2000

Contributions of Object Oriented Software Design Towards Limiting the Problems Caused by a Lack of Software Engineering

Jennifer Bevan
University of California - Santa Cruz, jbevan@cse.ucsc.edu

Contributions of Object Oriented Software Design towards Limiting the Problems Caused by a Lack of Software Engineering

Jennifer Bevan, Department of Computer Engineering, University of California at Santa Cruz
jbevan@cse.ucsc.edu

Abstract

The Radio Science Validation and Processing (RSVP) software suite was created to replace an outdated and disorganized set of legacy software. The development process applied no formal software engineering methods, but the initial phase did employ a rudimentary object-oriented approach. The modules developed during this initial phase were later recognized to be the most stable and easily maintained portion of the resulting software suite. This paper discusses some of the problems frequently caused by a lack of software engineering, and examines how an object-oriented foundation mitigated the effects of these problems on the overall quality of the RSVP project.

Introduction

While the benefits of software engineering are no longer considered to be strictly hypothetical quality improvements (Paulk, et al., 1997), the consistant use of software engineering principles is still far from widespread. While the number of software engineering baccalaureate programs worldwide increased from none in 1994 to dozens in 1999 (Hew, et al., 1999), the creation of a standard regarding the requirements of such a degree is still in progress. On the other hand, object-oriented languages are slowly becoming the rule rather than the exception in the required programming courses for computer science baccalaureate degrees. This education is creating a growing pool of programmers in the workforce who are somewhat familiar with the basics of object-oriented design (Parnas, 1995), yet who are not familiar with software engineering principles. Given the obsession with lowering a product's time-to-market, if it seems possible to create that product using familiar techniques, commercial developers will use what they know. New learning generally occurs only when a sufficient benefit is clearly shown to, and approved by, project management. While this increase in the use of object-oriented designs provides many benefits in software development, it is not a replacement for software engineering.

However, the use of object-oriented principles in software design is certainly not mutually exclusive with the use of software engineering principles, and in fact helps to ensure that some basic software engineering issues are addressed. This report identifies common problems incurred by the lack of software engineering, discusses the aspects of object-oriented design which overlap software engineering concerns, and examines how an object-oriented foundation mitigated the effects of these problems on the overall quality of a specific scientific data processing tool.

Development Overview

The development of the Radio Science Validation and Processing (RSVP) software suite was divided into three phases, the first of which was to allow the extraction of scientific data from any of several input formats, including two that had not been previously used by the Radio Science Team. The development team for the first phase involved a single programmer for the new code, and two programmers dedicated to porting the legacy fortran code from the old hardware to the new. This porting involved the removal of code that expected the presence of an array processor, as well as the selection of which versions of which functional modules were to be ported. The legacy code had evolved over twenty years in an extremely spaghetti-like fashion, and no single person or document could identify what each version of each module was supposed to do "differently" than the other versions. After the legacy code was selected and ported, the development team was reduced to only the single programmer who had been writing the new code, working at less than half-time. It is this new code that constitues the RSVP "core code". While the ported legacy software was updated during later phases, the use of this legacy-encapsulation approach (Bennet, 1997) during RSVP development did restrict the design of the core code; each functional module that transformed the data had to output a format identical to the existing input format of the next module. This approach did not allow for an object-oriented design for the entire project, and in fact, only the core code can be considered truly object-oriented.

In 1994, there were 11 input formats that the software was required to process. By 1999, this number had grown to 14. Within these formats, data of the same *data type* (such as received frequency, time-stamp, or other format elements) were not always represented in identical units or encodings. The developer for the RSVP core code was not familiar with any sort of formalized object-oriented design process (Booch, 1991) nor with software

engineering principles. In early 1994, after two days of attempting to force these data formats into the *struct* constructs of C, she realized that an object-oriented approach was probably a better way to go, and started over. Less than a year later, she had a working initial version that could handle all of her test data sets correctly. The transition to real data was more problematic, as unexpected deviations from the expected format were discovered one by one. By 1996, the core code was handling real data, could output data readable by the legacy code, and had already undergone a new data format addition; work then started on incorporating the legacy code hooks into the GUI front-end used by RSVP. In May, 1997, the first official release of RSVP and the ported legacy software occurred (Bevan, 1997), containing approximately 24,000 lines of new code, and implementing classes for 13 different data formats. Later versions added another data format class, and added in rewritten and reorganized replacement modules for the legacy software. The latest release is RSVP 3.1 (Bevan, 1999), containing approximately 59,000 lines of code.

