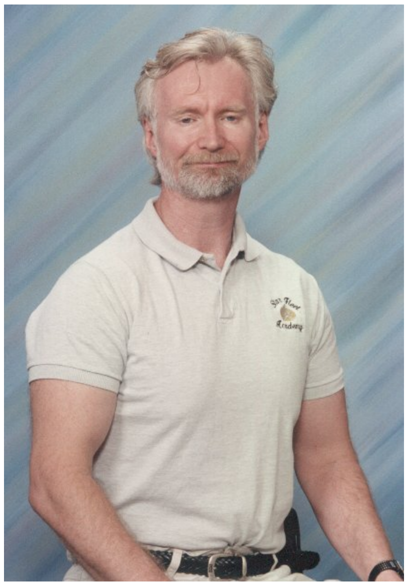
Computing Model Complexity



Bruce Powel Douglass
Chief Evangelist
IBM Rational

Metrics: The Good, the Bad, and the Visually-Challenged

Metrics have been applied – and misapplied – to icsoftware systems for a number of decades now. In a perfect world, metrics provide both a measure of quality, an estimate of defect rates, and even the likely location of defects in a given system. In general, a *metric* is a measurement of some characteristics of the system that is, in theory, related to an important system aspect that is difficult to measure or compute directly. Using metrics is akin to looking at a potential new-hire's college grades as a measure of what he or she knows, because measuring what they actually know is fairly difficult. You *hope* there is a strong correlation between what you're measuring and what you want. It's the old story of things that are easy to measure versus things that you want to know. Or, as Kant put it in *A Critique of Pure Reason,* "That which is known cannot be real, and that which is real cannot be known" – the so-called "analytic-synthetic dichotomy." All too often, though, the use of metrics is more like searching for your keys under the streetlight even though you lost them hundreds of yards away in the bushes because "the light is better." There is the risk that what you are actually measuring is *uncorrelated* with what you hope to estimate.

The application of metrics in a blind, simplistic fashion is unlikely to yield any benefit; however, if you understand the concepts, metrics can provide useful information that can improve the actual quality of the system under development. In this paper, we'll focus on UML *model* metrics rather than software metrics, even though the history of such metrics is primarily developed in the software industry. One advantage of model metrics is that it allows us to apply the very same techniques to disciplines other than software that also use models, especially systems engineering.

Types of Metrics

As stated above, a metric is a measurement used to estimate some information that you want but is difficult to obtain directly. Since there are different kinds of such information, there are different kinds of metrics. The metrics that we will be concerned with here can be divided into two groups: quality metrics, and fault metrics. Quality metrics estimate some aspect of the quality of the system, such as the ability to

- maintain, modify, adapt, and evolve the system over time,
- understand and comprehend the system,
- extend the system to new requirements,
- port the system to new environments,
- demonstrate adherence to one or more standards, such as DO-178B, for the purpose of regulatory agency approval

On the other hand, fault metrics attempt to estimate the likelihood and even the probable location of flaws in the system analysis, design, or implementation. The primary way this latter goal is achieved is by measuring behavioral complexity. The theory is that the more complex something is, the more likely it is to have defects in it.

Some metrics users have gone so far as to make strict guidelines, such as “The cyclomatic complexity of any module shall not exceed x.xx” or “The coupling among units shall be less than y.yy”. While understandable, ultimately these efforts are misguided. Metrics are guidelines and should never be rules. Metrics are not really measures of quality and error rate, but are instead measures of things one hopes correlate with those characteristics. Furthermore, some design problems are just harder than others so that there is a distribution of complexity in various modules. I do think there is a use for metrics – primarily as an indication that a module of class might require some special attention. Using metrics to guide testing, for example, or peer reviews, is a great idea. If a module or class is has higher coupling or cyclomatic complexity doesn’t mean that the module or class is *bad*, but does suggest that some extra testing or examination is probably a good idea.

Law of Douglass #72: “The Difference between Theory and Practice is Greater in Practice Than it is in Theory”

There are many different metrics available. Many of these metrics are straightforward adaptations of metrics from structured design approaches, others are less straightforward adaptations, and still others are entirely new, constructed to analyze object-specific features. Some of these metrics apply to large-scale structural aspects of the models and others apply to small (object/block) scale. The interested reader is referred off to the references for more information.

The main thing to remember is that metrics are *guidelines* – quantitative though they may be, they are not the same thing as the benefit or the quality measure we are trying to achieve. So it doesn’t make sense to rigidly adhere to these metrics but it does make sense to use them to identify potential “hot spots” or areas of potential concern.

Model Organizational Metrics

I like to organize structural metrics into two primary forms, following the two kinds of architecture identified in the ROPES approach. On one hand, we have the "logical architecture" – the organization of things that exist at design time, and on the other, we have "physical architecture" – the organization of things that exist at run-time. The former is also known as "model organization" because it concerns itself with how we organize models on our desktop, and this is a completely independent concern than how we organize pieces of the running system. In the logical model, we group analysis and design elements into units called packages. In the UML, a package is a design-time-only model element that contains other model elements. It is meant to be a work unit and also a configuration management unit[1].

Name	Purpose	Description
NP	Number of Packages	Identifies the number of "work" or "configuration" units in the model
DPC	Depth of Package Containment	A measure of the depth of the model organization unit
EP	Number of Elements In Package	The number of classes and other elements (such as use cases and types) in a specified package, a measure of the size and granularity of the package
AEP	Average Number of Elements Per Package	A measure of the overall granularity of the model organization
MEPP	Maximum Number of Elements Per Package	A measure of the maximum complexity of model organizational elements
PU	Package Utility	Number of developers the number of people who have read or usage access to a package / the number of developers who write element of a package

Requirements Metrics

The "functional model" of a system refers to the capabilities of a system and the various qualities of service (QoS) of those capabilities without regard to how those capabilities are achieved. In reality, this is nothing more (or less) than the set of black-box requirements of a system or system element. The UML elements used to represent these are use cases, various relations among the use cases, requirements elements (a Rhapsody extension to the UML 2.0[2]), statecharts and activity diagrams (to specify the requirements in a formal language), sequence diagrams, constraints, actors, associations among the actors and the use cases, information flows, and information items.

[1] For more discussion about this topic, see my white paper in reference [6].
[2] This extension is to be found in the SysML (Systems Modeling Language) specification work that is currently underway and should be submitted to the OMG for standardization later this year.

Name	Purpose	Description
NUC	Number of Use Cases	A measure of the number of independent capabilities of the system
FP	Function Points	An estimate of the complexity of the problem to be solved, maps well to the NUC metric
NA	Number of Actors	The number of actors associated with a given use case
NUCA	Number of Use Cases per Actor	Given an actor, the number of use cases associated with the it
NUCSD	Number of Use Case Sequence Diagrams	The total number of black-box sequence diagrams used as exemplars for use cases
AUCSD	Average number of Use Case Sequence Diagrams	NUCSD / NUC. This is a measure of the average scope of a use case
NUCS	Number of Use Case States	Total number of states + activities used to specify the use cases
UCDC	Use Case Decomposition Complexity	The number of use cases derived from a single use case – this includes generalization and dependencies (both «includes», and «extends» relations)
IIC	Information Item Count	Total number of Information Items in the use case model
IICUC	Information Item Count per Use Case	IIC / NUC

Figure 1 shows how requirements elements might be bound to a use case (either dependency or anchors can be used for this purpose). Figure 2 shows how a more complex requirements taxonomy from a use case might be represented – in this case, an alarm system for a medical device.

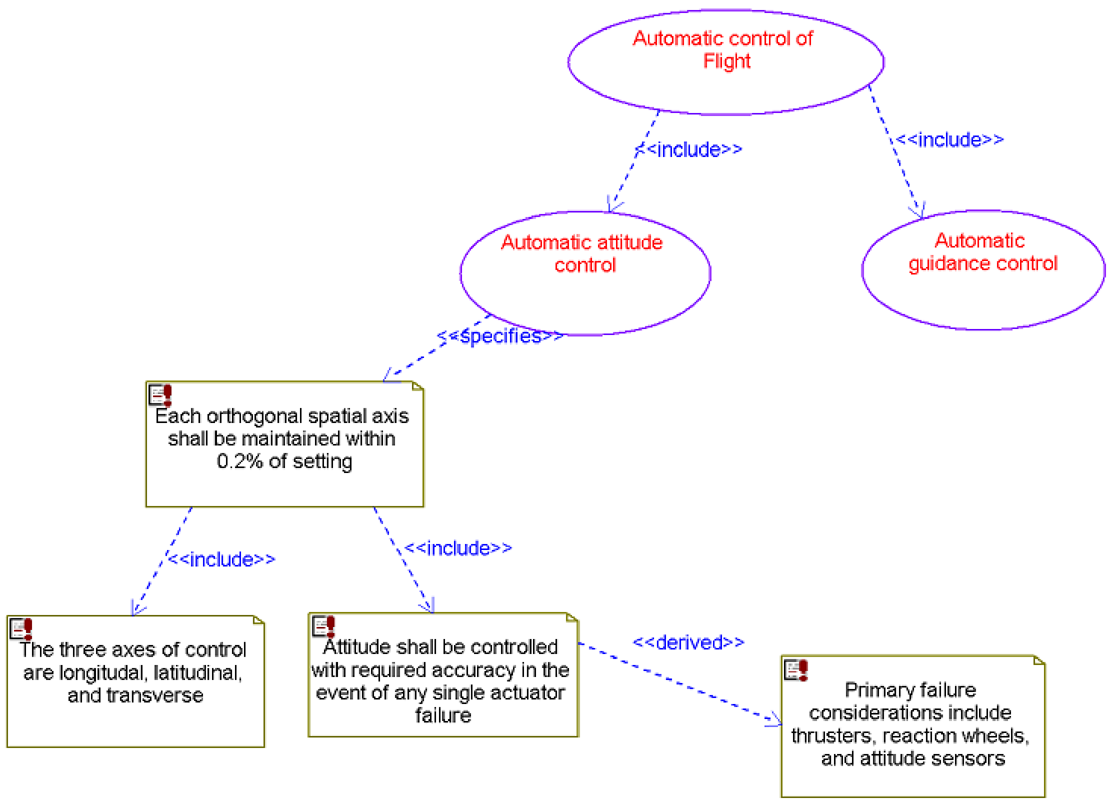
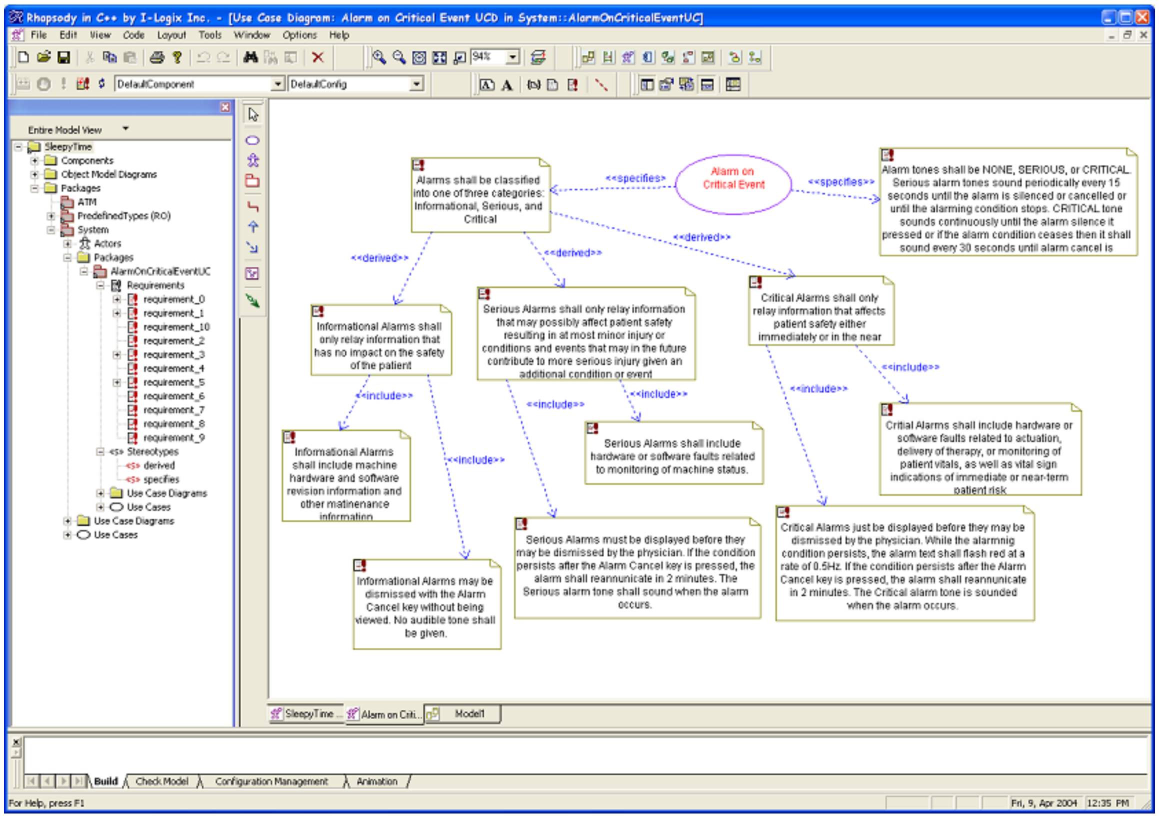

