A CHECKLIST FOR SOFTWARE REUSE

Terry W. Jackson
U. S. Nuclear Regulatory Commission
MS: T10-L1, Washington, D.C. 20555
Phone: (301) 415-6486 Fax: (301) 415-5074
Email: twj@nrc.gov

KEYWORDS
Software, Industrial Computing, Quality Improvement, Systems Integration

ABSTRACT

As nuclear power plants conduct analog-to-digital, and some digital-to-digital upgrades, reuse of software can decrease development time/cost and increase software quality. However, improper application of software reuse can lead to digital system failures, even when high-quality, proven software is reused. Many of the reuse-related faults are introduced through interface conflicts with hardware, operators, plant systems, and other software systems and modules. Faults also arise when the impact and limitations of all functions within the reused software are not fully acknowledged. This paper describes the types of software reuse and the ways that reuse-related faults are introduced. It also provides two examples of accidents that have resulted from poor reuse practice. At the end of the paper, a checklist is provided to motivate best practices in software reuse.

INTRODUCTION

Software reuse is the use of *software components* in more than one software system[1]. Software components include software specifications, code, manuals, tests, plans, design rational, and other items that are a product of software development[2]. In many cases, reusable components consist of only parts of the items listed above. For example, a programmer may copy a small portion of source code to a new software system under construction, versus copying the entire source code. In other cases, the developer may reuse the entire software system. Commercial off-the-shelf(COTS) software is an example where the entire software is reused.

Software reuse offers two benefits to developers[3]. One benefit is the reduced development time and resources. In one report, it is estimated that less than 15% of all software is unique to the application[4]. However, only 5% of all code is being reused. Through reuse, developers take advantage of previously developed solutions and focus on the problems for which no solution is available. Not only does software reuse save development time and effort, but it can also raise software quality. If a software system has proven reliability, there is some potential for quality assurance through the reuse of its components.

Despite the benefits, experience has shown that faults are sometimes introduced during software reuse activities[5]. The purpose of this paper to explore the ways in which those faults enter software-based systems, particularly when high-quality, proven software is being reused. By looking at the ways in which faults are introduced, a checklist is created to assist in the execution and acceptance of software reuse activities. While it is possible to reuse specifications, test cases, documentation, and design rational, this paper focuses on the reuse of software implementations (i.e., code, input parameters, and logic). However, treatment of the subject is at a generic level,

---

[1] The views expressed in this paper are solely those of the author and should not be construed as NRC views or positions.

enabling the generalization of reuse principles to other software components. To fully appreciate the introduction of reuse-related faults, it is necessary to look at the different types of software reuse.

I. TYPES OF SOFTWARE REUSE

Software reuse takes on several different forms, as shown in Fig. 1. The most commonly perceived form of software reuse is the case where a programmer copies code from one program and pastes it into the program he or she is developing. For the sake of discussion, this type of software reuse is called *traditional software reuse*. In traditional reuse, the developer looks into previously developed software and identifies those parts that could be reused. Often, this type of reuse is not planned in the software development process, but rather, the programmer sees an opportunity to reuse either their previous work, or that of another programmer[1]. This type of "opportunistic" reuse rarely takes into full consideration the interface, impact, and limitations of the software, which can lead to the problems discussed in Section II. However, a "systematic" approach to traditional software reuse incorporates processes and methods into the software development process to overcome the problems associated with opportunistic software reuse.

Another type of software reuse involves automated means. *Application generators* are becoming popular because they remove much of the complexity and development effort involving large software systems[6]. An application generator is a software tool that creates software code for a particular application[7]. For example, some human-machine interface (HMI) systems contain application generators. To generate a stand-alone HMI software system, the development software provides high-level methods for entering application-specific details. Such high-level methods include data entry/questionnaire forms and graphical modeling techniques. Once the development software obtains details specific to the application, it generates the stand-alone software system. Software common to all HMI systems, such as hardware drivers, graphical user-interfaces, and communication routines, are stored in the tool automatically integrated with other software routines. Thus, the application generator "reuses" these common components, and only needs the developer to supply application-specific details. Other examples of

