properties, and their relations in a UML model

SysML – the Systems Modeling Language, a profile of the UML standard used to model systems concepts

Type Architecture – a model of the structural elements of a set of elements, typically capture in a UML Class or OMD diagram or a SysML Block diagram

UML – the Unified Modeling Language, the pre-eminent standard for software modeling, and the basis of the SysML modeling language.

Use Case – a collection of system actions that represents a coherent usage of a system.

# Syntactic Standards

## General Modeling Standards

* All important model elements shall have a comment that describes why the element exists, how it contributes to the model, general usage of the element, and constraints on that element. This applies to (description includes standard fields expected for the kind of element)
  + Projects
    - **Purpose**
    - **Description**
  + Diagrams
    - **Mission**
  + Classes
    - **Purpose**:
    - **Represents**:
    - **Roles**:
    - **Description**:
    - **Security Constraints**:
    - **Pre-conditions**:
    - **Post-conditions**:
    - **Invariants**:
  + Attributes and variables  
    Note: the value property metadata valid range, units, accuracy, precision, and fidelity, disallowed values, etc. may be supplied via the application of stereotypes from the *project-specific profile*.
    - **Purpose**:
    - **Represents**:
    - **Roles**:
    - **Description**:
    - **Security Constraints**:
    - **Pre-conditions**:
    - **Post-conditions**:
    - **Invariants**
  + Operations
    - **Purpose**:
    - **Processing performed**:
    - **Algorithm/pattern**:
    - **Return value constraints**:
    - **Security constraints**:
    - **Pre-conditions**:
    - **Post-condition**:
    - **Timing constraints**:
  + Types  
    Note: the value property metadata valid range, units, accuracy, precision, and fidelity, disallowed values, etc. may be supplied via the application of stereotypes from the *project-specific profile*.
    - **Purpose**:
    - **Represents**:
    - **Description**:
    - **Security Constraints**:
    - **Pre-conditions**:
    - **Post-conditions**:
    - **Invariants**
  + Actors
    - **Purpose**:
    - **Represents**:
    - **Roles**:
    - **Description**:
    - **Security Constraints**:
    - **Pre-conditions**:
    - **Post-conditions**:
    - **Invariants**:
  + Use cases
    - **Purpose**:
    - **Processing performed**:
    - **Algorithm/pattern**:
    - **Return value constraints**:
    - **Security constraints**:
    - **Pre-conditions**:
    - **Post-condition**:
    - **Timing constraints**:
* Some of these properties may be omitted if inapplicable OR if captured as metadata in stereotype tags

## Model Organization

### Federating Rhapsody models

Basic federation organization shall be:

* One systems requirements & architecture model as shown in Figure 1.
  + Note: If the Systems model is large, it can be split into multiple models
* As can be seen, this would contain the use cases, requirements and architecture of the overall system.
* The recommended SE Model organization is shown in Figure 2.
* One model per subsystem (Figure 3). This would contain the model components for an individual subsystem.
  + One package shall contain the requirements, use cases, and other specification elements for the model, imported from the SE model.
  + One package shall represent the deployment architecture (electronics, mechanical, software allocations) of the subsystem under development
  + One package shall represent the software aspects and is further decomposed.
  + Referenced packages from the Shared Model permit access to shared data and interface specifications
  + Note: Large subsystem models may be broken up into multiple models.
* One (occasionally more) shared model containing common classes, interfaces, and types (Figure 4)

When transitioning from systems engineering model to the subsystem (including the software) models, the SE model organization shall provide a package for the specification of each subsystem, as shown in Figure 1

* The subsystem specifications in the SE model is organized as one package per subsystem contained by the Architectural Design package
* The subsystem model begins by importing the relevant subsystem specification package from the system SE model
  + The subsystem spec package is added *by value* (“as unit” in Rhapsody)
* The shared packages are added into the subsystem models (aka “client models”) *by reference* so that changes to those packages are reflected in the client models when they are loaded

