Harmony MBSE Modeling Standards for use with UML, SysML, and Rhapsody

Prepared by
Dr. Bruce Powel Douglass, Ph.D.
Chief Evangelist
IBM Rational
# Table of Contents

1.0 SCOPE ........................................................................................................................................... 4

2.0 REFERENCED DOCUMENTS ........................................................................................................ 4

3.0 MODELING STANDARDS ............................................................................................................. 4
   3.1 Naming Conventions .................................................................................................................... 4
   3.2 Modeling Guidelines .................................................................................................................... 8
   3.3 Model Reviews ............................................................................................................................ 13
   3.4 General Diagram Guidelines ..................................................................................................... 14
   3.5 Use Case Diagram Guidelines .................................................................................................. 17
   3.6 Class Diagram Guidelines ........................................................................................................ 25
   3.7 Sequence Diagram Guidelines .................................................................................................. 33
   3.8 State Diagram Guidelines ......................................................................................................... 36
   3.9 Activity Diagram Guidelines ..................................................................................................... 45
   3.10 Deployment Diagram Guidelines ............................................................................................. Error! Bookmark not defined.
   3.11 UML Extension Guidelines ..................................................................................................... Error! Bookmark not defined.

APPENDIX A – RHAPSODY ACTION LANGUAGE ................................................................................. 46

BASIC SYNTAX ..................................................................................................................................... 46

ASSIGNMENT AND ARITHMETIC OPERATORS .................................................................................. 47

PRINTING ............................................................................................................................................... 47

COMPARISON OPERATORS .................................................................................................................. 47

CONDITIONAL STATEMENTS ................................................................................................................ 48

INCREMENTAL LOOPING STATEMENTS ............................................................................................... 48

CONDITIONAL LOOPING STATEMENTS ............................................................................................... 48

INVOKING OPERATIONS ....................................................................................................................... 49

GENERATING EVENTS .......................................................................................................................... 49

REFERRING TO EVENT PARAMETERS IN TRANSITIONS .................................................................. 49

TESTING THE PORT ON WHICH AN EVENT ARRIVES ...................................................................... 49

TESTING THE STATE OF A STATE MACHINE ...................................................................................... 50
Figures

Figure 1: Naming Blocks and Features ................................................................. 5
Figure 2: Naming on an Internal Block Diagram .................................................. 6
Figure 3: Naming States and Transitions .............................................................. 7
Figure 4: Naming Use Cases ............................................................................... 8
Figure 5: Domain Vocabulary ............................................................................ 8
Figure 6: Recommended Model Organization Set .................................................. 9
Figure 7: Recommend SE Model Organization .................................................... 10
Figure 8: Subsystem Model Organization ............................................................ 11
Figure 9: Shared Model Organization ................................................................ 11
Figure 10: Domains ........................................................................................... 12
Figure 12: Subsystem Model Organization ........................................................... 13
Figure 13: Example Class Diagram with Mission ............................................... 15
Figure 14: Example Project Overview diagram ................................................... 17
Figure 15: Use Case Description ....................................................................... 18
Figure 16: Scenario with description ................................................................... 20
Figure 17: Executable Use Cases Models as Objects ........................................... 23
Figure 18: Executable Use Case Statecharts ....................................................... 24
Figure 19: Executable Use Case Scenarios .......................................................... 25
Figure 20: Parameterized Classes ..................................................................... 27
Figure 21: Interfaces and Ports Example ............................................................. 28
Figure 22: Multiple Interface Implementations .................................................... 29
Figure 23: Reflexive Associations ..................................................................... 30
Figure 24: Relations Example ............................................................................ 30
Figure 25: Role Names Example ........................................................................ 31
Figure 26: Association Class ............................................................................. 31
Figure 27: Parameterization and Generalization ................................................ 32
Figure 28: Role Names Subclasses Example ....................................................... 33
Figure 29: Sequence Diagram Example .............................................................. 34
Figure 30: Interaction Fragment ....................................................................... 34
Figure 31: Loop Operator .................................................................................. 35
Figure 32: Opt and Alt Operators ..................................................................... 36
Figure 33: Nested States .................................................................................... 37
Figure 34: Complex statechart before decomposition into submachines .......... 38
Figure 35: Statechart decomposed into submachines – level 0 ......................... 38
Figure 36: Statechart decomposed into submachines – level 1 ......................... 39
Figure 37: Statechart decomposed into submachines – level 2 ......................... 40
Figure 38: Coordinating And-States ................................................................. 41
Figure 39: Use Case Statechart ....................................................................... 42
Figure 40: Forcing action closure .................................................................... 43
Figure 41: Race Condition in And-States ............................................................ 44
Figure 42: Mixing activity diagrams and state machines ................................. 46
1.0 SCOPE

This document provides the modeling guidelines to be developers using UML and Rhapsody to development systems and software. This document does not discuss coding standards.

