Software Reuse in Safety-Critical Systems

Master’s Thesis
Rochester Institute of Technology

Leanna Rierson

May 1, 2000
Disclaimers:

1. This paper is not the official Federal Aviation Administration (FAA) position. The author is a FAA employee, and the paper is intended to be consistent with FAA policy; however, it has not been coordinated through the FAA’s approving officials and merely represents the opinions of the author.

2. The author and FAA are continuing to perform activities in the areas discussed in this paper. For example:
   - FAA policy on reuse if being developed;
   - A tutorial on software reuse is being considered; and
   - Research is being conducted in the areas of commercial-off-the-shelf software, software service history, and object-oriented technology.

Additional materials will be posted on the Aircraft Certification Software website as they become available.
1.0 Introduction

The use of computers in safety-critical systems is growing. Manufacturers of aviation, nuclear, and medical equipment are using increasing amounts of software. The complexity of the software is also increasing, as are the pressures to reduce development cost and time. Software reuse has been touted as a potential solution for reducing the complexity, cost, and time of software development. However, there are several concerns with the reuse of software in safety-critical systems. This paper will explore some of the key reuse topics and identify safety concerns. Since the author is heavily involved in the aviation field, many of the issues and concerns will be identified from an aviation perspective; however, they are also applicable to other safety-related domains. Additional research will be required in several areas to more completely address the reuse issues – this paper will identify areas where additional research is needed.

An overview of this research paper is provided below:

- **Section 2.0** addresses the growing use of software in safety-critical systems. Increasing numbers of manufacturers desire to improve their products by using software. Since software development costs and schedules tend to exceed what is planned, manufacturers are also looking for ways to reduce time and cost. Software reuse is being considered in many safety-critical projects in order to reduce time, cost, and complexity of development.

- **Section 3.0** explores how the civil aviation manufacturers typically address software in their safety-critical systems. The aviation industry has a good record for producing safe systems. This section provides an overview of the RTCA (formerly known as the Radio Technical Committee for Aeronautics) document DO-178B, “Software Considerations in Airborne Systems and Equipment Certification,” which is used by most manufacturers of airborne software. DO-178B provides objectives that will be used to identify the issues with software reuse. Although the aviation approach is used in this paper to identify the issues, these issues will also be applicable to other safety-critical domains (e.g., medical or nuclear).

- **Section 4.0** provides an overview of several approaches to software reuse. The software reuse “discipline” is still relatively immature; consequently, there are several different concepts and approaches being proposed. An actual reuse program tends to tailor their approach using portions of several different approaches. This section is not all-inclusive but does address several approaches that are commonly proposed in safety-critical systems.

- **Section 5.0** addresses the area of software components in safety-critical systems. Many manufacturers desire to reuse software components in multiple systems. This section will identify some of the characteristics needed for a good software component and some of the concerns regarding reusable software components in safety-critical systems. Much of this section is based on the author’s experience from a project for the Federal Aviation Administration’s (FAA) communication datalink program.
• **Section 6.0** provides an overview of object-oriented technology and the concerns of its use in safety-critical systems. Special attention is given to three object-oriented languages being considered in safety-critical systems: C++, Java, and Ada 95.

• **Section 7.0** provides a list of additional research needed to continue addressing the software reuse subject. This paper identifies issues and concerns. Follow-on efforts are needed to work on solutions to those issues. This list will be provided to the FAA, National Aviation and Space Administration (NASA), and industry for consideration in their aviation safety research programs.

• **Section 8.0** provides a brief summary of this paper.

2.0 The Growing Use of Software in Safety-Critical Systems

A safety-critical system is a computer, electronic, or electromechanical system whose failure may cause injury or loss of human life. In a safety-critical system, a “failure” may include the failure of a system to perform its intended function, a failure to warn the operator(s) of an unsafe condition, or a failure to display correct data. Examples of safety-critical systems include aircraft control systems, medical equipment, and nuclear power station control systems [1]. The use of computers in safety-critical systems is increasing at a rapid rate.

Software in safety-critical systems is here to stay, not only in aviation, but also in medical, nuclear, and other safety-critical fields. Development of software in safety-critical systems can be both costly and time consuming. Not only is effective software management needed to reduce time and cost, but also available software technologies should be investigated. **Software reuse** is one technical approach that many believe can reduce software development time and cost.

However, experience makes it clear that reuse is not always entirely safe. The explosion of the Ariane Five rocket is a vivid reminder to the developers of safety-critical software of what can happen if software is reused without proper care. The software used on the Ariane Five was originally intended for the Ariane Four and worked properly on that platform. However, the launch characteristics of the Ariane Four and Five rockets were different. Improper reuse of the Ariane Four software caused the Ariane Five to explode [2].

In her book *Safeware*, Dr. Nancy Leveson states that there is a myth that software reuse increases safety. She provides three examples of safety problems that arose from the reuse of software. First, the Therac-25 medical device reused parts from the Therac-20. An unknown error existed in the Therac-20 software. The error had no serious consequences on the Therac-20 operation, except that it would occasionally blow a fuse. However, on the Therac-25, the error led to massive radiation overdoses and led to the death of at least two people. Software reuse was not the sole reason for the Therac-25 incident; however, it was a major contributing factor. Second, air traffic control software used in the United States was reused in Great Britain. However, the users failed to account for the longitudinal differences between the U.S. and Great Britain. Third, software written for aircraft in the northern hemisphere and above sea level, often causes problems when reused in the southern hemisphere or below sea level [3].

The above examples provide a big picture look at some cases where software reuse was not carefully evaluated and implemented. As software becomes more complex and more widely
used, the concerns of software reuse in safety-critical systems increase. Reuse is a viable option in many cases; however, it must be evaluated and implemented with caution.

3.0 Civil Aviation Method of Addressing Software Assurance

The RTCA (formerly known as the Radio Technical Committee for Aeronautics) document DO-178B, entitled “Software Considerations in Airborne Systems and Equipment Certification,” is the guidance document that most civil aviation manufacturers use for certification approval of their airborne software. ED-12B is the European Organization for Civil Aviation Equipment (EUROCAE) equivalent of DO-178B. In order to assess how DO-178B/ED-12B applies to software reuse, it is important to understand the background and basics of the document.

DO-178/ED-12 (no revision) was first developed by the international civil aviation community in 1982. It was revised in 1985 to add more detail. In 1992, DO-178B/ED-12B was completed and has become the software “standard” for airborne software in civil aviation products. The DO-178/ED-12 document and all of its revisions were sponsored by RTCA and EUROCAE, with the involvement of aviation, software, and certification experts from across the world.

DO-178B/ED-12B focuses on the software aspects of system development. As part of the systems engineering task, a system safety assessment is performed before DO-178B/ED-12B can be applied to the software development effort. A system safety assessment is a process to identify the hazards, failure conditions leading to these hazards, and the effects of mitigation strategies. The system safety assessment task determines a software level based upon the contribution of the software to the potential failure conditions defined in the system safety assessment process. The five software levels, A to E, are summarized in Table 1 [4].

These software levels define differing degrees of rigor for the software development process. Annex A in DO-178B/ED-12B lists the objectives that must be met for each specific software level. These software levels define a number of key characteristics for the software development and verification processes. The difference in rigor is determined by the number of objectives that need to be satisfied, whether a specific objective is satisfied with independence, and the formality of configuration management of the software data produced during development. For example, the number of objectives for each software level is listed below:

- Level A: 66 objectives
- Level B: 65 objectives
- Level C: 58 objectives
- Level D: 28 objectives
- Level E: 0 objectives

DO-178B/ED-12B is divided into development processes and integral processes. The development processes include planning, requirements, design, code, and integration. The integral processes include verification, configuration management, quality assurance, and certification liaison. The integral processes are overlaid on each of the development processes (i.e., verification, configuration management, quality assurance, and certification liaison are applied to each development process).
<table>
<thead>
<tr>
<th>Failure Condition Category</th>
<th>Description</th>
<th>Software Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catastrophic</td>
<td>Failure conditions which would prevent continued safe flight and landing of the aircraft.</td>
<td>A</td>
</tr>
<tr>
<td>Hazardous</td>
<td>Failure condition which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operation conditions to the extent that there would be:</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>(1) a large reduction in safety margins or functional capabilities,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2) physical distress or higher workload such that the flight crew could not be relied on to perform their tasks accurately or completely, or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(3) adverse effects on occupants including serious or potential fatal injuries to a small number of occupants.</td>
<td></td>
</tr>
<tr>
<td>Major</td>
<td>Failure conditions which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operation conditions to the extent that there would be, for example,</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>(a) a significant reduction in safety margins or functional capabilities, as significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to occupants, possibly including injuries.</td>
<td></td>
</tr>
<tr>
<td>Minor</td>
<td>Failure conditions which would not significantly reduce aircraft safety, and which would involve crew actions that are well within their capabilities.</td>
<td>D</td>
</tr>
<tr>
<td>No Effect</td>
<td>Failure conditions which do not affect the operational capability of the aircraft or increase crew workload.</td>
<td>E</td>
</tr>
</tbody>
</table>