RSVP Architecture

The two main class heirarchies in RSVP are implemented as libraries, so that other scientists may more easily adapt their own analysis software to accept new datafile formats by reusing the extraction methods within RSVP. These heirarchies are both single inheritance, single-depth trees with a basic container class providing both virtual methods for the subclasses, and generic methods for subclass-independent data manipulation. The "main" program of RSVP gets user input to determine what type of data is being input, instantiates the appropriate classes, and then hands control over to the GUI's select-type loop. Figure 1 shows a high-level diagram of this control path.

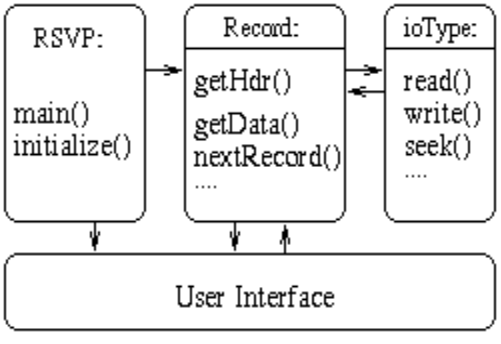

Figure1: RSVP High-Level Control Path

The first generic container class used by RSVP is the *Record* class, which holds all of the common data across the different formats, as well as data search functions. Format-specific classes inherit from this *Record* class and add the necessary data types and flags to correctly control the processing control and output. Virtual functions in the *Record* class lead to format-specific subroutines in the appropriate class, subroutines which generally consist of bit-level unpacking of data formats. Figure 2 shows a small subset of the classes currently used in the *Record* heirarchy.

The second generic container class in RSVP is the *ioType* class, a base class for multiple media-specific subclasses, in order to allow the access methods of half-inch magnetic tapes, 8mm magnetic tapes, CD/ROMs, and disk files to be isolated from the rest of the code. While the *ioType* class was being created, additional type-specific classes were added to handle byte streams and arrays. The structure of the *ioType* class heirarchy is identical to that of the *Record* class heirarchy. The virtual methods provided by the *ioType* class are *read, write, seek, tell, open,* and *close.*

The motivation behind structuring this portion of RSVP in this manner arose primarily from the volatility in data file formats. Specific issues related to the design of RSVP can be found in later sections.

Common Problems Involved with Informal Product Development

The purpose of software engineering is essentially to provide methods by which the probability of creating the right product, in the best possible way, is maximized. Checkpoints are inserted into well-defined developmental

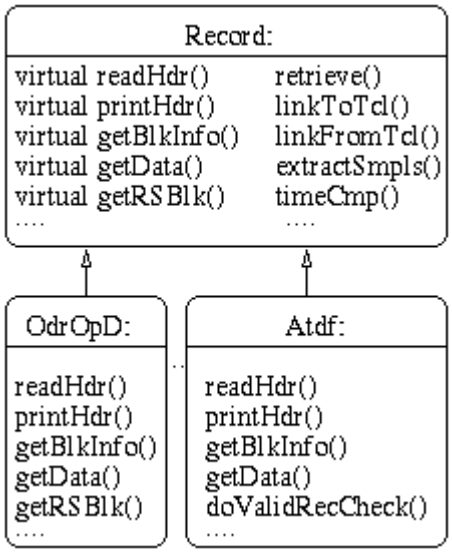

Figure 2: *Record* Class Heirarchy

stages which attempt to discover bugs and other "features" as early as possible, thereby reducing the number of problems found after product deployment. Software engineering methods encourage the development of software that is easily upgraded or otherwise maintained. In other words, the goal is to produce software with as much thoughtful planning and execution as is commonly applied in other engineering disciplines (Bauer, 1997).

