NASA-CR-197290

The Navigation Toolkit

William F. Rich[*]
Stephen W. Strom

Flight Design and Dynamics Department
Rockwell Space Operations Company, 600 Gemini Ave., Houston, TX 77058
73062.1465@compuserve.com, strom@acm.org
[*]*Current address:* HC65 Box 217A, Alpine, TX 79830

Abstract

This report summarizes the experience of the authors in managing, designing, and implementing an object-oriented applications framework for orbital navigation analysis for the Flight Design and Dynamics Department of the Rockwell Space Operations Company in Houston, in support of the Mission Operations Directorate of NASA's Johnson Space Center. The 8 person year project spanned 1.5 years and produced 30,000 lines of C++ code, replacing 150,000 lines of Fortran/C.

We believe that our experience is important because it represents a "second project" experience and generated real production-quality code — it was not a pilot. The project successfully demonstrated the use of "continuous development" or rapid prototyping techniques. Use of formal methods and executable models contributed to the quality of the code. Keys to the success of the project were a strong architectural vision and highly skilled workers.

This report focuses on process and methodology, and not on a detailed design description of the product. But the true importance of the object-oriented paradigm is its liberation of the developer to focus on the problem rather than the means used to solve the problem.

1. The problem

Navigation is the process of taking measurements and using them to improve the knowledge of the position and velocity of one or more vehicles. The software system we were to build for analysis purposes had to be able to model the dynamics of physical systems, and simulate as well as process measurements from various sensors. The current system comprises 300,000 lines of mixed Fortran and C. In this first increment, we decided to replace approximately half of this code with a completely re-engineered system written in C++.

2. Our solution

There is no "right" way to do any particular project, and there is certainly no single way to do all projects. Indeed, the means must be determined by the end. However, we believe the methodology and process we used have shown themselves to be highly successful in our domain, with our people.

2.1. Methodology

At the outset, we were most heavily influenced by Booch, though we tried to remain goal-oriented and not become "methodology slaves." The primary changes we made were heavier use of modeling and formal methods.

2.1.1. Language choices

We chose C++ as the language to implement the

(NASA-CR-197290) THE NAVIGATION TOOLKIT (Rockwell Space Operations Co.) 7 p

N95-22161

Unclas

G3/04 0039782

final applications. Despite its general excellence, however, three problems plagued us: Strong typing, usually a blessing but sometimes causing us to write more code (possibly introducing more errors than it prevented); programmer-supplied memory-management; and the lack of a *good* macro facility.

We also felt that a very-high-level language for modeling would be useful, primarily in support of requirements development. We chose Common Lisp, with the Common Lisp Object System (CLOS), as our modeling language for several reasons: It supports many programming paradigms, including object-oriented programming; it is (relatively) efficient; the implementation of Common Lisp we used (Macintosh Common Lisp) had a remarkably small footprint, allowing it to run on the 4 MB PowerBook we used for much of our modeling work; we had a long acquaintance and high comfort level with Lisp, particularly for object-oriented programming (Strom 1986); it is covered by an ANSI standard.

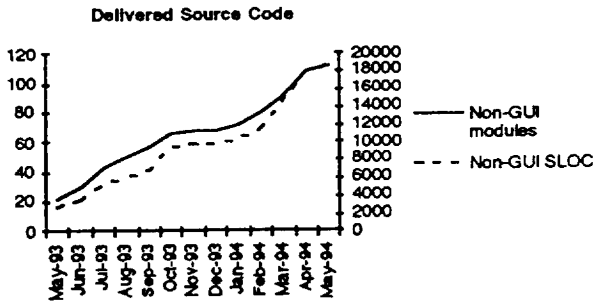
2.1.2. Domain analysis — steal but formalize

We were able to reuse much of the documentation on the existing system, primarily because of the relatively clean division that was maintained between "engineering" and "programming" documentation. The more fundamental analysis, however, was more difficult. This included the creation of classes describing space vehicles, the forces acting on them, transformations between reference frames, etc. Most advanced textbooks on classical mechanics (e.g., Goldstein 1980) take these concepts for granted. We therefore used a modern introductory text (Hestenes 1986) as the foundation for this analysis. We used algebraic specification techniques to capture the results of this domain analysis (as suggested in Srinivas 1990) and recorded them in the software requirements specification.