Table 1 – DO-178B/ED-12B Software Levels [4]

The objectives of DO-178B/ED-12B are listed in Annex A of the document and are organized around the development processes and integral processes previously described. There are ten tables in Annex A with objectives—the subject of each table is listed below:

- Table A-1: Software Planning Process
- Table A-2: Software Development Processes
- Table A-3: Verification of Outputs of Software Requirements Process
- Table A-4: Verification of Outputs of Software Design Process
- Table A-5: Verification of Outputs of Software Coding & Integration Processes
- Table A-6: Testing of Outputs of Integration Process
- Table A-7: Verification of Verification Process Results
- Table A-8: Software Configuration Management Process
- Table A-9: Software Quality Assurance Process
- Table A-10: Certification Liaison Process

Table A-4 objective 1 is used in Figure 1 to illustrate the Annex A table layout and structure. The first set of columns contains information about the DO-178B/ED-12B objectives: objective number, description, and reference to DO-178B/ED-12B paragraph where that
objective is further detailed. The next set of columns with headers A, B, C, D show the applicability of that particular objective to the software level. For example, objective 1 is applicable for levels A, B, and C; however, it does not need to be satisfied for software level D. If the circle indicating applicability is filled in, then that objective must be satisfied with independence.

The next series of columns describe the outputs produced as evidence that the objective is satisfied. The “Description” column lists where that evidence is documented. The “Ref.” column identifies the paragraph within Chapter 11 of DO-178B/ED-12B that details the attributes of that software data. The last 4 columns correlate the rigor of configuration management of the particular output with the associated software level. Control category 1 requires more configuration management activities than control category 2. For instance, control category 1 requires problem reporting and change control, whereas control category 2 requires only change control.

<table>
<thead>
<tr>
<th>Objective</th>
<th>Applicability</th>
<th>Output</th>
<th>Control Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-level requirements comply with high-level requirements.</td>
<td>6.3.2a</td>
<td>Software Verification Results</td>
<td>11.14</td>
</tr>
</tbody>
</table>

Figure 1 – Portion of Table A-4 in DO-178B/ED-12B [4]

Assessing conformance to DO-178B/ED-12B is performed through on-site reviews and/or desktop (data) reviews by regulatory authorities or appropriately designated industry representatives. The assessment evaluates the data to determine if the objectives listed in Annex A of DO-178B/ED-12B are met.

DO-178B/ED-12B is used on nearly every civil aviation airborne project involving software. Recently, DO-178B/ED-12B has also been applied to military software programs, non-airborne aviation software (such as ground-stations), and medical equipment software. DO-178B/ED-12B has earned the respect of those developing safety-critical software and hence serves as a “measuring stick” for evaluating proposed alternative methods of software assurance. In order to identify the potential safety and assurance issues of software reuse, reuse technology will be evaluated against the objectives of DO-178B/ED-12B in this research work.

4.0 Approaches to Software Reuse in Safety-Critical Systems

Steve Adolph’s article, “Whatever Happened to Software Reuse?” states, “If you ask five programmers what reuse is, you’ll get eight answers” [5]. He points out that software reuse and software salvaging are different. “Software reuse is software that was designed to be reused” and software salvaging is “using software that was not designed to be reused” [5].

In their book, Software Reuse – Domain Analysis and Design Process, Jag and Prince Sodhi define software reuse as “a process of implementing or updating software systems using existing software assets. Assets can be defined as software components, objects, software
requirement analysis and design models, domain architecture, database schema, code, documentation, manuals, standards, test scenarios, and plans. Software reuse may occur within a software system, across similar systems, or in widely different software systems” [6].

Regardless of the precise definition, the goal of software reuse is to use as much software data as possible from previous development efforts in order to reduce time, cost, and risks associated with re-development.

Even though the concept of software reuse seems simple in theory, implementing that concept into reality is quite complex. This section will provide an overview of the reuse approaches and on-going research. The discipline of software reuse is evolving and is still quite immature. While reuse promises many benefits, it is difficult to actually implement. These approaches and research efforts provide a toolbox from which software developers can plan their reuse effort. Each software organization will have to decide which approaches most appropriately meet their needs. The following approaches and concepts will be discussed in this section:

- Planning for reuse;
- Domain engineering (architectures that contribute to reuse);
- Software components;
- Object-oriented technology;
- Portability;
- Commercial-off-the-shelf (COTS) software; and
- Product service history.

4.1 Planning for Reuse

Reuse does not just happen, it must be well planned and managed. In his book, Practical Software Reuse, Donald Reifer emphasizes the importance of the managerial and organizational issues for software reuse development – software reuse is costly when it is first launched and must be properly planned to reap the follow-on benefits [7]. Reifer provides a “reuse adoption process” for companies striving to integrate software reuse into their software development programs. The ten steps of his process are listed below [7]:

1. Define the company vision and strategy;
2. Determine where the company currently is;
3. Establish an operation concept for the company;
4. Prepare a business case for the company;
5. Develop a company business plan;
6. Focus early efforts on the company infrastructure;
7. Make an initial success;
8. Try the ideas before they are solidified;
9. Strive for a success image; and
10. Iterate and refine the process based on results.

Steve McConnell’s book, Rapid Development, also provides some valuable information on the topic of software reuse. He lists the following as keys to success in software reuse – these
key issues must be well planned and managed in order for software reuse to offer its maximum benefits [8]:

- Take advantage of personnel continuity between old and new programs;
- Do not overestimate your savings;
- Secure long-term, high-level management commitment to the reuse program;
- Make reuse an integral part of the development process;
- Establish a separate reuse group whose job it is to identify reusable component candidates, create standards that support reuse, and disseminate information about reusable components to potential users;
- Focus on small, sharp, domain-specific components; and
- Focus design efforts on abstraction and modularity (e.g., information hiding and encapsulation).

Additionally, software reuse planning efforts should address safety, software integration with software and hardware, portability, maintenance, and re-verification.

The old eastern saying, “If you do not know where you are going, any road will lead you there,” holds much truth when it comes to planning for reuse of software. Having a plan and a vision is essential to success.

### 4.2 Domain Engineering

A domain is “a group or family of related systems. All systems in that domain share a set of capabilities and/or data” [6]. Domain engineering is a relatively new discipline. It is the process of creating assets through domain analysis, domain design, and domain implementation that can be managed and reused. The domain engineer suggests an architecture that meets the majority of the application needs and is suitable for future reuse.

A lot of energy and research is being put into the concept of domain engineering, particularly domain analysis. At an Institute of Electrical and Electronic Engineers (IEEE) conference on software reuse, a number of position papers and research initiatives in the area of domain analysis were presented.

Rafael Capilla, a presenter at the IEEE conference, presented two sides of reuse: *domain engineering* (development for reuse) and *reuse engineering* (development with reuse). He went on to define reuse engineering as “the process of reusing reusable components that have (or haven’t) passed through domain analysis process. Specific to our problem is the process of reusing stored knowledge which has passed through a previous process of domain analysis. It comprises the following steps:

1. Domain analysis;
2. Search and selection of solutions;
3. Knowledge reuse;
4. Generation of the solution;
5. Validation and verification;
6. Feedback to repositories (optional)” [9].
Capilla also defined domain analysis as “the analysis of relevant information of the problems to generate a model of information or knowledge that can be reused to build applications … the fundamental steps” to do the domain analysis are:

1. Analysis prior to reuse;
2. Semantic analysis;
3. Structure analysis;
4. Domain vocabulary construction;
5. Problem classification;
6. Verification of minimum information to reuse; and
7. Domain model generation [9].

One of the main goals of domain analysis is to develop general-purpose software architecture in order to increase software reuse (i.e., the “open architecture” concept). The paper “Architecture Reuse through Domain-Specific Language Generator” by S. A. White and C. Lemus emphasizes the reuse of architectural domain knowledge, reuse of architectural designs, and conditions for reuse of software components. They state, “Our position of software reuse is that while reuse of code is important we feel that in order to make truly large gains in productivity and quality, reuse of software designs and patterns offer the greater potential for return on investment” [10].

Rosario Girardi and Bertrand Ibrahim’s position paper, “Software Architectures to Improve the Effectiveness of Reuse Techniques,” emphasizes the need to reduce cognitive distance in software development. Cognitive distance is “the intellectual effort that is required to take a software system from one stage of development to another. Moreover, a reuse technique should reduce the cognitive distance between the conception of a system at the highest level and its final executable implementation” [11]. Girardi and Ibrahim propose that “reuse techniques based on software architectures reduce cognitive distance.” They believe the establishment of “general-purpose libraries of software architectures appears as a promising direction for successful reuse”[11].

A good domain engineering process is essential to successful software reuse.

