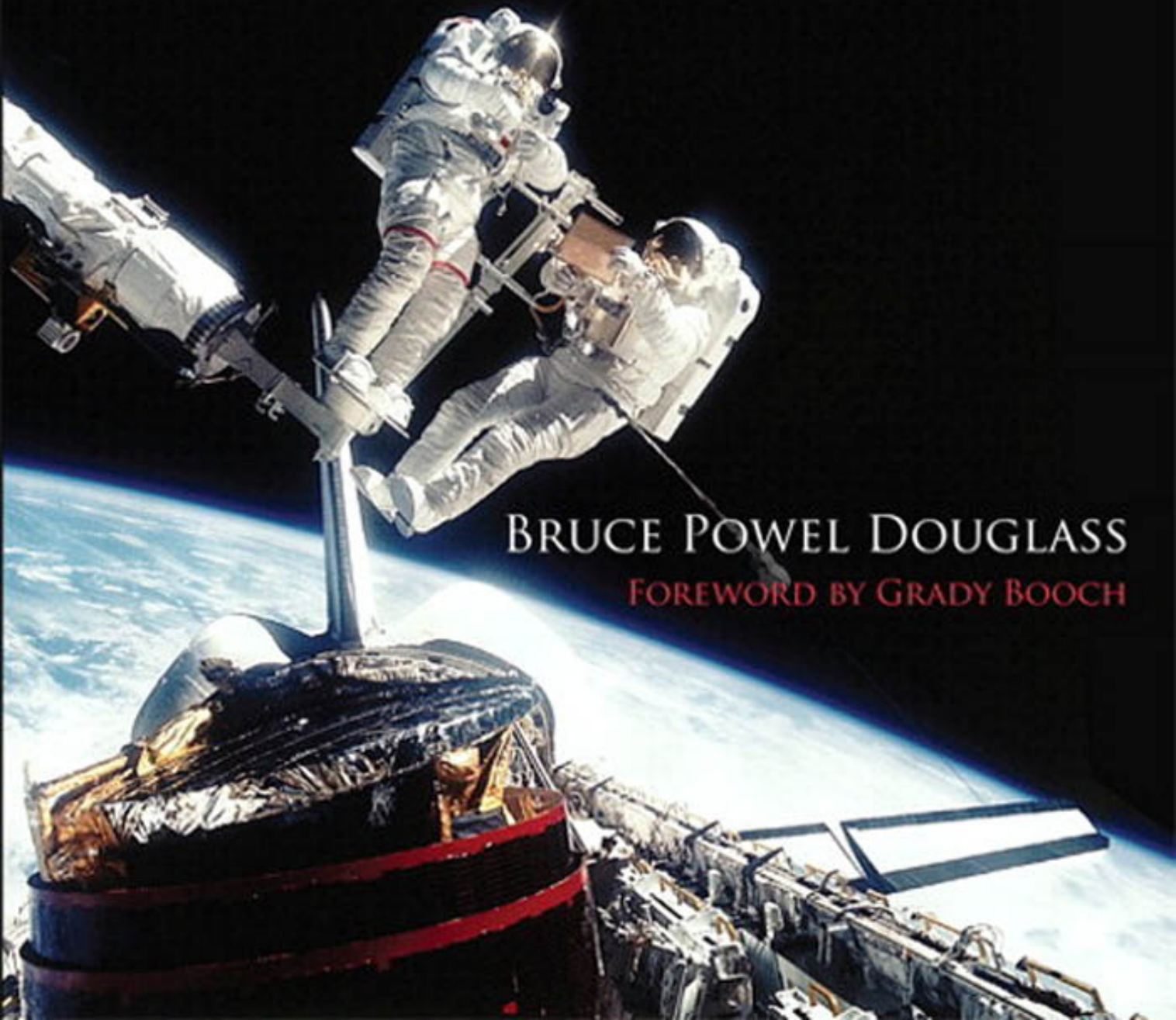
# Real-Time Agility

## The Harmony/ESW Method for Real-Time and Embedded Systems Development



Bruce Powel Douglass

Foreword by Grady Booch

## Praise for *Real-Time Agility*

"Regardless of your perceptions of Agile, this is a must read! Douglass's book is a powerful and practical guide to a well-defined process that will enable engineers to confidently navigate the complexity, risk, and variability of real-time and embedded systems—including CMMI compliance. From requirements specification to product delivery, whatever your modeling and development environment, *this is the instruction manual.*"

*—Mark Scoville, software architect*

"This book will provide you with the framework of agile development for real-time projects ranging from embedded systems to web-based, data collection applications. I wish I had this book three years ago when we began a real-time, embedded drilling control system project, but all my engineers will be getting copies now that it is available. And, for my academic colleagues, this is the perfect book for graduate seminars in applied software development techniques."