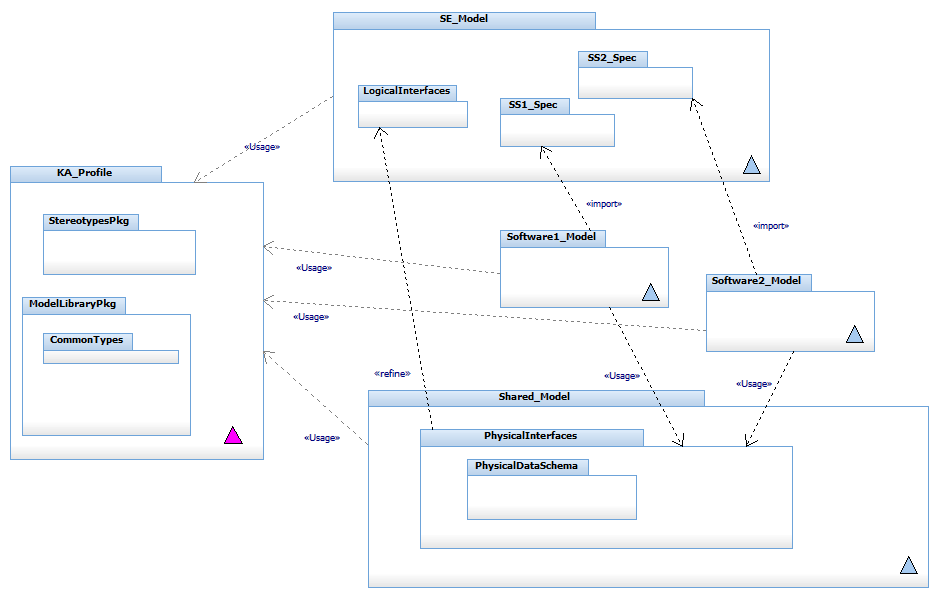


Figure : Recommended Model Organization Set

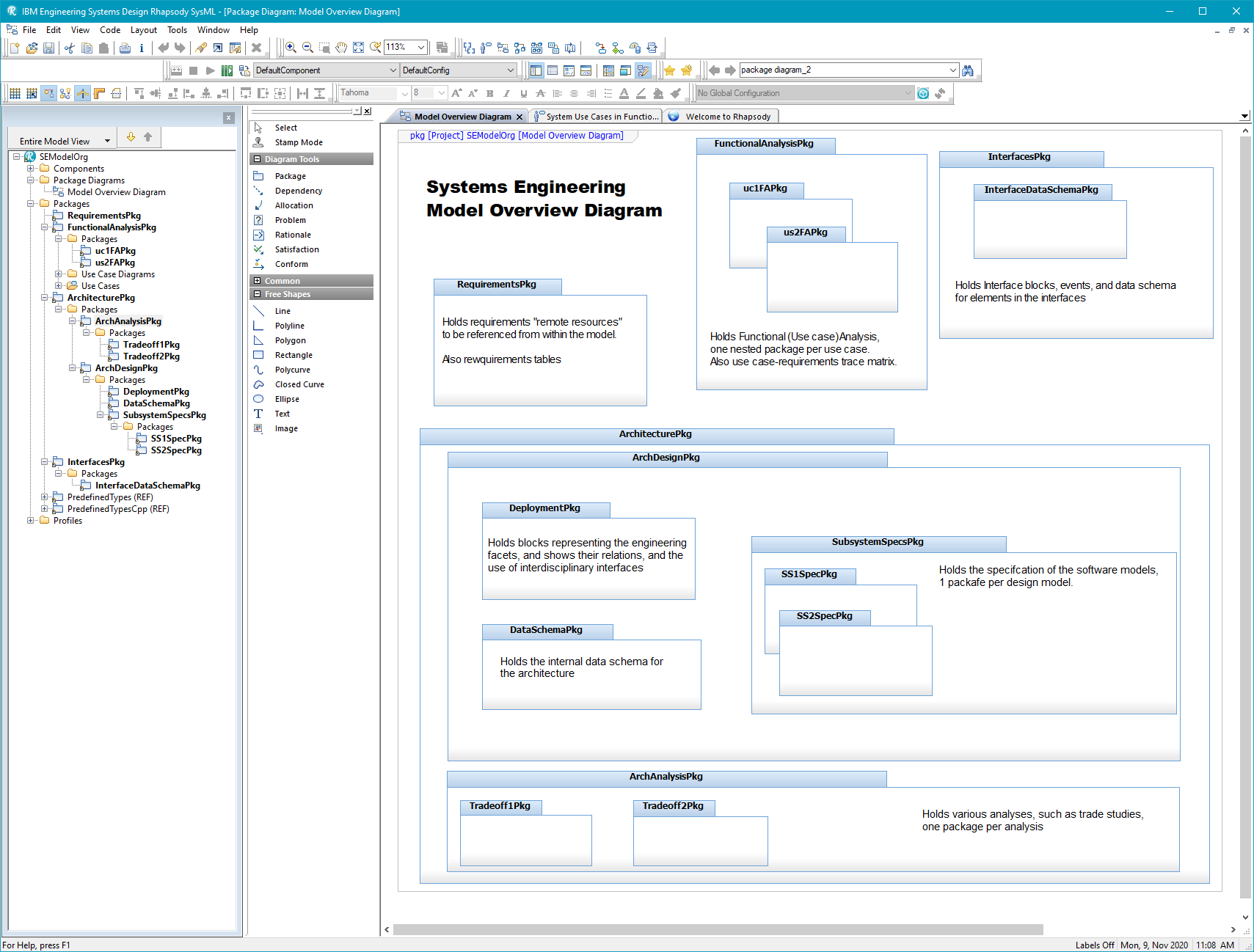


Figure : SE Model Organization

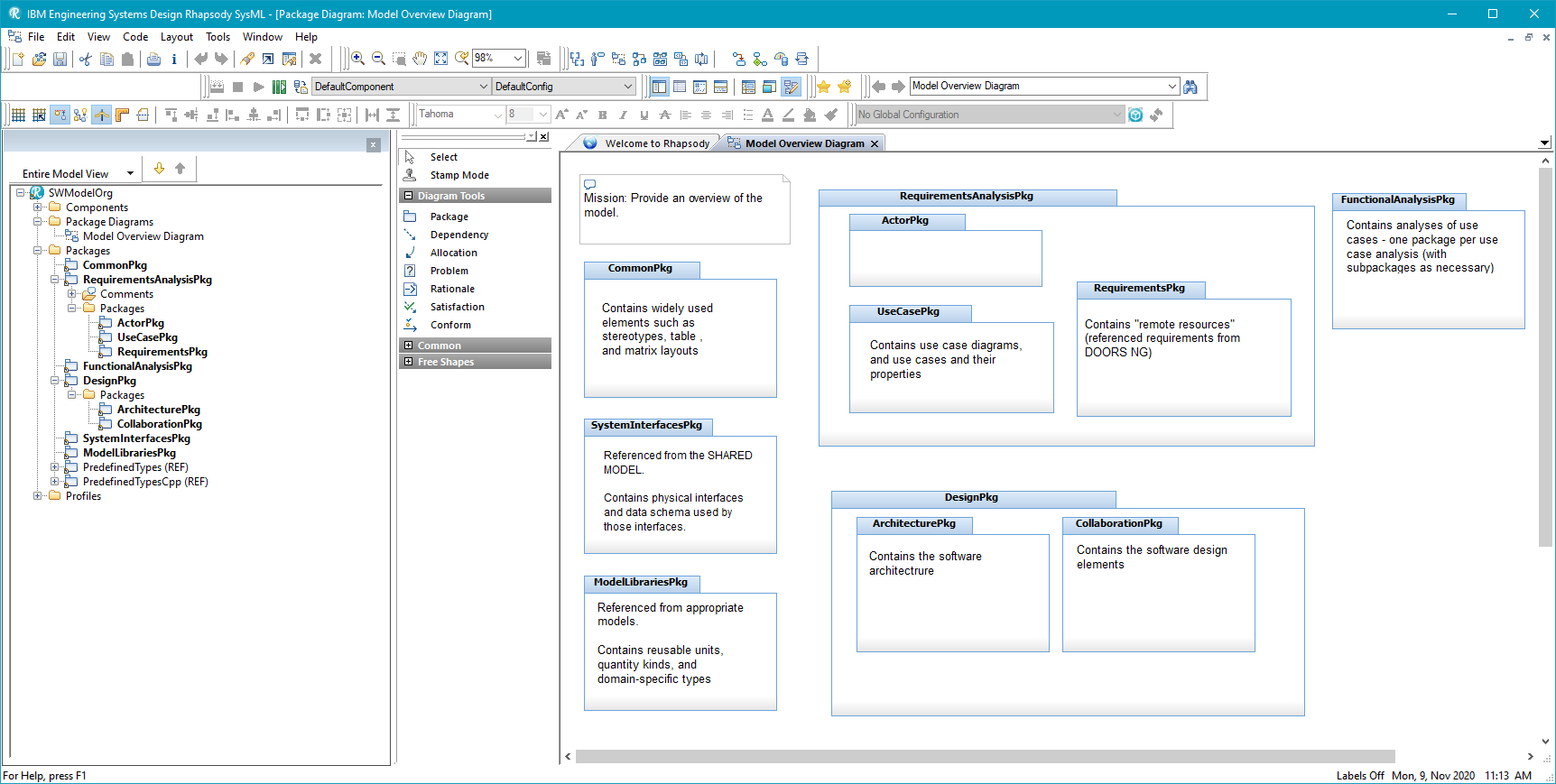


Figure : Software Model Organization

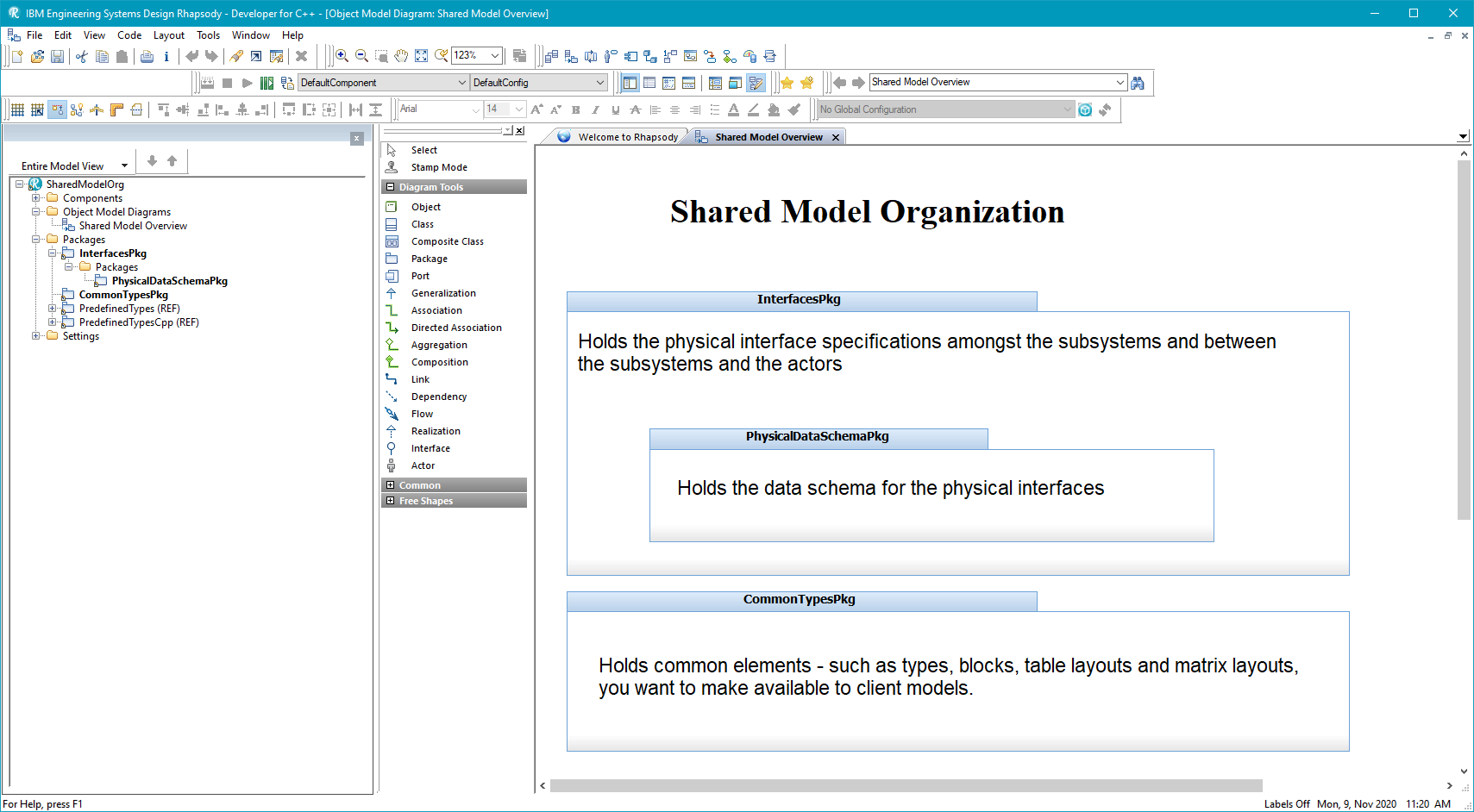


Figure : Shared Model Organization

### Specify Design Classes, Objects, and Types in the Software Models

* Software models contain information about the development of a software component or library, including the requirements, software use cases, software architecture, and software design, as needed to achieve the objectives of the model.

## Naming Conventions

### Basic Naming convention

Figure 5 through Figure 8 show examples of the following naming conventions:

* Model element names shall be taken from the appropriate domain vocabulary, whether it is a problem-domain (e.g. Tracking or Navigation) or technical design (e.g. middleware or hardware) domain
  + In general, no “default names” assigned by the tool are acceptable in a model under formal review
  + Exceptions: There are some exceptions where the name is almost never used, such as the name of the state machine for a block; it is almost always referenced as being owned by the block and so the default name is acceptable.
* Model element names shall not contain white space or “special characters” except for the underscore (“\_”), except for use case and actor names, which may contain white space
  + In general, names must be “compilable”. This is necessary to support executable models and code generation.
* Classes, interfaces, data types, value types and event names begin with an uppercase letter
* Class property names will begin with a lowercase letter. Class features include
  + Attributes
  + Functions (operations)
  + Event receptions
  + Ports
  + Association roles
  + Parts
  + Exception: When then name of the property is normally capitalized, for example O2flow (for oxygen flow), then it may begin with an uppercase letter.
* Class instance (object) name shall begin with a lowercase letter.
* “Things” (e.g. objects, classes, attributes) shall be named with strong nouns or noun phrases
* Packages names will begin with an uppercase letter, have a name that is consistent with its purpose, and end with “Pkg” as a suffix. For example, “DesignPkg” or “UseCasePkg”.
* “Actions” (e.g. operations, use cases, functions) shall be named with strong verb or verb phrases
* Association role names shall refer to the usage or role an instance of the class plays with respect to the class at the other end of the association, e.g.
  + myDataSource
  + itsDataQueue
  + theCommController
* In multiple word names, make the first letter of every word (after the first) uppercase. This is known as *camel case* as in ThisIsAClass (for a class) or thisIsAProperty (for a property).
  + Underscores may be inserted between words, if desired as in This\_Is\_A\_Class (for a class) or this\_Is\_A\_Property (for a property).
* Port names shall begin with a lower case “p” and be named either by
  + Their semantic content (e.g. pConfigData)
  + The element to which they connect (e.g. pTracker) (preferred)
* Interfaces shall be named either
  + in terms of their semantic concept (e.g. iNavData or iFLIRCommands) and their names shall be prefixed with a lower case ‘i’ OR
  + by the name of the contributing elements (e.g. iSource\_Target, where “Source” refers to the provided side of the interface and “Target” refers to the required side) – see the iComm\_Controller interface in Figure 5. (preferred)
* Named constants shall be named with ALL CAPS, as in MAX\_FLOW\_RATE.