4.3 Software Components

The term “software component” has many definitions. In an article entitled “The Significance of Components,” Bertrand Meyer proposes that there are two key properties of a software components: (1) the component may be used by other program elements, and (2) the users and developers of the software component do not need to know each other [12]. Software components are basically pieces of software data that can be used in multiple applications. Because the software component concept is one of the central elements of software reuse and is being seriously considered by developers of safety-critical software, it will be further explored in Section 5.0.

4.4 Object-Oriented Technology (OOT)

Many people promote the object-oriented (OO) design and development as an approach which allows software reuse to be utilized. OO technology leads to reusable classes and
libraries. According to Sodhi, object-oriented approach improves the following software development reuse aspects [6]:

- Interoperability through standard object classes used across various software systems through inheritance.
- Maintainability and reliably of software systems through object encapsulation.
- Flexibility through polymorphism, as well as simplification of software development.
- Availability of domain-specific objects and software reusable components repository.

Object-oriented technology (OOT) has gained quite a following over the past few years, due to its potential to reduce development cost and to promote software reuse. Many developers of safety-critical software are considering OOT in their systems. Therefore, this aspect of software reuse will be covered in more detail in Section 6.0.

4.5 Portability

4.5.1 What is Portability?

James D. Mooney of the University of West Virginia has carried out a number of portability and reuse research efforts with funding from the National Science Foundation. He defines software portability as an attribute which may be possessed by a software unit to a specific degree with respect to a specific class of environments. It may also be an attribute of auxiliary elements such as data, documentation, and human experience [13]. Portability is typically concerned with the reuse of complete applications on new platforms and/or environments; it is a form of reuse [14]. Portability is becoming universally recognized as a desirable attribute for most software products, because it enhances the value of a software package both by extending its useful life cycle and by expanding the range of installations in which it can be used [15].

There are typically two types of portability considered: binary portability (porting the executable form) and source portability (porting the source language representation). Binary portability is clearly desirable, but is possible only across strongly similar processors (e.g., same binary instruction set) or if a binary translator is available. Source portability assumes availability of source code, but provides opportunities to adapt a software unit to a wide range of environments [14].

Mooney wrote that “the porting process has two principal components which may be called transportation and adaptation. Transportation is physical movement; this may not be trivial since compatible media must be used and various types of representation conversion may be required. Adaptation is any modification that must be performed on the original version; we take this to mean both mechanical translation such as by language processors, and manual modification by humans” [14].

4.5.2 Designing for Portability

Mooney brings out the point that “the designer must determine how (or if) the specified portability goals can be met most effectively. In general, incorporating portability may call for design strategies such as the following:
1. Identify the minimum necessary set of environmental requirements and assumptions.
2. Eliminate all unnecessary assumptions throughout the design.
3. Identify specific environment interfaces required (procedure calls, parameters, data structures, etc.). For each interface, either:
   4a. Encapsulate the interface completely in a suitable module, package, object, etc. Anticipate the need to adapt this interface to each target system; or:
   4b. Identify a suitable standard for the interface, which is expected to be available in most target environments. Follow this standard throughout the design. Anticipate the need to provide a software layer to ‘bridge the gap’ for environments which do not implement the standard satisfactorily. A special case of 4b is the selection of a suitable programming language. This choice must consider the availability of translators for the anticipated target environments. Moreover, it is desirable that as many of the required interfaces as possible be captured as elements of the language itself. For example, a language with a variety of standard file operations built in, such as Ada or C, would allow these operations to be used freely throughout the program. Choice of a language with limited facilities, like Pascal, would require more attention to isolating these operations if portability is to be maintained. This consideration might require selection of a different language than would be preferred based on other criteria” [15].

Using good specification methods is also essential for portability design. Mooney suggests the need for two types of language-independent specifications:

1. Descriptive specifications which characterize the environmental requirements for a software unit, and the facilities available in a specific environment, in terms of interfaces and intrinsic attributes such as reliability;
2. Quantitative specifications which characterize the actual or required degree of portability of an application with respect to a specific environment or class of environments [15].

4.5.3 Technical Issues Related to Portability

There are a number of technical issues associated with portability and reuse. They are summarized below:

**Classification:** Classification of reusable artifacts is a major issue in reuse and portability. The emphasis is on classifying complete applications according to their environmental interfaces and requirements. Classification can assist in matching applications to environments, and in reusing experience gained from porting other applications with a similar classification [14].

**Specification:** Effective specification for reuse is an area of technical consideration. For example, given a set of functions or objects to be incorporated in a collection of components, effective choices must be made on partitioning, parameterization, etc. Mooney states that “portability issues arise after the functional requirements of a software unit have been identified. In this sense the specifications are already determined. However, when an application is to be developed (or redeveloped), the role of portability considerations in the development process must be determined by the portability requirements of the project. These requirements form a part of the overall requirements specification, and should answer questions such as ‘how much’ portability is required, to ‘what kinds of environments,’ and ‘what costs’ can be accepted to achieve it” [14].
Measurement: Measurement techniques are needed for the cost-effectiveness of portability-based approach to software development. These measurements must take into account both the increased costs of initial development and the cost-savings later realized through porting [14].

Design: Both portability and reuse are software development goals which have a significant impact on the design process (as well as other phases of the complete development cycle). This impact is felt especially in the task of partitioning an overall design into a suitable collection of entities (procedures, modules, objects, etc.). Many objectives must be considered and balanced in choosing an effective partitioning [14].

Cultural Adaptation: In many cases, the detailed behavior that is most desirable for an application ported to a new environment is not identical to its behavior in a previous environment. Instead, the behavior must be adapted to the conventions of the new environment and the intended users [14].

Verification and Validation: An essential part of the software development process is establishing a high level of confidence in the correctness of an implementation. Techniques ranging from systematic testing to formal verification are used to establish this confidence. Testing and verification activities must be repeated for each new application or implementation. However, it may be possible to demonstrate that some types of correctness and other properties are preserved during porting or reuse. This could lead to a reduction in effort in revalidating the new application or implementation [14].

Other technical issues that commonly arise in software reuse and portability programs are operating system inconsistencies, different compiler options and effects, library incompatibilities, run-time problems, underestimation of the integration effort (both software/software and software/hardware integration), and architectural consistency. Many of these problems can be overcome by planning ahead, breaking the project into small/abstract components, implementing risk management, and defining portable specifications.

4.5.4 The Real-Time Issues of Portability

One area where portability has particular challenges is in the area of real-time software design and implementation. For real-time software design, the following requirements must be considered for portability and reuse [13]:

1. Timing (e.g., Guaranteed bounds, Accurate interval and interrupt timing);
2. Memory allocation (e.g., Pools and fixed-size blocks, Memory locking, Virtual memory disable);
3. Memory deallocation (e.g., Garbage collection, Reformatting memory to remove unused or dead areas);
4. Dynamic Task Creation;
5. Scheduling Control (e.g., Preemptive scheduling, Cyclical scheduling, Susendable round robin, Thread management, Static scheduling, Resource scheduling);
6. Synchronization and Communication (e.g., Semaphores, Messages, Locks, Event Flags, etc.);
7. Events and Input/Output (e.g., Direct interrupt handling, Direct device control); and
8. File Access (e.g., Contiguous files for direct indexing).
Portability is an important aspect of software reuse that should be considered by developers of reusable software.

4.6 **Commercial-off-the-shelf (COTS) Software**

DO-178B/ED-12B defines COTS software as “commercially available applications sold by vendors through public catalog listings. COTS software is not intended to be customized or enhanced. Contract-negotiated software developed for a specific application is not COTS software” [4]. COTS software can contribute to reuse by providing commercially available components for implementation.

One of the most common uses of COTS software in embedded systems is operating systems and real-time operating system (RTOS) kernels. There are many concerns regarding the use of COTS software in safety critical systems, since the development rigor and robustness are often not documented (i.e., documented evidence is not available).

The FAA has recently launched a research effort to complete guidelines, verification methods, and assessment criteria for COTS software. There is substantial concern in the aerospace industry for investigating whether methods are available and could be used for evaluating COTS software used in airborne systems. The use of COTS software could potentially offer significant cost savings for small aircraft and rotorcraft. There could also be a potential for increased aircraft safety, if lower cost systems could be shown to be safe and would allow the replacement of older, less capable systems. Moving maps, graphical weather, situational awareness, and cockpit display of traffic information (CDTI) could be used in general aviation application, if efficient methods of assessing COTS software were available. However, the FAA is concerned that optimism regarding the use of COTS software in safety-critical applications is not justified. There is a substantial body of opinion in the field of safety engineering that the use of COTS software in safety critical applications is inherently unsafe. On the other hand, it is the opinion of some that the potential for economic savings is so great (perhaps driving possible safety enhancements in the use of new, more capable systems), it would be in the public’s and industry’s interest to have a thorough investigation of the competing claims regarding COTS software. In a joint RTCA and EUROCAE software meeting, the following questions were posed for consideration:

- What verification techniques can be employed to assure that COTS products are safe?
- How is the system protected from the undeclared additional functionality that might be introduced by a COTS product?
- How is configuration control gained for COTS products?
- Is there a COTS graphic operating system that can be used in airborne systems?
- Could testing techniques be developed to verify that unintended functions in COTS software do not exist or do not have an adverse affect?
- Could there be a platform foundation for integrating a variety of COTS products? (i.e., Could there be an integration standard for aviation, similar to HP Soft Punch?)
- Are there software packages available that perform multi-tasking and multi-processing that can be certified?
- Are there tools or techniques to create requirements from code?
• How can large operating systems be approved (i.e., How can critical processing be encapsulated within a non-secure environment? This is particularly a concern for ground systems.)?

