DOT/FAA/AR-07/52

Air Traffic Organization
Operations Planning
Office of Aviation Research
Washington, DC 20591

Object-Oriented Technology Verification Phase 2 Report—Data Coupling and Control Coupling

August 2007

Final Report

This document is available to the U.S. public through the National Technical Information Service (NTIS), Springfield, Virginia 22161.

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

This document is disseminated under the sponsorship of the U.S. Department of Transportation in the interest of information exchange. The United States Government assumes no liability for the contents or use thereof. The United States Government does not endorse products or manufacturers. Trade or manufacturer's names appear herein solely because they are considered essential to the objective of this report. This document does not constitute FAA certification policy. Consult your local FAA aircraft certification office as to its use.

This report is available at the Federal Aviation Administration William J. Hughes Technical Center's Full-Text Technical Reports page: actlibrary.tc.faa.gov in Adobe Acrobat portable document format (PDF).

Technical Report Documentation Page

1. Report No.

DOT/FAA/AR-07/52

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle

OBJECT-ORIENTED TECHNOLOGY VERIFICATION PHASE 2 REPORT–DATA COUPLING AND CONTROL COUPLING

5. Report Date

August 2007

6. Performing Organization Code

7. Author(s)

John Joseph Chilenski and John L. Kurtz

8. Performing Organization Report No.

9. Performing Organization Name and Address

The Boeing Company
P.O. Box 3707
Seattle, WA 98124-2207

10. Work Unit No. (TRAIS)

11. Contract or Grant No.

12. Sponsoring Agency Name and Address

U.S. Department of Transportation
Federal Aviation Administration
Air Traffic Organization Operations Planning
Office of Aviation Research and Development
Washington, DC 20591

13. Type of Report and Period Covered

Final Report–Phase 2

14. Sponsoring Agency Code

AIR-120

15. Supplementary Notes

The Federal Aviation Administration Airport and Aircraft Safety R&D Division COTR was Charles Kilgore.

16. Abstract

This Report documents the results of an investigation into issues and acceptance criteria for the verification (i.e., confirmation in DO-178B) of data coupling and control coupling (DCCC) within object-oriented technology (OOT) in commercial aviation. OOT has been used extensively throughout the non-safety-critical software and computer-based systems industry. OOT has also been used in safety-critical medical and automotive systems and has been introduced in the commercial airborne software and systems domain. However, as with any new technology, there are concerns and issues relating to its adoption within safety-critical systems.

The intent of the structural coverage analyses (confirmation) of DCCC is to provide an objective assessment (measurement) of the completeness of the requirements-based tests of the integrated components. A review of the published literature concerning integration verification found that coverage of intercomponent dependencies as an acceptable adequacy criterion of integration testing in both non-OOT and OOT software was well motivated. This approach is known as coupling-based integration testing. This Report, therefore, recommends the coverage of intercomponent dependencies as a measurable adequacy criterion to satisfy RTCA DO-178B/EUROCAE ED-12B Table A-7 Objective 8.

Three issues requiring additional work are identified. These issues are concerned with whether different levels of coverage for different software levels should be allowed for verification of DCCC. One major issue requiring further work concerns the level of coverage required for the adequate testing of polymorphism with dynamic binding and dispatch. Defining adequate testing for this OOT feature is an active research area with no definitive answer yet. As such, the recommendation in this Report may only be considered an interim solution where polymorphism with dynamic binding and dispatch are concerned.

17. Key Words

Object-oriented technology, Data coupling, Control coupling, Verification

18. Distribution Statement

This document is available to the U.S. public through the National Technical Information Service (NTIS) Springfield, Virginia 22161.

19. Security Classif. (of this report)

Unclassified

20. Security Classif. (of this page)

Unclassified

21. No. of Pages

61

22. Price

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

CAST	Certification Authorities Software Team
CBIT	Coupling-based integration testing
CFG	Control flow graph
DC	Decision coverage
DCCC	Data coupling and control coupling
DDG	Data dependency graph
DUA	Definition-use association
DUM	Definition use matrix
FAA	Federal Aviation Administration
HIT	Hierarchical integration testing
IEEE	Institute of Electrical and Electronics Engineers
MCDC	Modified condition decision coverage
OOT	Object-oriented technology
OOTiA	Object-Oriented Technology in Aviation
PDG	Program dependency graph
PIE	Propagation, infection, execution
RCC	Receiver-classes criterion
SDG	System dependency graph
SQL	Structured Query Language
TMC	Target-methods criterion
UML	Unified Modeling Language

EXECUTIVE SUMMARY

Object-oriented technology (OOT) has been used extensively throughout the non-safety-critical software and computer-based systems industry. OOT has also been used in safety-critical medical and automotive systems and has been introduced in the commercial airborne software and systems domain. However, as with any new technology, there are concerns and issues relating to its adoption within safety-critical systems. Previous Federal Aviation Administration (FAA) research and two Object-Oriented Technology in Aviation (OOTiA) workshops with industry indicate that there are some areas of OOT verification that are still concerns in safety-critical systems.

The FAA is sponsoring this 3-year, three-phase research to provide information for developing FAA policy and guidance for the use of OOTiA systems and to support harmonization with international certification authorities on the use and verification of OOTiA. Phase 1 gathered information on the current state of the industry with respect to the use and verification of OOTiA through an industry survey. Phase 2 (this Report) addresses the verification of data coupling and control coupling (DCCC) (satisfaction of Objective 8 of RTCA DO-178B/EUROCAE ED-12B Table A-7) in OOT software. Phase 3 will address structural coverage at the object-code level (satisfaction of Objectives 5-7 of DO-178B/ED-12B Table A-7) for OOT software.