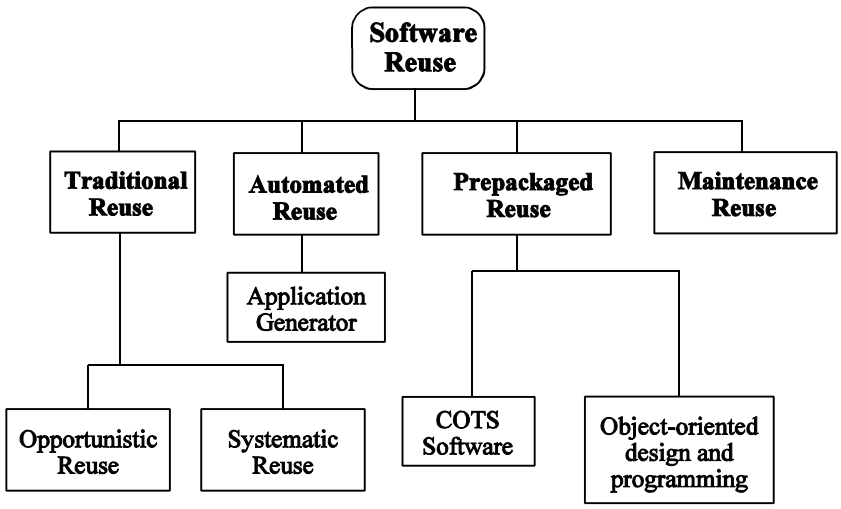

Figure 1. Types of Software Reuse

application generators include programming environments, such as LabVIEW™. These programming environments remove much of the code development (i.e., operating system interface and graphical user interface) from the developer, and require only application-specific detail or code to complete the software system.

Application generators make software development easier by hiding much of the complexity associated with the interface between the operating system and hardware. This same benefit may be a source of potential interface problems, particularly when the configuration of the hardware and operating system is slightly different from the application generator's expectation. However, there does not appear to be any studies which affirm, or deny, the existence of such interface problems.

Besides traditional and automatic means of software reuse, software can be developed specifically for reuse. COTS software provides the same advantages of traditional and automatic software reuse, including reduced development resources and increased software quality[8]. It is evident that much development time is saved if there exists a software package that meets system requirements. Also, many COTS software packages have an appreciable amount of operational experience proving their quality. Another "prepackage" approach to software reuse is object-oriented design and programming[2]. Object-oriented design and programming is a way of developing software by focusing on software objects and the interconnections between those objects[7]. An object consists of data and the operations on that data, along with a well-defined interface. Software systems are then created by "connecting" the objects, which may be reused from previously developed systems.

A fourth type of software reuse includes maintenance activities. One of the major problems in software maintenance is understanding the software and the context with which it operates. For example, if a correction or upgrade is needed, the programmer attempts to make the change without affecting the rest of the software. In effect, the programmer is "reusing" the rest of the software. Whether reuse or maintenance is performed, the programmer is concerned with how the correction, upgrade, or software component interfaces with the rest of the software system.

II. INTRODUCTION OF FAULTS DURING SOFTWARE REUSE

Although it seems logical that software reuse would decrease software costs and increase quality, such expectations are not always met. One reason for the shortfall is the introduction of faults during the software reuse process. Despite the type of software reuse, these faults are generally manifested in the following ways:

- interface conflicts occur between the reused component and other system components,
- incomplete understanding of the capabilities and limitations of the reused component, and
- assuming the reuse of proven, high-quality software automatically makes the new software of high quality

Interface Conflicts

Interfacing reusable software to a new software system is the majority of the effort in software reuse. Interfaces not only include other software, but also include hardware, humans, plant systems, and the environment. A full understanding of the interface is important in software reuse[2,9]. Reused software assumes that its interfacing components provide a particular type of information in a specified amount and pattern. If that information, or its protocol, is different from what the reused software expects, a software failure may occur. Vice-versa, the reused software provides its interfacing components with information in a particular format. If there are differences between the expected information/format and the delivered information/format, a software failure may occur.

LabVIEW is a registered trademark of National Instruments.