United Technology Research Center is considering these and other questions as they carry out COTS software research for the FAA. The FAA hopes that this research effort will help to address many of the concerns regarding COTS software in safety-critical systems.

4.7 Product Service History

DO-178B/ED-12B defines product service history as, “a contiguous period of time during which the software is operated within a known environment, and during which successive failures are recorded” [4]. The purpose of product service history is to gain confidence in software over a period of time. According to DO-178B/ED-12B, the acceptability of this method is dependent on:

• Configuration management of the software;
• Effectiveness of problem reporting activity;
• Stability and maturity of the software;
• Relevance of product service history environment;
• Actual error rates and product service history; and
• Impact of modifications [4].

The international Certification Authorities Software Team (CAST) paper entitled, “Guidance for Assessing the Software Aspects of Product Service History of Airborne Software Systems and Equipment” lists the following attributes that should be evaluated in determining the credit that can or cannot be granted for service history [16]:

• Service duration length;
• Change control during service;
• Proposed use versus service use;
• Proposed environment to service environment;
• Number of significant modifications during service;
• Number of software modifications during service;
• Number of hardware modifications during service;
• Error detection capability;
• Error reporting capability;
• Number of in-service errors; and
• Amount/quality of service history data available and reviewed.

To date, in the civil aviation approval process, product service history credit has only been granted for projects with similar or identical environments. However, many manufacturers now desire to use service history credit for software that may be run in different environments. For example, Windows NT has thousands of hours of history in personal computer and network applications. Developers of safety-critical systems may desire to use that history to gain credit for use in their applications. (Note: This approach would likely only be possible for the lower software levels of DO-178B/ED-12B.)
In order to address this area of software reuse, the FAA is striving to begin a research program to further evaluate the use of product service history in safety-critical systems.

4.8 Summary of Software Reuse

This section has highlighted several of the aspects of software reuse that may be relevant in a software development program desiring to utilize reusable software. All of these issues are of interest to developers of safety-critical software. However, there are two areas that are currently causing lots of concern in the safety-critical development environment: software components and object-oriented technology. The use of software components and object-oriented technology have had some success in the mainstream software development environment (e.g., desktop application software, web-based software, etc.); however, their use in safety-critical software has been very limited. Section 5.0 will explore the concept of software components in more detail. Section 6.0 will further explore the use of object-oriented technology in safety-critical systems.

5.0 Software Components in Safety-Critical Systems

5.1 What is a Software Component?

As discussed in section 4.3, there are numerous definitions of the term software component. Ann Rhodes’ paper entitled “Component Based Development for Embedded Systems” refers to a software component as “prewritten elements of software with clear functionality and well-defined interfaces” [17]. Anthony Lattanze defines a component as “an atomic software element that can be reused or used in conjunction with other components; ideally, it should work without modification and without the engineer needing to know the content and internal function of the component. However, the interface, functionality, preconditions and post-conditions performance characteristics and required supporting elements must be well known” [18]. These definitions are wide and general in nature and are most appropriate for purposes of this thesis, since developers of safety-critical systems face many of the more general issues.

Some manufacturers desire software components to become like electronic components. It is easy to buy resistors or electronic circuits to plug into an overall system. However, this concept has not yet become reality in the software world. In theory, software components can help reduce development costs and allow developers to focus on the more unique aspects of a program. Examples of software components being considered in safety-critical systems are real-time operating systems (RTOS), communication protocol stacks, and run-time libraries.

5.2 Characteristics of a Good Software Component

As mentioned in Section 4.3, Bertrand Meyer proposes that there are two key properties of a software component: (1) the component may be used by other program elements, and (2) the users and developers of the software component do not need to know each other [12]. Similarly, other sources propose that there three major attributes of a software component: (1) it is reusable;
The first attribute, *it is reusable*, is relatively obvious. If a software component is not reusable, it is just another application or module in a project.

In order for a component to be reusable, *its functionality must be clear* and well-defined interfaces. A function defines what a component does. The function should be a high-level “what” the component does – not “how” it is implemented. Some factors of clear functionality are described below [17]:

- **Single purpose** – it implements a single concept or purpose.
- **Encapsulates related functions** – it only provides functions related to its purpose.
- **Properly sized** – it is not too small for use and not too large to become unmanageable.

In order to allow smooth and successful integration, a software component must have a well-defined interface. An interface defines how the user interacts with the component. A successful interface has consistent syntax, logical design, predictable behavior, and consistent method of error handling. A good interface is complete, consistent, and cohesive – it provides what the user needs to make the component work [17].

Bertrand Meyer’s article, “Rules for Component Builders,” provides eight qualities of a software component. These qualities are essential to developing good componentware and are summarized below [19]:

- **Careful specification.** A component must have precisely specified functionality and interface.
- **Correctness.** The component must work as stated in the specification.
- **Robustness.** The component must not fail, when it is used properly.
- **Ease of identification.** It should be easy for users to identify if the component meets their needs.
- **Ease of learning.** Users should be able to learn how to use the component quickly.
- **Wide-spectrum of coverage.** The component should be useful by integrators with a wide range of experience (i.e., novices to experts).
- **Consistency.** Components in a library should have consistency.
- **Generality.** The component should be general enough for use in multiple environments.

These qualities are essential for successful implementation of software components.

### 5.3 The Software Component Library

One of the major purposes of developing software components is to develop a component library. The library stores reusable assets and serves as a primary reuse distribution mechanism. According to Donald Reifer in his book, *Practical Software Reuse*, the library should be selected because it provides applications engineers with the following minimum set of user-oriented capabilities [7]:

Leanna Rierson 15 May 1, 2000
1. Provide seamless access to available library capabilities and populations for authorized users (may consist of several libraries networked together).

2. Be able to quickly search, browse, and retrieve quality reusable assets (or abstracts) that satisfy user’s requirements. The paper entitled “Applying Multi-Media to the Reuse of Design Knowledge” encourages the documentation of “design narratives” that can quickly be retrieved and reused [12].

3. Be an integral part of the software engineering environment (permits engineers to promote assets from one library to another using common service modules – configuration management, SQL queries, …).

The software reuse library allows reusable software assets to be stored, controlled, distributed, and managed. Assets include more than just code. In the best of all possible worlds, the software reuse library is an integral part of the software engineering environment; therefore, it also includes requirements, plans, design, test cases, … It allows software engineers to find, browse, and access reusable assets as they are performing their applications engineering work tasks. The task of asset management is typically broken into four areas: library management, library population, library operations, and library maintenance [7]. Each will be discussed below.

- **Library Management** is conducted to administer the reuse library, maintain access control, and manage the day-to-day operations. The complexity depends on the size of the organization and the plans for reuse.

- **Library Population** is the process of adding components to the library. This can be a large and tedious task. Each asset must be abstracted, classified, and qualified based upon predefined criteria before it is made available to potential users. The criteria usually requires component name, function, parameters, etc. In some cases, the asset may need to be re-engineered. The main product of this task is a listing of qualified assets.

- **Library Operations** is the process of operating the library and providing the advertised goods and services to the users. This includes keeping the users informed of up-to-date information, etc. Configuration control and quality control must also be maintained.

- **Library Maintenance** is the process of keeping the library software up to date. The database software used to implement the library often changes or requires addition of new features, etc. [7].

In order to effectively use a software component, its **design rationale** should be well documented. In his paper entitled, “Design Rationale and Reuse,” Dean Allemang states, “Design rationale information is important in software reuse for at least two reasons; first, the design decisions that went into a component will help to determine where it can appropriately and advantageously be reused, and second, the design decisions made in a system in which a component is to be reused determine what components can be appropriately reused” [20]. Allemang divides design rationale into two categories: (1) **internal rationale** (the relation of parts of the design to other parts of the same design – the way in which a component interacts with other components) and (2) **external rationale** (the relation of parts of the design to parts of different designs – use of component in multiple systems) [20].
5.4 Concerns of Software Components in Safety-Critical Systems

The author of this thesis is leading the Federal Aviation Administration’s Technical ReUsable Software Team (TRUST). The purpose of TRUST is to identify issues of software component reuse and to develop guidelines to address those issues.