2.0 REFERENCED DOCUMENTS


Douglass, Bruce Powel Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns (Addison-Wesley, 1999)


Douglass, Bruce Powel Real-Time UML Workshop (Elsevier Press, 2006)

Douglass, Bruce Powel Real-Time Agility (Addison-Wesley, 2009)

Douglass, Bruce Powel Design Patterns for Embedded Systems in C (Elsevier Press, 2010)

Douglass, Bruce Powel Agile Systems Engineering (Elsevier Press, 2015 (2016))

3.0 MODELING STANDARDS

As much as possible, standards will be enforced using a common properties file, macros, and explicit procedures.

3.1 Naming Conventions

3.1.1 Naming convention

Figure 1 through Figure 4 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. NDDS or hardware) domain
  - In general, no “default names” assigned by the tool are acceptable in a model under formal review
- Model element names will 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”
- Blocks, data types, value types and event names begin with an uppercase letter
- Block feature names will begin with a lower case letter. Block features include
  - Block values (attributes)
  - Block functions (operations)
  - Block event receptions
  - Block ports
  - Block association roles
  - Block parts
- Block instance (object) name will begin with a lower case letter.
• “Things” (e.g., objects, blocks, attributes) should be named with strong nouns or noun phrases
• “Actions” (e.g., operations, use cases, functions) should be named with strong verb or verb phrases
• Association role names shall refer to the usage or role an instance of the block plays with respect to the block at the other end of the association, e.g.
  o myDataSource
  o itsDataQueue
  o theCommController
• In multiple word names, make the first letter of every word (after the first) upper case
• Port names shall begin with a lower case “p” and be named either by
  o Their semantic content (e.g., pConfigData)
  o The actor with which they connect (e.g., pTracker)
  o As an alternative, the primary name may be suffixed with the word “port” instead of using the prefix “p”
• Interfaces shall be named in terms of their semantic concept (e.g., iNavData or iFLIRCommands) and their names shall be prefixed with a lower case ‘i’
• Interface blocks shall be named in terms of their semantic concept and their names shall be prefixed with a lower case “ib” (e.g., ibRadarCmd)
• Interface names shall begin with a lower-case “i” and follow standard identifier naming conventions

Figure 1: Naming Blocks and Features
Figure 2: Naming on an Internal Block Diagram
Figure 3: Naming States and Transitions
3.1.2 Domain Vocabulary
Where possible, use domain vocabulary to name elements as shown in Figure 5, e.g.
- Targeting domain: Target, combatID, Track
- Communication domain: Socket, Message
- Operating System (OS) domain: Thread, Task, Semaphore
- Hardware domain: Interrupt, Address, MemoryMappedDevice

3.2 Modeling Guidelines

3.2.1 Large models or projects should be split across multiple Rhapsody models
Basic large model organization should be:
- One systems requirements & architecture model as shown in Figure 6.
  - 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 7.
- One model per subsystem (Figure 8). 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 9)
When transitioning from systems engineering model to the software model, the SE model organization shall provide a package for the specification of each subsystem, as shown in Figure 6

- The SubsystemSpecification package in the SE model is organized as one package per subsystem
- 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 6: Recommended Model Organization Set**
Figure 7: Recommend SE Model Organization

Holds imported requirements, requirements tables, and the like.

Holds use cases, use case diagrams, and a package per use case for functional analysis with blocks (or classes), state machines, and scenarios.

Holds trade studies and proposed architectures for analysis.

Holds selected subsystem architecture, logical subsystem interfaces and types. One package for each subsystem specification.
Figure 8: Subsystem Model Organization

Figure 9: Shared Model Organization
3.2.2 Use Domains to organize common design elements

In Platform Independent Models (PIMs), blocks (henceforth known as “analysis blocks”) fall into two broad categories – reusable and non-reusable. In this section “blocks” shall refer to all structural elements, e.g. classes, objects, and types.