Automatic control of Flight
<<include>>
<<include>>
Automatic attitude control
Automatic guidance control
<<specifies>>
Each orthogonal spatial axis shall be maintained within 0.2% of setting
<<include>>
<<include>>
The three axes of control are longitudinal, latitudinal, and transverse
Attitude shall be controlled with required accuracy in the event of any single actuator failure
<<derived>>
Primary failure considerations include thrusters, reaction wheels, and attitude sensors

Figure 1: Combining Requirements Elements with Use Cases



Alarms shall be classified into one of three categories: Informational, Serious, and Critical

Alarm tones shall be NONE, SERIOUS, or CRITICAL. Serious alarm tones sound periodically every 15 seconds until the alarm is silenced or cancelled or until the alarming condition stops. CRITICAL tone sounds continuously until the alarm silence it pressed or if the alarm condition ceases then it shall sound every 30 seconds until alarm cancel is

Informational Alarms shall only relay information that has no impact on the safety of the patient

Serious Alarms shall only relay information that may possibly affect patient safety resulting in at most minor injury or conditions and events that may in the future contribute to more serious injury given an additional condition or event

Critical Alarms shall only relay information that affects patient safety either immediately or in the near

Informational Alarms shall include machine hardware and software revision information and other matinenance information

Serious Alarms shall include hardware or software faults related to monitoring of machine status.

Critial Alarms shall include hardware or software faults related to actuation, delivery of therapy, or monitoring of patient vitals, as well as vital sign indications of immediate or near-term patient risk

Informational Alarms may be dismissed with the Alarm Cancel key without being viewed. No audible tone shall be given.

Serious Alarms must be displayed before they may be dismissed by the physician. If the condition persists after the Alarm Cancel key is pressed, the alarm shall reannunicate in 2 minutes. The Serious alarm tone shall sound when the alarm occurs.

Critical Alarms just be displayed before they may be dismissed by the physician. While the alarmnig condition persists, the alarm text shall flash red at a rate of 0.5Hz. If the condition persists after the Alarm Cancel key is pressed, the alarm shall reannunicate in 2 minutes. The Critical alarm tone is sounded when the alarm occurs.

Figure 2: Modeling a Requirements Taxonomy

It is also possible to represent information flow in UML 2 and Rhapsody 5 as well. Many people find this a useful analytic technique for understanding the information processing required of a system. Figure 3 shows how such a diagram might be used to represent the information management for an air traffic control system. Such diagrams can be done for the entire system, but in complex systems, they should be done independently for each information-rich use case.

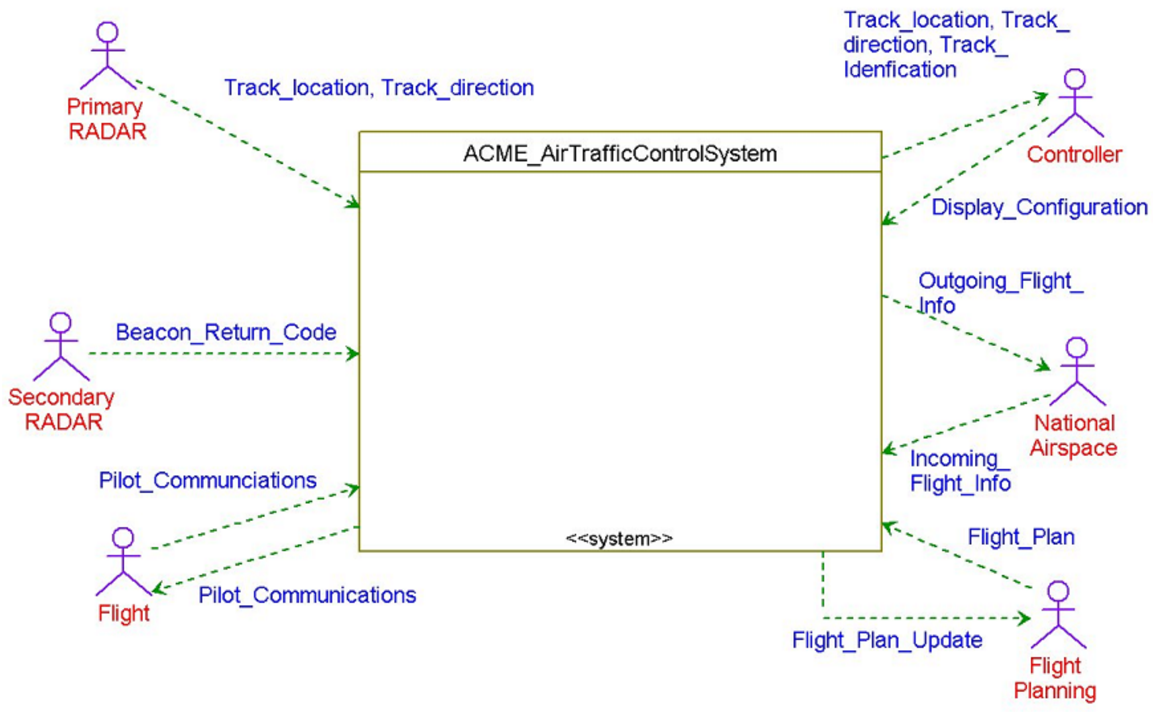


Figure 3: Information Flows in UML 2 and Rhapsody

Model Architectural Structural Metrics

The model structural metrics are focused on model analysis and design content, rather than on how it is organized or how its requirements are specified. These metrics can be applied to large models or small and to architectural aspects or to semantic elements. The difference is that architectural elements such as subsystems, tasks, channels, etc. exist to organize and deploy the semantic elements, and the semantic elements perform the “real work” of the system functionality.

Name	Purpose	Description
NS	Number of Subsystems	The number of large-scale architectural units of a system.
NT	Number of Tasks	Number of «active» objects in a system
NAS	Number of Address Spaces	A measure of the scope of the distribution of a model across address spaces or computers.
CASC	Cross-Address Space Coupling	A measure of the cohesion within address spaces versus cohesion across address spaces.
RAS	Redundant Architecture Scope	Number of redundant architectural units for use in the Safety and Reliability Architecture (either homogeneous or heterogeneous)
NP	Number of Processors	Number of processor nodes in the system. This *may* (or may not) be identical with the NAS metric
NCP	Number of Components per	Measures the cohesion of functionality within a processor node, assuming that a component

	Processor	provides a coherent set of functionality.
NUCS	Number of Use Cases per Subsystem	For systems that decompose system use cases into subsystem level use cases (see references [5] and [7])

For example, Figure 4 shows the subsystem architecture for an anesthesia device. NS for this system would be 6.