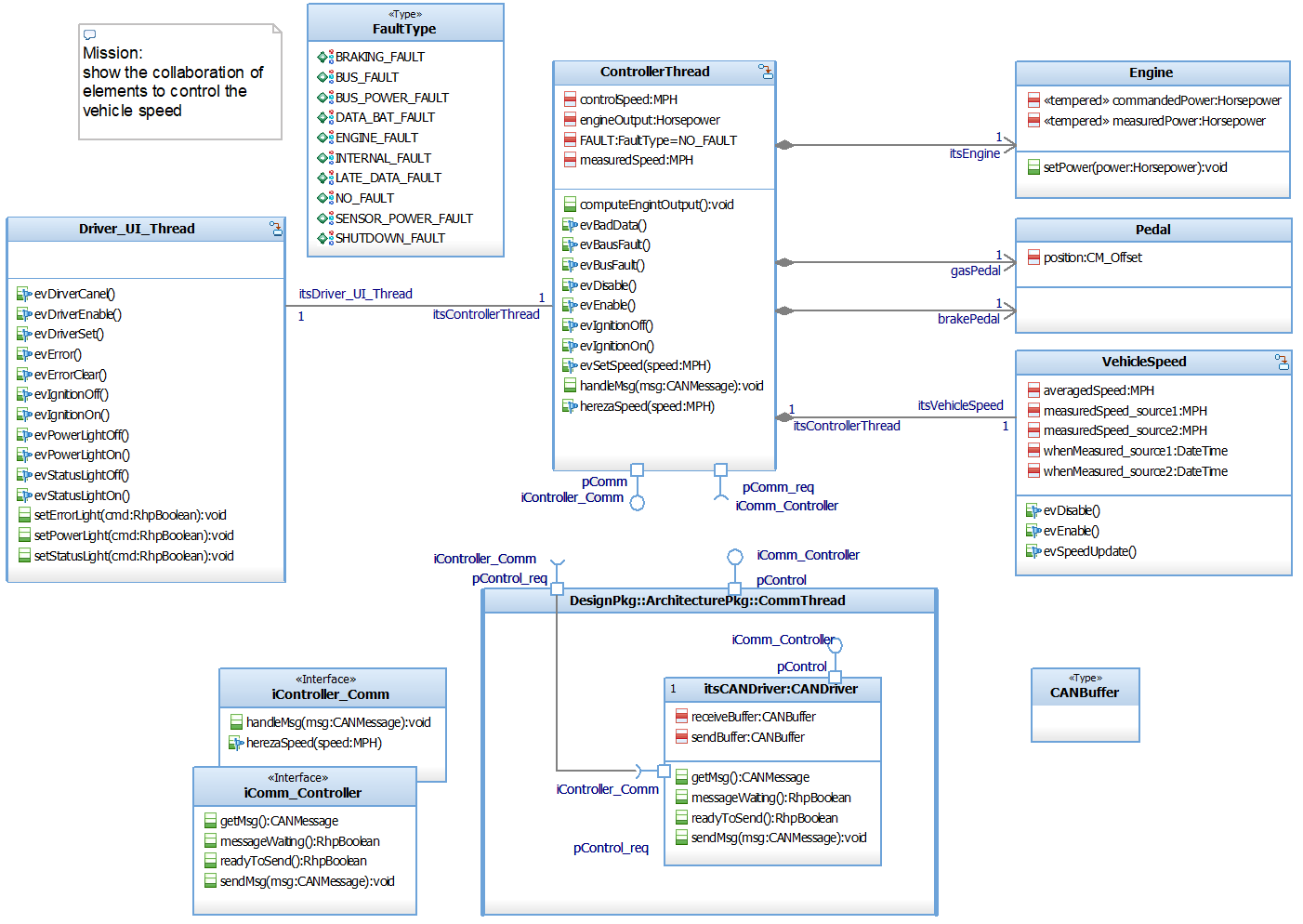


Figure : Naming Classes and Features on an OMD

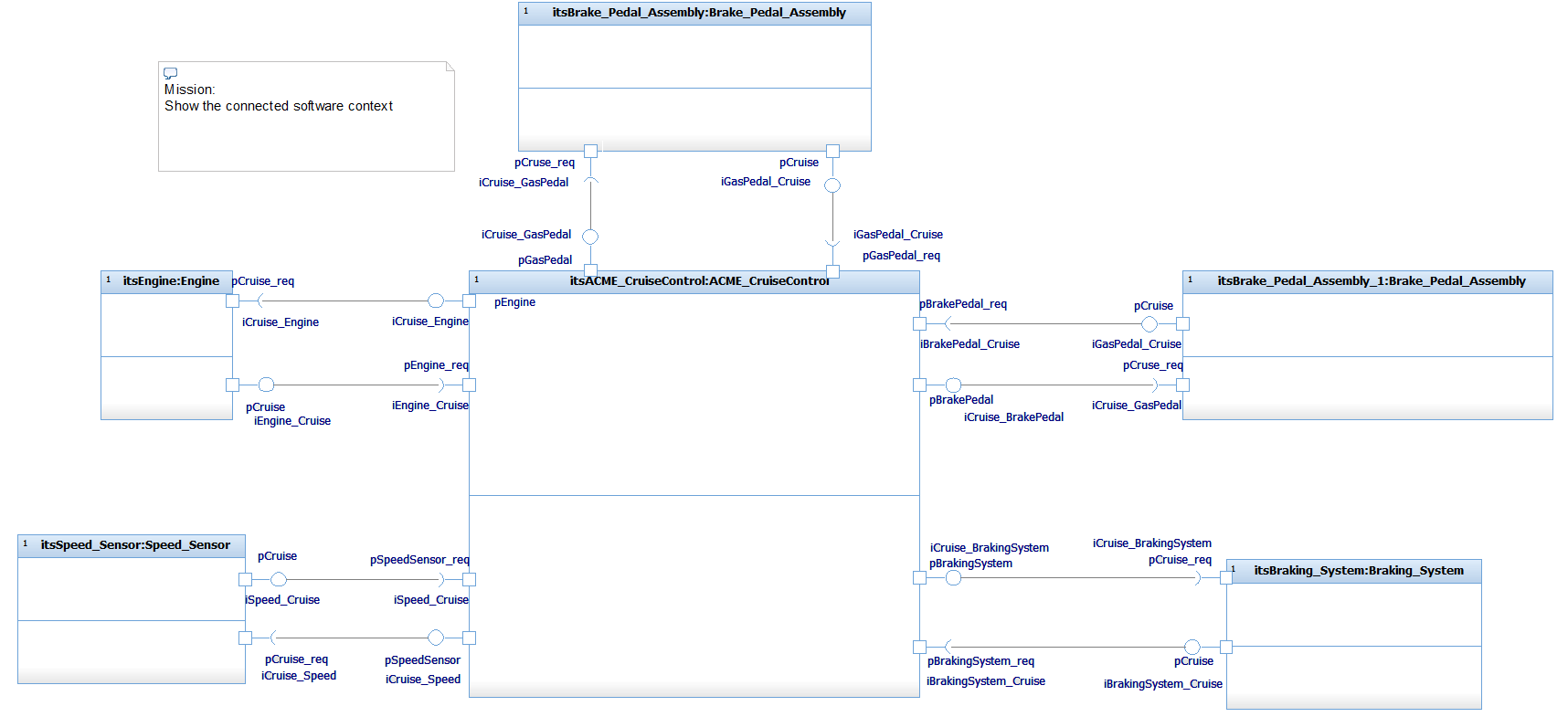


Figure : Naming on a Structure Diagram

Figure : Naming States and Transitions

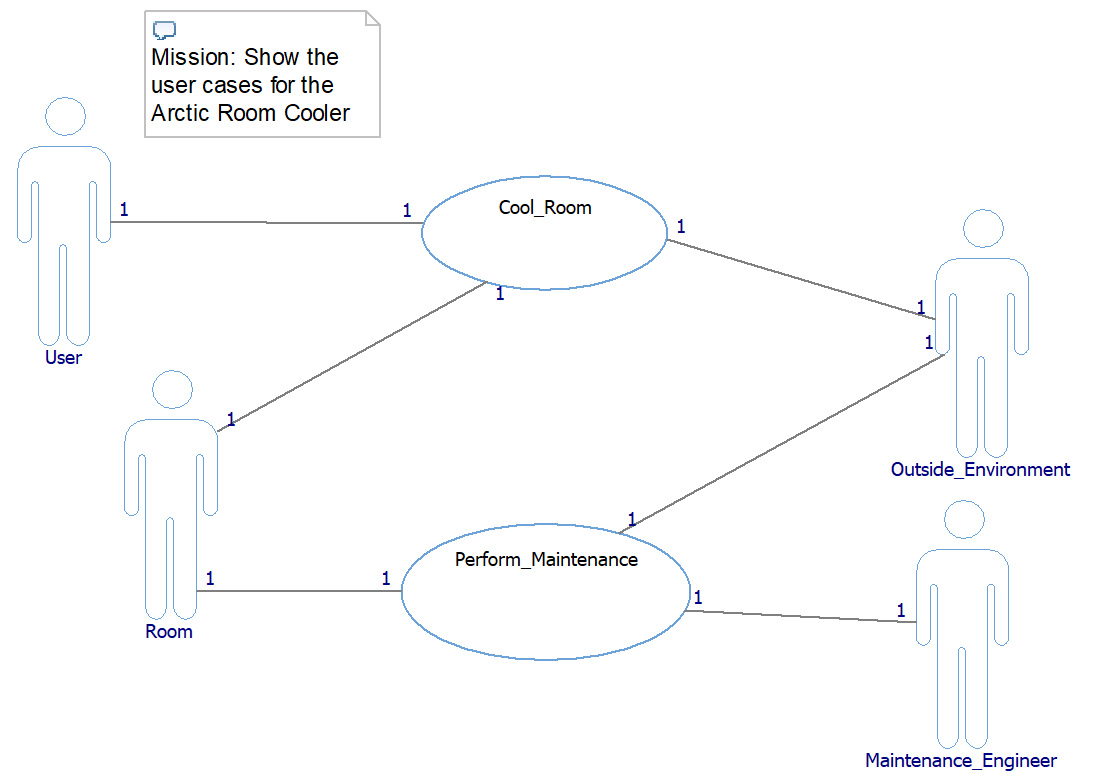


Figure : Naming Use Cases

## General Diagramming Conventions

### Every diagram shall have a singular mission

Every diagram shall have a single important concept that it is trying to show as shown in Figure 9. This is called its *mission*. This is especially important for class, activity, timing, and sequence diagrams (less important for state diagrams due to their “built-in” mission).

A diagram shall show elements related to a single purpose – a singular concept, a question (or answer), or to support a line of reasoning. This is the *mission* of the diagram.

The mission shall be stated on the diagram. Some common class diagram missions include

* Show a use case execution context
* Show a design-level collaboration
* Show a single generalization taxonomy
* Show a logical or physical data schema
* Show the contents of a package
* Show an aspect of system architecture
  + Show the subsystem architecture
  + Show the concurrency architecture
  + Show the distribution architecture
  + Show the deployment architecture
  + Show the safety architecture
  + Show the reliability architecture
  + Show the security architecture
* Show a design pattern

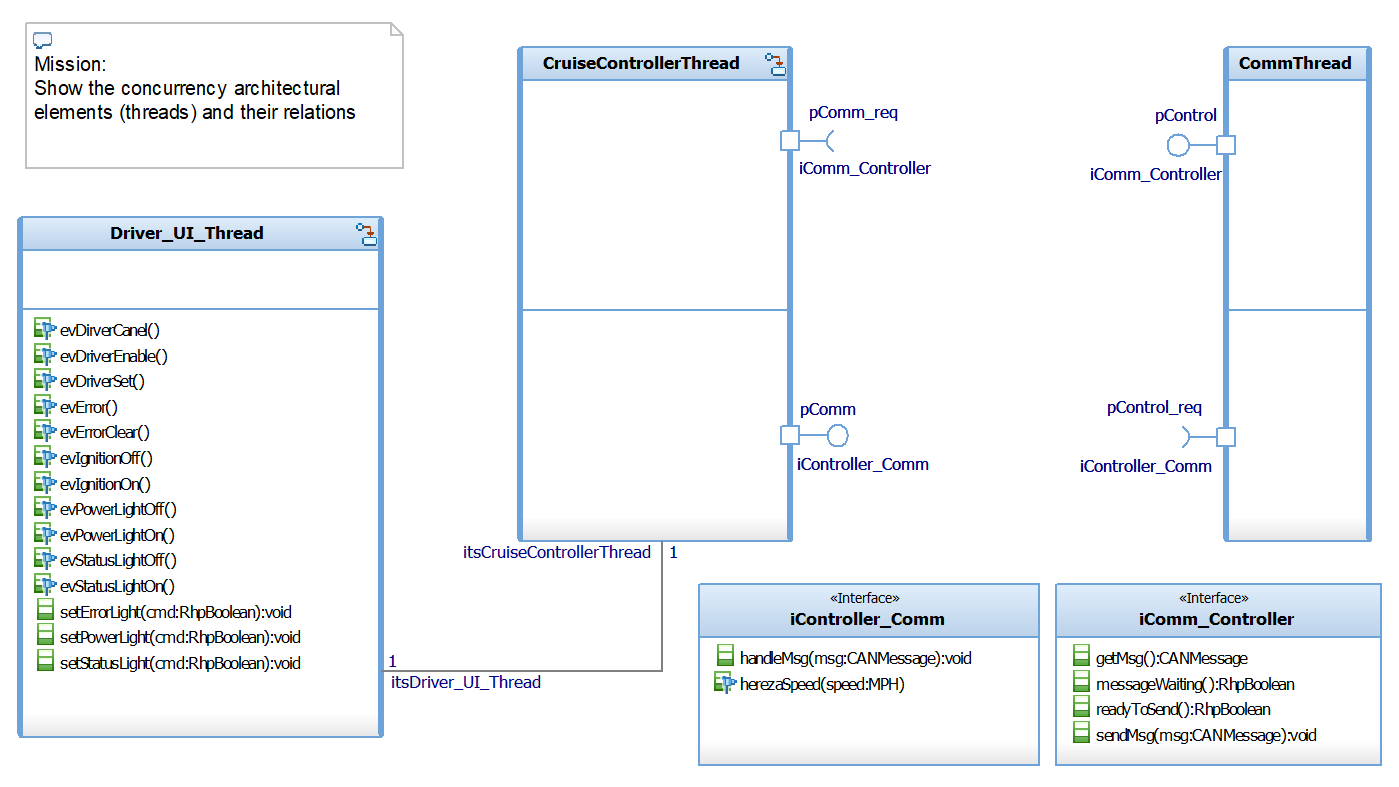


Figure : Example Class Diagram with Mission

* Very complex diagrams usually have too many missions or only a vague purpose

### Other diagramming conventions

* Every structural diagram shall have a comment explicitly describing the mission or purpose of that diagram
* Structural diagrams include
  + Class Diagram
  + Object Model Diagram (OMD)
  + Structure Diagram
* OMDs are the preferred structural diagrams as the others are simply constrained OMDs in Rhapsody.
* Every interaction diagram shall have a shall have a comment explicitly describing
  + The corresponding use case (if applicable)
  + The mission or purpose of that diagram
  + The pre- and post-conditions of the diagram
* Interaction diagrams include
  + Sequence diagram (most common)
  + Timing diagram
* Functional diagrams include:
  + Use case diagrams
  + Requirements diagrams
* Minimize line crossing. To avoid line crossing:
  + Use diagram connectors. These are elements for showing line connections when it is inconvenient to draw the line between the two elements
  + Replicate a class or diagram element. It is perfectly permissible to show a class or type multiple times on a diagram. Do this when it minimizes diagrammatic complexity.
  + Use multiple diagrams
* As much as possible, inputs shall come from the left or top and output shall leave at the right or bottom
* Use similar line styles (straight, rectilinear or curved) for semantically similar lines (e.g. rounded rectilinear for associations control and object flows, and state transitions, straight for generalization and dependencies).
* Shapes should be similarly sized except for containment hierarchies (i.e. structured classes may be larger than their contained parts)
* Avoiding using color to show semantics
  + Color may be used for emphasis or to improve readability
* Add constraints to show special or custom semantics or user-defined semantic rules
* Create a Model Overview Diagram (usage of Package Diagram or OMD and stereotyped with «Model Overview Diagram» from the *project-specific profile*) as a project-level (i.e. not nested within a package) that contains
  + Model organization description
  + Package structure shown as packages
  + Provide hyperlinks on the model overview diagram to important model views such as
    - High-level use cases
    - Type Context
    - Connected Context
    - Composition Architecture
    - Connected Architecture
    - Design Collaboration