There are two main reasons for the introduction of interface-related errors during software reuse[10]. First, programmers try to force the application requirements to fit a structure for which they know a solution, even if the solution fails to satisfy part of the original specification. And second, the programmer is unfamiliar with the interface of other components. To avoid the introduction of interface-related errors, the developers should first identify the application requirements, and then, compare the capabilities, format, etc. of the candidate reusable software against those requirements. Any differences should be noted and evaluated to see if the reusable software can be modified sufficiently to fit all the requirements. Not only should the software-related requirements be analyzed, but also those related to the computer hardware, plant system, and operator. If the reusable software is not sufficiently documented, or it is difficult to understand, it may be better to develop the solution from the beginning instead of attempting reuse. Once a reusable software module is placed in the software system, integration testing of the entire, fully assembled software system is required[3].

Incomplete Understanding of Capabilities and Limitations

Understanding the capabilities and limitations of reusable components is a fundamental practice in all engineering disciplines. Consider computer hardware engineering and its use of integrated circuits(IC). Basically, engineers assemble various ICs together to build the hardware for a computer. The engineers understand the capabilities and limitations of each IC because that information is provided in data sheets. The data sheets describe the interface(pins), basic functions the chip provides, how to access those functions, timing requirements, temperature limitations, and power limitations. In contrast to IC reuse, there is little information provided on the capabilities, limitations, and interface of software. In situations where a programmer reuses previously developed software, he or she may know about its functionality, but not fully appreciate its limitation, impact, and interface. In situations involving COTS software, often the information is gathered from promotional literature, which scarcely covers the interface and limitation issues. Assembling a "data sheet" for reusable software provides a beneficial tool against the introduction of reuse-related faults. The difficulty is determining what information should be included in the data sheet. The checklist at the end of the paper demonstrates some of the information that could be included in a software data sheet.

Assuming Quality Transfer

Simply having operational experience with no previous problems does not automatically imply that reused software will have no problems in its new application[11]. Unless the software is being reused exactly as it was in the previous application, additional reviews and tests should be performed to complete its qualification. Previous tests and operational experience can only clear software errors for the operational and test profile of its previous application. If the new application uses the software in a different fashion, any original errors that have not been discovered by test or operation may arise.

Modification to the reused software is another reason for testing. For many reuse situations, it is necessary to modify the component in order for it to meet design specifications. This is especially true with traditional software reuse and software maintenance. Once software has been modified, it is necessary to re-validate the software to ensure that it meets the new requirements.

While previous tests and operational experience should not be used alone to verify the quality of reused software, it does not mean that previous testing and operational experience is discounted[6]. It merely suggests that one should consider the type of testing and operational experience and how it applies to the new application. The presence of previous testing and operational experience should decrease the amount of testing and review needed for a reusable component.

Examples

Loss of life and property have occurred in the past due to faults related to software reuse. The Ariane 5 self-destruction and the Therac-25 accidents are two examples where reuse-related faults led to system failures.

On June 4, 1996, the maiden flight of the Ariane 5 launcher ended in failure when the launcher veered from its flight path, broke up, and self-destructed[12]. The Ariane 5 was the newest spacecraft launcher developed for the European Space Agency. The accident was caused when the Inertial Reference System (IRS) sent diagnostic data instead of flight data to the flight computer. On the Ariane 5, there are two redundant IRS systems; one operating and the other in hot standby. Due to a common-cause software error, both IRS systems shut down and began to send diagnostic data. When the flight computer read the diagnostic data, it commanded the launcher to veer sharply, causing it to break up and self-destruct. The IRS requirements for the Ariane 5 and its predecessor, the Ariane 4, are essentially the same. Therefore, most of the IRS software was reused. One of the reused software functions caused the common-cause software error. This function involved the computation of horizontal bias (acceleration), and it was not required in the Ariane 5, but kept in order to maintain commonality between the two launchers. The common-cause software error occurred when a 64-bit floating-point number became large enough that it could not be converted to a 16-bit integer. This caused an operand error, forcing both IRS units to shut down. The same error could have occurred in the Ariane 4 except the horizontal bias never got big enough to cause a conversion problem. This example provides several lessons in software reuse. First, reusing proven software does not ensure that it will have the same record of performance in the new application. Second, functions within reused software may impact performance whether or not they are needed. And third, it is important to understand how the reused software interfaces with other systems. In this example, there was no consideration given to the system interface(increased horizontal bias).