*—Don Shafer, chief technology officer, Athens Group;*
*adjunct professor, Cockrell School of Engineering,*
*The University of Texas at Austin*

"We have used Dr. Douglass's books on real-time (*Doing Hard Time, Real-Time UML,* and *Real-Time Design Patterns*) for years. His books are always informative, accessible, and entertaining. *Real-Time Agility* continues that tradition, and I can't wait to introduce it to my colleagues."

*—Chris Talbott, principal software designer*

"Until now, agile software development has been mostly applied within the IT domain. This book breaks new ground by showing how to successfully traverse the perceived chasm between agility and real-time development. Although embedded systems impose challenging constraints on development teams, you can always benefit from increasing your agility."

*—Scott W. Ambler, chief methodologist/Agile, IBM Rational;*
*author of* Agile Modeling

that subsequent plans will be more accurate than earlier ones. Uncertainty can be accounted for in several ways. One way is to allow the end date to vary. With this approach, the computed end date has an associated expected range of variation that diminishes as the end date approaches. Alternatively, you can fix the end date but vary the delivered functionality. This is often captured by identifying features as either required or "upside" (to be delivered if there is adequate time). Another approach is to vary the effort applied to the work, either as a percentage of available effort or by adding resources. What you cannot do—although many managers repeatedly try—is to fix all three aspects simultaneously.

### *Replanning Often*

Because you learn from project execution, you can use that increasing knowledge to make increasingly accurate plans. This is known as **replanning.** In the Harmony/ESW process, this replanning occurs at a specific point in the microcycle, although intermediate replanning can be done more frequently if desired. The microcycle contains a short activity known as the increment review or, more popularly, the "party phase." During this phase, one of the things reviewed is the project schedule. A typical microcycle is in the range of four to six weeks but may be as short as two weeks or as long as four months. During this schedule review the team looks at the project accomplishments against what was planned, computes the project velocity, and replans to take actual project progress into account.

## Minimize Overall Complexity

As long as requirements are met, simpler is better. Simpler requires less effort to create. Simpler is easier to get right. Simpler is more understandable. Simpler is more maintainable. Of course, having said that, we spend a lot of time developing systems that have quite complex functionality. One of the things we want from our designs is that they be as simple as possible, but no simpler than that.

As a general rule, we can have complexity in the large (architecture or collaboration level) or in the small (class, function, or data structure level). In some cases, we improve overall simplicity by making individual elements slightly more complex because it greatly simplifies the interaction of those elements. Or, in other cases, we improve overall simplicity by simplifying some very complex elements but at a cost of increasing the number of elements and their relations in the collaboration.

Complexity comes in two kinds. First, there is **inherent complexity.** This is the complexity that results from the fundamental nature of the feature being developed. Without changing the feature, you are stuck with the inherent complexity. The second kind is **incidental complexity.** This is the complexity resulting from the idioms, patterns, and technologies chosen to realize the feature. This latter kind of complexity is under your control as a developer. You can trade off complexity in the small for complexity in the large in your effort to minimize the overall complexity.

For example, consider the collaboration shown in Figure 3.22. It contains a class `Vaporizer` for delivering anesthetic drugs and has the complex state machine shown in Figure 3.23. Note that the structural diagram is simple because the elements within it are complex.

We can simplify the `Vaporizer` class at the expense of complicating the collaboration by extracting either composite or and-states within the class and making them separate classes. Figure 3.24 shows the resulting more complex collaboration. Note the icon in the upper right-hand corner of various classes; this indicates that the class has a state machine. Portions of the state machine from the original `Vaporizer` class now populate these various classes. Each class