- Non-reusable analysis blocks and objects shall be specified in the package which owns the block diagram in which they appear
- Reusable analysis blocks and objects should be clustered into domains as shown in Figure 10.
  - A domain is a stereotype of package that contains elements within a single subject matter (e.g. communications, GUI, alarming, medical concepts, target tracking, navigation, generic types, etc).
  - Each reusable analysis block and type should be specified in a single domain
    - If a block appears to belong in multiple domains, break up the block into domain-specific concepts
      - e.g. “Alarm” into AlarmDomain::alarm and GUIDomain::TextDisplay
    - If a reusable block doesn’t appear to belong to any domain, then create a new domain for that block and concepts within its subject matter
  - Specialized blocks (i.e. blocks which subclass other blocks) which are reusable reside in the same domain as their parent block, in packages nested within that domain, or in specialized from that domain.
  - Specialized blocks which are not reusable shall be treated as non-reusable blocks, as specified above.

Figure 10: Domains
3.2.3 **Specify Design Objects, Blocks, and Types in the Subsystem Models**

- In this section “classes” shall refer to all structural elements, e.g. blocks, objects, and types.
- Design block are specified within either the subsystem model or, if they are to be shared, within specialized domains as shown in Figure 11
  - E.g. CommunicationsDomain can be specialized for a particular technology, such as 802.11g to create a Comm802-11gDomain. Blocks defined within the CommunicationsDomain are subclassed in the Comm802_11gDomain to add design details.

![Figure 11: Subsystem Model Organization](image)

3.3 **Model Reviews**

In an MBSE environment, models and associated views (diagrams) are the primary artifacts to be reviewed, not the source code.

3.3.1 **General guidelines for Reviews**

- Models should be reviewed in coherent subunits, rather than all at once, for example
  - Requirements Models, including related work products, for example, on a per use case basis, review related
    - Requirements
    - Trace table(s)
    - Use case diagram(s)
    - Use case
    - “Black box” sequence diagrams
    - Activity diagram
    - Use case execution context
    - State machine
    - Generated sequence diagrams
  - Architecture Model
    - Subsystem diagram (BDD or IBD)
    - Interfaces or Interface Blocks
    - Allocated requirements


- Allocated use cases
- “White box” sequence diagrams
  - Subsystem Specification Review (per subsystem)
    - Allocated requirements
    - Subsystem use cases
    - Trace table(s)
    - Use case diagram(s)
    - Use case
    - Activity diagram
    - Use case execution context
    - State machine
    - Generated sequence diagrams

- Model Review
  - Harmony Review Workflow (based on Fagan Inspection protocol)
    - Determine objectives and purpose for review
    - Prepare materials for review
    - Disseminate materials to reviews 5-14 days prior to review
    - Schedule review
    - Reviewers individually review material and create issues/questions
    - During meeting, address raised issues and questions
    - Any unresolved issues result in work items
    - Subsequent reviews only review changed items and work items from previous reviews
    - See Perform Review task in Harmony aMBSE workflow
  - Models will be reviewed for
    - Achievement of model objectives
    - Correctness, accuracy, completeness and level of fidelity
    - Adherence to the modeling guidelines (this document) – this is performed by QA personnel
    - Low complexity

3.4 General Diagram Guidelines

3.4.1 Every diagram should have a singular mission

Every diagram should have a single important concept that it is trying to show as shown in Figure 12. This is called its mission. This is especially important for block, activity, parametric, and sequence diagrams (less important for state diagrams due to their “built-in” mission).

A diagram should 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 should be stated on the diagram. Some common block 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

![Example Class Diagram with Mission](image)

**Figure 12: Example Class Diagram with Mission**

**3.4.2 Show only the elements that contribute to the mission on the diagram**

- Very complex diagrams usually have too many missions or only a vague purpose
- Large diagrams can be decomposed into multiple diagrams in two primary ways
  - Grouping (common purpose)
  - Abstraction level
- Add comments to your diagrams (note this may (optionally) be in the diagram description instead if preferred)
  - Comments should address WHY; let the diagram itself address WHAT
  - Every structural diagram shall have a comment explicitly describing the mission or purpose of that diagram
  - Structural diagrams include
    - Block Definition Diagram (BDD)
    - Internal Block Diagram (IBD)
    - Deployment Architecture Diagram
  - Every interaction diagram 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
    - Communication diagram (UML)
    - Timing diagram (UML)
  - Every functional diagram shall have a commend explicitly describing
3.4.3 Create user-defined model navigation links with hyperlinks