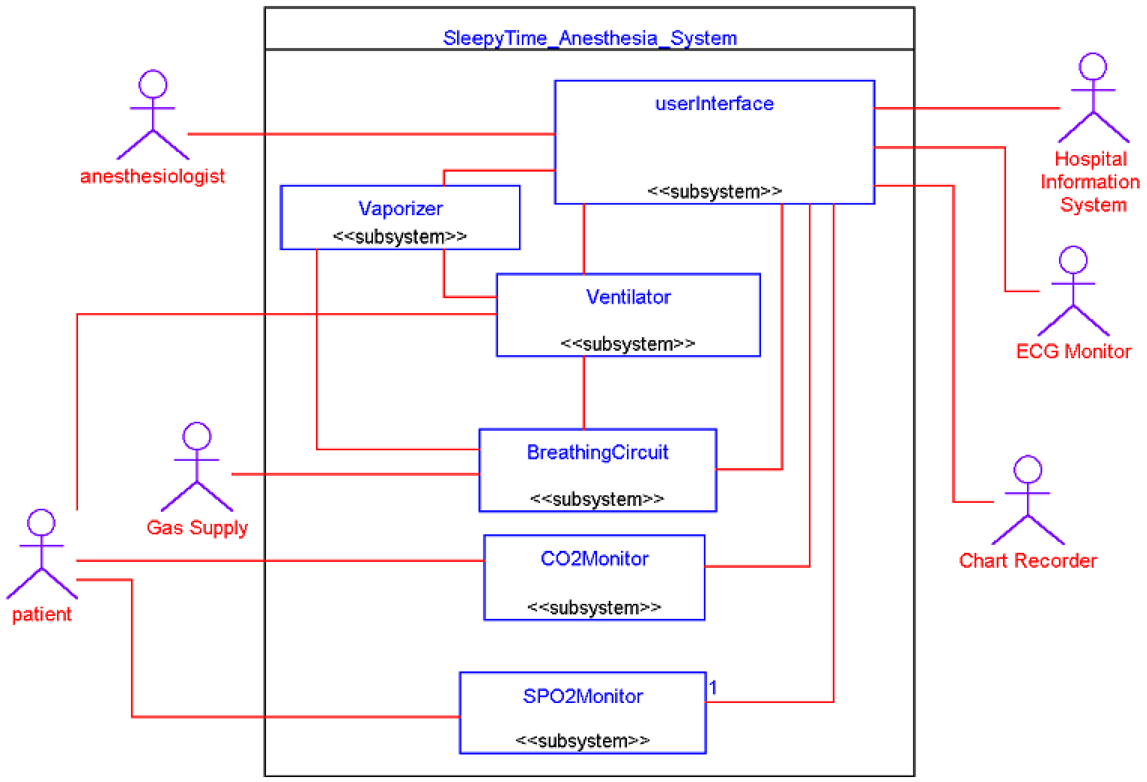


Figure 4: Subsystem Architecture

For the User Interface Subsystem, the NUCS is 4 as shown in the next figure.

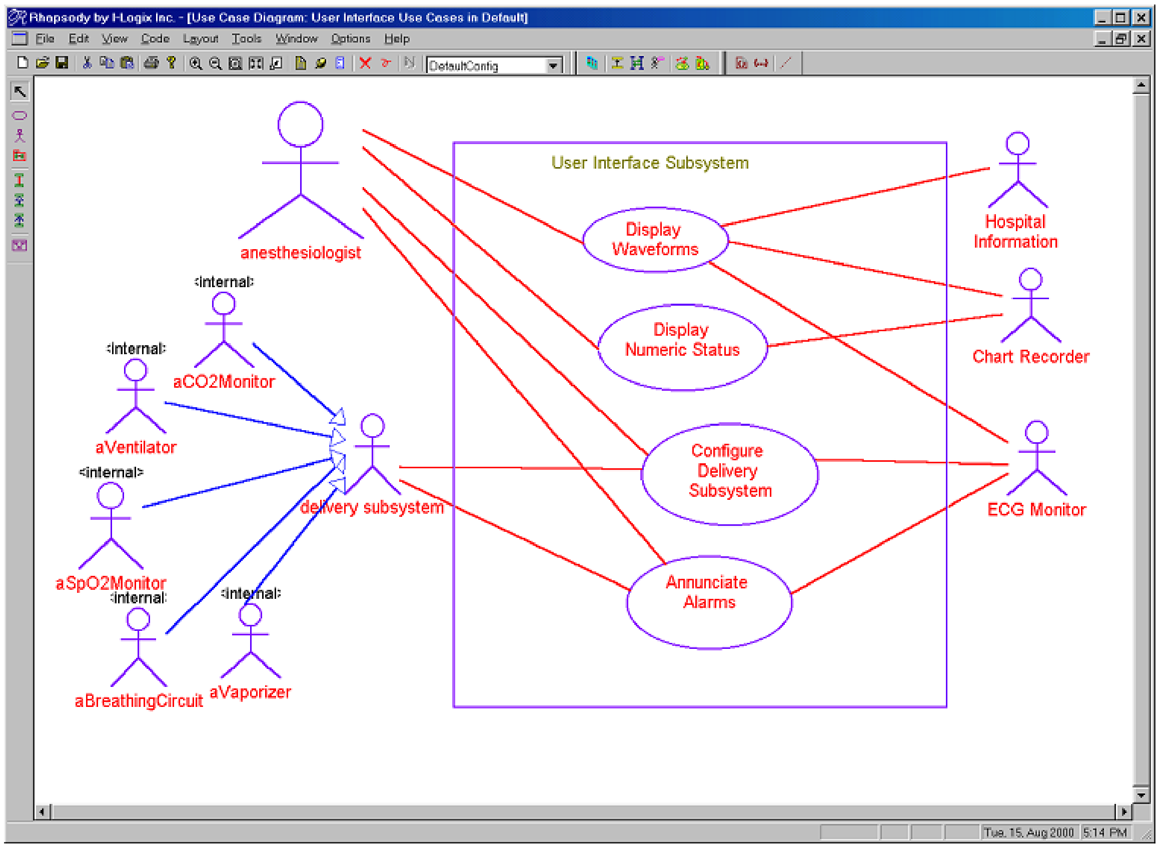


Figure 5: User Interface Subsystem Use Cases

Model Semantic Structural Elements

While the architectural elements organize the system into units, the semantic elements do "real work". The semantic elements (aka "primitive objects and classes") perform the structural and behavioral semantics of the system within the confines of the architecture. The architectural elements organize, optimize, and delegate behavior to these semantic elements but they don't perform the actual desired functionality of the system. That falls to the semantic elements.

Name	Purpose	Description
NoC	Number of Classes	A measure of the number of model size
CC	Class Coupling	Measures the cohesion of the classes by computing the number of associations a class has with its peers
TSC	Total number of subclasses	Measures the global use of generalization within a model
CID	Class Inheritance Depth	The maximum length of a given class generalization taxonomy
NC	Number of Children	Number of direct descendent (subclasses), a

		measure of the class reuse
NM	Number of Methods	Number of methods within a class
NP	Number of ports	Number of unique identifiable connection points of a class
EF	Encapsulation Factor	Number of class features (attributes, methods, and event receptions) publicly visible divided by the total number of such features – a measure of the degree to which the internal structure of a class is encapsulated
SF	Specialization Factor	The number of operations and statechart action sets which are specialized in subclasses

A couple of comments are in order. With the Class Coupling (CC) metric, we are concerned not about the number of clients a class has, but rather the number of servers it uses. Therefore, we only add the association when the association is navigable from the current class to another; if it is one way from another class, it doesn't count in the class coupling. This means that this metric estimates how many other classes I might have to touch if the current class must be updated.

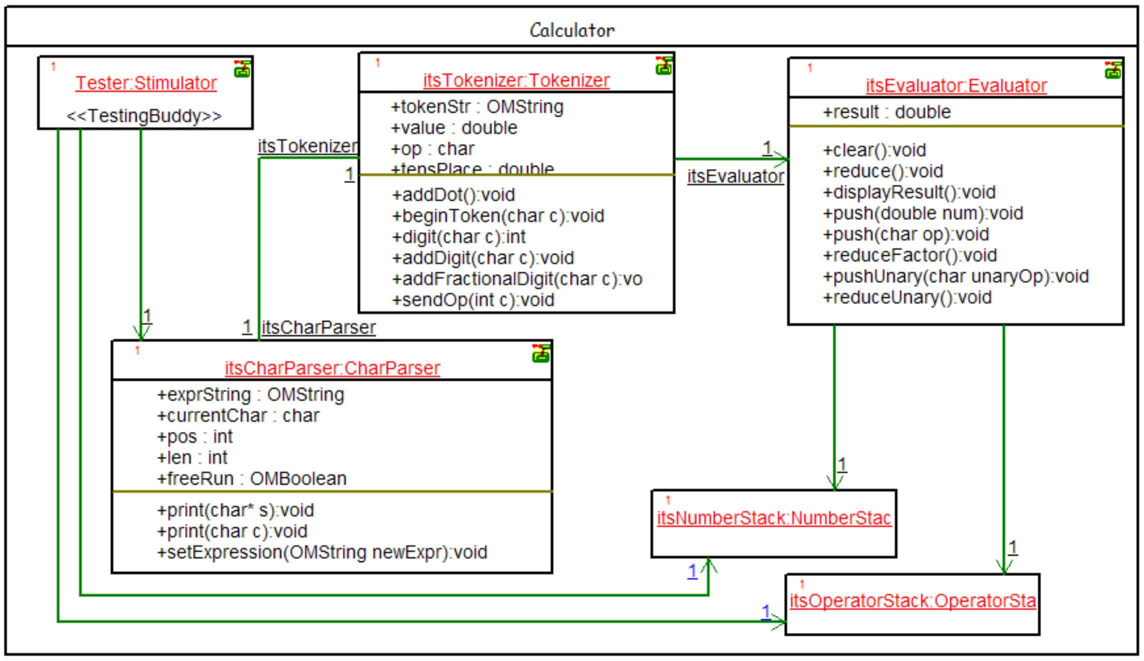


Figure 6: Example Class Diagram

What is the class coupling of the classes shown in Figure 6? The Calculator class has a coupling of 6, as it has composition relations with 6 classes. The Evaluator has a class coupling of 2 since it only knows how to "talk to" 2 other classes (the NumberStack and OperatorStack classes).

The Encapsulation Factor (EP) is a good metric for maintainability because it measures the degree to which the internal structure of a system is hidden from its clients. Figure 7 shows a CachedQueue class with a EP metric value of 0.56 (5/9).

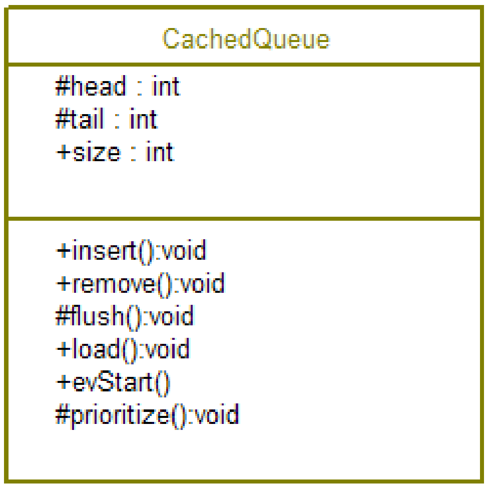


Figure 7: Encapsulation

Behavioral Metrics

The previous section discussed some of the metrics for assessing structural elements. This section provides some metrics for assessing behavioral semantics. A couple of these metrics look at the method bodies to assess their quality or complexity, but most are firmly focused at the model level. The UML has three primary behavioral views – sequence diagrams for collaborative behavior (and some alternative notations which are, in some sense, equivalent – communication and timing diagrams), and statecharts and activity diagrams for the behavioral specification of individual elements.