Some common problems which frequently result from a lack of such formal development techniques include an incomplete or incorrect understanding of the product requirements and designs that, even if formally established, are not flexible in the direction of the most likely upgrade path. Additionally, implementations are created that require system-wide changes for maintenance or upgrade modifications; these changes are frequently of the type that cause unintended side effects in unexpected parts of the code. While software engineering does not guarantee an error-free product development cycle, it certainly does provide a framework which aids developers in avoiding these problems.

Software Engineering Issues Addressed by Object-Oriented Design

Object-oriented design, while not a process that can be applied to a product's entire life-cycle, certainly addresses some of the same concerns as software engineering. The re-classification of a problem domain into objects enforces a fairly in-depth examination of the requirements of a project. The resulting object-oriented models create an understanding of the behavior of the system (Booch, et al., 1999), including the volatile functional- and object-based upgrade paths. Object-oriented implementation is more tightly connected with a verified design, and the software engineering metric of high cohesion is analogous to the object-oriented design goal of encapsulation.

Object-Oriented Design Principles and RSVP

The object-oriented approach in the development of RSVP addressed several software engineering aspects. Each of these aspects, and the extent to which this approach achieved the goals of the aspects, is discussed in turn.

Requirements Engineering

Given the dynamic nature of the set of input formats that RSVP would be required to process, the creation of an object-based representation ensured that the relationship between the data-processing software and the input data was thoroughly examined. This process uncovered the fact that the same nominal data type across different formats might have different encodings, use different units of measurement, or allow a different range of valid data values. It also identified exactly which data types were required by the processing, in what format they had to be presented to the processing software, and in which data formats this data would appear. Links between the input format and the allowable set of processing functions could then be created without modification of actual data processing code, which for the majority of the development process remained legacy code. Data value or format modification could be done in the format-specific subclass before the extracted data were presented to the rest of the code.

Design

In order to encapsulate all format-specific issues, such as bit-packing formats and units of measure, the *Record* class was designed to be the only class that allows access to the input data. High-level code can only ask for certain types of data, which are obtained through *Record* class methods. This level of information hiding was specifically designed to make the addition of new data file format subclasses easier. The *ioType* class was created for the same purpose (the use of new media was an expected upgrade path), and used the same heirarchy structure as the *Record* class. The inheritance depth of RSVP is only one, between these parents and their specific subclasses: the project was not objectified to the extent that an encompassing *DataProcessing* class was created.

Implementation

The *Record* class, as the central access point for all input data, publicly owns the most common data types (e.g. spacecraft identifiers, antenna and ground system identifiers, time-stamps). Format-specific data types, where either the data type itself was not common or where the representation was not format-unique, were isolated in format-specific subclasses. The *Record* class provided the means to traverse the data records and the procedures to output the data to the scientific processing programs, while the format-specific subclasses manipulated the data types to provide a uniform representation and generic valid-data flags.

The addition of new input format subclasses does not affect any existing code, with the exception of a single localized modification to the *Record* class initialization procedure. Because any instance of a *Record* has the same interface, the high-level code to manipulate data records did not need to change over the later, updated, versions of RSVP. Similarly, new subclasses of i*oType* did not affect the high-level code, for exactly the same reasons.

Object-Oriented Induced Quality Aspects of RSVP

The development of RSVP is currently considered complete, with respect to the original project goals. While it is certainly not a perfect product, especially with respect to mathematical problems in specific scientific data processing stages, the overall stability and flexibility of the software is very satisfactory. New input media and data formats are not a cause for concern to the scientists using the product: they have worked through several format additions and found no adverse effects unintentionally introduced. The core code has not changed beyond the addition of initialization routines for new subclasses. Modifications to the data-extraction code does not affect the data-processing code, and vice versa. While there are certainly some quality aspects that a formal software engineering process would have improved, those aspects affected by the object-oriented design process are recognized by the users as of high quality.

Given that there are very few specific, universally applicable standards for software quality, the best source of quality information is from the users. The quality of RSVP is generally divided into three categories by its users. The first is the ability to quickly adapt to new formats with respect to simple data-extraction and reformatting issues. The second is the ability to protect working portions of the project through numerous maintenance-level changes. The third is mathematical correctness. The first two issues were directly affected by the use of object-oriented design in the initial phase of development. The third issue, on the other hand, was not covered by object-oriented design techniques; the legacy code was converted in later development phases, and due to a lack of formal verification and validation methods, suffered during the translation.