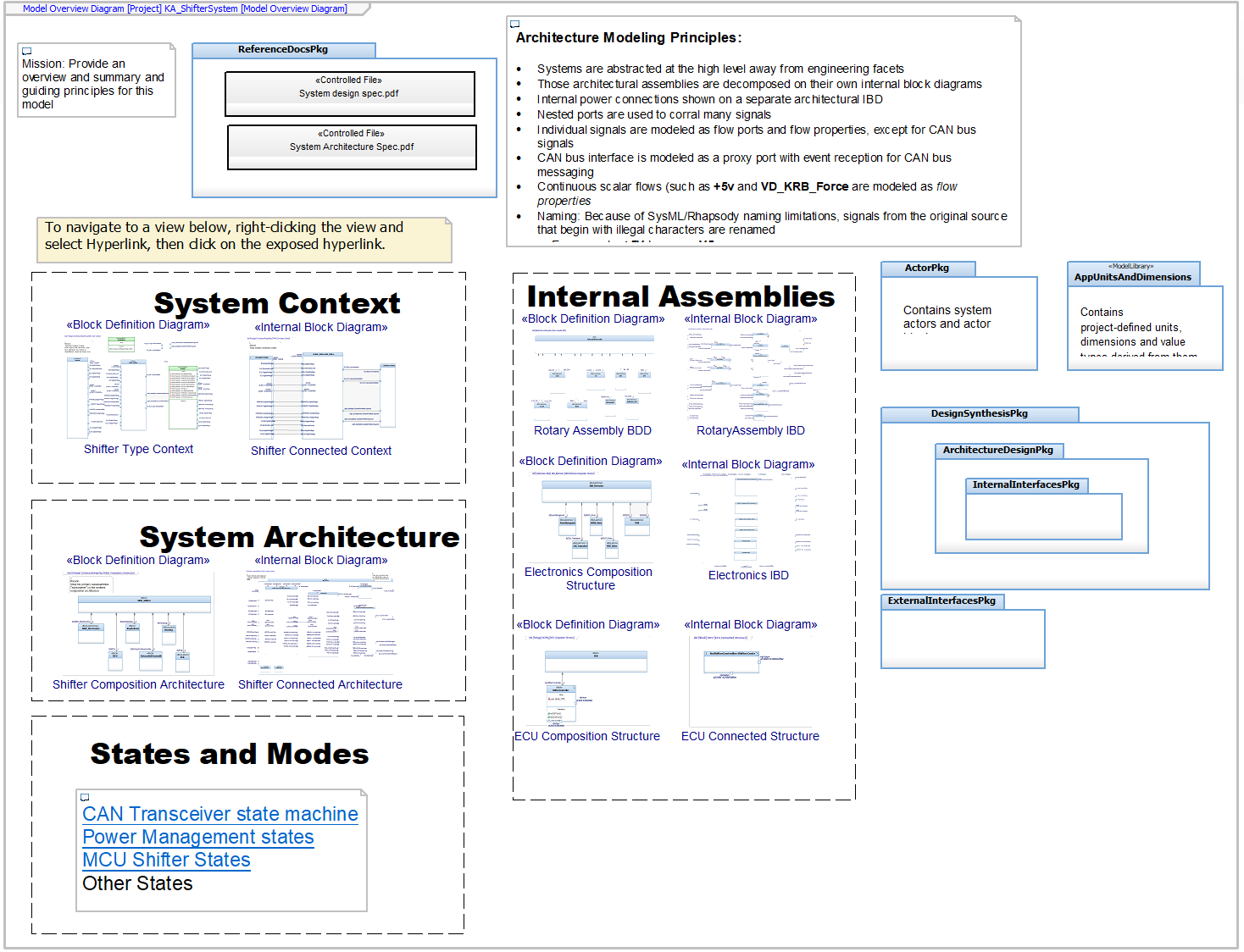


Figure : Example Project Overview diagram

## Object Model Diagrams (OMDs)

OMDs and Class diagrams are the same except that OMDs may also show object (instances). This section applies to both diagram types with that exception.

### Content

OMDs are generally limited to

* Class
* Types
* Class properties
  + Attributes
  + Operations
  + Event Receptions
  + Ports
  + Nested parts (instances)
* Interfaces
* Class relations:
  + Association, Aggregation, Composition
  + Generalization
  + Dependency (including stereotypes of dependency)
* Comments
* Constraints
* Requirements

In Rhapsody, OMDs may also contain parts and connectors and may be used instead of structure diagrams, if desired.

### Classes and Objects

* Classes, objects, and attributes shall have names that are noun or noun phrases.
* Classes, objects, and attributes shall have *singular* nouns for names
  + Plurality is handled with multiplicity of the element
* Operations shall be named with a strong verbs or verb phrases
* Only show class features if they contribute to the diagram mission.
* Features shall have *protected* visibility unless their access is directly required by clients

### Interfaces

May be use to represent explicit interaction contracts between classes, whether or not ports are used.

### Interfaces and Ports

* Standard UML ports may be used but are not mandatory
* Ports are typed by Interfaces. Interfaces shall not have implementation or attributes.
* Interfaces shall not define or reveal design or implementation of the class realizing the interface.
* Ports shall be named in terms of the client role that connects to it or the semantic content of the data or services that traverse it, as shown in Figure 11.

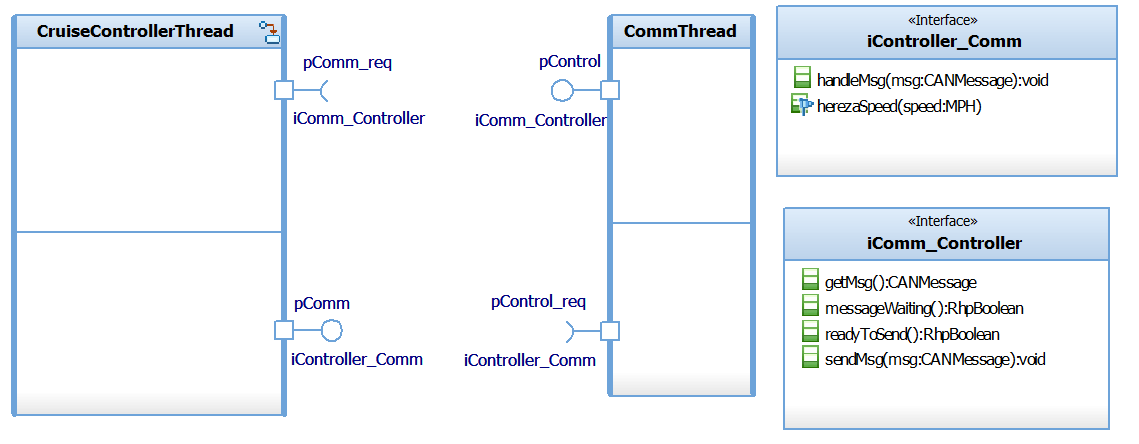


Figure : Interfaces and Ports Example

* Do not use bidirectional contracts on UML Ports – that is, ports that contain both offered and required interfaces. The port connections from Figure 11 should not be represented as in Figure 12. This is because of a defect identified in Rhapsody 8.4 and 9.0.



Figure 12: Disallowed Interfaces and Ports Example

### Relations

* The use of ports is an *alternative* to the use of associations:
  + Ports can only be connected between instance or instance roles but they are specified on classes.
  + Associations are relations between class; *links* are instances of associations
* Always show multiplicity on associations
  + Exception: you need not show the multiplicity on the “whole” end of a composition, as this is always “1”
* Use role ends names on associations
  + Show responsibilities via good role end naming
  + Role end names shall specify a usage that one or more instances at that role end fulfill at run-time
  + Role end names are always more important that association names
* Always show role names on associations
  + When there are multiple associations between a pair of classes
  + Exception: only show role end names for navigable ends of the association; that is, for a directed association, only show the role end of the navigable end (the end with the arrow).
* Use a single association with non-unary multiplicity when all the objects play the same role; use multiple associations to the same class when the objects play different roles
  + Figure 13 shows two different compositions to MessageQueue because the roles of the two are distinct (one is for input, the other for output)
  + The figure shows a \* multiplicity for Message, because each message fulfills the same role with respect to the MessageQueue



Figure : Relations Example

* Role end names (see Figure 13) shall indicate the role that an instance of one class plays for the other, e.g.
* Most associations should be unidirectional
  + Most associations model a client-server relation with the client being able to navigate to the server, but not vice versa.
  + Early models may use bidirectional associations when the direction of message flow isn’t clear but later models should resolve most of these into unidirectional associations
* Use rectilinear lines for association (including aggregation and composition) and straight lines for generalization and dependency
* Minimize use of dependency relation
  + For requirements traceability, use the «trace» dependency (if done in the model rather than in a separate requirements traceability tool)

## Structure Diagrams

Structure diagrams are used to show how objects (instances) connect via *links*, with or without ports. OMDs may be used instead of UML Structure Diagrams, if desired.

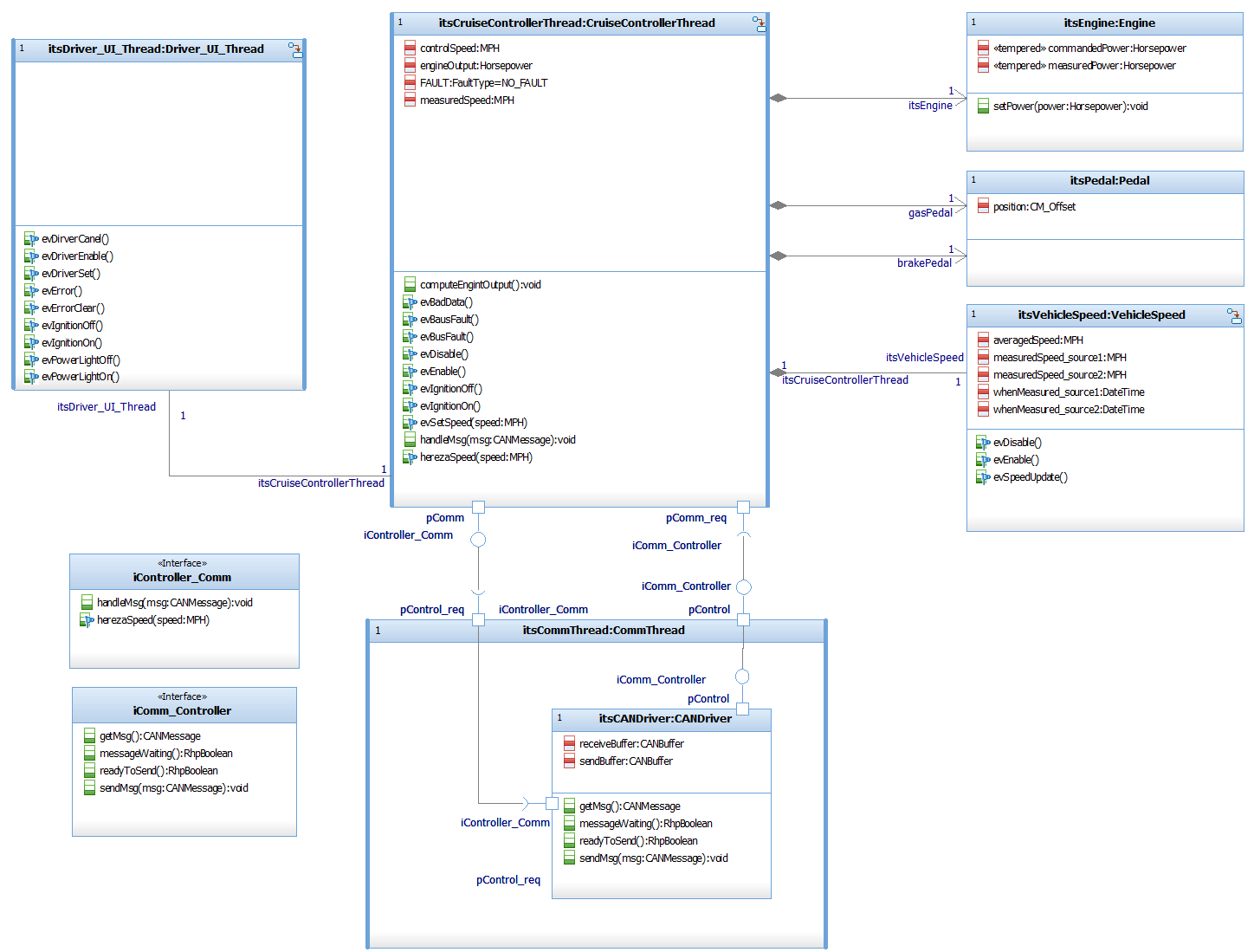


Figure : Example Structure Diagram

* Structure diagrams are generally owned by a composite class but may be owned by a package
* Ports are linked via *links*
* Connected ports must be interface-compatible (what is offered on one side is required on the other).