TRUST was initiated to provide guidance to software developers designing a router system for the communication datalink, which will likely be used in the modernization of the National Airspace (NAS). The goal of the router developer is to reuse the router in numerous avionics units and ground-based applications, with different target computers. The goal of the FAA has been to make sure that the router (a software component) is reused safely. The FAA has used DO-178B/ED-12B as the measuring stick to identify potential issues. In order to identify the issues, software reviews of the router development were performed by the FAA (led by the author of this thesis). The findings from the router project were then generalized to identify general issues that might face any developer or user of a software component. Additional workshops and reviews are being held to continue identifying and addressing other issues that may arise.

This section will summarize issues identified to date regarding the reuse of software components in safety-critical systems.

5.4.1 Planning

Both the developer and user of the software component should agree and plan for reuse early in the program. The developer of the software component will have plans at the component level to address the development, configuration management, verification, and quality assurance of the component. The user will have system-level plans to address the overall software development and reuse effort.

DO-178B/ED-12B places much emphasis on the existence of plans and adhering to those plans. In order for a software component to be well-designed and used, the planning process must be thorough.

5.4.2 Multiple Requirements Levels

When there are multiple manufacturers involved in a program, numerous “levels” of requirements typically exist. Figure 2 illustrates an example of the numerous requirement levels that might exist in the national airspace system. Level 1 is the airspace level. The national airspace includes satellites, ground stations, and aircraft. For this example, Level 2 is an aircraft within the airspace. A third level of requirements is illustrated by Level 3, the black-box level. An aircraft contains a number of black-boxes. Level 4 is a component with a black-box – it might be the requirements for a RTOS. Level 5 illustrates the sub-components (e.g., modules) within the component.
In the traditional airborne software world, levels 2 and 3 tend to be most common. DO-178B/ED-12B was written with these levels in mind. In DO-178B/ED-12B, “high-level requirements” are defined as “Software requirements developed from analysis of system requirements, safety-related requirements, and system architecture” [4]. “Low-Level Requirements” are defined as, “Software requirements derived from high-level requirements, derived requirements, and design constraints from which source code can be directly implemented without further information” [4]. Basically, high-level requirements are the level just below the system requirements and low-level requirements are the lowest level of refinement before implementation into source code. A number of DO-178B/ED-12B objectives refer to high-level and low-level requirements.

The reason that this issue of requirements level proposes a concern for safety, is that both the high-level and low-level requirements must be verified. If requirements are verified at too high or too low of a level, problems can arise. The goal is to have low-level verification to catch problems within the components/modules and high-level verification to catch integration issues. The proper coverage of verification is needed in order to gain assurance that safety-related errors have been detected and removed from the safety-critical software.
5.4.3 Re-verification

In order to determine a re-verification position, the following assumptions are made: source code is not changed, high-level requirements are not changed, low-level requirements are not changed, derived requirements are not changed, software architecture is not changed, integrator is using the same high-level requirements for the reusable software component as did the developer. If these assumptions are true and a new compiler or a new target environment is used, two common questions arise and are addressed below:

(1) How much re-verification is required with a new compiler, new target computer, or new target environment and why?

In general, all verification accomplished by test should be repeated on a new target computer, new target environment, and/or new compiler. However, for a given program it might be possible to use analysis to show that specific test items will not be affected by a compiler or target environment change. Analysis situations should be handled on a program by program basis.

Also, in general, non-test verification activities such as analyses, reviews, inspections, and walkthroughs may not have to be repeated when the software is recompiled and installed in a new target computer. However, there are certain exceptions. For instance, it should be assured that the new target environment system requirements are compatible with the software requirements of the reusable software component. If they are not completely compatible, modifications may be needed to the reusable software component.

(2) Will structural coverage analysis need to be repeated on the target computer?

DO-178B/ED-12B suggests four kinds of structural coverage, depending on the software level: statement coverage (Levels A, B, and C); decision coverage (Levels A and B); modified condition/decision coverage (Level A only); and data and control coupling coverage (Levels A, B, and C).

Structural coverage analysis should not have to be repeated for Level A if the newly compiled object code is directly traceable to source code statements, as in this case the analysis may be performed on the source code, for which it is assumed that no changes have been made. Structural coverage analysis should be repeated for Level A software if the newly compiled object code is not directly traceable to source code statements. Structural coverage analysis should not have to be repeated for Levels B and C software, as coverage analysis may be performed on source code for these levels.

5.4.4 Interface Documents

Interface documents should be provided by the reusable software component developer. The interface documents should explicitly define what activities are required by the component user to ensure that the component will function in accordance with its approval basis. Typical items included in interface documents would be any configuration parameters, any restrictions on tools, additional verifications, memory and timing requirements, etc. Additionally, the interface documents will define the communications mechanisms between the reusable component and the other software programs. Typical items included would be variables that are
accessible and their characteristics, variables/data that are required from the system and their characteristics, access mechanisms, etc.

5.4.5 Partitioning and Protection Considerations

The purpose of partitioning and protection in safety-critical systems is to protect higher integrity software from software with less integrity. For example, a partition may be used to separate a Level B function from a Level C function in an airborne application.

Although partitioning and protection will most likely be a function at the system architecture level, the software component may have some requirements for partitioning and protection that will need to be satisfied. For example, there may be some maintenance code that is at a different software level than the operational flight program for the component. In some cases, the component might have specific protocols that facilitate protection and partitioning. These will need to be documented and evaluated by the integrator and the regulatory authorities.

5.4.6 Artifacts

One of the most common questions that arises when addressing the reuse of software components, is “what artifacts should be transferred to the user of the component?” In order to support the continued airworthiness of a project (i.e., ongoing safety assurance), the FAA proposes that the following data should be supplied by the re-useable software component developer to the user:

- The type design data listed in Section 9.4 of DO-178B/ED-12B (i.e., Software Requirements Data, Design Description, Source Code, Executable Object Code, Software Configuration Index, and Software Accomplishment Summary) [4].
- The Plan for Software Aspects of Certification (PSAC).
- Interface information (e.g., interface control document, porting guide). This should include any hardware and software resource requirements (e.g. memory for program, data, stack, timing requirements, etc).
- Installation procedures and limitations. The limitations should be sufficient to ensure that the software, when installed in accordance with the installation procedures, continues to meet the requirements. The limitations should also be sufficient to identify any unique aspects of the installation. The limitations should include at least the following:
  1. Equipment specifications required for proper operation and performance of the software.
  2. List of the sub-components (by part number) that compose the reusable software component.
  3. Instructions for periodic maintenance and calibration that are required for continued airworthiness once the software is installed.
- Listing of DO-178B/ED-12B objectives with the credit sought for each.
- Data to support the integrator’s completion of objectives. For example, if partial credit was sought for objective 1-1 (Software development and integral processes are defined), it needs to be clearly defined to the integrator what that partial credit entails and what they (the
integrator) need to do to complete the credit. The necessary data to support that “partial” should also be made available to the integrator. Another example would be a requirement to perform specific testing on the target computer in order to completely meet the specific objective.

- Test cases and procedures to be re-run on the target computer. With a listing of test cases and procedures affected by any integrator settable parameters.

Additionally, any DO-178B/ED-12B artifacts not listed above used in the software development and approval process should be made available to the integrator (e.g., SQA records). Also, data needed to support changes to the reusable component should be identified. If the developer should go out of business, this data will help support continued safety and data retention regulations.

Whether the above data is actually delivered to the user or made accessible for any future considerations is a business decision that needs to be worked out between the developer and user of the software component. The above information is needed to address continued safety of the software component. Who owns that information is not as important as to its existence and accessibility.

5.4.7 Maintenance

Although DO-178B/ED-12B does not actually address maintenance issues, it does suggest the existence of a post-certification change process. Maintenance of software components should be carefully planned and considered by all stakeholders during the original development of the component.

5.5 Summary of Software Components

This section has provided an overview of software components, good practices regarding development and use of software components, and concerns regarding the reuse of software components in safety-critical systems. The author will continue work in this area, as the field of software components continues to grow.

6.0 Object-Oriented Technology in Safety-Critical Systems

Object-Oriented Technology (OOT) is seen by many in the mainstream software community as the “silver bullet” that will take us into the new millennium of software development. OOT is appealing because of the number of available tools, the emphasis on reuse, and the appeal to software designers. It is touted as a technology that saves money, improves quality, and saves time.

However, to date, few safety-critical computer systems have implemented OOT. Safety-critical designers tend to use proven technologies and, as a result lag, a few years behind the mainstream designers of non-safety software. Since OOT has proven to be cost-effective and technically sound for many projects, manufacturers of safety-critical systems are now considering its use.
There are some concerns when using OOT that must be carefully considered. This section will provide an overview of OOT and some of the concerns of using OOT safety-critical software. This is merely the beginning of a more in-depth study and will likely be followed by other studies.

6.1 Overview of OOT

OOT is a software development technique that is centered around “objects.” IEEE refers to OOT as “a software development technique in which a system or component is expressed in terms of objects and connections between those objects” [21]. An object can be compared to a “black box” at the software level – it sends and receives messages. The object contains both code (functions) and data (structures). The user does not have insight into the internal details of the object, thus giving it the comparison to a black box. An object can model real world entities, such as a sensor or hardware controller, as separate software components with defined behaviors.