- Create a Model Overview Diagram (BDD) as a project-level (i.e. not nested within a package) that contains:
  - Model organization description
  - Package structure shown as packages
  - Model view paths with hyperlinks based on purpose of the view, e.g.
    - Requirements
    - Architecture
    - PIMs
    - PSMs
    - Design
    - Performance Data
    - Data Definitions
    - Tests

  *Hyperflows* are trees or acyclic graphics constructed from sets of hyperlinks. Create hyperflows to support user navigation through the model.

- Every time you construct a diagram, add links into that diagram that make sense and add links from elements of this diagram to subsequent diagrams that make sense:
  - There should be very few diagrams that are not the targets or sources of hyperlinks or do not contain elements that are the sources of hyperlinks.

- Add hyperlinks to related views and diagrams *as you construct the diagram* not after the fact.

- Hyperlinks to *controlled files* (including the DoDAF generated products) can be added by simply dragging the product to the diagram. Double clicking on that artifact will open the product in its native application.

- Add hyperlinks to diagrams:
  - Create a comment and enter text. Select appropriate text, right click, and select Hyperlink. From there you can either select an external file or an internal model element as the target for the hyperlink.
  - Primarily use diagrams and controlled files as the targets for hyperlinks.

- Add hyperlinks to elements.
3.5 Use Case Diagram Guidelines

3.5.1 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 14)
  - 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
- Quality of Service (QoS) constraints, such as reliability, safety, worst case execution time, performance, etc. when needed for your system

Figure 14: Use Case Description

- A use case is not a single requirement; it is a container for a (possibly large) set of requirements
  - Note: A typical use case might be 10-100 requirements but this varies widely depending on the project, level of abstraction, and complexity of the problem
  - Example: “Turn On” would not be a use case if it contained a single message “On” to the system from an actor, but it might be if there was a complex interaction involved in turning on the system. In the former case, the single message would be part of a more complex use case that might have, as its first activity, turning the system on.
- Use cases should be named with a strong verb or verb phrase
- Use cases should be independent in terms of the requirements (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
- Most, but not all, use cases are initiated by an actor
  - Use cases can have autonomous and time-based behavior
• All use cases should interact at least one actor
• Define a use case with both scenarios and specifications
• Use packages and/or to manage many use cases, organized by some criterion, such as
  o Common actors
  o Common purpose
  o Common area of concern
  o Common specification team

3.5.2 Scenarios
A scenario is a specific sequencing of messages between the actors the system while executing the use case. Scenarios are invaluable for both requirements capture and validation (via walking through the desired behavior with the customer) and for verification (via execution of the system, automatic construction of the resulting sequence diagram, and automatic comparison with the requirements sequence diagram).

• Use sequence diagrams to capture scenarios
  o For system-level use cases, only actors and the system (or use case) may be shown as lifelines, not internal elements
  o For subsystem-level use cases, actors, the system, and the subsystems may be shown as life lines
  o System- and subsystem-level views of the same scenarios should be linked with “vertical decomposition” between the system-level lifeline and the subsystem-level scenario
• Use “horizontal decomposition” of scenarios (via referenced interaction fragments) when a group of messages is to be reused or when it simplifies the overall model complexity
• For each use case scenario, specify all additional pre- and post-conditions above and beyond those for the use case
  o e.g. “Precondition: This scenario starts with a received Associated Measurement Report (AMR) and assumes the target is visible to sensors.”
• Focus first on “sunny day” scenarios; later identify “rainy day” scenarios
  o Sunny day scenarios are those that depict normal or typical operations
  o Rainy day scenarios are those that depict unusual, error, or exceptional scenarios
• Create fault classifications for rainy day scenarios
  o This means for a given fault classification, create a single (or small set) of scenarios, rather than one scenario for each potential fault in the fault class
  o A fault class is a set of faults that are treated identically from the system perspective
    ▪ E.g. “Gas supply failure”, “Gas pipeline leak”, and “Gas pipeline obstruction” may be all treated identically and therefore would form a fault class “Gas Supply Fault”
  o This is to avoid “scenario explosion” as there are almost always orders of magnitude more fault scenarios than “sunny day” scenarios
  o Specify the set of faults within the fault classification either as a class, enumerated type, and/or or in a constraint
  o See Figure 16 for an example of fault classification
Figure 15: Scenario with description
Figure 16: Fault Classification in Scenarios