Name	Purpose	Description
SDS	Sequence Diagram size	Number of messages x number of lifelines
DCC	Douglass Cyclomatic Complexity	Modified McCabe cyclomatic complexity to account for nesting and and-states
WMC	Weighted Methods per Class	A measure of (non-reactive) class complexity = sum of methods x complexity for all methods. For classes without activity diagrams, method complexity can be estimated by Lines of Code in the method.
ND	Nesting Depth	State and activity nesting depth – number of levels of nesting
NE	Nesting Encapsulation	Number of transitions (other than default) that cross levels of nesting

NAS	Number of And-States	Total number of and-states within a statechart
SCN	Statechart completeness	Number of events received by a statemachine / number of transitions
NPS	Number of pseudostates	Number of non-default pseudostates such as history, conditional, fork, join, junction, and stubs

Cyclomatic complexity is probably the most common metric used for behavioral complexity and can be applied directly to both statecharts and activity diagrams. The basic computational rule for cyclomatic complexity is the number of edges – number of nodes (e.g. states) + 2.

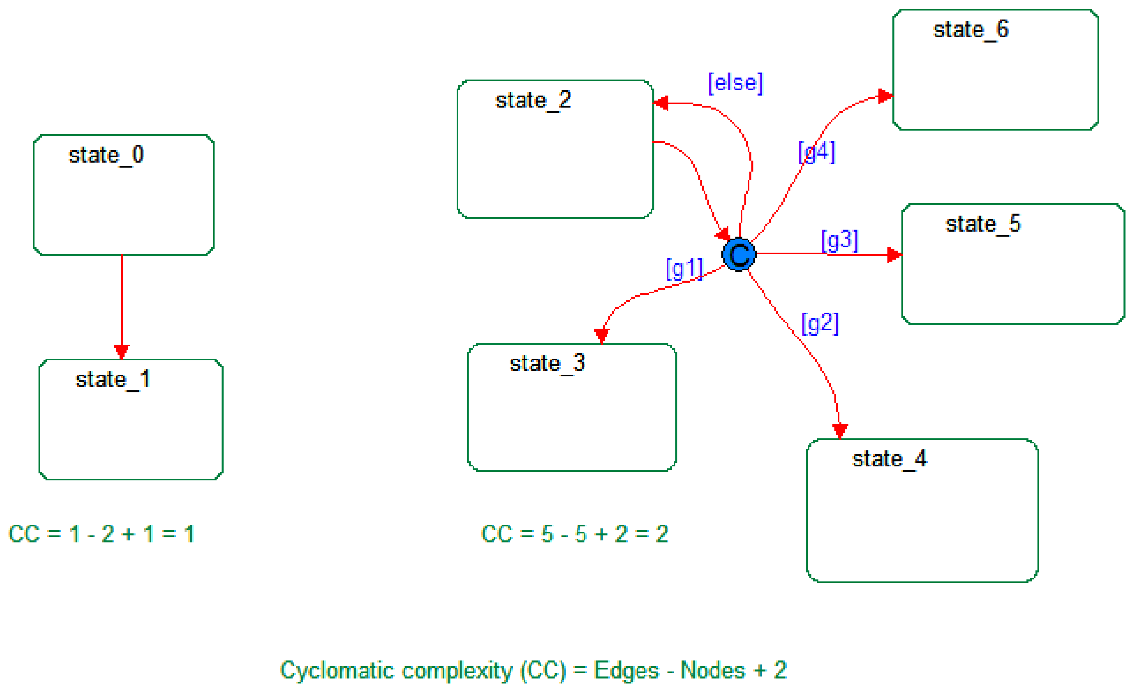


Figure 8: Cyclomatic Complexity for States

Figure 8 shows the computation of cyclomatic complexity for two different state machines. They are both straightforward, other than to note that an edge is a complete transition path to another state so that the number of such paths in the right hand side of the figure is 5 and not 6 (you don't count the path to the conditional connector as a separate edge). However, CC doesn't account for nesting of states, nor does it account for and-states.

Consider Figure 9. The two statemachines are behaviorally equivalent. In either case, you have 4 states and from each state there is a transition path to the other three. If you compute the standard CC for the left statemachine you'll get a cyclomatic complexity of 1 (4-5+2), while you'll get 10 (12-4+2). Which is right?

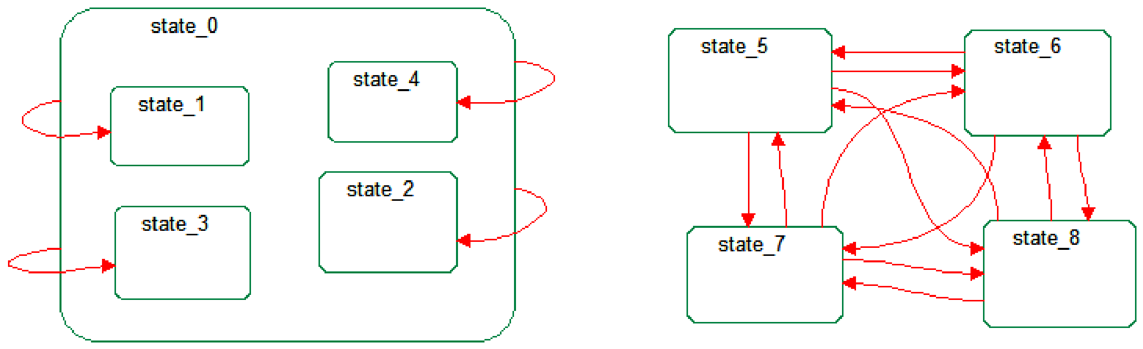


Figure 9: CC for Nested States

To answer the question, we need to remember why we're doing this. We want to ascertain a measure of the likelihood that the statechart is incorrect due to complexity-induced errors. Therefore, in this case I would prefer to go with the former computation rather than that of the semantic equivalent (1 instead of 10). However, nesting does itself add some amount of complexity. Even though this statechart and the left-most statechart in Figure 8 have the same computed CC, surely the latter is more complex. Thus, I add "1" for every level of nesting. Thus the Douglass Cyclometric Complexity (DCC) for the left hand figure is 2, since there is one level of nesting.

Next, what about and-states?

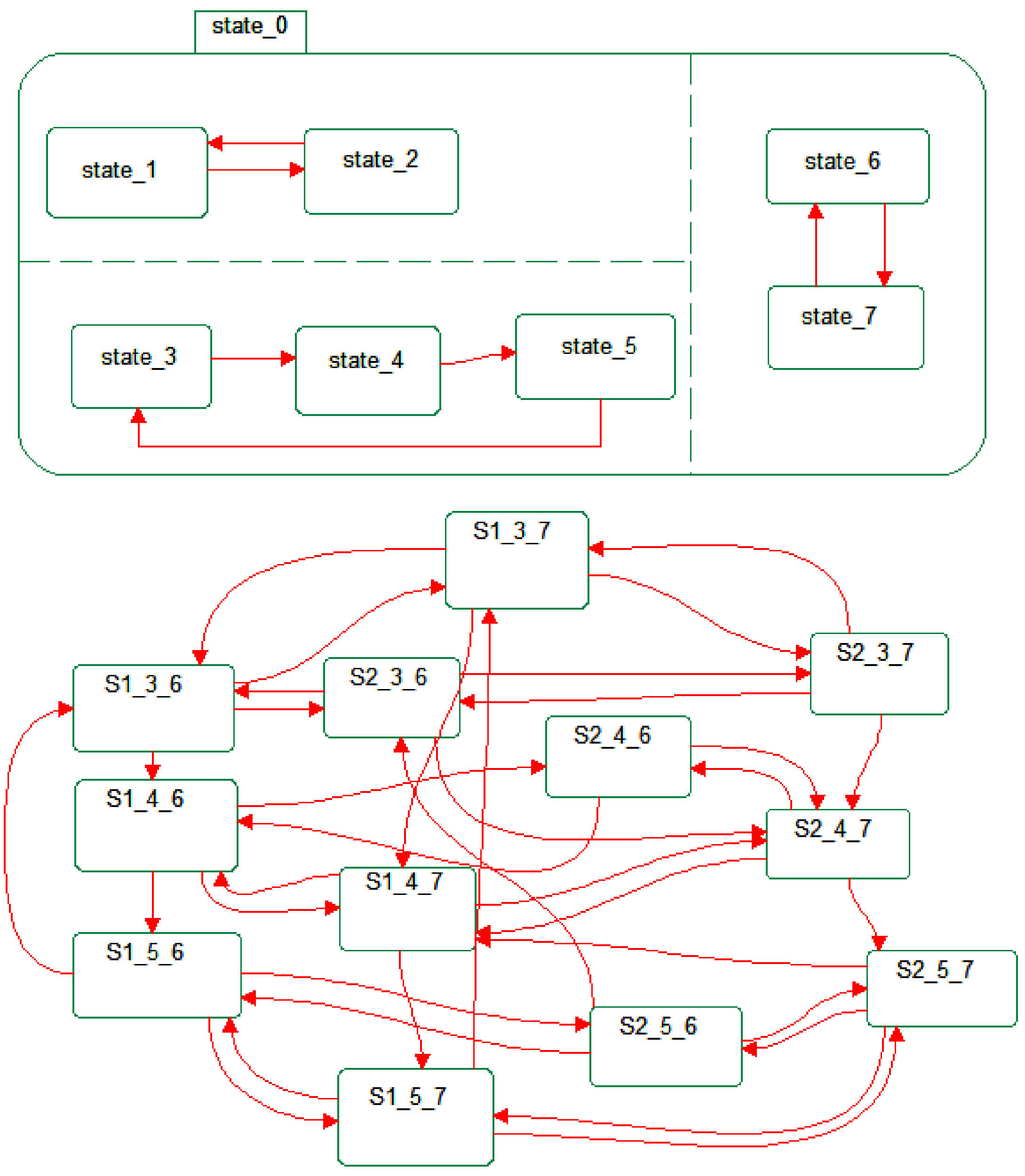


Figure 10: CC for And-States

First, let me say that the upper and lower statecharts in Figure 10 are semantically equivalent (you *knew* you liked statecharts for a reason, didn't you!). The statechart in the upper part of the figure, use and-states to represent three independent aspects of the class while the lower part is the same thing without the and-states. A straightforward cyclomatic complexity of the upper part of the figure yields 1 (7-8+2) while the same computation on the semantically-identical lower statechart yields 25 (35-12+2). Arguably, both values are correct, but which is the more useful? I think that the lower part of the figure is *far* more difficult to understand and get right that the upper part, so I'm very much inclined to use the computation for the upper part. On the other hand, using and-states is more complex than not. Surely, the upper part of this figure is more complex than the left-most statechart in Figure 8. Therefore, I add a "1" for every and-state, yielding a DCC value of 4 for the upper figure.