A major concept in OOT is the “class.” Grady Booch, a champion in OOT methodology, defines a “class” as “a set of objects that share a common structure and a common behavior” [22]. A class contains the attributes and operations that are required to describe the characteristics and behavior of a real world entity. Figure 3 illustrates a representation of a class definition for an object.

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes:</td>
</tr>
<tr>
<td>Operations:</td>
</tr>
</tbody>
</table>

Figure 3 – Object-Oriented Class Representation

6.1.1 Principles of OOT

There are seven principles that form the foundation for OOT: abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence [22]. Not all of these principles are unique to OOT, but OOT is the only development methodology that embodies all seven as a consistent model. Abstraction, modularity, concurrency, and persistence are principles that are commonly used in other development methodologies. However, encapsulation (using a technique called information hiding), hierarchy (using a technique called inheritance), and
typing (using a concept called *polymorphism*) are relatively unique to OOT. Each of the seven principles is described below.

**Abstraction** is one of the fundamental ways that complexity is addressed in software development. “An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer” [22].

**Encapsulation** is the process of hiding the design details in the object implementation. Encapsulation can be described as “the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse” [23]. Encapsulation is generally achieved through *information hiding*, which is the process of hiding the aspects of an object that are not essential for the user to see. Typically, both the structure and the implementation methods of the object are hidden [22].

**Modularity** is the process of partitioning a program into logically separated and defined components that possess defined interactions and limited access to data. Booch writes that modularity is a “property of a system that has been decomposed into a set of cohesive and loosely coupled modules” [22].

**Hierarchy** is simply the ordering of abstractions. Examples of hierarchy are *single inheritance* and *multiple inheritance*. In OOT, when a sub-class is created, this new class “inherits” all of the existing attributes and operations of the original class, called the “parent” or “superclass” [24]. Inheritance is a relationship between classes where one class is the “parent” (also called “base,” “superclass,” or “ancestor”) class of another [25]. One author puts it this way, “Inheritance is a relationship among classes where a child class can share the structure and operations of a parent class and adapt it for its own use” [26].

**Inheritance** is one of the key differences between OOT and conventional software development. There are two types of inheritance: *single inheritance* and *multiple inheritance*. In *single inheritance*, the sub-class inherits the attributes and operations from a single superclass. In *multiple inheritance*, the sub-class inherits some attributes from one class and others from another class. *Multiple inheritance* is controversial, because it complicates the class hierarchy and configuration control [27].

**Typing** is a principle that is used in OOT that has many definitions. Booch presents one of the most clear and concise definitions by stating, “Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways” [22]. Examples of OOT typing are strong typing, weak typing, static typing, and dynamic typing. Each OOT programming language varies in its implementation of typing.

Another OOT concept closely related to typing is *polymorphism*. *Polymorphism* comes from the Greek meaning “many forms.” It allows one name to be used for two or more related but different purposes [23]. It is the ability of an object to assume or become many different forms of object. *Polymorphism* specifies slightly different or additional structure or behavior for an object, when assuming or becoming an object [25]. This allows different underlying implementations for the same command. For example, assume there exists a vehicle class that includes a steer-left command. If a boat object was created from the vehicle class, the steer-left
command would be implemented by a push to the right on a tiller. However, if a car object was created from the same class, it might use a counter-clockwise rotation to achieve the same command.

ConcURRENCY is the process of carrying out several events simultaneously.

Persistence is “the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object’s locations moves from the address space in which it was created)” [22].

6.1.2 OOT Methodology

Everyone seems to have a slightly different perspective of what OOT actually entails. OOT can be described in four phases: Object-Oriented Analysis (OOA), Object-Oriented Design (OOD), Object-Oriented Programming (OOP), and Object-Oriented Verification/Test (OOV/T). The implementation of these phases is typically iterative or evolutionary. An overview of each phase will be addressed below.

6.1.2.1 OOA is the process of defining all classes that are relevant to solve the problem and the relationships and behavior associated with them [27]. A number of tasks occur to carry out the OOA as shown in Figure 4. The tasks are reapplied until the model is completed. As shown in Figure 4, use cases, class-responsibility-collaborator (CRC) models, object-relationship (OR) models, and object-behavior (OB) models are methods typically used to carry out the OOA. The use case is a method utilized to identify the user’s requirements. The CRC model is used to identify the class attributes, operations, and hierarchy. The OR model is used to illustrate the relationship between the numerous objects. And, the OB model is used to model the behavior of each object.

6.1.2.2 OOD transforms the OOA into a blueprint for software construction. Four layers of design are usually defined: subsystem layer, class and object layer, message layer, and responsibilities layer. The subsystem design layer represents each subsystem that enables software to achieve the requirements. The class and object design layer contains class hierarchies and object designs. The message design layer contains the internal and external interfaces to communicate between objects. The responsibilities design layer contains the algorithm design and data structures for attributes and operations of each object.

The Unified Modeling Language (UML) is becoming the standard technique for graphical design of embedded systems [28]. The UML provides a way of smoothly transitioning from OOA to OOD.

6.1.2.3 OOP is the coding phase of the design project, using an object-oriented (OO) language. There are dozens of OO languages. Three of the most well known are C++, Smalltalk, and Java. C++ and Java are of particular interest for designers of embedded software. Java’s platform independence and C++’s tool support make these two languages very appealing to the developers of safety-critical systems. Because of their extreme popularity and the desire for use in safety-critical systems, both C++ and Java will be discussed in sections 6.3 and 6.4, respectively.
6.1.2.4 **OOV/T** is the process of detecting errors and verifying correctness of the OOA, OOD, and OOP. OOV/T includes reviews, analyses, and tests of the software design and implementation. OOV/T requires slightly different strategies and tactics than the traditional structured approach. The variance in the approach is driven by characteristics like inheritance, encapsulation, and polymorphism. Most developers use a “design for testability” approach to begin addressing any verification and test issues early in the program.

### 6.2 Overview of C++

To date C++ has been used in only a few safety-critical systems. However, many developers are seriously considering the use of C++ in future developments. This section will provide an overview of C++, with emphasis on embedded systems, since most safety-critical systems fall into this category.

C++ is an object-oriented language that has many similarities to C. In fact C++ is a superset of C. Typically, C and C++ can run on the C++ compiler. Because of the fact that embedded systems are becoming larger and more complex, the tool support and object-orientation support of C++ are appealing [29]. C has gained quite a bit of popularity in the embedded software community; however, the limited tool support and difficult maintenance issues surrounding C are leading many to start looking at other options. A paper by Dan Saks entitled “How to Evaluate C++ as a Language for Embedded Programming” lists the following advantages of C++ over C [29]:

- “C++ is a better C.” C++ is a superset of C. C++ applies stricter translation-time checking than C does; consequently, C++ compilers detect more errors.
- “C++ supports data abstraction.” C++ supports objects and classes, which makes for easier program maintenance.
• “C++ supports object-oriented programming.” OOA and OOD are easy to implement in C++.

• “C++ partitions the name space better than C.” This reduces the chance of global naming conflicts.

Saks also lists that a couple of disadvantages of C++ are its complexity and its hidden run-time burdens [29]. He states that C++ “compiler diagnostics can be very cryptic. Run-time bugs can be very subtle. C++ places higher demands on tools such as linkers and debuggers” [29]. Overall, C++ typically increases execution time and data and code space [30].

6.2.1 Embedded C++

Because of the above-mentioned weaknesses of C++ in embedded systems, in 1995 a consortium of companies began development of a version of C++ that addresses the concerns of embedded software developers. The dialect of C++ that has emerged is known as “Embedded C++”. The language definition and style guide for Embedded C++ are available free of charge at http://www.caravan.net/ec2plus. Several companies have already implemented Embedded C++ compilers.

Embedded C++ (EC++) differs from C++ as follows [55]:

• Templates were omitted because they can result in code explosion.

• Exceptions were omitted because of the additional run-time overheads in calling functions. (Note: In some cases in C++ overhead is increased, even when code makes no direct use of exceptions.)

• Namespaces were omitted because they add complexity and do not appear to have an adequate payoff in embedded systems.

• Run-time type identification (RTTI) was omitted because it adds complexity without the benefits for embedded systems.

• Multiple inheritance and virtual base classes were omitted because they add overheads, even for code that makes no direct use of these features.

Basically, Embedded C++ is a simplification of C++. This simplification improves the size and efficiency of the run-time library. For example, “eliminating exceptions from the EC++ language eliminates the need to provide exception handling functions or classes in the runtime library. And the lack of RTTI eliminates the need to provide a type-info class” [56]. An article in Computer Design stated, “For embedded systems, a language generating a great deal of interest today is Embedded C++, a subset of ANSI C++ that offers certain advantages for real-time development.” Embedded C++ (EC++) omits some of the “problematic” features of ANSI C++. For example, multiple inheritance, virtual base classes, run-time identification, templates, exceptions, and namespaces are deleted [31].