## Use Case Diagrams

### Use Cases

* Adhere to general diagram guidelines, above
* Model an operational capability as a use case
* For each use case, specify in the description (see Figure 15)
  + Use case name
  + Purpose or goal
  + Description
  + Preconditions – invariants that must be true before the use case is started
  + Post-conditions – invariants the system guarantees when the use case completes
  + Invariants – assumptions or expectations that must be true throughout the execution of the use case behavior

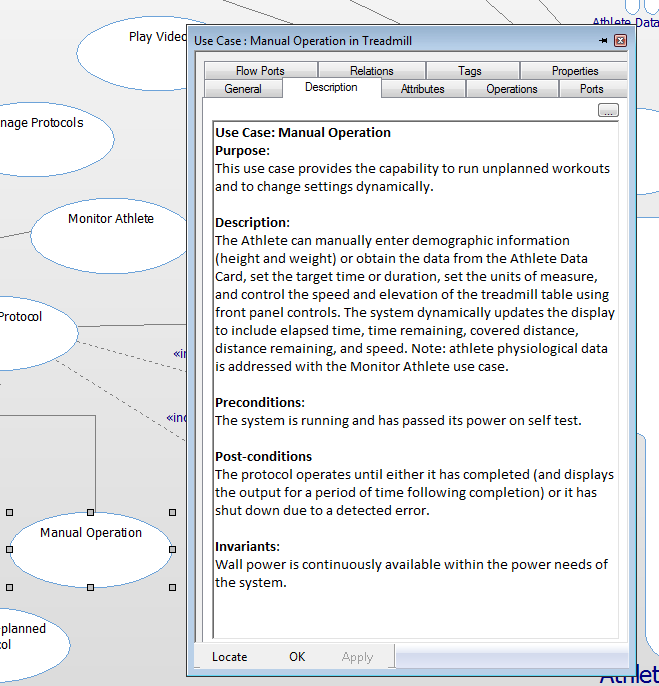


Figure : Use Case Description

* Use cases shall be named with a strong verb or verb phrase
* Use cases shall be independent in terms of the requirements as much as reasonable (but needn’t be in terms of their design realization)
  + Example: “Configure Sensor” and “Acquire Data” are independent in terms of their requirements, but in the implementation, the configuration settings are used in the acquisition of the data
* If use cases are tightly coupled in terms of operational requirements, join them into a single use case
  + E.g., “Manage Pedestrian Traffic” and “Manage Vehicular Traffic” are tightly coupled; join these into a “Manage Traffic” use case
* Constrain a use case only when that constraint applies to all scenarios of the use case
* All use cases shall interact at least one actor
* Define a use case with both scenarios (sequence diagrams) and specifications (activity or state diagrams)
* Use packages and/or to manage many use cases, organized by some criterion, such as
  + Common actors
  + Common purpose
  + Common area of concern
  + Common specification team

### Actors

* Actors are objects outside the scope of the system that interact with the system when it executes the use case
* Actors shall be given singular noun names from the problem domain
* If it’s “in the box” then it’s not an actor
  + If it connects to or interacts with your system at the customer site, then it is an actor
* Avoid identifying technology as actors; the actors should be the problem domain element of interest, not the means by which the actor connects to your system
  + The actor might be “Hospital Information System” not the “Network Interface Card” that connects the HIS to your system
* Don’t model interaction among actors
  + Since you’re not building the actor, their interaction is unimportant in your system design – focus on what you’re trying to build
* NEVER model “Time” as an actor; a use case can initiate behavior on its own with internal timeout events

### Relations

* Use directed associations only when message flow is unidirectional
* Subclass actors when the specialized actor participates in special relations or additional use cases over the base actor
* Use generalization to indicate specialized forms when technology realizations add distinct requirements
  + E.g. “Identify User” use case holds requirements common to all its specialized forms, but “Identify via Fingerprint Scan” adds some unique requirements over “Identify via Password”
* Use «include» to map system-level use cases to subsystem-level use cases
* Use «include» to encapsulate capabilities that are used in multiple (larger) use cases
* «include» arrowhead shall point to the “part” use case, not the “whole”
* Use «extend» infrequently
* Use «extend» for optional functionality that can be inserted at a specific extension point
* «extend» arrowhead points toward the “whole” use case, not the “part”
* Use «trace» to show the relation between a use case and a requirement allocated to it

## Requirements Diagrams

Requirements diagrams are not a UML standard view, however, OMDs or Class diagrams may be used for this purpose.

Requirements will be managed primarily in DOORS. The use of requirements diagrams and relations are limited to adding relations from model element to requirements. Requirements diagrams and tables may be used to visualize requirements in the model, if desired.

### Relations

* Dependency relations («trace» and «satisfy») go FROM the source element (use case, class, or other design element) TO the requirement
* Use «trace» for use case – requirement relation
* Use «satisfy» or «allocate» for design element – requirement relation

## Requirements Tables

Requirements tables are generally the preferred way to work with sets of requirements in Rhapsody.

The *project-specific profile* provides a predefined **Requirements Table Layout**. An example of its use is shown in Figure 16.

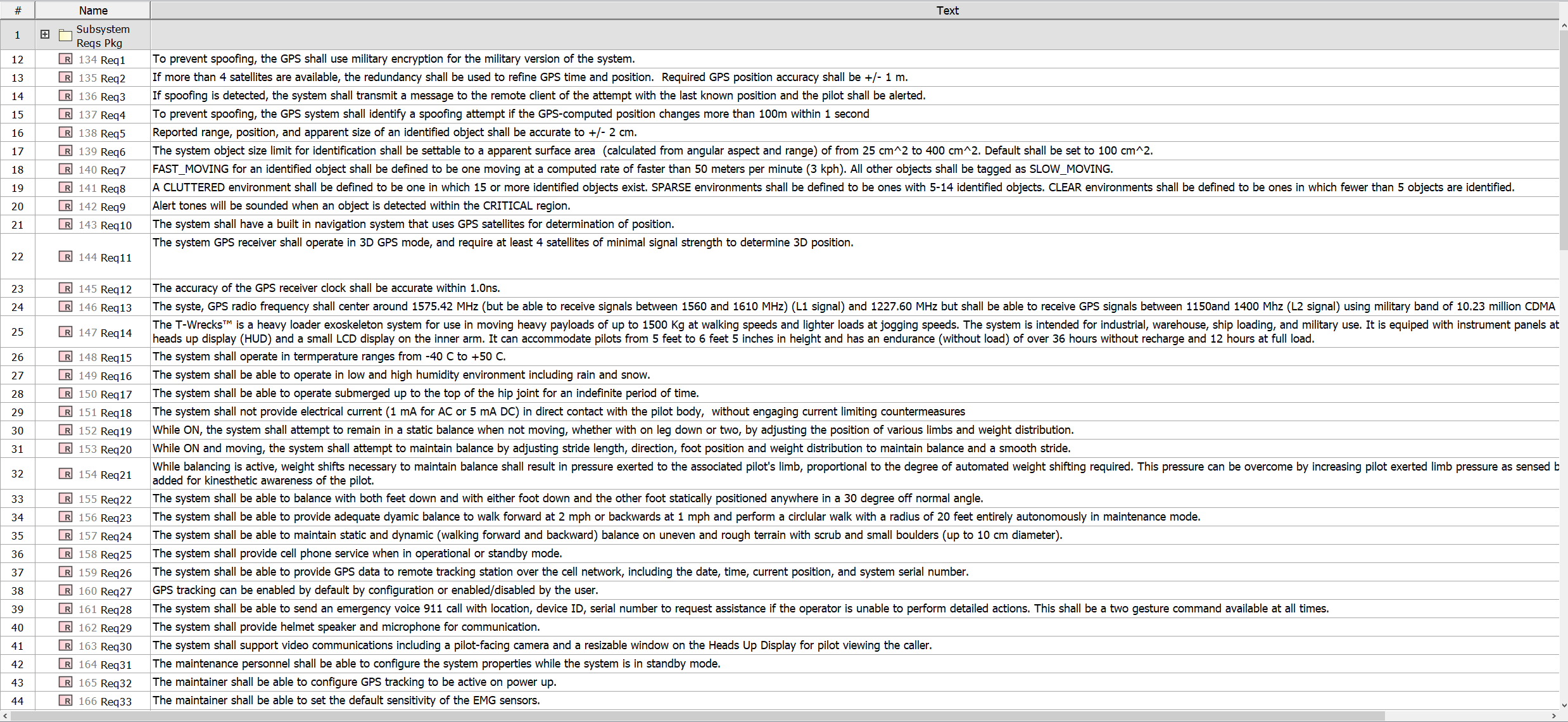


Figure : Requirements Table

## Matrices

Matrices show the relations between sets of elements. Common uses for matrices include:

* Stakeholder – System Requirements
* Requirements – Use Cases
* Requirements – Design Elements
* Other matrix can be easily developed as needed.

## Sequence Diagrams

* For a set of sequence diagrams around a common context – such as a use case or design collaboration – it is preferrable to have the lifelines appear in the same order, even if a lifeline is not used in a particular sequence diagram. This is to facilitate comparison among the sequences.
* As much as possible, arrange the lifelines to make messages go from left to right.
* Use horizontal lines for synchronous messages
* Use horizontal or slanted lines for asynchronous messages
* Use execution occurrences (“activation bars”) shall be avoided.
* Add comments to the sequence to describe why steps are being taken and to describe parallel activities not shown in the messages
* To shorten long scenario, wrap up sets of related messages into a referenced interaction fragment, as shown in Figure 17.

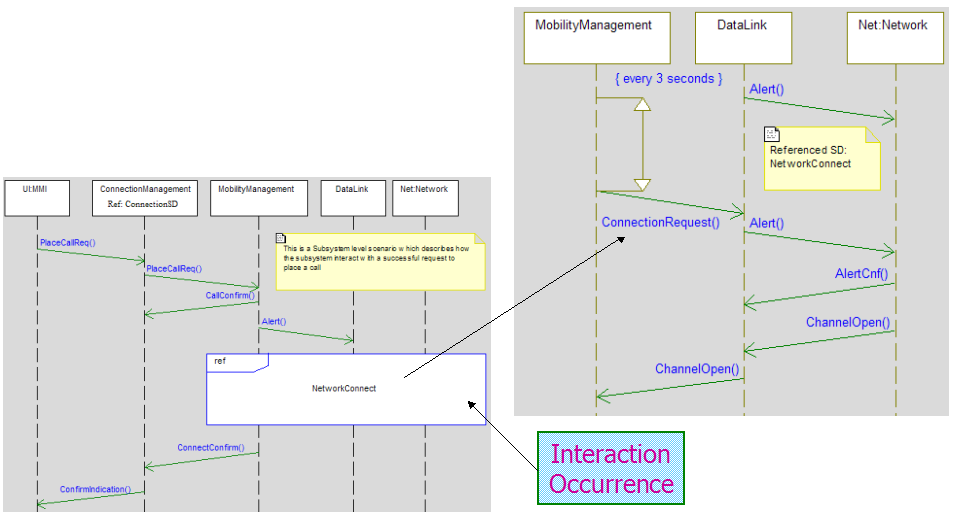


Figure : Interaction Fragment

* To reuse sets of messages, use a referenced interaction fragment
* Name messages the same as the operations or events they represent or invoke
* For special semantics use relevant interaction fragment operators, e.g.
  + Loop
  + Parallel
  + Opt (optional) for “if”
  + Alt (alternative) for “select case”

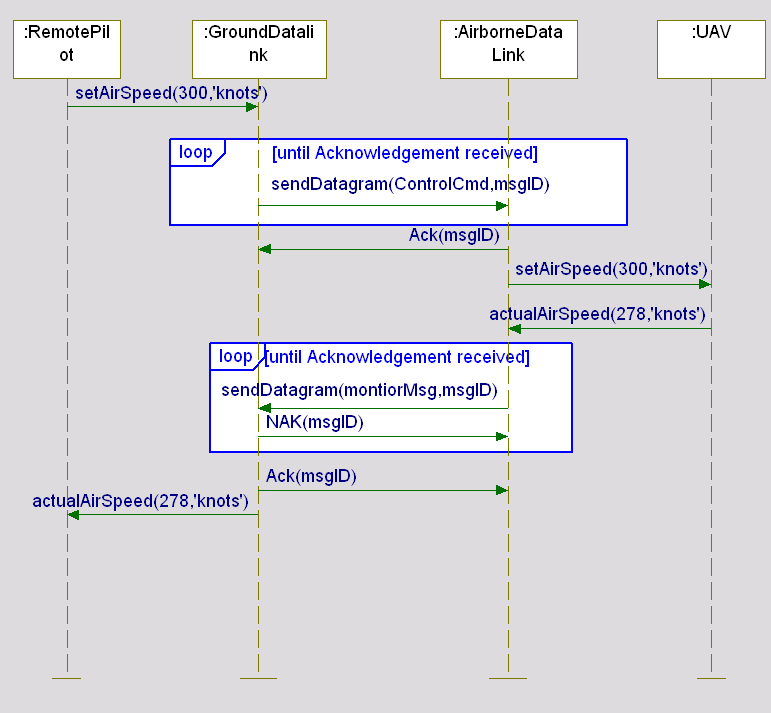


Figure : Loop Operator

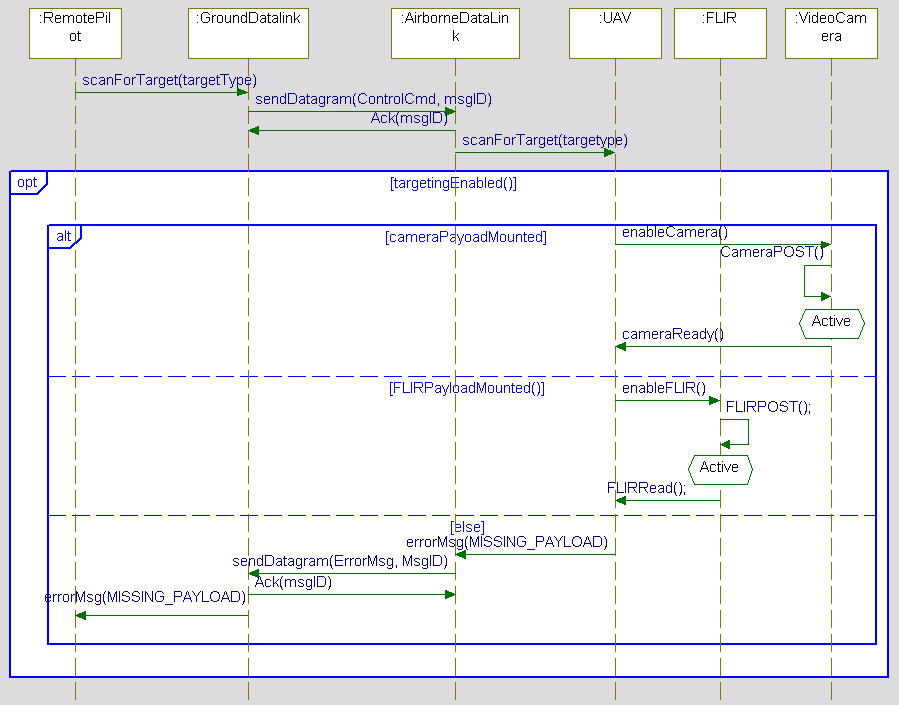


Figure : Opt and Alt Operators

* Avoid nesting interaction operations more than three levels deep.