- A use case scenario is a path within a specification (use case) statechart or activity diagram
- Create a minimal spanning scenario set (MSSS). This is the set of scenarios such that every requirement is represented in at least one scenario, and every action and event on the use case state machine is represented in at least one scenario

3.5.3 Specifications

- Use text to explain “why”; use a formal language (i.e. statecharts or activity diagram) to define “what” or “how”
- Representing formal behavioral specifications
  - Use a statechart especially for discrete-time and event-driven use case behavior
  - Use an activity diagram to specify behavior that is primarily flow-based
  - Use an activity diagram to specify behavior that is continuous in time or value
  - The formal specification (i.e. statechart or activity diagram) must represent every possible scenario of that use case

3.5.4 Actors

- Actors are objects outside the scope of the system that interact with the system when it executes the use case
- Actors should 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
  o 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

3.5.5 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
  o 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 should point to the “part” use case, not the “whole”
• Use <<extends>> infrequently
• Use <<extends>> for optional functionality that can be inserted at a specific extension point
• <<extends>> arrowhead points toward the “whole” use case, not the “part”
• Use <<allocate>> to show the relation between a use case and a requirement allocated to it

3.5.6 Executable Use Cases
Executable use cases are done mechanistically by representing the actors and the use case as block instances connected with ports, typed by interfaces, such as shown in Figure 17. This has some technical advantages in terms of interface specification, ease of execution, and specifying the operational contracts among elements.

• Use executable use cases when
  o The cost of requirements errors is high
  o The requirements are vague, imprecise, ambiguous
  o The requirements are complex
  o The requirements must be well-validated
  o The requirements or their effect must be demonstrated to stakeholders
• Use one port on the “use case object” per actor
• Ports shall explicitly be typed by their interfaces
  o “Rapid ports” (i.e. uptyped ports) may be used early on, but before the model is formally reviewed, ports should be explicitly typed
Figure 17: Executable Use Cases Models as Block Instances

- The behavior of the executable use cases shall be represented as statecharts, such as Figure 18.
- The interaction of the executable use cases with the actors shall be depicted on sequence diagrams, as shown in Figure 19.
Figure 18: Executable Use Case Statecharts
3.6 Block Diagram Guidelines

3.6.1 Block and Objects

- Blocks, objects, and attributes should have names that are noun or noun phrases.
  - Analysis elements should be named using domain vocabulary
- Blocks, objects, and attributes should have *singular* nouns for names
  - Plurality is handled with multiplicity of the element
- Operations and event receptions should be named with a strong verb
• Only show block features if they contribute to the diagram mission. These include
  o Operations
  o Events
  o Attributes (values)
  o Relations
  o Interfaces
  o Ports
  o Constraints
• Features should have protected visibility unless their access is directly required by clients
  o Attributes should always have protected visibility; their mutators and accessors should be
    public only when direct access to the attributes is required by a client
• Use friend dependency infrequently
  o Friend dependencies are particularly useful for “testing buddy” classes
• Use usage dependency to show a relation when an instance of a specified block is passed as a
  parameter but not otherwise the object of an association
• Use bind dependency to show the relation between a parameterized class (one with unbound formal
  parameters) and an instantiated parameterized class (an instantiable class with actual parameters
  bound to the formal parameters) – see Figure 20\(^1\).

---

\(^1\) In this case, the instantiated parameters are shown in a “free form box” because Rhapsody doesn’t depict
that aspect although it does depict the formal parameters (dashed box).
3.6.2 Interfaces and Ports

- Ports are a design pattern (see *Real-Time Design Patterns* by Dr. Bruce Powel Douglass) and therefore have both costs and benefits
  - Benefit: ports enforce object encapsulation and reuse of classes
  - Cost: Ports are objects and have memory and performance impact
- Ports are best used between large-scale classes (e.g. systems, subsystems, and components) but not, as a general rule, between parts inside a structured class
  - If the architecture is many levels deep, this rule may apply only at the bottom or “leaf” level of abstraction
- Ports should always be defined in terms of the contract that they support
Offered or provided interfaces specify synchronous operations or asynchronous events provided or handled by the element owning the port.