Simple Modification to Include New Formats

The simple, yet very structured design of single-point data access via the *Record* class allows for new formats to be added easily and quickly. The initial step, that of creating the format-specific subclass, generally took only a few hours after the new format documentation was provided to the developer. If verification of the new subclass with test or actual data uncovered any implementation-level errors, they could generally be fixed in a matter of minutes or hours. Because this stage of the data processing generally involves only data extraction and necessary modifications to conform to the uniform data type representations, all such errors could only occur within the subclass definition. This localization of potential errors greatly increased the speed with which bugs could be found and fixed.

Localization of Code Modification Effects

Because RSVP was developed in a three planned phases, changes to each phase were incorporated with the expected work of the next phase. The deployment environment was limited to a released version and a beta version, which was essentially the developer's copy with minor privilege restrictions. Scientists who had requested a modification could, if necessary, only wait for as long as it took the developer to implement that single modification in the beta tree, then run that version to see if the modification made a difference to the processed data. During these well-defined yet very informal maintenance phases, it was crucial that a modification not be able to affect calculations in unintended parts of the code.

The object-oriented structure of the code, which inherently promotes the low coupling and high cohesion principles, increased the quality of RSVP in this respect. Modifications to the transformation routines of format specific data types were forcibly restricted, affecting only that subclass. Modifications to one portion of the data processing code, while not created under an object oriented paradigm, were nonetheless called on object oriented data. This shielding allowed the scientists to get the earliest possible returns on modification requests without worry that a given change could affect the data outside the expected modification "effect field".

Conclusions

The creation of RSVP was the work of a single developer who, like many in the workforce now, had neither heard of software engineering nor understood its principles and yet was considered a "good" programmer. As is typical in situations where there is a push to produce working code in a short time frame (in this case due to the unreliable nature of the hardware on which the legacy code could run), the programmer applied the principles she did understand, those of basic object oriented design. While the overall quality and acceptance of the RSVP package by its users is considered good, certain important aspects were overlooked due to the total absense of formal software engineering techniques. However, the quality of RSVP is not accidentally, nor coincidentally, good. The object-oriented design used in the initial phase, where the legacy code had no impact on the new code except in determining output formats, created the most stable and reliable portion of the entire RSVP package. While this design process did not address all of the issues which should have been addressed, the quality of RSVP is almost certainly much higher than it would have been without the benefit of an object-oriented foundation.

References

Bauer, F. "Forward: Software engineering-a european perspective.", *Software Engineering*, 1993, M. Dorfman and R. H. Thayer, IEEE Computer Society Press, 1997. pp. 75.

Bevan, J. *RSVP 2.1 Release Pages*, http://radioscience.jpl.nasa.gov/info/software/rsvp2.1/, 1997

Bevan, J. *RSVP 3.1 Release Pages*, http://radioscience.jpl.nasa.gov/info/software/rsvp.3.1/, 1999

Bennett, K. "Software Maintenance: A Tutorial", *Software Engineering*, 1997, M. Dorfman and R. H. Thayer, IEEE Computer Society Press, 1997, pp. 289-303.

Booch, G. *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991, pp. v-vii.

Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modeling Language User Guide*, Addison-Wesley, Object Technology Series, 1999, pp. 3-11.

Hew, D., Sinderson, E., and L. Spirkovska. "The state of software engineering: Body of knowledge, education, certification, and licencing.", Final Project, Software Engineering Graduate Course CMPE276, Fall 1999, University of California Santa Cruz, Nov. 1999.

Parnas, D., to Brooks, F. in "No Silver Bullet, Refired", *Software Engineering*, 1997, M. Dorfman and R. H. Thayer, IEEE Computer Society Press, 1997, pp. 221.

Paulk, M., Curtis, B., Chrissis, M., and Weber, C. "Capability maturity model for software.", *Software Engineering*, 1997, M. Dorfman and R. H. Thayer, IEEE Computer Society Press, 1997, pp. 427-438.