So the computation of the DCC is simply this:

DCC = edges – nodes + 2 + levels of nesting + number of and-states

This gives a measure of complexity that correlates well, I believe, with the ability to model the state behavior without making an error. By way of an example, what is the complexity of the statechart shown in Figure 11

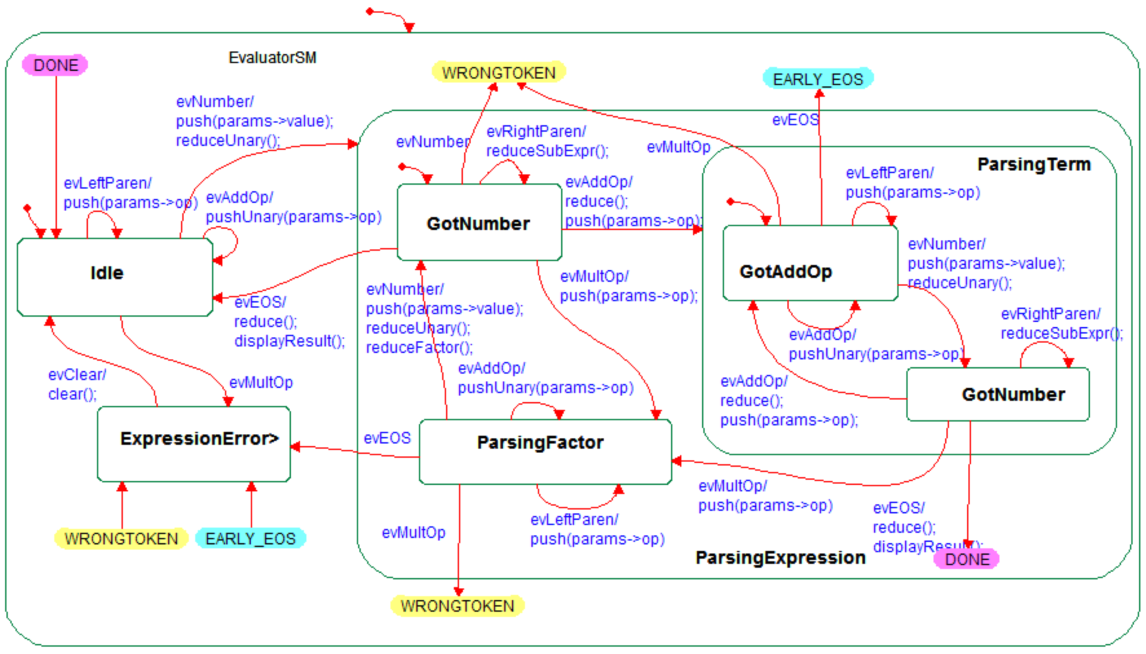

EvaluatorSM
DONE
WRONGTOKEN
EARLY_EOS
Idle
GotNumber
ExpressionError>
ParsingFactor
GotAddOp
ParsingTerm
ParsingExpression
evNumber/ push(params->value); reduceUnary();
evLeftParen/ push(params->op)
evAddOp/ pushUnary(params->op)
evEOS/ reduce(); displayResult();
evClear/ clear();
evMultOp
evRightParen/ reduceSubExpr();
evAddOp/ reduce(); push(params->op);
evMultOp/ push(params->op);
evNumber/ push(params->value); reduceUnary(); reduceFactor();
evLeftParen/ push(params->op)
evEOS
evEOS/ reduce(); displayResult();

Figure 11: Example Statechart

This figure represents the statechart for a class that evaluates tokens in a calculator. Much of the complexity of this statechart arises from the numerous actions that are executed. The computed CC for this statechart would be (22 – 7 + 2 + 2 (for 2 levels of nesting)) for a total of 19.

Diagram-Specific Metrics

The metrics commonly used are all, if you will, *model metrics*. That is, the hope to analyze some aspect of the quality of the system as opposed to the presentation. The main reason for that, I believe, is because in source code these two issues are the same thing. However, in a language such as UML, the visualization of a model and the semantics of the model are independent issues, with the possible exception of statecharts and activity diagrams. Therefore I think it makes sense to have separate metrics for the visualization aspects.

Why are statecharts and activity diagrams the exceptions? Simply put, because these diagrams are fully constructive visual representations of the underlying semantic model. That is, for a given model element (Classifier or method, for example), a statechart or activity diagram completely specifies the behavior of that element. Ok, it might invoke "submachines," breaking the single diagram into a hierarchy of diagrams, but logically it's a single specification and a single diagram. The other diagrams are not. I can choose virtually any scheme to organize my class diagrams, for example into a set of, at best, loosely coupled diagrams.

When writing books, authors (and I am as guilty as any in this regard) select and simplify examples so that they illustrate the point at hand. Thus we see single diagrams showing a model or aspect of a model. Real systems, however, are complex and cannot be shown on a single diagram – even if you use E-size plotter paper and 2-pt font. Some rule or guideline must be applied to decide how to break up the system views across multiple diagrams.

A simple one is the famous, and counterproductive, 7 +/- 2 rule. The idea, misinterpreted from some neurolinguistics research is that the human brain can only hold 7 (+/- 2) things at once, so this is, therefore, a good rule to apply to diagram construction. The flaw is that the authors of this rule failed to understand the research. First of all, the rule, such as it is, is a rule about *short-term memory* not comprehension. That is, most humans can hold 7 (+/- 2) things in short term memory at once. However, if you're looking at a diagram, you don't need to hold those things in memory *because they are right in front of you!* Duh…

So on the face of it, the rule is a misinterpretation of the basic research. But how does it work in practice?

The answer is "poorly." Back in the good old Data Flow Diagram (DFD) days of the 80s, this rule was widely applied as a rule of good diagram construction and rigidly enforced in many environments. The results were diagrams that were virtually unreadable, with arbitrarily-deeply nested diagrams just to meet the criterion.

A better rule, I think, is the ROPES[3] rule, which has to do with the coherence of a diagram. The coherence of a diagram may be thought of as the degree to which the diagram represents a single important concept; what ROPES refers to as the *mission* of the diagram. The ROPES rule is "every diagram should convey a single important concept." That is, every diagram should represent all elements necessary to convey its mission, whether that is one element or 50. Statecharts and activity diagrams already meet this criteria; their mission is to describe the possible sequences of actions and event receptions for a single model element. The other diagram types are more flexible in this regard and so coherence becomes more important for those diagrams.

[3] Rapid Optimizing Process for Embedded Systems

Consider the standard class diagram (OMD in Rhapsody parlance). It is a structural diagram showing some aspect of the system structure. Some standard ROPES missions for class diagrams are:

- Collaboration – show all of the object roles collaborating to realize a single use case
- Package contents – show all the classes within a single package
- Domain diagram – show the set of “domains” and their relations (single subject matter packages); this is sometimes called a “package diagram”
- Subsystem architecture – show the set of subsystems[4] and their relations
- Distribution architecture – show how objects are distributed across address spaces with distribution patterns such as Broker or Publish-Subscribe.
- Safety and reliability architecture – show how faults are identified, isolated, and corrected at run-time through the application of redundancy
- Concurrency architecture – the so-called “task diagram”; show the set of concurrent elements («active» and other concurrency objects such as mutex semaphores, message queues, etc)
- Deployment architecture – show how elements from different engineering disciplines (e.g. software, mechanical, electrical, and chemical) collaborate[5]
- Generalization – show the a single generalization taxonomy
- Class structure – show the decomposition of a “structured” class into its internal parts

Diagrams that adhere to this guideline only contain elements that contribute to the mission of the diagram and avoid others. This means that a class typically participates in multiple missions (several collaborations, it resides in a single package, might be part of a generalization taxonomy, etc).

The good news is that the consistent use of diagram missions leads to highly readable and comprehensible systems, and it scales very well from tiny to huge systems. The bad news is that it is very difficult to compute adherence to the guideline because coherence to a concept is difficult to quantify. The best way to ensure adherence to this guideline remains (1) specify on the diagram what the mission is, and (2) check adherence during a peer review.

[4] Note that a subsystem is just a “big architectural object” that contains and delegates services to, its internal part object roles

[5] This can be shown on a “deployment diagram” which is arguably just a weak form of a class diagram. However, the SysML (Systems Modeling Language) effort has rejected deployment diagrams because they are not as powerful as normal class diagrams, and use class diagrams with various kinds of stereotyped classes («electrical», «mechanical», «chemical», and «software») instead.

Qualitative Guidelines for Model Quality

Metrics, as has already been stated, use easily-computed values to estimate more ethereal-but-desirable system and model properties. While computational metrics can be useful when properly applied, it behooves us to understand what are the qualitative goals that these quantitative measures seek to measure, so that they may be optimized.

Model Organization Guidelines

- Models should be organized to optimize developer workflow
- Packages are the primary organizational unit of UML modeling
- Use of domains (i.e. «domain» stereotype of package) maximizes reusability of classes
- Use of subsystems allows flexible organization of objects into coherent run-time artifacts

A typical model organization looks Figure 12:

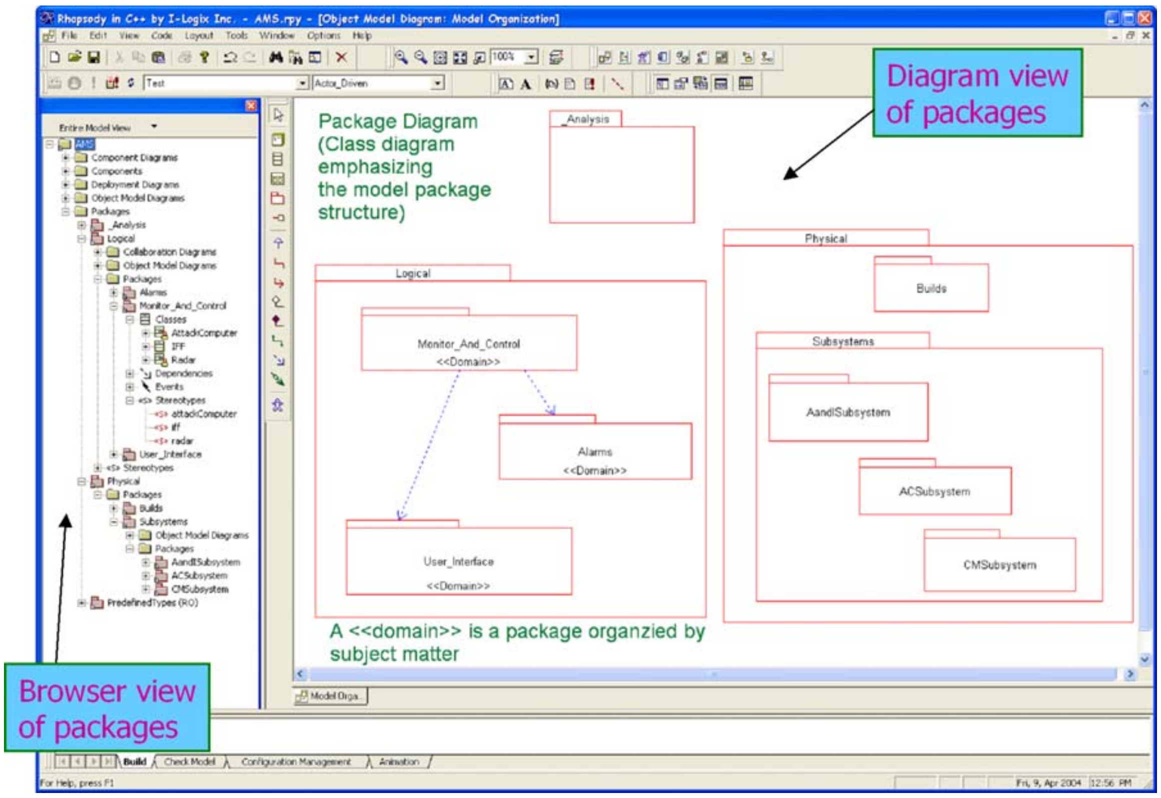


Figure 12: Model Organization

In Figure 12, we see an three "high-level" packages – Analysis, Logical, and Physical. The Analysis package is where we locate requirements and "system" things, such as the system actors, the system class, system use cases, sequence diagrams and so on. The Logical package contains subpackages stereotyped «domain» which contain all of the classes and types identified in object analysis. The Physical package contains primarily, the architectural design elements, such as subsystems, and collaborations instantiated from the classes and types specified in the domains of the Logical package.

Requirements Guidelines

Requirements need not be done in UML or Rhapsody, but there are advantages to doing so. Requirements written and maintained in text have problems with completeness, consistency, accuracy, and correctness. These properties are much easier to ensure in a UML/Rhapsody view because of the increase in formality of the representational semantics of the forms of use cases, statecharts, activity diagrams, and sequence diagrams. Some important guidelines for requirements include:

- A use case should be a coherent representation of a complete system usage at a given level of abstraction
- There should be 6-24 use cases at the highest level
- In complex system, use cases may be decomposed with «include» and «extends» dependencies, and with generalization among use cases to create more specific, detailed uses cases at a lower level of abstraction
- Use cases should not reveal or imply system design, nor specific system interface technologies
- Actors should represent elements outside the scope of the system with which the system must interact
- *Time is not an actor* – sometimes systems provide autonomous behavior
- A use case should return a result visible to at least one actor.
    - For capabilities within a system that do not return a result to an actor, use requirements elements inside a package or stereotype a use case to be «internal»
- Use cases should be approximately independent in their requirements, although not necessarily in terms of their ultimate design or implementation
- Use case names should be strong verbs, never nouns

Model Architectural Structural Guidelines

These guidelines have to do with overall organization of the running system.

- Architectural design is all about optimizing the system through organization in-the-large
- Physical architecture should be represented with at least one class diagram in each of the Five Views of Architecture
    - Subsystem and Component View
    - Distribution View
    - Concurrency and Resource View
    - Safety and Reliability View

    - Deployment View
- Each subsystem contains and delegates behaviors to its internal parts.
- The internal parts of a subsystem are ultimately instances of the types and classes specified in the domains

Figure 13 schematically shows the 5 views of architecture, while the next figure shows an example of the safety and reliability architecture. Note in Figure 14 that the subsystems (using the Channel Pattern from [5]) contain semantic objects as parts.

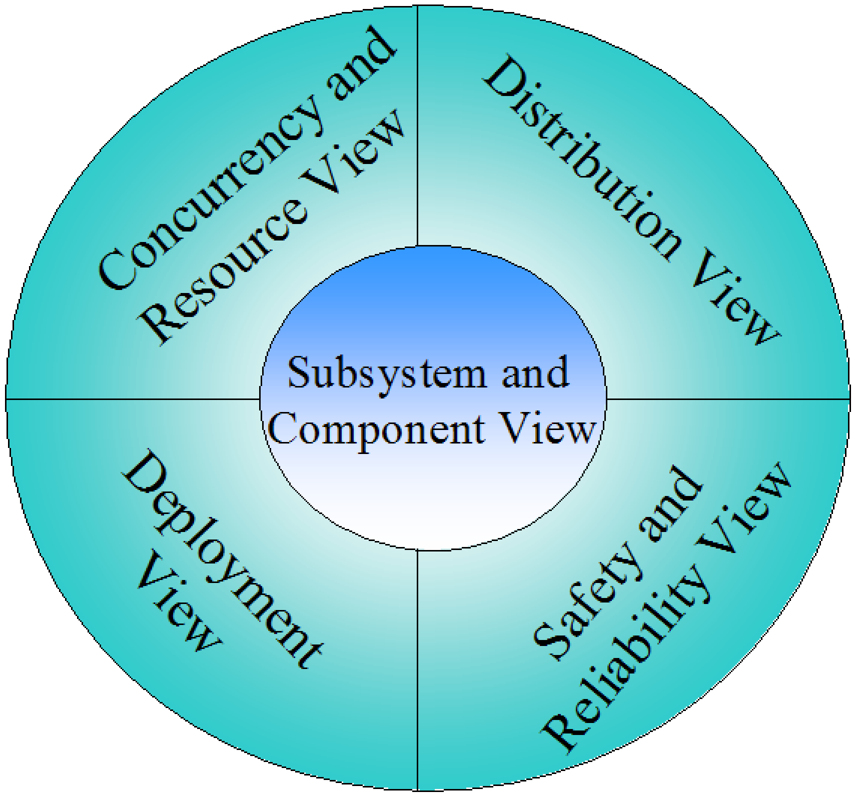

Concurrency and Resource View
Distribution View
Subsystem and Component View
Deployment View
Safety and Reliability View

Figure 13: Architectural Views in the ROPES Process

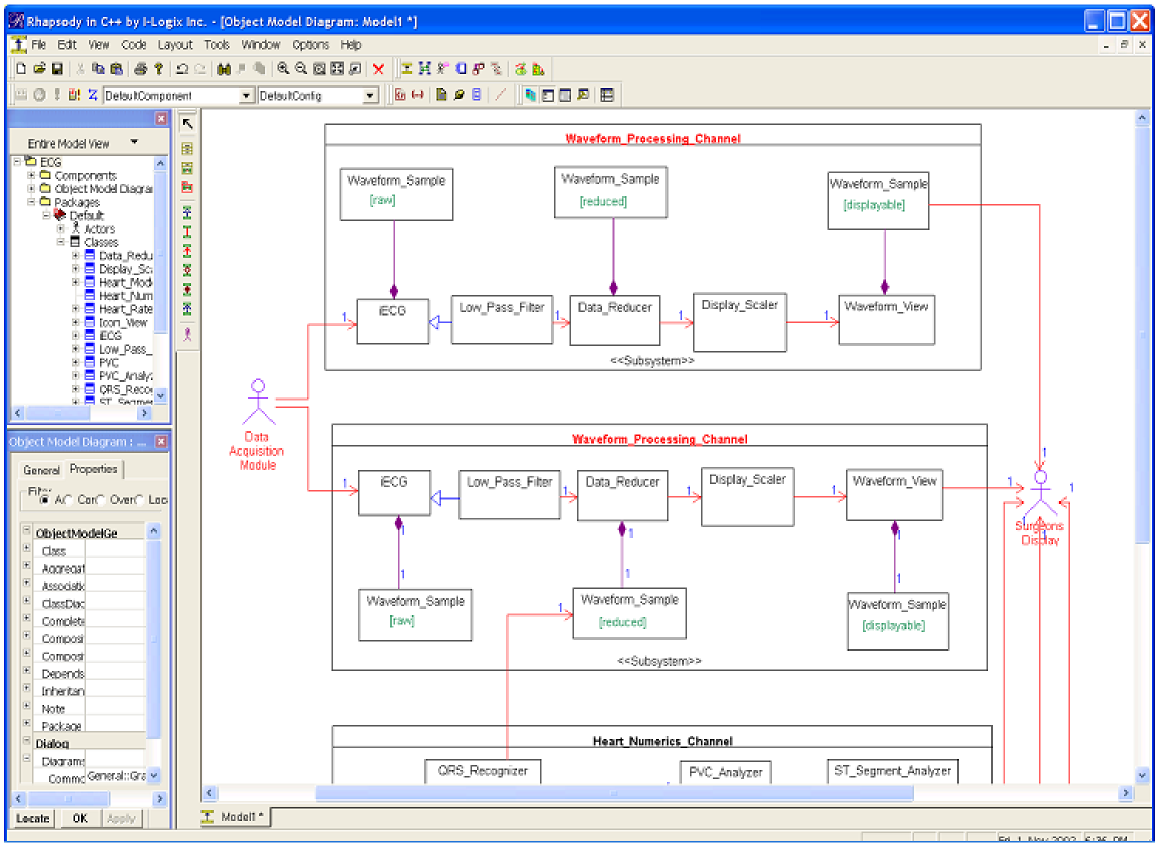


Figure 14: Safety and Reliability Architecture View

Figure 15 is a typical task diagram from the concurrency architecture view. It uses «active» objects as the primary elements, with passive object running in the context of those threads as parts, resources shown as shared among those threads, and various other elements of the concurrency architecture including semaphores, queues, and concurrency properties.