In general, in SE models, asynchronous communications are preferred in logical interfaces.

Required interfaces specify synchronous operations or asynchronous events required by the element owning the port

- Ports should 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 21.

**Figure 21: Interfaces and Ports Example**

- Interfaces should not define or reveal design or implementation of the block realizing the interface.
- Always use an interface if there is more than one design or implementation used to meet a responsibility, as shown in Figure 22.
- QoS constraints on services in interfaces must narrow in subclassed interfaces to preserve Liskov Substitution Principle, i.e. a block realizing the subclassed interface may be substituted for a block realizing the base interface without violating the constraints.
3.6.3 Relations

- The use of ports is an alternative to the use of associations:
  - Ports can only be connected between instance or instance roles
  - Associations are relations between blocks; links (connectors) are instances of associations
- Whenever the instances of two blocks must exchange a message, an association may be used
  - Exception: when an object is passed as a parameter to an operation, don’t draw an association to the class of that parameter, but use a usage dependency to the class of the parameter
- 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 should specify a usage that one or more instances at that role end fulfill at run-time
  - Role end names are almost always more important that association names
  - Don’t show both role end names and association names, unless the association name clearly aids understanding
- Always show role names on associations
  - When there are multiple associations between a pair of classes
  - When the association is to and from the same class (see Figure 23).
Figure 23: Reflexive Associations

- 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 24 shows two different aggregations 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 24: Relations Example

- Role end names (see Figure 25) should 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 association classes when the association is rich, either in information or behavior, e.g.
    - Socket
    - Marriage

If the association class must participate in relations, then use a (regular) block instead of an association class, and draw associations from it to the two other participant classes
  - Note: This is the most common design realization of association blocks anyway

Use rectilinear lines for association (including aggregation and composition) and straight lines for generalization
- Use composition only when you want to enforce creation and destruction semantics, e.g. the whole is responsible for creation and destruction of the part.
- Composition is a relationship between blocks (i.e. specifications); a structured block with parts is a relation between a block and object roles that specify it (i.e. block and part).
- Minimize use of dependency relation:
  - When used, use stereotypes to classify the dependency (e.g. bind, friend, etc)
  - For requirements traceability, use the "trace" dependency (if done in the model rather than in a separate requirements traceability tool).
- Use generalization only when the specialized block extends and/or specializes the base block.
- Don't use generalization to indicate object roles:
  - "Up button" and "Down button" are not different subclasses – difference is in usage not in specification.
  - Indicate role either as an association end role or as a part name inside a structured class.
- Use generalization and parameterization properly:
  - Use generalization when you want the subclass to work on exactly the same data types but you want the behaviors on that data to work differently.
  - Use parameterization ("templates" in C++ or "generics" in Ada) when you want exactly the same behavior but want it to operate on different data types.

![Figure 27: Parameterization and Generalization](image)

- Graphically position subclasses and bound parameterized blocks below their base elements as shown in Figure 28.
3.7 Sequence Diagram Guidelines

- As much as possible, arrange the lifelines to make messages go from left to right, as shown in Figure 29.
- Use horizontal lines for synchronous messages and slanted lines for asynchronous.
- Use execution occurrences (“activation bars”) sparingly and only for synchronously invoked messages.
- Add comments to the sequence to describe why steps are being taken and to describe parallel activities not shown in the messages.
- To show scenarios at different levels of abstraction, use a structured class on the abstract scenario, decompose the lifeline on a nested diagram to show how parts of the structured class interact.

Figure 28: Role Names Subclasses Example
Figure 29: Sequence Diagram Example

- To shorten long scenario, wrap up sets of related messages into a referenced interaction fragment, as shown in Figure 30.

Figure 30: 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 31: Loop Operator
3.8 State Diagram Guidelines

- State diagrams should be used to specify reactive (i.e. “reacts to events”) behavior for blocks and use cases, including blocks at different levels of abstraction (e.g. systems and subsystems)
- Use a statechart 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

3.8.1 States

- Use names for states that come from the problem vocabulary (domain)
- 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 33).
Figure 33: Nested States

- Use submachines to simplify complex state machines such as shown in the following sets of figures. Figure 34 shows a complex statechart with 3 levels of abstraction before it is decomposed into nested submachines. Figure 35 through Figure 37 show exactly the same statechart decomposed into a set of layered submachines.
Figure 34: Complex statechart before decomposition into submachines

Figure 35: Statechart decomposed into submachines – level 0
Figure 36: Statechart decomposed into submachines – level 1
Use and-states when the order of execution of actions between the and-states is irrelevant.