## Activity Diagrams

* Use activity diagrams to model behavior that is flow (rather than event-) based such as algorithms or when modeling behavior that is continuous in time or value such as continuous pressure on a pedal
* Model algorithmic behavior (flow of control) with activity diagrams
  + Operations can be specified with activity diagrams
* Always indicate starting activity or action state
* Primitive actions shall be specified in the target action language (C or C++)
* You may optionally use forks and joins to model “logical concurrency” in activity diagrams but this will not create task threads (use active classes for that).
* Optionally, swim lanes may be used to allocate activities to classifiers
* Use guards only on transitions exiting a condition or branch point
* Complexity in activity diagrams is managed by decomposition
  + Use *Call Behavior* actions to invoke behavior on a separate activity diagram
  + Use *Call Operation* actions to invoke behavior defined by a function or operation
  + Decompose the operation into multiple operation calls
    - Each operation can have its *own* activity diagram
    - Note: Not all operations must have a flow chart or activity diagram
* Activity diagrams and state machine should generally be mixed with the State Machine “on top”

## Flow Charts

Flow charts in Rhapsody are constrained activity diagrams that may be created only for operations; they lack forks, joins, and call behaviors. They are optimized to provide code for single-threaded operations.

* Flow charts may be used for operations, especially when
  + The code is intended for deployment
  + The operation is complex and it is desirable to document its design

## State Diagrams

* State diagrams should be used preferentially to specify reactive (i.e. “reacts to events”) behavior for classes and use cases, including classes at different levels of abstraction (e.g. systems and subsystems)
* Use a State Machine to model behavior of classifiers when that behavior
  + is event driven or
  + is modal, i.e. the behavior differs depending on state
* Always, at every level of nesting, indicate the default state
* State diagrams may be either asynchronous (asynchronous event-driven) or synchronous (call-driven) but not usually both
  + Great care must be taken with state machines with both synchronous and asynchronous triggers to avoid race conditions

### States

* Use names for states that come from the problem vocabulary (domain)
  + Exception: In some cases, a state may be added to explicit force the completion of a run-to-completion step using the [*Forced Closure Rule*](#_Definitions_and_Acronyms). Such states will have names outside of the problem domain because they exist solely to resolve a design technical issue.
* Use composite states when one transition exiting the composite applies to all nested states or when the composite state logically contains the nested states (e.g. note the evEOS event from ParsingNumber state in Figure 20).



Figure : Nested States

* Use submachines to simplify complex state machines such as shown in the following sets of figures. Figure 21 shows a complex State Machine with 3 levels of abstraction before it is decomposed into nested submachines. Figure 22 through Figure 24 show exactly the same State Machine decomposed into a set of layered submachines.



Figure : Complex State Machine before decomposition into submachines

This complex state machine contains a composite state Parsing Expression, which in turn contains a composite state Parsing Term. This can be decomposed into three layers:



Figure : State Machine decomposed into submachines – level 0



Figure : State Machine decomposed into submachines – level 1



Figure : State Machine decomposed into submachines – level 2

* Pay attention to action placement
  + Add actions to state entry only when the actions should be executed whenever the state is entered regardless of which transition is taken
  + Add actions to state exit only when the actions should be executed whenever the state is exited regardless of which transition is taken
  + Add actions to transitions when the above conditions are not met
* In the presence of and-states, avoid race conditions
  + Race conditions are defined to be when a computational result depends on a specific order of execution, but that order is not knowable
  + Race conditions occur when the same event is processed in simultaneously active and-states and
    - Incompatible target states are specified, or
    - Actions on the transitions manipulate the same attributes, or
    - Incompatible actions are executed
  + See Figure 25 for an example of an avoidable race condition

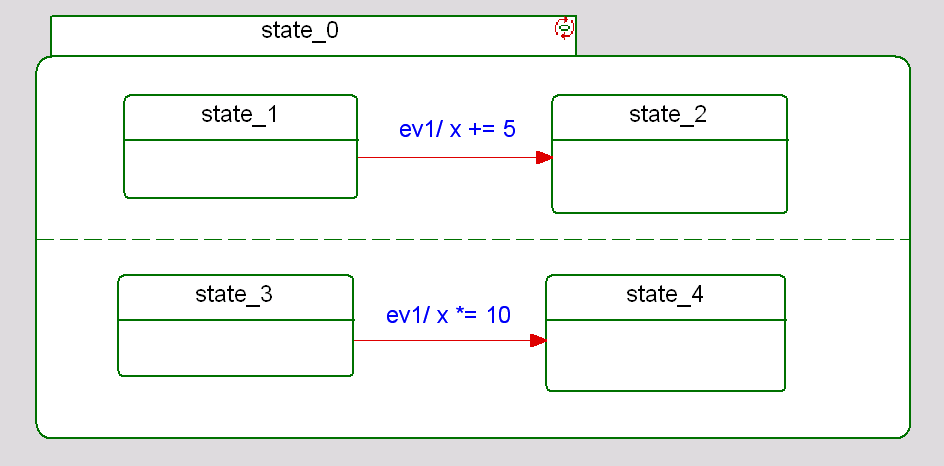


Figure : Race Condition in And-States

### Actions

* Actions are run-to-completion, therefore actions should generally have a short execution time
* Actions may be direct attribute manipulations, operations defined on the class, or operations defined on classes to which the current class has an association
* Complex actions shall be modeled as operations; those operations can then be specified using activity diagrams
  + E.g rather than “/x = foo(z); y = sin(x)^2 – tan(x); z = hsin(x + y);”, wrap the set of actions in to an operation and invoke it “/computeZ();”
* Simple actions may be direct action language statements to manipulate attributes of the object
  + E.g. ev1/ x = sin(y) + cos(z);

### Guards

* Guards shall not have side-effects
  + In C, C++, or Java, “x = 0” shall not be used as a guard; “x == 0” would be better
* Guards on transition segments exiting conditional connectors shall have non-overlapping conditions
  + For example, “[x>0]” and “[x>10]” would not be good guards from the same conditional connector as if x==20, both guards would evaluate to TRUE
* Don’t use the result of actions in guards on the same transition
  + Guards are evaluated prior to the execution of actions
  + See [Forced Closure rule](#_Definitions_and_Acronyms)
* Use an [else] guard when the event triggering the transition must always be handled

### Submachines

A *submachine* is a set of nested states that are placed in a separate diagram to decrease the visual complexity of a state machine.

* Remember that submachines are “syntactic sugar” only – the submachine is still logically a part of the containing state machine; submachine merely aid in visualization of complex state machines
* Use submachines to manage complexity
  + when a composite state is part of a complex state machine, that composite state can be decomposed on a separate state diagram (submachine)
  + recommended when there are few, if any, non-default transitions
* Use exit and entry points only when non-default transitions are used
  + If there are more than a very small number of non-default transitions, it is recommended *not* to decompose the nested state into a submachine
* See State Guidelines and Figure 22 through Figure 24, above.

# Semantic Standards: Using SysML

## Model Overview Diagram

* Each model shall have a Model Overview diagram located “above” the packages.
* This shall be either a Package or OMD diagram and the stereotype *«Model Overview Diagram»* from the *project-specific profile* shall be applied to the diagram
* This diagram should contain information about the content and use of the model.

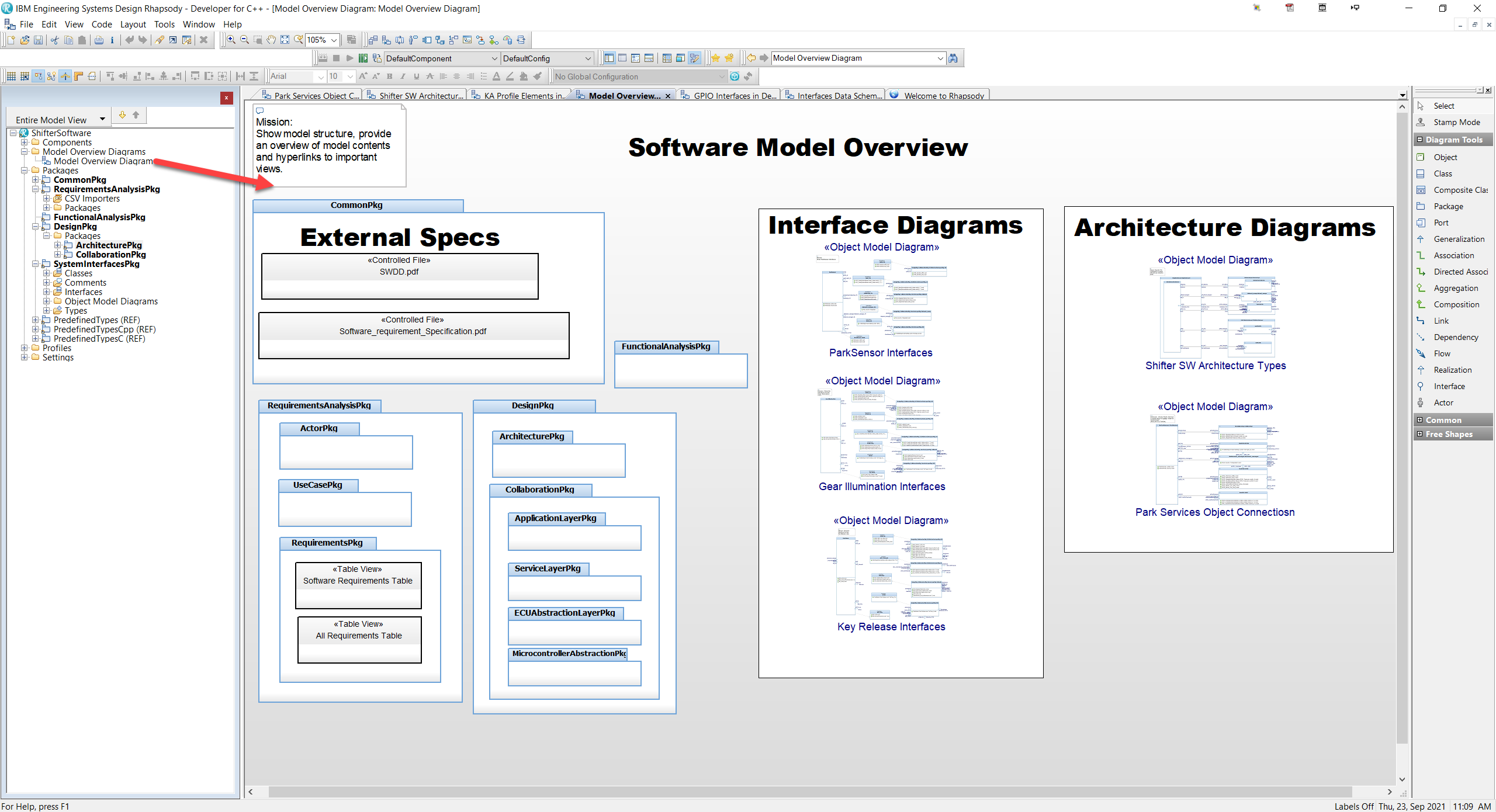


Figure : Model Overview Diagram

Note the location of the diagram in the browser on the left of Figure 26.