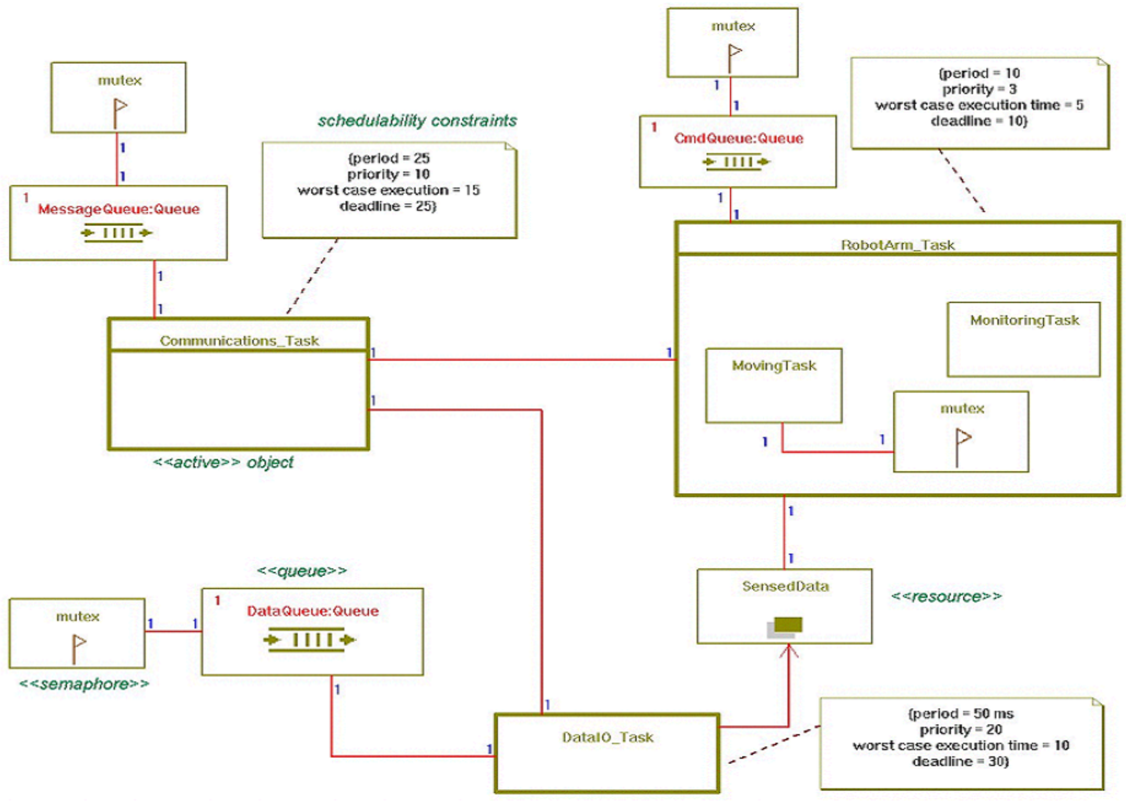


Figure 15: Concurrency Architecture View

Model Structural Guidelines

These guidelines refer to the semantic elements used to perform the “real work” of the system.

- Each semantic[6] class belongs in exactly one domain; when this is not true, either refactor the class or refactor the domains
- A class is not the same as a role; a role is a usage of a class in a specific context while a class is a specification of a thing that may fulfill a number of different roles
    - E.g. “attack dog” is a role – “Dog” is the class, and “myAttackDog” is the name of the association role end, or the name of the instance
- Generalization taxonomies always stay within a single domain
- All use case collaborations contain elements from multiple domains
    - Associations cross package boundaries
- When packages represent concepts at different levels of abstraction, associations should only be navigable from more abstract -> less abstract
- All class diagrams should depict a single important concept
    - Only show class features that contribute to the mission of the diagram

[6] As opposed to classes added to optimize the design of the system.

  - Show role names on associations when they contribute the diagram mission
- Classes should have noun or noun phrases for names
  - Operations should have verbs for names
  - Operation names should reveal semantic role of the operation not internal implementation
  - Attributes should have nouns or noun phrases for names
- Attributes should be structurally simple
  - Complex attributes should be modeled as classes with a composition relation from the primary class
- Always indicate multiplicity

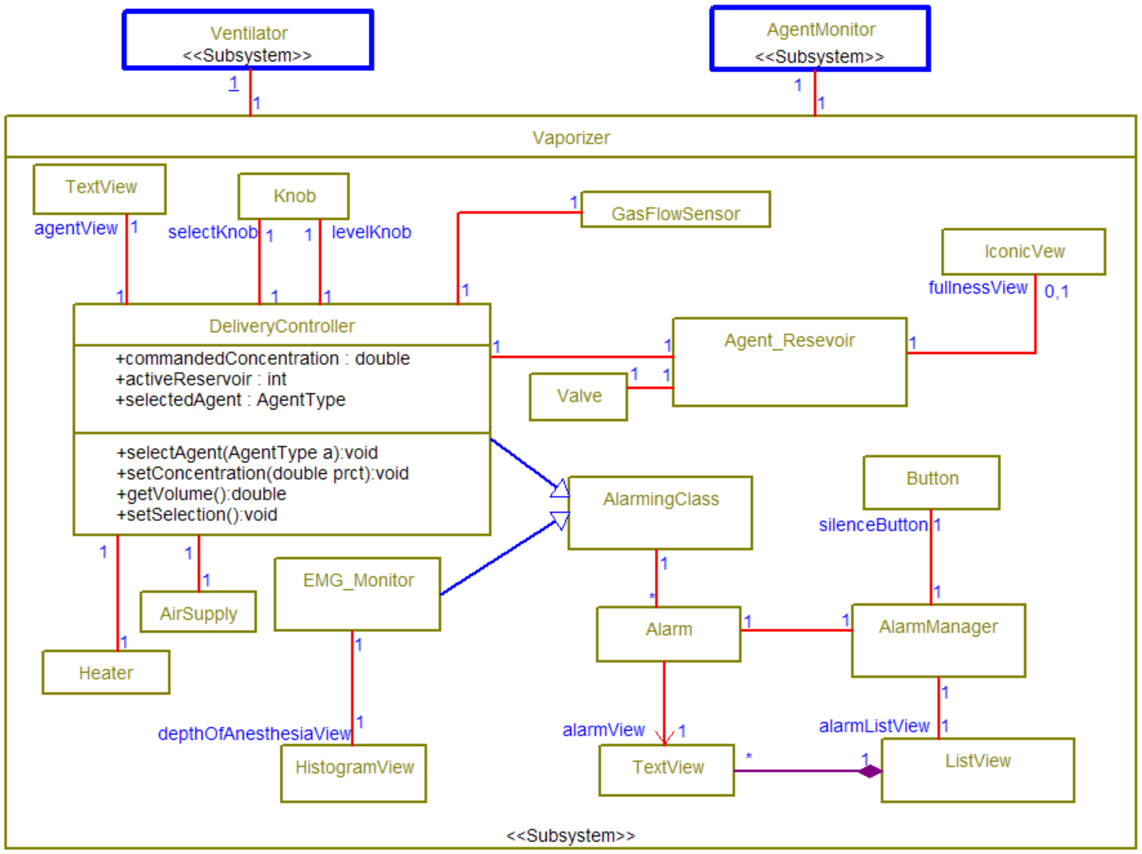


Figure 16: Class Diagram

Figure 16 shows a class diagram, emphasizing the internal structure of one of the subsystems.

Model Behavioral Guidelines

Statemachine and Activity Diagram Guidelines

- Use a statechart to specify class (or use case) behavior when the behavior is driven primarily in response to incoming events
- Use an activity diagram to specify class (or use case) behavior when the behavior is primarily control flow (algorithmic) behavior
- Always identify initial states
- When a one or more transitions have the same effect for a set of states, consider making those sets nested within a composite state
- Use and-states to specify independent aspects of state behavior
- Guards should never overlap; that is, at most one guard on a set of related transitions should be true at any time
- Place actions on transitions if those actions are only performed under some circumstances when leaving or entering a state
- Place actions on entry to a state when they must always be executed when the state is entered; similarly, place actions on exit from a state when they must always be executed when a state is left
- When a set of states is accessible from all other states in that set, make them all nested states of a single composite and make the transitions from the composite to the nested states
- For use case statecharts, message from the actors to the system should be modeled as transition events; messages from the system to the actors should be modeled as actions
- Use swimlanes on activity diagrams to map activities to different objects
- Use submachines to simplify diagrams when a statechart or activity diagram becomes too complex
- “Wrap” long sequences of actions into single operations defined on the class to simplify the diagrams

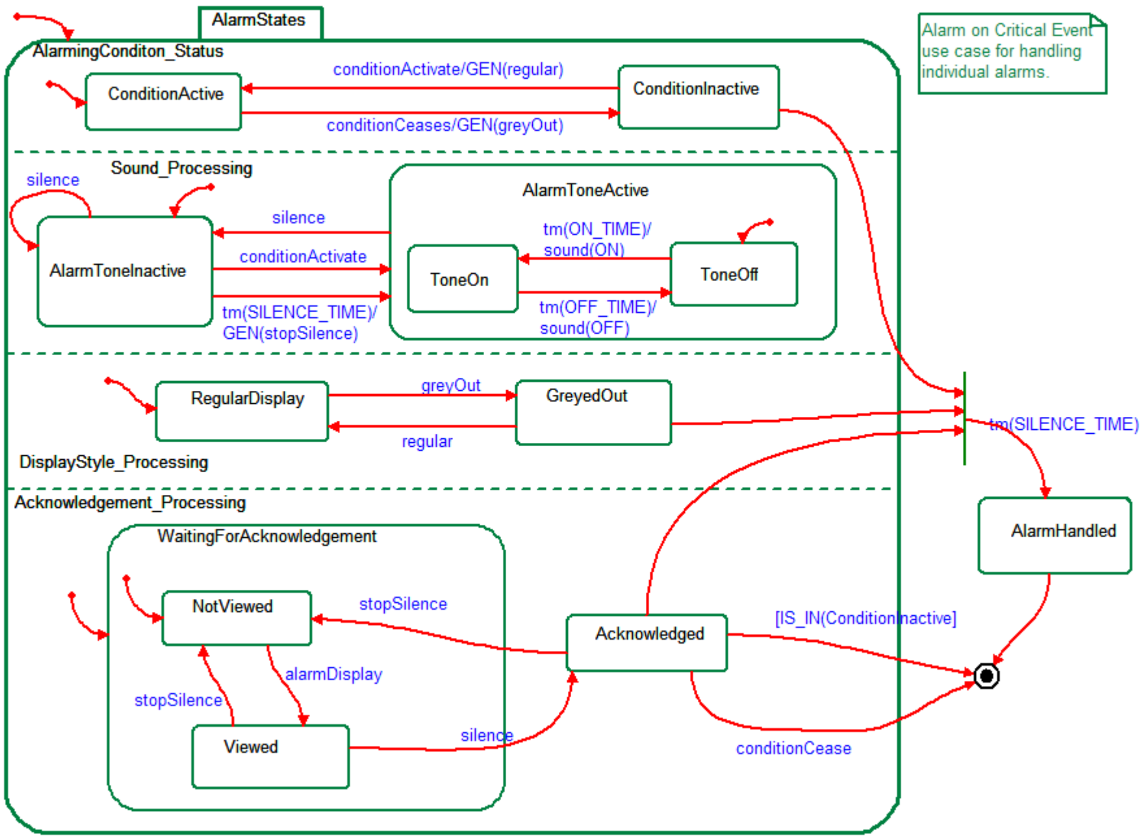


Figure 17: Statechart diagram

Figure 17 shows a somewhat typical statechart diagram for a Manage Alarms use case.