2.1.3. Rapid prototyping

From the outset we were convinced of the need to verify the integrity of the architecture with working prototypes. We were also convinced that, if the change processes were controlled correctly, these prototypes did not have to be disposable. We could achieve evolutionary development if all subprocesses contributed to the ease of rapid prototyping. For example, configuration management facilitated the change process, rather than constricting it. We tracked, rather than restricted, the changes to our software.

Rapid prototyping lowers the overall risk to the funding organization by providing almost immediate payback in the form of executable code. Here is a plot of the number of ultimately *delivered* modules and lines of non-user interface code for our project, as a function of time:



2.1.4. The role of abstract data type (formal) specifications

We previously worked in C with abstract data types. This experience led us to start this new project by focusing on structure. This approach was insufficient to properly describe the desired behavior of a class, particularly under inheritance. For example, when we introduced forces, we wanted to be able to express the following design constraint: $\vec{F} = m\vec{a}$. Structural descriptions could not do this. We turned, therefore, to abstract data type, or algebraic, specifications. The power of formal specifications to describe the interface (including behavior) of a class became immediately apparent.

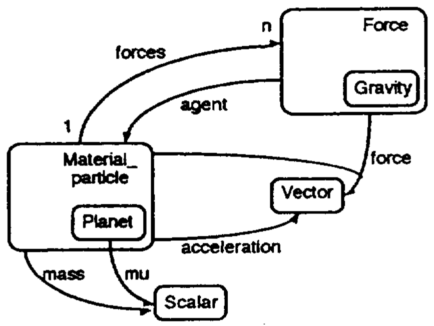
Here is part of the current interface to the classes Particle and Material_particle:

```
class Particle {
public:
        Vector position_wrt(const Body&);
        Vector velocity_wrt(const Body&);
        Vector acceleration_wrt(const Body&);
protected:
        virtual Vector position();
        virtual Vector velocity();
        virtual Vector acceleration();
};

class Material_particle : public Particle {
public:
        Vector sum_of_forces(const Body&);
        double mass();
protected:
        virtual vector acceleration();
};

// For all m in Material_particle and
// inertial reference frames b:
//      m.acceleration_wrt(b) ==
//      m.sum_of_forces(b) / m.mass()
```

We had worried that algebraic specifications might be "too abstract" for users and developers. These fears proved to be unfounded.

2.1.5. A rigorous definition of software architecture and detailed design

Consider the following apparently plausible design for a Force class, based on Hestenes 1986 (we use Harrel's higraph extension of Venn diagrams, see Harrel 1988):



Planet, as a subclass of Material_particle, can function as an agent. When Gravity is asked to compute the force on a material particle, it uses Newton's law of universal gravitation, the mass of the material particle, and the gravitational parameter μ stored in the agent (here a Planet).

There is only one problem with this "design" — *It cannot be implemented in C++!* Static typing prevents the Gravity force from seeing the agent's μ. *This diagram should not be considered to be a "bad" design — it is simply not a design at all (for implementation in C++).* (It could, however, be a design for a dynamically-typed language such as Smalltalk, CLOS, or Dylan.) (A related problem is that other forces, e.g., drag, may require properties of the particle being acted upon besides mass.)

The tendency to postpone the greatest risk, namely, the software architecture, leads us to propose the following definition of software architecture — **Software architecture is a description, in the implementation language, of the interfaces between the software components.** This definition has several advantages: the interface can be compiled, providing a rigorous test for syntactic compatibility of the interfaces; it addresses the greatest risk, i.e., implementability of the software architecture, early in the project. The software detailed design *is* the code. In accordance with IEEE Std 1016-1987, the design specification presents views of this design. Rapid prototyping is increasingly detailed elucidation of the software design.

2.2. Process

The process we created for development of the Navigation Toolkit was driven by the problem we had to solve and the people we had to solve it. We held to the maxim that "Processes don't write software — people write software." Our intent was to balance the need to give our people the freedom to develop good solutions against the need to continuously monitor the progress of the project.

2.2.1. *Team organization*

We organized the team along orthogonal Work-type (or W-type) and Application-type (or A-type) lines (Swanson and Beath, 1990). The W-type organization followed Booch 1994, Brooks 1975, and Stroustrup 1991. None of these roles was a full-time position. Instead, each team member was primarily a programmer. Most of the classes to be developed required considerable technical expertise, requiring the additional A-type organization.