When and-states must synchronize, use one or more of the four primary methods:

- **Broadcast events** – each event is processed independently by all active and-states.
  - This is logically equivalent to each and-state receiving its own copy of the event sent to the object.
- **Propagated events** – an action is executed when an event is executed by one and-state, and this event is consumed in the next modeling step by another and-state, enforcing a specific order of transition execution.
  - E.g. `/GEN(evStep2);`
- **Guards** – guards may be used to ensure that another and-state is in a particular state using the `IS_IN()` operator,
  - E.g. `IS_IN(DOOR_CLOSED)`
- **Common attributes** – the scope of a state machine is the object, so all object features are “visible” to all and-states.

See Figure 38 for examples of and-state synchronization.
- Separate and-states into different objects when the features they use don’t significantly overlap
  - If and-states manipulate different attributes of the object and use different services of the object, separate them into multiple objects with associations
- Use of and-states in use cases is not uncommon, but it is relatively rare to find them in the class model. Figure 39 is a use case statechart with and-states.
3.8.2 Transitions

- Use null-triggered events when you add a state for the purpose of forcing closure of the state machine step
  - This is typically done to force action completion so that the result of those actions can be used in guards

Figure 39: Use Case Statechart
Figure 40: Forcing action closure

- 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 41 for an example of an avoidable race condition
3.8.3 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 should 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);

3.8.4 Guards
- Guards should not have side-effects
  - In C, C++, or Java, “x = 0” should not be used as a guard; “x == 0” would be better
- Guards on transition segments exiting conditional connectors should 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 Forcing Closure rule above
- Use an [else] guard when the event triggering the transition must always be handled
- If you use a choice point, you must always have an [else] guard on an exiting transition
3.8.5 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 statemachine; 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 35 through Figure 37, above.

3.9 Activity Diagram Guidelines
- 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
- Use forks and joins to model concurrency in activity diagrams
- Use swim lanes to allocate activities to classifiers
- Transitions should activate upon completion of the previous activity rather than the reception of an event
- Use guards only on transitions exiting a condition or branch point
- Complexity in activity diagrams is managed by decomposition
  - Use activity submachines to manage complex activity diagrams, or
  - Decompose the operation into multiple operation calls
    - Each operation can have its own activity diagram
- Activity diagrams and state machine should be mixed with the statechart “on top”
APPENDIX A – RHAPSODY ACTION LANGUAGE

Basic syntax
The following appendix provides a brief introduction to Rhapsody’s action language. Action language statements are used in five primary places:

- In operation implementation bodies (Open the Features dialog for the operation, and select the Implementation pane), e.g.
  ```
  For (j=0;j<10;j++) {
    x += foo(j);
  }
  ```
- As actions on statecharts either
  - As state entry actions
  - As state exit actions
  - As transition actions
    ```
    E.g. ev1[x>=15]/foo(params->z);
    ```
- As guards on transitions
  ```
  E.g. [IS_IN(Closed) && (x < 10)]
  ```
- As action specifications in actions or activities on activity diagrams
- As guards on transitions in activity diagrams
The language is case sensitive. That is, “evmove” is different from “evMove”.

Each action statement not within a guard must end with a semi-colon.

All names must start with a letter and may contain additional letters, numbers, and the underscore characters, but cannot contain spaces. Special characters are not permitted in names, except for underscores (_). However, a name should never start with an underscore.

The following words are reserved and should not be used for names: asm, auto, break, case, catch, char, class, const, continue, default, delete, do, double, else, enum, extern, float, for, friend, GEN, goto, id, if, inline, int, IS_IN, IS_PORT, long, new, operator, OPORT, OUT_PORT, params, private, protected, public, register, return, short, signed, sizeof, static, struct, switch, template, this, throw, try, typedef, union, unsigned, virtual, void, volatile, while.