Between 1985 and 1987, six known accidents involving massive radiation overdoses were caused by software errors in the Therac-25 medical linear accelerator[11]. The Therac-25 was preceded by the Therac-6 and Therac-20 linear accelerators. The Therac-6 was the first accelerator to have software-based systems, although the system could operate without software assistance. The Therac-20 succeeded the Therac-6, and reused much of its software. Both the Therac-6 and Therac-20 operated without major problems. The Therac-25 succeeded the Therac-20 but more reliance was placed on software in the control and safety systems. Many of the hardware interlocks and safety devices found in the Therac-6 and Therac-20 were replaced by software safety interlocks on the Therac-25. Therac-25 reused much of the software from the Therac-20, and, because of the safety success with the Therac-6 and Therac-20, it was assumed the software was of high quality. In determining the cause of the accidents, it was discovered that one of the software errors in the Therac-25 could be traced back to the Therac-20. The hardware interlocks and safety devices in the Therac-20 had prevented radiation overdoses from occurring with the Therac-20, but not with the Therac-25. Thus, a change in the hardware environment had revealed faults that resided in the software all along.

III. A CHECKLIST FOR SOFTWARE REUSE

As mentioned earlier, faults manifest themselves in reuse activities despite the type of reuse. The purpose of the checklist is to provide help in practical software reuse activities. Using the checklist does not guarantee the absence of reuse-related faults, and the checklist is not inclusive. Rather, the checklist is a compilation of best practices from literature and lessons-learned, which may need periodic update to take into account new findings[9,13,14]. The checklist supports the idea of a software data sheet by addressing the type of information that should be included in the sheet. The checklist is generic in nature and may be applied, with some modifications, to the various types of software reuse.

A Checklist for Software Reuse

1. FEATURE CAPABILITIES:

☐ 1.1 What are the requirements/design specifications the reused software should meet?

☐ 1.2 Are all the features of the reused component described in sufficient detail?

☐ 1.3 Do the features meet the specifications identified in question 1.1?

☐ 1.4 Is there a potential for any reused software feature to negatively impact the new software system's operation/safety? If so, what has been done to mitigate those negative impacts?

2. FEATURE LIMITATIONS:

Memory

☐ 2.1 What are the maximum memory requirements for the reused component, and can those requirements be met with the current hardware platform?

☐ 2.2 What type of memory is required (i.e., dual-access memory, non-volatile RAM, etc.)?

☐ 2.3 What is the access time, wordlength, and other such memory specifications for the reused software?

☐ 2.4 What are the requirements for allocating memory to the reused software?

Processor

☐ 2.5 What type of processor is required to support the software?

☐ 2.6 What necessary features must the processor possess (i.e., floating-point capability, watchdog timer, etc.)?

Initialization

☐ 2.7 What data elements must be initialized prior to executing the reused software? What are the initialization values?

☐ 2.8 What software routines or procedures must run prior to the execution of the reused software?

☐ 2.9 What hardware elements must be initialized prior to executing the reused software?

Accuracy

☐ 2.10 What accuracy can the reused software's outputs achieve? Are those limits satisfactory for the new application?

Data Throughput

☐ 2.11 How fast can the reused software process data and is that rate satisfactory for the new application?

Timing and Synchronization

☐ 2.12 What timing requirements must the reused software meet, and can it meet those requirements?

☐ 2.13 What are the functions within the reused software which must operate without interruption? What has been done to ensure they are not interrupted?

☐ 2.14 In a multitasking environment, what priority does the reused software require?

Exception Conditions

☐ 2.15 What are the exception conditions associated with the reused software and how does the software handle those conditions?

A Checklist for Software Reuse(cont.)

3. INTERFACE:

Software

☐ 3.1 What are the interfaces (inputs and outputs) between the reused software and other software modules in the system? Are those interfaces in agreement as far as data type/size, amount, protocol/format, and range are concerned?