Sequence Diagram Guidelines

Sequence diagrams can easily become unmanageable with long sequences and random ordering of messages. Use of the following guidelines will improve the readability of your sequence diagrams.

- Place a note box in an upper corner of each sequence diagram to identify the diagram, which use case (or collaboration) it elaborates, its pre- and postconditions, and the purpose of the sequence
- Try to get left-to-right ordering of messages as much as possible
- Avoid activation instance boxes – they only apply in a subset of sequences
- Use partition lines when you want to add special semantics to a set of messages – including loops, iterations, or constraints
- Use text freely along one side of the sequence diagram with comments as to *why* the sequences are flowing as they are
- Use lifeline decomposition when you want to show sequence diagrams at multiple levels of decomposition. At the high level, the life lines can represent architectural

units (such as subsystems) which may be composed into sequence among their parts in the nested sequences
- Use interaction fragment decomposition when you have a coherent set of messages that you want to remove to either simplify the current sequence diagram or because you want to reuse the nested sequence in multiple places
- In a lifeline decomposition sequence, use a System Border element to represent message crossing the nesting boundary

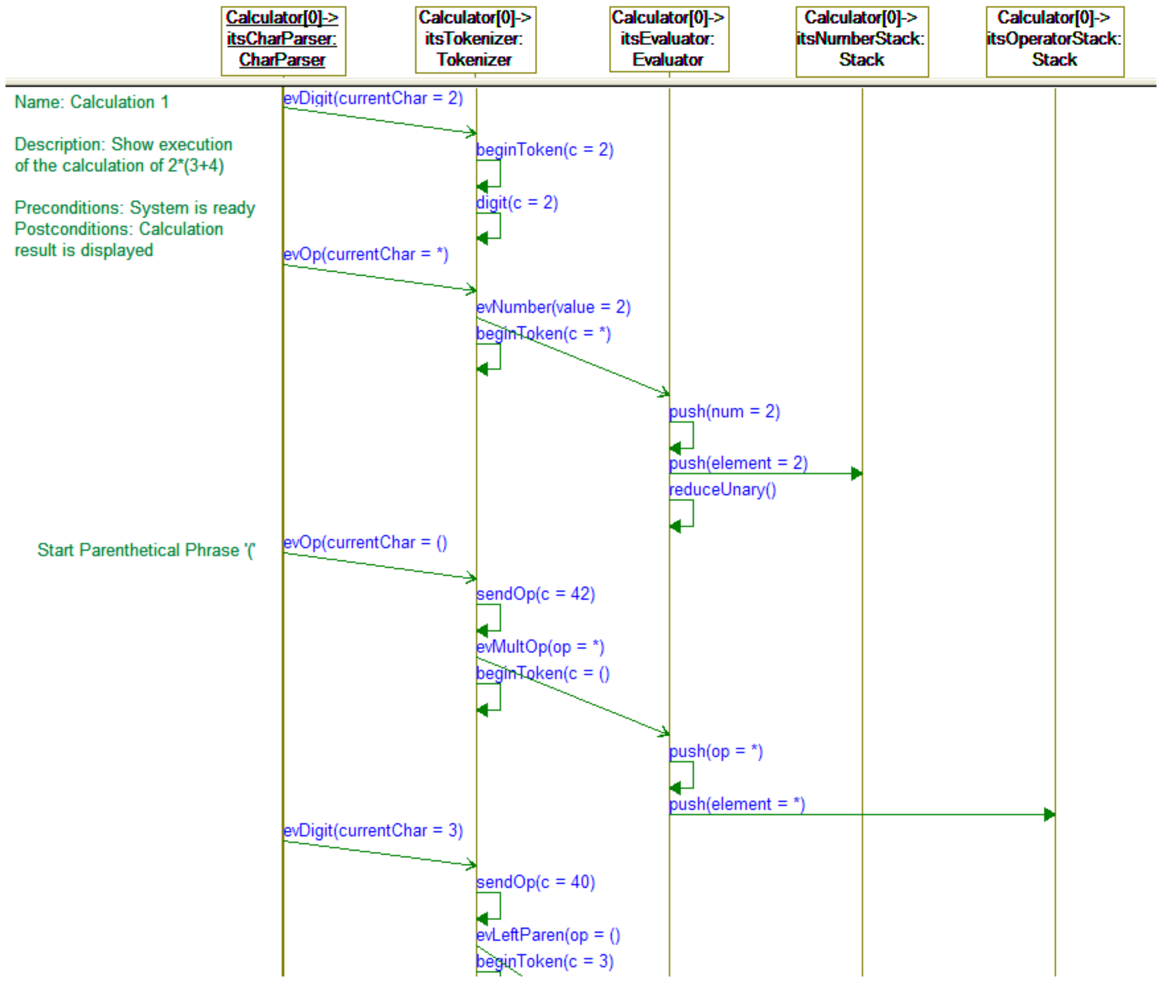

Calculator[0]-> itsCharParser: CharParser
Calculator[0]-> itsTokenizer: Tokenizer
Calculator[0]-> itsEvaluator: Evaluator
Calculator[0]-> itsNumberStack: Stack
Calculator[0]-> itsOperatorStack: Stack
Name: Calculation 1
Description: Show execution of the calculation of 2*(3+4)
Preconditions: System is ready
Postconditions: Calculation result is displayed
Start Parenthetical Phrase '('
evDigit(currentChar = 2)
beginToken(c = 2)
digit(c = 2)
evOp(currentChar = *)
evNumber(value = 2)
beginToken(c = *)
push(num = 2)
push(element = 2)
reduceUnary()
evOp(currentChar = ()
sendOp(c = 42)
evMultOp(op = *)
beginToken(c = ()
push(op = *)
push(element = *)
evDigit(currentChar = 3)
sendOp(c = 40)
evLeftParen(op = ()
beginToken(c = 3)

Figure 18: Sequence diagram

General Diagram Guidelines

Some guidelines apply to all diagrams:

- Eliminate or minimize line crossing
- Use color only as hints, never to convey semantic information
- Always include a diagram note naming the diagram and identifying its purpose (mission), context, and pre- and postconditions where appropriate

- Each diagram should represent exactly one important concept and contain only those elements that support that mission
    - It is perfectly permissible to have the same model elements show up on multiple diagrams
- Use consistent naming conventions; recommended is
    - Class names between with uppercase letters
    - Class features (operations, triggered operations, events, and attributes) begin with lowercase letters
- Use "pin notes" for To-Do items use other notes for persistent comments about model structure and semantics
- When there are many diagrams of the same type, use packages to organize them into coherent sets

Computing Metrics of Rhapsody Models

Oops, I seem to have run out of space. In a future column I'll pick several of these metrics and show how the Rhapsody API can be used to access the model with Visual Basic, and provide some VBA macros to compute these metrics.

Conclusion

We've discussed, albeit briefly, some of the metrics that can be applied to models to gain an insight into their quality and fault proneness. Which are best? They all have their place but the various metrics measure or emphasizes different measures of "goodness."

To improve collaboration in large-scale teams, it is important to follow the organizational guidelines given, even though they are difficult to estimate with quanitative metrics. Some metrics that are valuable to estimate team collaboration potential include package methods Package Usefulness – the ratio of the number of people who use a package versus the number of people of can use elements of a package. When this ration is large, then the package tends to be coherent and "tight", while at the same time being highly useful to other teams. Deeply nested packages are of less concern when the nested elements are not visible to the client of the primary package; however, when this is not true, then a high DPC (Depth of Package Containment) can seriously inhibit the usability of elements within a package. Class coupling is also important, because it is a measure of how many elements I might need to modify when I change a class. This is especially important for Cross-Address Space Coupling (CASC), where we identify the number of clients in other address spaces.

For maintainability, we want to maximize encapsulation so that when we change some internal part of a model element we have minimal, if any, effect of the clients or users of that element. For maintainability, the most important metric is the Encapsulation Factor (EF) metric (number of public class features / total number of class features). When this number is low, then the classes hide significant internal design. This generally improves maintainability. Similarly, low class coupling implies that changes will be more localized, simplifying maintenance.

For fault estimation, we want to simplify the behavior for the simple expedient that simple things are easier to get right. For getting behavior right, the Douglass Cyclomatic Complexity metric is my favorite, a perhaps minor tweak on a classic measurement. For non-reactive systems, the same metric can be applied to activity diagrams. When the behavior of classes isn't modeled (such as when you write the methods using the target source language), then Weighted Methods Per Class is a good measure for predicting faults. WMC is sums the product of each method and its complexity, such as measured in Lines of Code.

In a future article, we'll actually write some VBA macros to extract this information from your models and compute the results.

References

[1] Lorenz and Kidd "Object-Oriented Software Metrics" Prentice-Hall, 1994

[2] In, Kim, and Barry "UML-Based Object-Oriented Metrics for Architecture Complexity Analysis" Texas A&M, 2003
http://sunset.usc.edu/gsaw/gsaw2003/s8e/in.pdf

[3] Rosenberg, Linda "Applying and Interpreting Object Oriented Metrics"
http://www.rspa.com/reflib/ProductMetrics.html#OOMetrics

[4] Douglass, Bruce Powel "Real-Time UML 3rd Edition: Advances in the UML for Real-Time Systems" Addison-Wesley, 2004

[5] Douglass, Bruce Powel "Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems" Addison-Wesley, 2002

[6] Douglass, Bruce Powel "Beyond Objects: Organizing Models the Right Way"
http://www.ilogix.com/whitepapers/whitepapers.cfm

[7] Douglass, Bruce Powel "Dr. Douglass' Guided Tour Through the Wonderland of Systems Engineering, UML, and Rhapsody"
http://www.ilogix.com/whitepapers/whitepapers.cfm

[8] Ambler, Scott "The Elements of UML Style" Cambridge Press, 2003

About the Author

Bruce Powel Douglass has over 20 years experience designing safety-critical real-time applications in a variety of hard real-time environments. He has designed and taught courses in object-orientation, real-time, and safety-critical systems development. He is an advisory board member for the Embedded Systems Conference, UML World Conference, and Software Development magazine. He is a cochair for the Real-Time Analysis and Design Working Group in the OMG standards organization. He is the Chief Evangelist at Telelogic, a leading real-time object-oriented and structured systems design automation tool vendor. He can be reached at bpd@ilogix.com.