As mentioned earlier, there have been a few C++ applications in safety-critical systems. There are still quite a few concerns regarding C++ in this area of application.

6.2.2 C++ and Safety
A web-site discussion entitled “Moderated Discussion on C++ and Safety” led by Brian Wichmann generated a lot of discussion on this topic. This discussion attracted world-wide input about C++ applicability to safety-critical projects. The discussion demonstrated how controversial the use of C++ is for safety-critical applications.

Wichmann began the discussion with this thought: “Although the major problem with safety-critical software is getting the requirements correct, the impact of the language is significant... The main problems I see with C++ arise from its 'high-level' nature. For instance, it is hard to show that there is no storage leak or bound the storage requirements statically. Another problem is that in several cases, the order in which an execution is performed is not defined, making it effectively impossible to guarantee predictable execution” [32].

The discussion to Wichmann’s question goes on for eight pages. Some of the more enlightening and relevant discussions are included below in order to illustrate the controversy of this subject. Peter Fenelon stated the following regarding sub-sets of C++: I'd be highly reluctant to see C++ in any safety-related or critical environment. By the time "unsafe" or "difficult" features are ruled out -- I'm referring particularly to exception handling, the use of templates and Standard Template Library (STL), multiple inheritance, and so on -- what's left isn't much more than ANSI C (a language I have far fewer quibbles with, if sensible guidelines are followed)...” [32].

Another interesting comment by Bob Gorman endorsed the possibility of using C++ on safety-critical systems: “It seems to me that many people here are evaluating C++ only as a sum of its features. Of course we can pick apart any language if we only focus its features or lack of them. However, it's the real world implementation of the features that makes the application safe and robust...” [32].

Jim Jaskol wrote: “… The tools, knowledge base, and experience surrounding C/C++ gives it tremendous advantages in many areas over other languages--advantages that can translate into safer systems. C++ has too much of a following to be ignored…” [32].

Additional concerns regarding the use of C++ in safety-critical systems are listed below:

- **Built-in functions.** One area of great concern for C++ programs is the use of certain built-in functions. For example, the use of “new” and “delete” functions can lead to non-deterministic behavior due to dynamic memory management functions (i.e., memory leakage can occur). The developers of safety-critical systems have found that the use of C++ functions and built-in libraries require extensive verification and testing in order to understand their behavior.

- **Run-time bugs.** The “subtle” aspect of C++ run-time bugs is a concern in safety-critical systems [29]. (Please note that all run-time bugs are a concern in safety-critical systems; however, the ones in C++ are more subtle and difficult to detect.)

- **Constructors and destructors.** C++ allows insertion of constructors and destructors by the compiler outside the control of the programmer [33]. Constructors are automatically invoked at entry to a block containing variable declarations. Therefore, in C++ a variable declaration can cause run-time overhead. Additionally, destructors can be invoked by the “delete” statement or automatically when an object goes out of scope [30]. These “automatic” features and run-time overhead cause concern in safety-critical systems.
• **Hidden fields.** Classes in C++ have hidden fields, making them difficult to map into existing data structures [29].

• **Overloading function names.** Two or more functions in C++ can have the same name. “Function name overloading places additional demands on development tools such as compilers, linkers, and debuggers” [34]. This ambiguity causes great concern in safety-critical systems.

• **Multiple inheritance.** This can lead to some unexpected code to be generated, depending on the compiler; therefore, the programmer needs to be aware of how the compiler organizes objects in memory [30]. Any type of unpredictable compiler actions or addition of dead code causes concern for developers of safety-critical systems.

In 1996 David Binkley of the National Institute of Standards and Technology (NIST) published a paper entitled “C++ in Safety Critical Systems.” The paper outlined guidelines for using C++ to create safe software. Adherence to these guidelines can lead to safer and more maintainable C++ programs. The paper also outlined a series of techniques and examples for creating safer C++ programs. The paper is available on the world wide web [35]. Implementation of such guidelines can help to address the short-coming of the C++ language.

### 6.3 Overview of Java

Java was first introduced by Sun in 1995. It was originally conceived as a programming language for embedded software in consumer electronics (e.g., cable television boxes) [36]. However, because of its platform independence, it has become widely used by developers of internet software and desktop applications. Because of its platform independence, widespread use by the mainstream software development community, and original embedded conception, Java has recently become a desirable language for developers of safety-critical systems. Some aviation companies are currently performing studies to determine the feasibility of using Java in safety-critical systems.

Java is “an easy-to-use, object-oriented programming language designed specifically for platform-independence and protection from common programming errors” [37]. Java’s portability is due to the fact that it is an “interpreted language.” The Java compiler produces what is known as “bytecode” that is then fed into a virtual machine. The virtual machine is then responsible for the conversion of bytecode into machine language. Basically, Java bytecode is the same for any platform [38]. Figure 2 shows the Java development and execution infrastructure [37].

The Java virtual machine is currently undergoing a lot of evolution. In some cases, it may actually become part of the microprocessor (i.e., the virtual machine may be implemented in silicon).

#### 6.3.1 Advantages of Java

There are a number of positive aspects of Java. Many of them are listed below:

• Its syntax is similar to C.
• Its primitive datatypes are always of a fixed size [37].
• It has automatic bounds-checking that prevents writing or reading past the end of an array [37].
• It is object-oriented, so it allows encapsulation, polymorphism, and inheritance.
• It simplifies inheritance by eliminating multiple inheritance.
• Its automatic garbage collection simplifies dynamic memory management and reduces memory leaks [37]. (Please note that leaks for safety-critical systems must be contained (e.g., heap management) or eliminated.)
• It organizes exceptions into a logic hierarchy and prevents them from being ignored by the programmers.
• It allows incorporation of software libraries, which encourages separation between programming method and executable object and allows more software packaging (see Figure 5) [37].
• It has strong exception handling capability [39].
• It does not have pointers. This reduces memory leaks and program complexity [40].
• It is independent of the underlying operating system and hardware [41].
• It does not allow the ambiguity of the overloaded operator [40].

![Figure 5 – The Java Development & Execution Infrastructure](image-url)
6.3.2 Concerns with Java in Safety-Critical Systems

As with any other language known to mankind, Java also has a number of disadvantages, particularly for real-time systems. An article in Computer Design explored the use of Java in safety-critical systems. The article stated that a positive aspect of the language is that the absence of pointers and automatic checking for common errors reduces the potential for memory errors. However, Java’s use of garbage collection is problematic, because it is a non-deterministic memory management technique [31]. Below is a list of commonly-know weaknesses of Java that are of particular concern for developers of safety-critical systems:

- It does not allow access to hardware registers and physical memory locations. This is a major downfall for those who develop device drivers and embedded systems [37].
- It requires a large virtual machine and libraries. This takes a lot of memory and time that do not exist in many embedded environments [37]/[42].
- There are features in Java that take a lot of time to run (e.g., automatic garbage collection, dynamic linking, and exception handling) [37].
- The default garbage collection algorithm makes Java non-deterministic (i.e., its maximum length of time to do something cannot be determined in advance) [37]/[43].
- The garbage collector can allow unrecoverable memory leakage [39]/[44].

In order to address the limitations of Java for real-time systems, a “Real-Time Expert Group” was formed in March 1999. The group is developing a real-time Java specification that will address three main areas [45]:

- **Thread model** – to make the behavior of Java’s threads more predictable.
- **Synchronization** – to avoid lower priority threads blocking higher priorities.
- **Memory management** – to provide more predictable garbage collection and freedom from dependency on dynamic allocation from the garbage-collected heap.

Additional areas addressed by the real-time specification are: response to external stimuli, providing accuracy of timing, and asynchronous transfer of control [45]. It is hoped that augmentation of Java to the real-time environment will make it more suitable for real-time, embedded systems [46].

Java is still relatively immature for safety-critical systems. Java is an extremely powerful language; however, it lacks robustness. There are new releases of Java every few months; this fast production of the language does not allow it to become robust. One software developer who uses Java for browsers claims that the browsers “crash” a lot. Frequent crashing may be tolerable for non-safety applications; however, it is not acceptable for a safety-critical system, such as an airplane or a nuclear power plant. It will be interesting to see how this language develops over the next few years.

6.4 Ada 95

Ada 95 can be implemented as an object-oriented language. Many developers of airborne software believe that Ada 95 is the only OO language mature enough for all levels of safety-critical systems. An ISO document entitled, “Guide for the Use of Ada Programming Language in High Integrity Systems” covers the Ada 95 issue well. Therefore, it will not be further explored in this paper.
6.5 Relationship Between OOT and Reuse

One of the main reasons for the industry’s move to OOT has been reuse. The OOA and OOD process helps designers break complex software systems into manageable pieces (i.e., classes and objects) [47]. This makes the design process less complex and more manageable [48]. Additionally, OO-based programming languages make it easy to implement the design into code. When the pieces are clearly defined and easily implemented, they are more likely to be reused on future systems.