Before initial delivery, the resulting team looked like this:

Education	Experience (yrs)	Principal process role	Application domain						
			Measurements	Integrators	Environment	Filter	Utilities	Programs	GUI
Doctorate	17	Architect	●			●	●	●	
Master's	28	Class designer				●	●		
Master's	21	Configuration manager					●	●	●
Master's	13	Project manager	●	●	●		●	●	
Master's	8	Subsystem lead							●
Bachelor's	5	Application engineer					●	●	
Bachelor's	4	Application engineer			●		●		●

2.2.2. *Macro and micro models*

The macro process model we adopted is the spiral model (Boehm 1988). The spiral model is risk driven and incorporates prototyping as a fundamental component. It provides a rich set of project milestones and supporting documentation. We modeled the micro process with Meyer's cluster model, in which a set of staggered waterfalls describes the development of "clusters" (groups of closely related classes).

3. Assessment

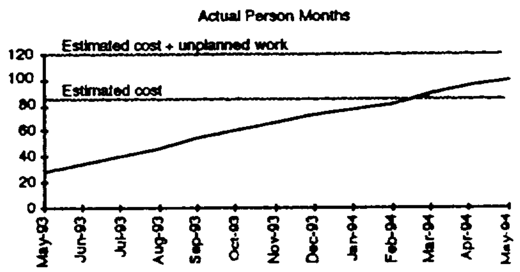
Time to look back, to assess (sometimes painfully) how well the project went.

3.1. Cost

How well did we do in predicting the course of the project? Here is a comparison of our predictions and the delivered lines of non-UI code:

Class category	Predicted SLOC	Actual SLOC
1. Measurements	3000	1473
2. Integrators	1000	762
3. Environment	4000	4434
4. Filter	4000	957
5. Utilities	2000	2389
6. Programs	5000	8148
Total	19000	18163

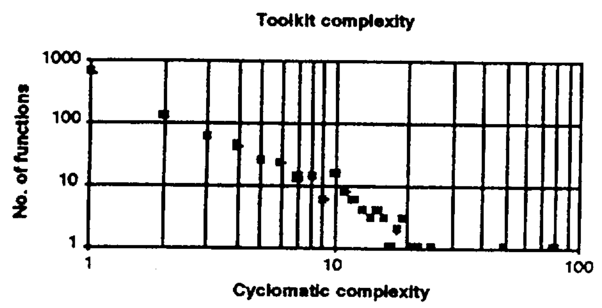
We also estimated that there would be 11,000 lines of UI code, or 30,000 lines of source code in all. Simple COCOMO, organic mode (Boehm 1981), predicted 85 person months of work. The actual cost of the project as a function of time is shown below:



We exceeded our cost estimates. Most of this is attributable to unplanned work in late 1993, associated with coordination with another company project. We expended 35 person months on this effort. Excluding this unplanned work, the cost of our project was extremely close to our original projection. This implies that COCOMO is a reasonably valid cost model for object-oriented projects.

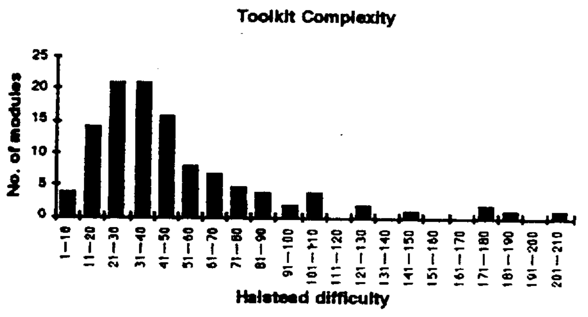
3.2. Quality metrics

Here is the cyclomatic complexity (metric 16 in IEEE Std 982.1-1988) of the functions that comprise the Toolkit:



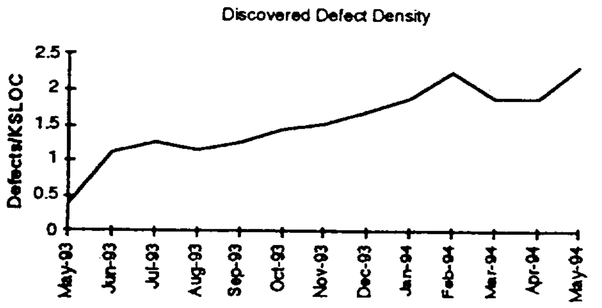
There was no observable correlation between cyclomatic complexity and defect density in our code.