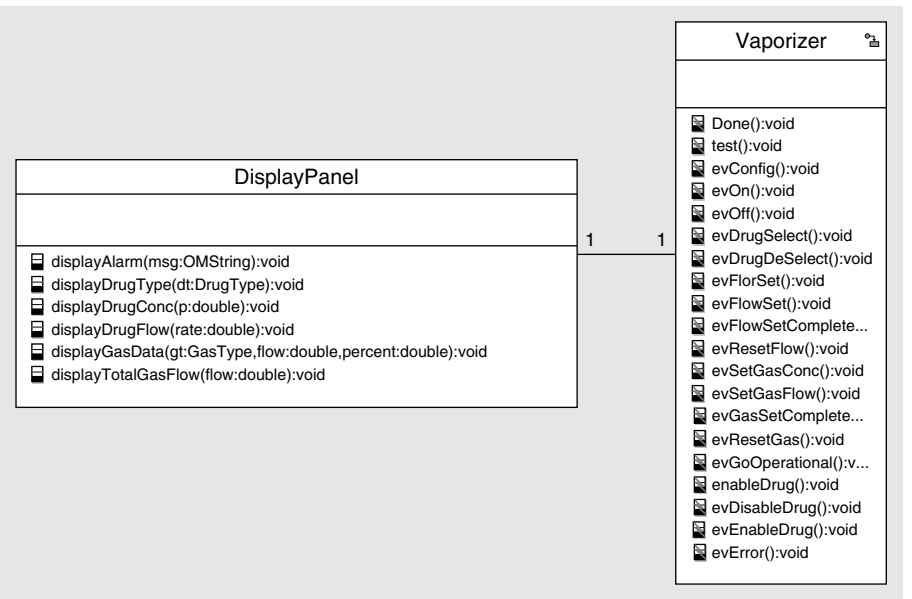



**Figure 3.22** *Sample collaboration*

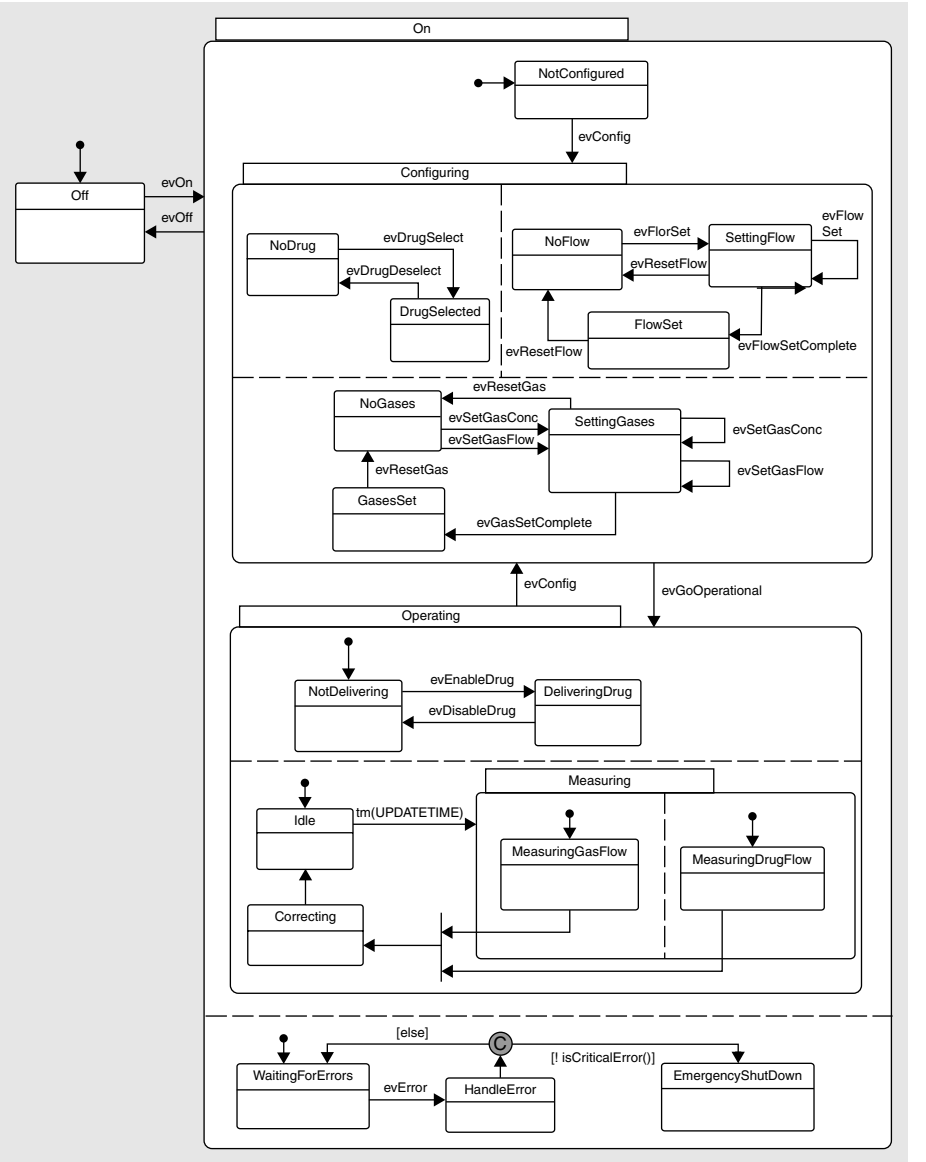



**Figure 3.23** *Class* `Vaporizer` *state machine*

in the collaboration has a narrower focus than the previous `Vaporizer` class, but the overall collaboration is more complex. Figure 3.25 shows the much simpler state machine for the `SimplifiedVaporizer` class. The other state machines are of a similar complexity.

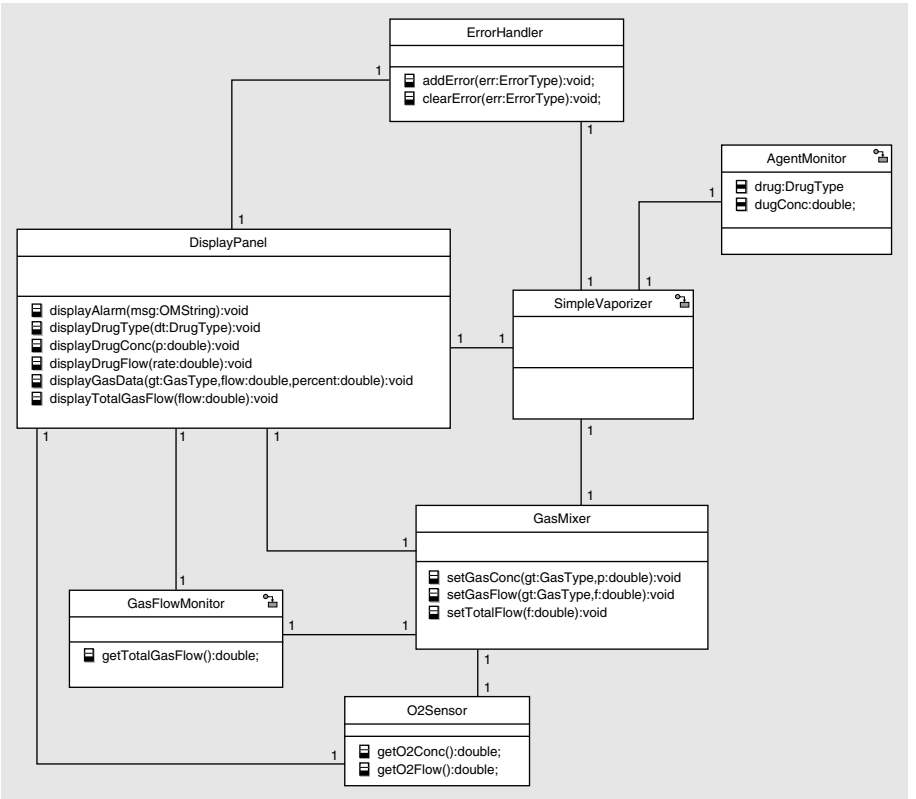



**Figure 3.24** *Elaborated collaboration*

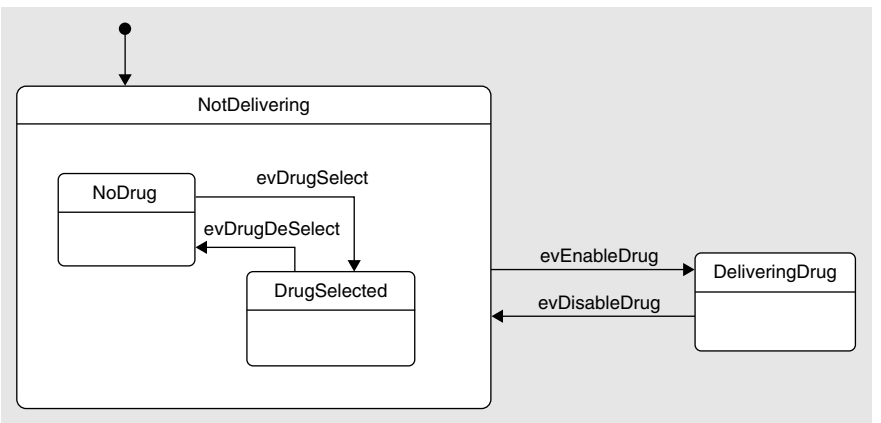



**Figure 3.25** *State machine for class* `SimplifiedVaporizer`

## Model with a Purpose

A model is always a simplification of the thing in the real world that it represents. As George Box says, "All models are wrong but some are useful."[25] For a model to be useful, it must represent the aspects of the elements relevant to the purpose of the model. In addition, we need to represent aspects of that model in views (diagrams) that present that information in a fashion that is understandable and usable for our purposes.