Figure : Another Model Overview Diagram

## Modeling Software Architecture Context

* The system context shows the system as a Class, the actors with which it communicates, and the interfaces.
* Use an OMD for the type context
* Use an OMD or Structure Diagram for the connected context

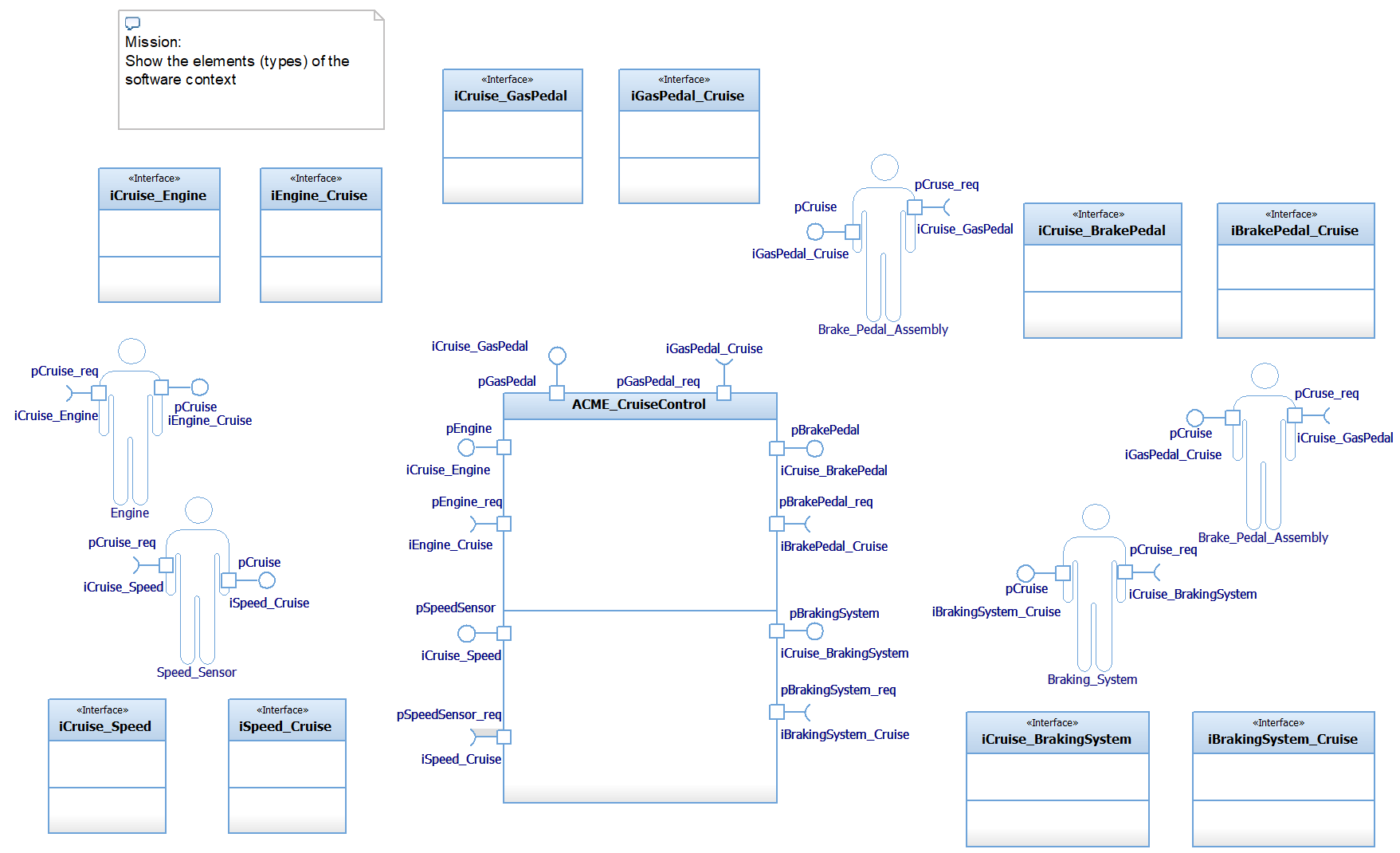


Figure : Type Context

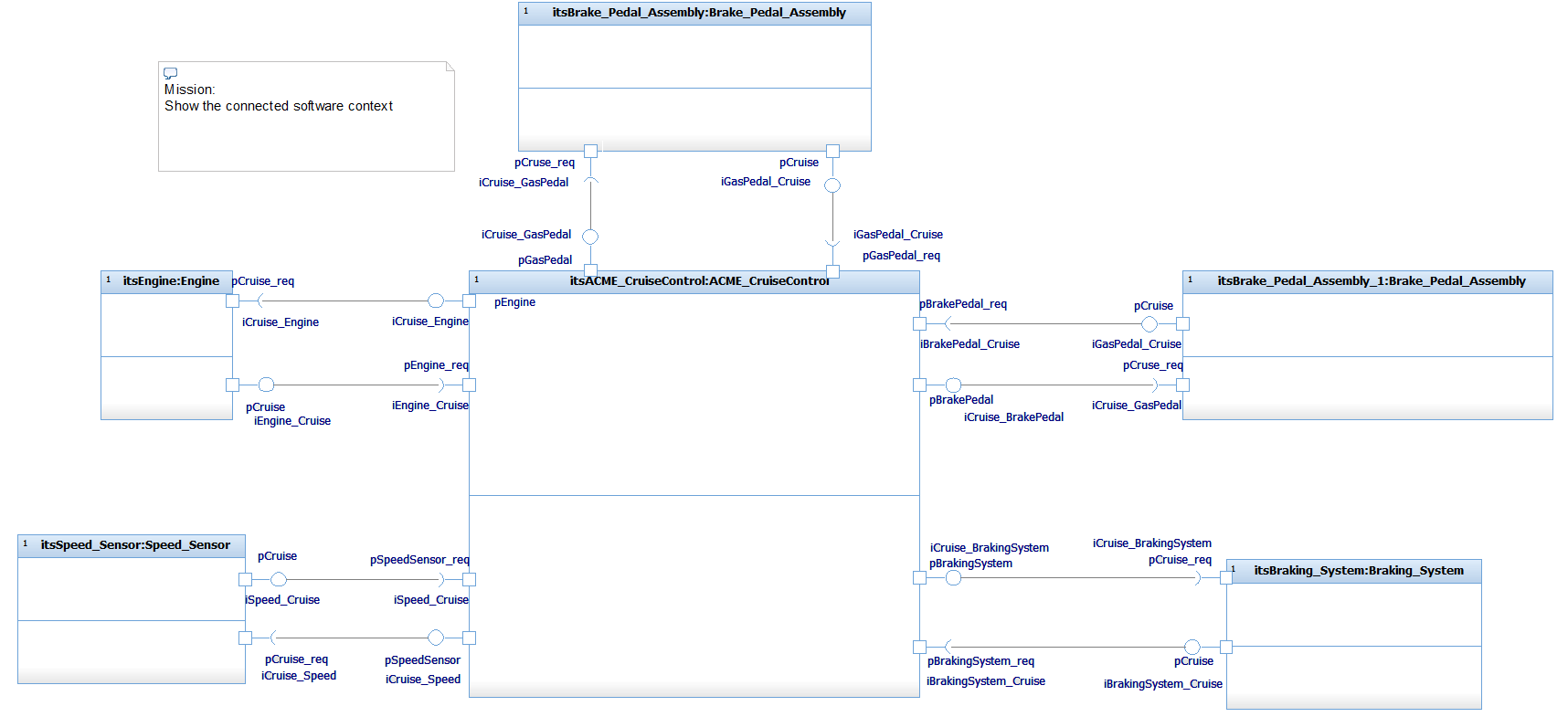


Figure : Connected Context

Modeling Systems Architecture Structure

* Use OMD to show the system composition architecture
  + The term “composition architecture” identifies the system and subsystems and show their relations.
* Use OMD or Structure Diagram to show the connected architecture
  + The connected architecture shows how the subsystems and related elements connect.