Halstead's complexity metrics (metric 14 in IEEE Std 982.1-1988), derived from information-theoretical concerns, appear to have more utility for our code. Here is a histogram of the Halstead difficulty of the Toolkit modules:



The modules with higher Halstead difficulty turned out to be those which had been extensively optimized, and have exhibited a higher number of defects than modules with lower Halstead difficulty.

Defects are usually tracked beginning with the completion of integration testing. We began tracking defects following unit test to demonstrate that the code that emerged from unit testing was of production quality. This contention is born out by the density of discovered defects (metric 2 in IEEE Std 982.1-1988):



It seems likely that the defect density will stabilize at well under 5 defects per KSLOC. Again, it must be emphasized that this is counting defects *following unit test.* Rational has reported a defect density of 2.21 defects per KSLOC for the Beta 1 iteration of their Rose CASE tool (Walsh 1992). The quality of our code, measured in defect density, is on a par with the best industry standards.

4. Acknowledgments

This work was performed under NASA contract NAS 9-18000.

We acknowledge the hard work and dedication of the following employees of Rockwell and Unisys who at various times were members of the Navigation Toolkit team: Gene Brownd, Kevin Buie, Francis Cerbins, Mick Chang, Lynn Keller, Steve Labar, and Jigesh Saheba. We also thank the following users who were able to break our code in the most ingenious ways: Ed Brown and Carolyn Propst; interested colleagues from other areas who variously contributed time, ideas, valued criticism, encouragement, and equipment: Walt Pace, Matthew Verona, and George Wu; and the members of our management who actually trusted us to do what we said we could do: Gene Brownd and Bill Heilman.

5. References

Boehm, Barry W. 1981. *Software engineering* *economics*. Englewood Cliffs, New Jersey. Prentice-Hall.

Boehm, Barry W. 1988. A spiral model of software development and enhancement. *Computer* 21(5): 61-72.

Booch, Grady. 1994. *Object-oriented analysis and design with applications*, 2d ed. Redwood City, California: Benjamin/Cummings.

Brooks, Frederick P., Jr. *The mythical man-month: essays on software engineering*. Reading, Massachusetts: Addison-Wesley.

Brownd, J. E. 1992a. Post FADS software requirements for navigation. Flight Design and Dynamics Department, Rockwell Space Operations Company.

Brownd, J. E. 1992b. Navigation software status and actions, 11/6/92. Flight Design and Dynamics Department, Rockwell Space Operations Company.

Goldstein, Herbert. 1980. *Classical mechanics*, 2d ed. Reading, Massachusetts: Addison-Wesley.

Harrel, David. 1988. "On Visual Formalisms," *Communications of the ACM* 31, no. 5 (1988): 514-530.

Henderson, Peter. 1993. *Object-oriented specification and design with C++*. Maidenhead, Berkshire, England: McGraw-Hill.

Hestenes, David. 1986. *New foundations for classical mechanics*. Dordrecht: Kluwer.

IEEE Std 982.1-1988. IEEE standard dictionary of measures to produce reliable software. Piscataway, New Jersey: IEEE Press.

IEEE Std 1016-1987. IEEE recommended practice for software design descriptions. Piscataway, New Jersey: IEEE Press.

Meyer, Bertrand. 1988. *Object-oriented software construction*. Hemel Hempstead, Hertfordshire: Prentice-Hall.

Prieto-Díaz, Rubén and Guillermo Arango. 1991. *Domain analysis and software systems modeling*. Los Alamitos, California: IEEE Computer Society Press.

Srinivas, Y. V. Algebraic specifications for domains. Technical report ASE-RTP-102, Department of Information and Computer Science, University of California. Reprinted in Prieto-Díaz and Arango 1991.

Strom, S. W. 1986. Object-Oriented Programming in Lisp. Houston: TRW Defense Systems Group.

Swanson, E. Burton and Cynthia Mathis Beath. 1990. Departmentalization in software development and maintenance. *Communications of the ACM* 33(6): 658-667.

Walsh, James F. 1992. Preliminary defect data from the iterative development of a large C++ program. In *OOPSLA '92 conference proceedings*, edited by Andreas Paepcke. New York: ACM Press.