One of the time-honored (and fairly useless) idioms for creating diagrams is to invoke the $7 \pm 2$ rule. The idea is adapted from neurolinguistics—we know that human short-term memory can hold $7 \pm 2$ things. In the seventies, someone misinterpreted this to mean that data flow diagrams should have no more than seven things on them, and if there are more than seven things, the right thing to do is to create a hierarchy.

This is, of course, stupid[26] and has created countless examples of extremely hard-to-follow diagram structures. The Harmony/ESW process recommends an entirely different practice:

> Each diagram should have a single key principle or aspect it is trying to represent; the diagram will contain all elements contributing to that principle or aspect but no more. This mission should be explicitly stated on the diagram unless obvious.

The mission for some diagrams—such as state charts—is obvious (specify the behavior of a `Classifier`) but for the other diagrams it is not. Some common missions for class diagrams include the following:

- *Show the concurrency architecture*—what tasks are in the system, what resources they share, and how they interact.
- *Show the distribution architecture*—how objects are split across address spaces, how they communicate and collaborate.
- *Show the deployment architecture*—how objects map to hardware.
- *Show the safety and reliability architecture*—how redundancy is managed to make the system robust in the presence of faults.
- *Show the subsystem architecture*—the large-scale pieces of the system and how they interact.

---

25. George Box, "Capability Maturity Model Integration (CMMI®) Version 1.2 Overview" (2007), at www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview07.pdf.

26. Not that I have an opinion or anything! ☺

- *Show the interfaces supported and required* by a subsystem.
- *Show a class taxonomy*—the generalization taxonomy of a related set of classes.
- *Show a collaboration of roles*—how a set of classes interact to realize a system-level capability.
- *Show the structure of a composite class*—the parts within the structured class.
- *Show the instances and links* at a specific point in time.

Whenever you create a diagram, *you should do so with an explicit purpose*. This mission is usually placed in a comment. I generally place that comment in the upper-left or upper-right corner of the diagram. For example, Figure 3.26 shows the subsystem architecture for an anesthesia machine. Figure 3.27 shows a sequence diagram from the same model that depicts a scenario from one of the use cases. The mission appears in the comment that names and describes the scenario, along with the pre- and postconditions. Figure 3.28 shows a UML package diagram showing the organization of the model. To recap, each diagram should have an explicit purpose, and that purpose should be clearly stated on the diagram.

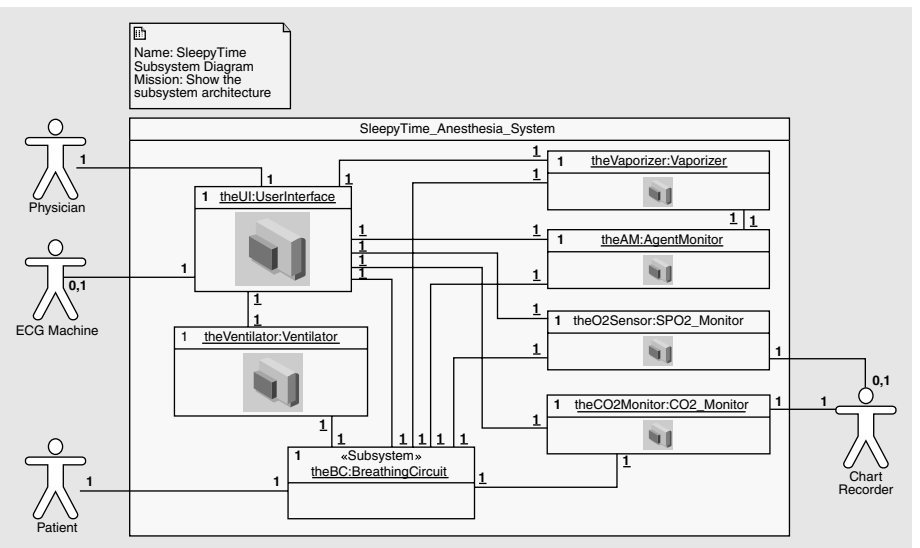



**Figure 3.26** *Subsystem diagram with mission statement*