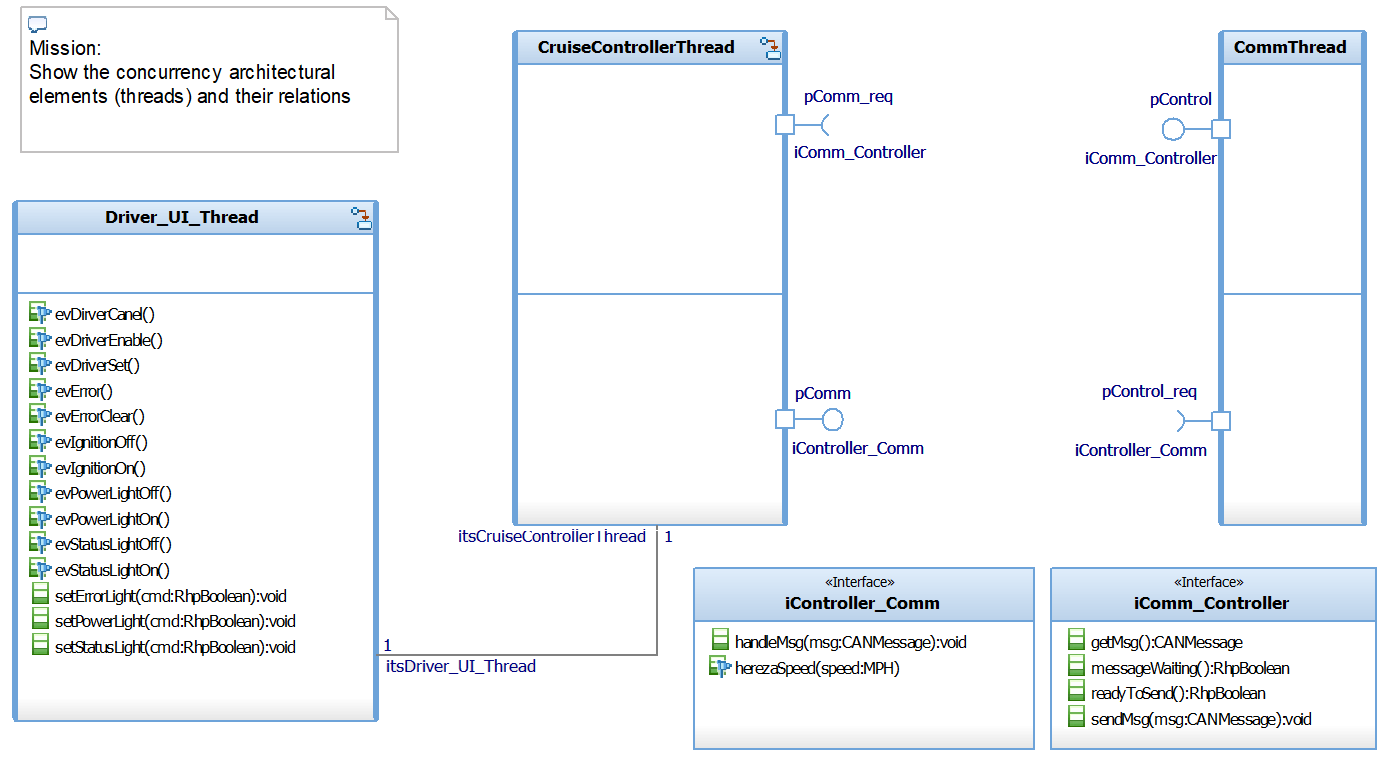


Figure : Software Concurrency Architecture

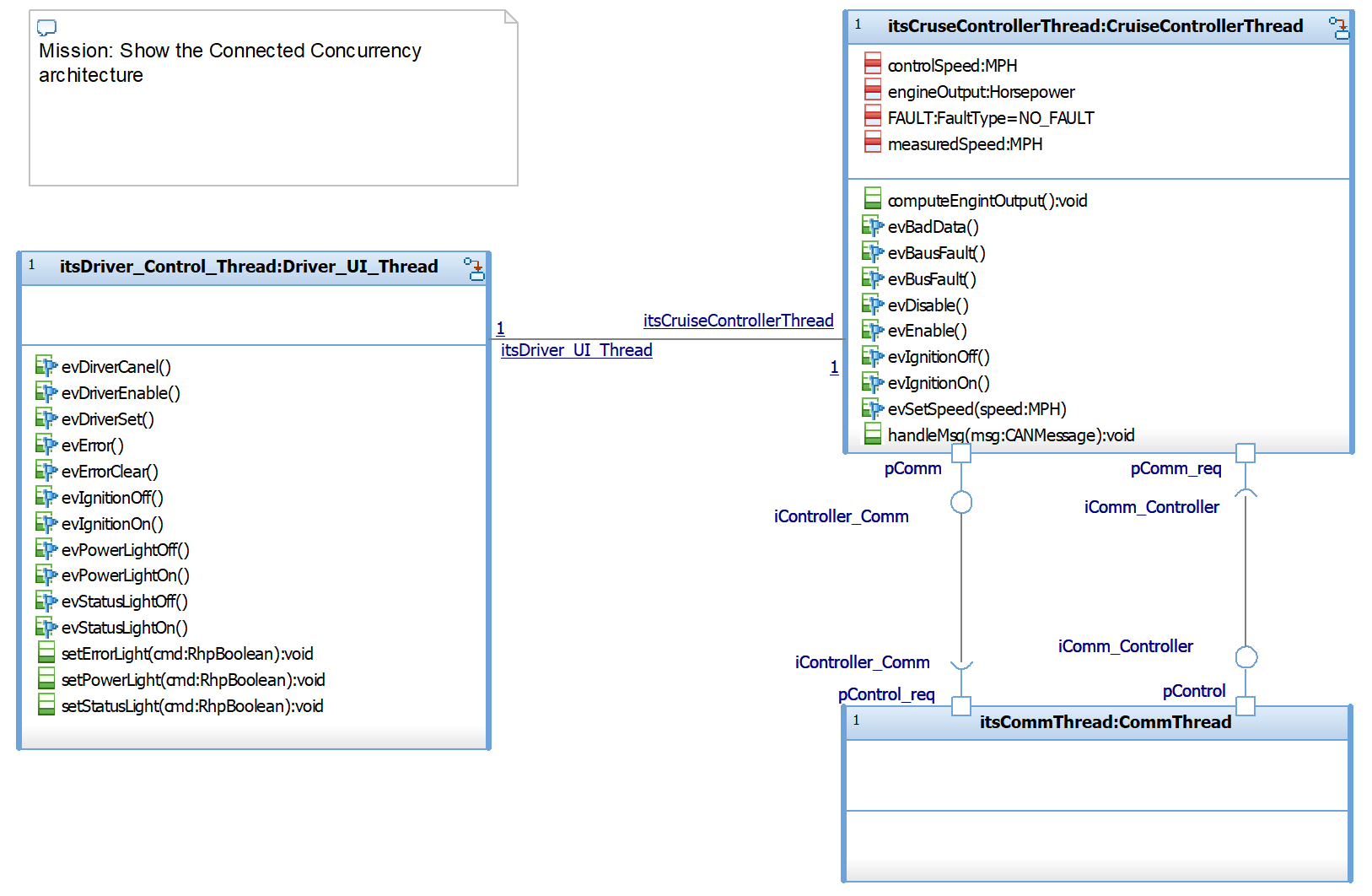


Figure : Connected Concurrency Architecture

Modeling Concurrency Architecture

* «active» classes will be used to represent OS threads.

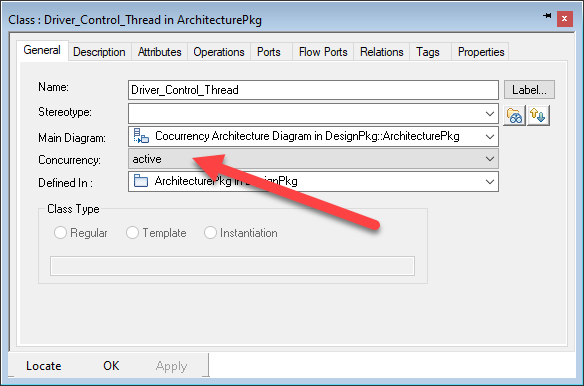


Figure : Active Classes are for threads

* Guarded property in features will be used to mark operations that are non-interruptible and should use semaphore to enforce that semantic

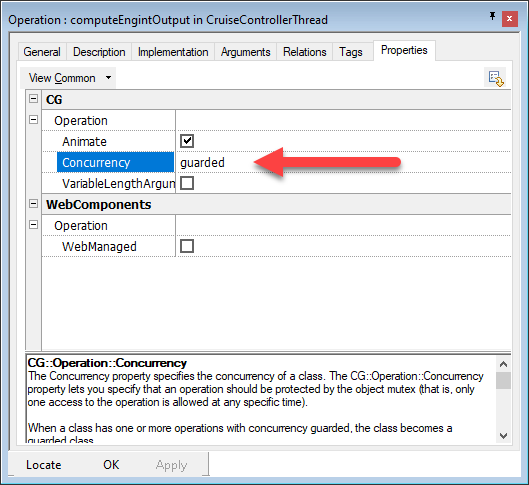


Figure : Guarded Calls invoke a semaphore

## Modeling Systems Architecture Behavior

Prefer state machines for behavioral specification of architectural elements.

### Modeling Data Schema

* Use OMDs to show the relations of classes, types, and units.
* Define and apply *units* preferentially over the use of primitive types
  + For example, define a unit KiloPascal and use it instead of just a double or Real.
* Use named constants rather than *magic numbers*

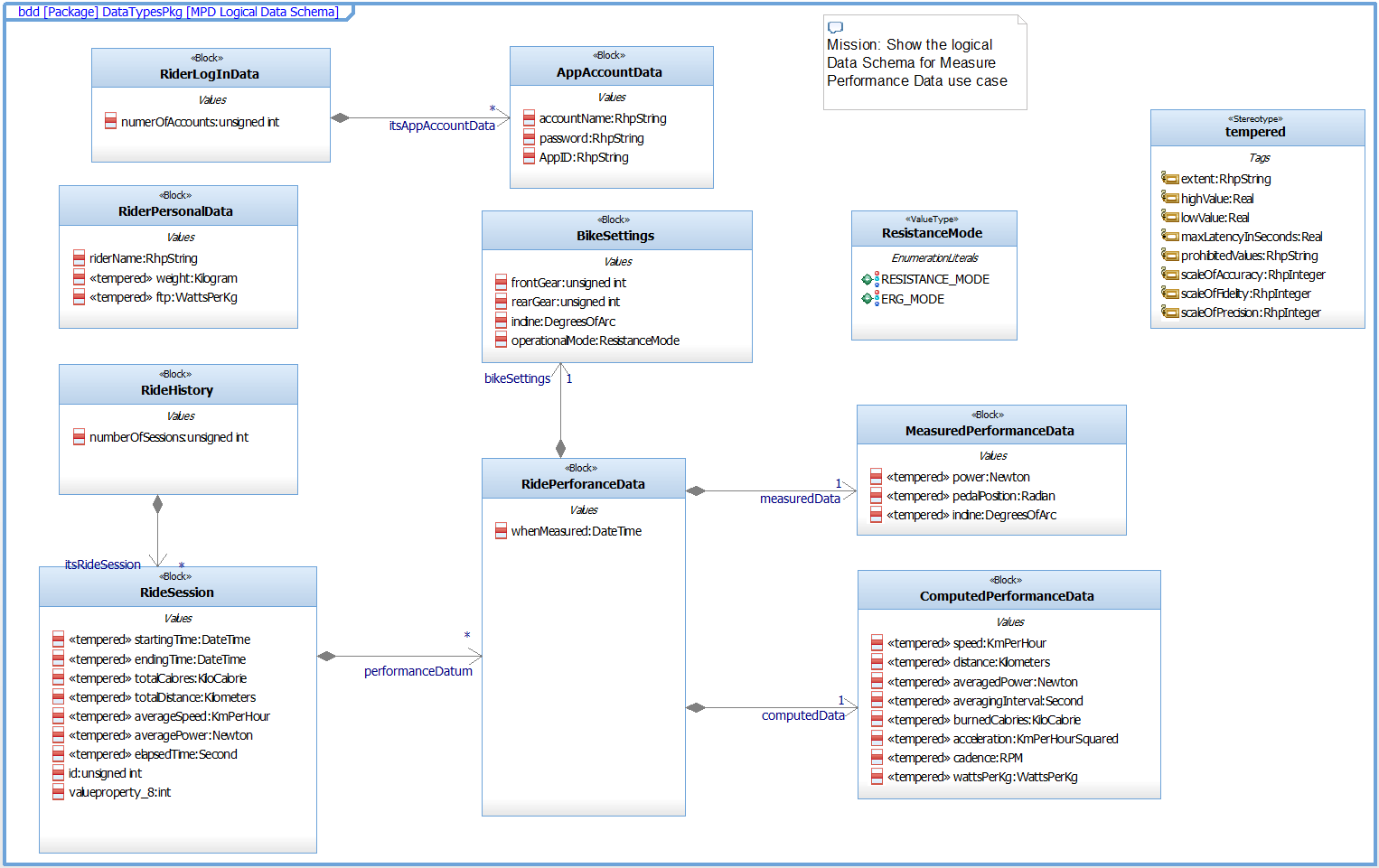


Figure : Data Schema

### Defining Units

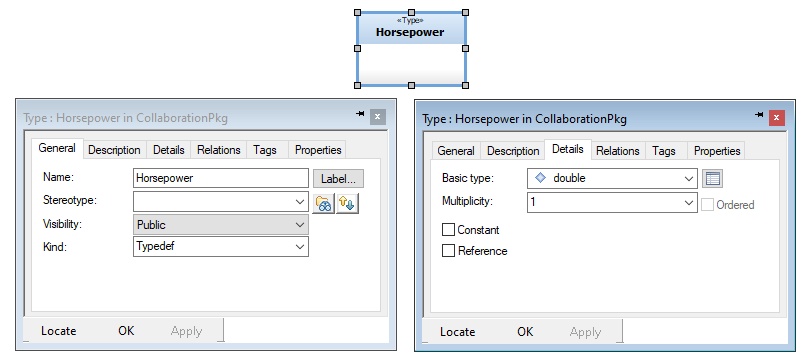


Figure 35: Definition Units as Typedef’d Primitives

### Defining Named Constants

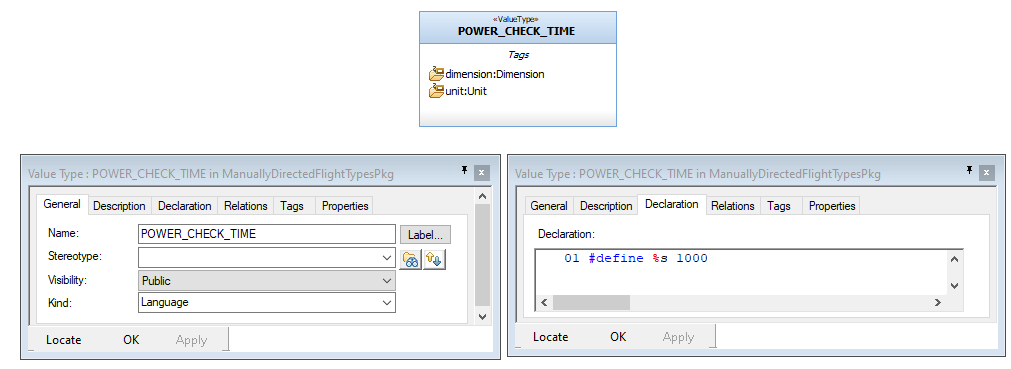


Figure : Defining a named Constant

## Relating to ASPICE and ISO 26262

To support ASPICE, the following content and views – at minimum – will be contained within a systems model or model federation:

* Model Overview Diagram
* System Context Type
* System Connected Context
* System Composition Architecture
* System Connected Architecture
* Requirements-Architectural design elements trace matrix
* System class state behavior identifying modes/primary system states and their transition
* State machines for the subsystems and indications of how they map to system states
* Activity and/or state diagrams depicting dynamic system behavior
* Sequence diagrams showing specific interactions (scenarios) of the architectural elements that is consistent with the state and activity models
* Interface diagram and/or tables showing the interfaces and their properties (event receptions and functions invoked over the connections)
* Data schema
* Deployment architecture (OMD and/or Structure Diagrams) with interdisciplinary interfaces)

The appropriate stereotypes from the project-specific profile will be applied to hold ASPICE and ISO 26262 model relevant metadata such as

* Capacity/memory needs
* ASIL Level
* Internally vs externally developed software
* Likelihood (such as for a fault)
* Severity (such as for a fault)
* Risk (such as for a fault)
* Fault Tolerance Time (such as for a fault)
* Worst case execution time (such as for a behavior)

## Generating Reports

Reports may be generated to for SE Architecture models using provided report templates or by creating custom ones.

# References

**Automotive SPICE: Process Reference Model** Version 3.1 (November 2017)

**ISO 26262-6:2018: Road Vehicles – Functional Safety – Part 6: Product development at the software level** (2018)

**Agile Model-Based Systems Engineering Cookbook** by Dr. Bruce Powel Douglass, (2021)

**Agile Systems Engineering** by Bruce Powel Douglass, (2016)

**Real-Time Agility** by Bruce Powel Douglass, (2009)

**Real-Time Agile Systems and Software Development** website [www.bruce-douglass.com](http://www.bruce-douglass.com)