Additionally, the model-based development approach used in OOT provides great opportunities for utilizing development tools [49]. For example, code generators can be developed from OO models, making the generation of code more accurate and more easily reused.

Unfortunately, many of the benefits of OOT, including reuse, are not being utilized in embedded systems. Alan Zeichik states, “Whether we like it or not, in most situations object-oriented programming has not succeeded in fostering code reuse, except in the most limited way” [50].

Two different articles by Bertrand Meyer reinforce this circumstance. Meyer states that even though the industry knows about OOT, “only a minority has really understood the deeper concepts and started applying them thoroughly” [51]. In another article he states that “almost everyone pays lip service to the principle of information hiding” [52]. Meyer also states that “most of the industry is far from having integrated in its daily practices the deeper principles of object technology” [51]. He goes on to promote strict discipline of all OO principles (e.g., information hiding, inheritance, polymorphism, etc.) in order to reap the benefits of reuse. As he puts it, “it’s hard to be a little object-oriented” [51]. Without truly implementing OO principles, the benefits of reuse cannot be realized.

6.6 Concerns in Use of OOT in Safety-Critical Software

Most software currently developed for installation in aircraft meets the objectives of DO-178B/ED-12B, as appropriate for the software level. This section will look at some of the issues to be addressed by a software development team using OOT in airborne civil aviation software in order to meet the objectives of DO-178B/ED-12B. Similar issues will exist in other safety-critical domains (e.g., medical or nuclear). This should not be considered a comprehensive study of the issues – there are likely additional issues to be addressed, depending on the specific project details (for example, each OO language and/or compiler may have different certification issues).

Planning. The OOT software development process should be carefully planned and documented. Additionally, the development standards should address any special limitations for the development team to consider (e.g., no multiple inheritance, language sub-sets, etc.).

Traceability. DO-178B/ED-12B suggests traceability is needed from requirements to design to code to test cases and test results. When inheritance is used in the design, special care must be taken to maintain traceability. This is particularly a concern, if multiple inheritance is used. Overall, multiple inheritance is a concern to certification authorities. If used, it should be very carefully applied and addressed in the development standards for the project.
Traceability is made more difficult because there is often a lack of OO methods or tools for the full software lifecycle. For example, tools and methods often cover OOA or OOD but not both. New tools are beginning to address this gap [54].

**Target Compatibility.** A number of DO-178B/ED-12B objectives address the topic of target compatibility. Using classes, instantiation, and automatic memory management typically implies the use of dynamic memory allocation. In typical implementations, dynamic memory algorithms require periodic reorganization of the memory to reduce the inevitable fragmentation. This leads to indeterminate execution profiles. As an alternative to the typical implementation provided by most OO languages, the developer might consider the feasibility of designing a deterministic memory allocation subsystem. Another approach which might be feasible, is to pre-allocate objects during program initialization and avoid creating or deleting them after that. Dynamic memory allocation must be verified in terms of both space (available memory) and execution time in order to determine compatibility with the target.

**Structural Coverage.** DO-178B/ED-12B addresses four forms of structural coverage, which are applicable depending on the software level: statement coverage (Levels A, B, and C); decision coverage (Levels A and B); modified condition/decision coverage (MC/DC) (Level A only); and data and control coupling coverage (Levels A, B, and C). The use of inheritance and polymorphism might cause difficulties in obtaining structural coverage, particularly decision coverage and MC/DC. Source to object code traceability will vary between compilers for inheritance and polymorphism.

**Dead/Deactivated Code.** DO-178B/ED-12B defines dead and deactivated code as follows:

“*Dead code - Executable object code (or data) which, as a result of a design error cannot be executed (code) or used (data) in a operational configuration of the target computer environment and is not traceable to a system or software requirement. An exception is embedded identifiers*” [4].

“*Deactivated code - Executable object code (or data) which by design is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component, or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options*” [4].

DO-178B/ED-12B basically states that any dead code should be removed and deactivated code should be verified (analysis and test) to prove that it cannot be inadvertently activated.

When superclass methods are replaced by sub-class methods (i.e., overridden methods), there is a possibility that dead or deactivated code could be introduced. Structural coverage analysis is intended to address the dead and deactivated code. However, any such occurrences would need to be addressed.

**Verification/Testing.** Test coverage of high-level and low-level OO requirements will likely require different testing strategies and tactics than the traditional structured approach. The characteristics of inheritance, encapsulation, and polymorphism drive the need for the different strategies and tactics. Most developers are using a “design for testability” approach to begin addressing any test issues early in the program.
**Overuse of Inheritance.** Overuse of inheritance, particularly multiple inheritance, can lead to unintended connections among classes [54]. This could lead to difficulty in meeting the DO-178B/ED-12B objective of data and control coupling.

**Ambiguity.** Inheritance, polymorphism, and operator overloading through dynamic or run-time linkage can lead to ambiguity. Polymorphic and overloaded functions may make tracing and verifying the code difficult [54]. Since DO-178B/ED-12B states that the source code should be verifiable, attention should be paid to such issues.

**Coding Issues.** Some OO languages have “features” that could make it extremely difficult or impossible to satisfy the objectives of DO-178B/ED-12B. In many cases, a well-defined sub-set of the language may be identified and documented in the coding standards that will allow compliance to objectives for a given software level. As an example, ANSI C++ has some “features” that might make meeting the objectives of DO-178B/ED-12B very difficult.

For example, some C++ concepts lead to memory leaks [53] or non-determinism [59]. These obstacles might be addressed by including the following restrictions in the coding standards:

- Minimize dynamic binding;
- Minimize operator overloading;
- Minimize control flow complexity;
- Use “new” only at initialization;
- Avoid using “delete”;
- Avoid use of exception handling;
- Avoid multiple inheritance; and
- Avoid type-cast pointers.

**Library Dependence.** The dependence on libraries is a concern for safety-critical systems—it is often unclear as to what is happening in the object libraries. Libraries may not have been developed with safety-critical applications in mind and may not have the integrity required for such applications. Use of libraries must be carefully considered and verified for proper functionality.

### 6.7 Conclusions Regarding OOT

An article in *Computer Design*, entitled “Building Tomorrow’s Embedded Software,” stated, “Size, complexity, and time-to-market issues are causing fundamental changes in how embedded software is written. While there are solutions to borrow from desktop development, they have to be chosen judiciously” [57]. Such is the case with safety-critical systems’ development. The use of OOT in aviation systems is being considered by many developers of airborne software. The jury is still out with respect to its use. There are advantages and disadvantages for all software development methods – OOT is no exception.

Developers should carefully weigh their program needs with the benefits and risks of OOT. There are a number of potential certification concerns, as discussed in this paper. The author and the FAA intend to further investigate each of the certification concerns in more depth in order determine potential risk mitigation strategies.
7.0 Additional Research Needed for Software Reuse in Safety-Critical Systems

This paper has taken a big picture look at some of the software reuse issues in safety-critical systems. Additional research needs to be performed to address some of the details of software reuse. Research in the following areas could help to further address issues of software reuse in safety-critical systems:

- Commercial-off-the-shelf software;
- Software service history;
- Object-oriented technology issues (particularly structural coverage analysis);
- Real-time operating systems;
- Software development and verification tools; and
- Software reuse integration and verification.

8.0 Summary

Software is becoming more dominant in safety-critical systems. In order to reduce the complexity, cost, and time of software development, many manufacturers are looking for ways to reuse software.

This paper has taken a big picture look at software in safety-critical systems and identified some areas of concern. Software components and object-oriented technology in safety-critical systems was explored in some detail to determine issues with such approaches.

For software components, some of the major issues to be addressed are:
- Planning;
- Multiple requirements levels;
- Re-verification in new target computers;
- Porting guidelines and interface documents;
- Partitioning and protection considerations;
- Artifacts; and
- Maintenance.

For object-oriented technology some of the major issues to be addressed are:
- Planning;
- Traceability;
- Target compatibility;
- Structural coverage;
- Dead and deactivated code;
- Verification and testing;
- Overuse of inheritance;
- Ambiguity;
- Coding issues; and
- Library dependence.
These issues lists are not all inclusive. Many of the issues can be overcome with adequate effort.

In addition to this research, additional research is being conducted or needs to be conducted in the following areas for safety-critical systems:

- Commercial-off-the-shelf software;
- Software service history;
- Object-oriented technology issues (particularly structural coverage analysis);
- Real-time operating systems; and
- Software development and verification tools.

The author intends to continue work in the area of software reuse in order to assure safe implementation of software reuse. Additional papers and research findings will be posted on the Aircraft Certification Software web-site.
9.0 References


**Additional Articles Read but not Referenced**

**Java**


**C++**


**Object-Oriented Technology**


**Software Components**


**General Reuse**


**Safety**