Hardware

☐ 3.2 What are the interfaces between the reused software and the hardware platform and peripherals? Are those interfaces in agreement as far as type of hardware, data type, access procedures, and privilege are concerned? Example hardware includes registers, input/output ports, buffers, memory, hard drives, switches, keypads, and display devices.

Operator

☐ 3.3 What are the interfaces between the reused software and the operator? Is the interface that the reused software presents consistent with the interface the operator expects? Example interfaces include displayed instructions, status indicators, keypads, touch screens, etc.

☐ 3.4 Is there any timing or sequence involved in the computer-operator interactions?

Plant Systems

☐ 3.5 What are the interfaces between the reused software and the plant system/environment? Are those interfaces in agreement as far as signal range, anticipated signal noise, and other signal characteristics are concerned?

VERIFICATION AND VALIDATION:

☐ 4.1 What unit tests been conducted on the reused software to ensure it meets specifications?

☐ 4.2 Have all modifications related to the software reuse effort been adequately verified and validated?

☐ 4.3 What integration tests have been conducted to ensure the interfaces between the reused software and other software, hardware, and plant systems are correct?

CONCLUSION

Software reuse can aid the production of high quality software at reduced time and cost. Types of software reuse include traditional "cut-and-paste" of software, application generators, COTS software, object-oriented programming, and software maintenance. Despite its benefits, careless application of software reuse can introduce faults. Many of these faults are introduced into the software because (1) it was assumed that a high quality product would automatically result from the reuse of proven software, (2) existence of interface conflicts between the reused software and other systems, and (3) the capabilities and limitations of functions within reused software are not fully appreciated. A checklist is provided in this paper to assist developers and auditors in software reuse activities.

REFERENCES

1. NIST. *Glossary of Software Reuse Terms*, Special Publication 500-222. Gaithersburg, MD: National Institute of Standards and Technology, Dept. of Commerce, December 1994.

2. Hall, P. A. V., ed. *Software Reuse and Reverse Engineering in Practice.* New York: Chapman & Hall, 1992.

3. Cusumano, M., ed. *Software Reuse in Japan*. Colorado Springs, CO: Technology Transfer International, 1992.

4. Jones, T. "Reusability in Programming: A Survey of the State-of-the-Art," *IEEE Transactions on Software Engineering*, SE10(5), September 1984.

5. Tracz, W. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Reading, MA: Addison-Wesley Publishing Co., 1995.

6. Hooper, J. and R. Chester. *Software Reuse*. New York: Plenum Press, 1991.

7. IEEE. "IEEE Standard Glossary of Software Engineering Terminology," IEEE Std. 610.12. New York: IEEE, 1990.

8. Reifer, D. *Practical Software Reuse*. New York: John Wiley & Sons, Inc., 1997.

9. Bullard, Guindi, Ligon, McCracken, and Rugaber. "Verification and Validation of Reusable Ada Components," *Guidelines Document for Ada Reuse and Metrics (Draft).* Lesslie, Chester, and Theofanos, eds. Oak Ridge, TN: Martin Marietta Energy Systems, Inc., 1989.

10. Curtis, B. "Cognitive Issues in Reusing Software Artifacts," *Software Reusability: Applications and Experience*, Vol. II, eds., Biggerstaff and Perlis. Reading, MA: ACM Press, Addison-Wesley, 1989.

11. Leveson, N. and C. Turner. "An Investigation of the Therac-25 Accidents," *IEEE Computer*, Vol. 26, No. 7, July 1993.

12. Lions, J. "Ariane 5: Flight 501 Failure," Report by the Inquiry Board. Paris: European Space Agency, 1996.

13. Barsotti, G. and M. Wilkinson. "Reusability - Not an Isolated Goal," *Proceedings of the Conference on Software Reusability and Maintainability*. Tysons Corner, VA: The National Institute for Software Quality and Productivity, Inc. 1987.

14. St. Dennis, R. "Reusable Ada Software Guidelines," *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, eds., Shriver and Sprague, Jr. Kailua-Kona, Hawaii, Jan. 1987.