Assignment and Arithmetic Operators

\[ \begin{align*}
X &= 1; & \text{(Sets } X \text{ equal to 1)} \\
X &= Y; & \text{(Sets } X \text{ equal to } Y) \\
X &= X + 5; & \text{(Adds 5 to } X) \\
X &= X - 3; & \text{(Subtracts 3 from } X) \\
X &= X \times 4; & \text{(Multiplies } X \text{ by 4)} \\
X &= X / 2; & \text{(Divides } X \text{ by 2)} \\
X &= X \% 5; & \text{(Sets } X \text{ to the remainder of } X \text{ divided by 5)} \\
X &= X + 1; & \text{(Increments } X \text{ by 1)} \\
X &= X - 1; & \text{(Decrement } X \text{ by 1)}
\end{align*} \]

Printing

The “cout” operator prints to the screen. Elements to be printed are separated by the “<<” operator. Text strings are surrounded by double quotes. Attributes are referenced using their names. The “endl” operator prints a carriage return. So, to print out the current value of X, use the following command:

\[ \text{cout} << \text{“The value of } X \text{ is ”} << X << \text{endl;} \]

If the current value of X is 5, this statement prints the following message on the screen:

The value of X is 5

Comparison Operators

\[ \begin{align*}
X &= 5 \text{ (X equal to 5)} \\
X &\neq 5 \text{ (X not equal to 5)}
\end{align*} \]
X<3 (X less than 3)
X<=3 (X less than or equal to 3)
X>4 (X greater than 4)
X>=4 (X greater than or equal to 4)
X>2 && X<7 (X greater than 2 and X less than 7)
X<2 || X==7 (X less than 2 or X equal to 7)

Conditional Statements
Conditional statements begin with the keyword “if” followed by a conditional expression in parenthesis, followed by the statement to execute if the condition evaluates to true. You can optionally add the “else” keyword to execute a statement if the condition evaluates to false. The “else” clause can contain another nested “if” statement as well. For example:

```cpp
if (X<=10)
    X++;
else
    X=0;
```

Multiple statements can be grouped together by placing them in curly braces.

```cpp
if (X<=10) {
    X++;
    cout << “The value of X is” << X << endl;
} else {
    X=0;
    cout << “Finished” << endl;
}
```

Incremental Looping Statements
Incremental looping is accomplished using the “for” statement. It holds three sections separated by semicolons to specify: 1) an initialization statement, 2) a conditional expression, and 3) an increment statement. For example, to iteratively set the value of X from 0 to 10 while printing out its value:

```cpp
for (X=0; X<=10; X++)
    cout << X << endl;
```

Conditional Looping Statements
The “while” statement is used for conditional looping. This statement has a single conditional expression and iterates so long as it evaluates to true. The previous example could be implemented using a “while” statement as follows:

```cpp
X=0;
while (X<=10)
    cout << X << endl;
```
while (X<=10)
{
    cout << X << endl;
    X++;
}

**Invoking Operations**
To invoke an operation on a block, use the operation name followed by parenthesis. For example, to invoke the “go” operation:

```
go();
```

If an operation takes parameters, place them in a comma-separated list. For example, to invoke the “min” operation with two parameters:

```
min(X,Y);
```

**Generating Events**
The “OUT_PORT” and “GEN” keywords are used to generate events through ports. For example, to send an event named “evStart” out the port named “p2”, issue the following statement:

```
OUT_PORT(p2)->GEN(evStart);
```

To generate an event with parameters, place them into a comma-separated list. For example, to generate an event named “evMove” with two parameters for velocity and direction:

```
OUT_PORT(p2)->GEN(evMove(10,2));
```

NOTE: The “OPORT” keyword can be used in place of “OUT_PORT”.

**Referring to Event Parameters in Transitions**
The “params” keyword followed by the “->” operator is used to reference the parameters of the event that caused the current transition. For example, if an event named “evMove” has a parameter named “velocity”, that parameter can be referenced using “params->velocity”. This syntax can also be embedded in statements within the action on the transition. For example,

```
if (params->velocity <= 5)…
```

**Testing the Port on which an Event Arrives**
The “IS_PORT” keyword is used to test whether the event that caused the current transition arrived through a specific port. For example:

```
if (IS_PORT(p2))…
```
Testing the State of a State Machine
The “IS_IN” keyword is used to test whether a state machine is in a specific state. For example, to test whether the state machine of a block is in a state called “Accelerating”:

```plaintext
if (IS_IN(Accelerating))...
```