This Report documents the results of an investigation into issues and acceptance criteria for the verification (i.e., confirmation in DO-178B) of DCCC within OOT in commercial aviation. Review of a number of publications allows one to conclude that the intent of the structural coverage analyses (confirmation) of DCCC is to provide an objective assessment (measurement) of the completeness of the requirements-based tests of the integrated components. A review of the published literature concerning integration verification found that coverage of intercomponent dependencies as an acceptable adequacy criterion of integration testing in both non-OOT and OOT software was well-motivated. This approach is known as coupling-based integration testing. This report, therefore, recommends the coverage of intercomponent dependencies as a measurable adequacy criterion to satisfy Objective 8 of DO-178B/ED-12B Table A-7.

A previous study identified a number of DCCC issues with certain OOT features. This project uncovered some additional issues related to OOT integration testing. This Report provides analyses of how the coverage of intercomponent dependencies addresses the DCCC aspects of these issues. Three issues requiring additional work are identified. These issues are concerned with whether different levels of coverage for different software levels should be allowed for verification of DCCC.

One major issue requiring further work concerns the level of coverage required for the adequate testing of polymorphism with dynamic binding and dispatch. Defining adequate testing for this OOT feature is an active research area with no definitive answer yet. Accordingly, the recommendation in this Report may only be considered an interim solution where polymorphism with dynamic binding and dispatch are concerned.

1. INTRODUCTION.

1.1 PURPOSE.

This Report documents the results of an investigation into issues and acceptance criteria for the verification (confirmation) of data coupling and control coupling (DCCC) within object-oriented technology (OOT) in commercial aviation. This investigation is the second of a three-phase research project undertaken by The Boeing Company on behalf of the Federal Aviation Administration (FAA). The results of the investigation provide information to the FAA to develop policy and guidance for the use of Object-Oriented Technology in Aviation (OOTiA) systems and to support harmonization with international certification authorities on the use and verification of OOTiA. The results also provide guidance for future research projects.

This investigation was guided, in part, by the results from an industry survey conducted during the first phase of this project [1]. Those results provided current and proposed interpretations and practices for the confirmation of software DCCC to satisfy the objectives of RTCA DO-178B/EUROCAE ED-12B (DO-178B hereinafter) [2], with the clarifications given in RTCA DO-248B [3], and how those interpretations and practices relate to OOT.

1.2 BACKGROUND.

OOT has been used extensively throughout the non-safety-critical software and computer-based systems industry. OOT has also been used in safety-critical medical and automotive systems and has been introduced in the commercial airborne software and systems domain [1 and 4]. However, as with any new technology, there are concerns and issues relating to its adoption within safety-critical systems. Previous FAA research [1, 4, and 5] and two OOTiA workshops with industry (see http://shemesh.larc.nasa.gov/foot/ for more information) indicate that there are some areas of OOT verification that are still concerns in safety-critical systems [6].

The FAA requested a 3-year, three-phase research program to investigate OOTiA verification. Phase 1 gathered information on the current state of the industry with respect to the use of OOTiA and the current and proposed verification practices for the resulting OOT software [1]. The Phase 1 results provided input to the Phase 2 work on the confirmation of DCCC (satisfaction of Objective 8 of DO-178B Table A-7 [2]) in OOT software reported herein.

1.3 DOCUMENT OVERVIEW.

Section 1 provides the purpose, background, and general overview of this Report.

Section 2 discusses the intent behind DCCC confirmation, establishes dependency relations as an appropriate identifier of coupling, and establishes verification/coverage of dependencies as an appropriate confirmation measure.

Section 3 contains the specific OOT features investigated, solutions identified, and issues raised during the course of this study.

Section 4 summarizes the results of the study and identifies issues for further study.

Section 5 provides a list of references used in this Report.

Section 6 identifies activities and documents related to the work reported herein.

Appendix A provides a brief overview of the Unified Modeling Language (UML) 2.0 [7] conventions used in this Report.

2. COUPLING = DEPENDENCY.

This section discusses the intent behind DCCC confirmation, establishes dependency relations as an appropriate identifier of coupling, and establishes verification/coverage of dependency relations as an appropriate confirmation measure.

2.1 INTENT OF COUPLING CONFIRMATION.

All processes need an exit criterion assessing the adequacy and completeness of the work performed by that process. Within DO-178B, requirements-based and structural coverage analyses are used to ensure that the requirements-based testing process adequately exercises a program's functions and structure [2]. Typically, structural coverage criteria are divided into two types: control flow and data flow.

Control flow criteria measure the flow of control between statements and sequences of statements in terms of statement invocations, Boolean expressions evaluated, and control constructs exercised (branches taken). Within DO-178B [2], three control flow criteria are used.

- Statement coverage for software Levels A through C
- Decision coverage (DC) for software Levels A and B
- Modified condition decision coverage (MCDC) for software Level A

These measures are traditionally applied intraprocedurally to individual statements within a single component. In OOT terminology, the control flow criteria are applied within the methods of a class.

Data flow criteria measure the flow of data along subpaths (sequences of statements and control constructs) between assignments of values to objects (generally variables) and references to the values of those objects in subsequent uses. Data flow criteria are further subdivided into intraprocedural measures for internal coupling (dependencies) and interprocedural measures for external (i.e., integration) coupling (dependencies). In OOT terminology, the intraprocedural measures are applied within the methods of a class and the interprocedural measures are applied between, or across, the methods of classes (i.e., between multiple methods within the same class, and between multiple methods of different classes). Within DO-178B, no data flow criteria are used, but confirmation of DCCC structural coverage is used for software Levels A through C, as called out in Objective 8 of Table A-7 [2]. This objective references Section 6.4.4.2, Structural

Coverage Analysis, subsection c. Note that, just as with control flow criteria, there are multiple data flow criteria providing varying levels of thoroughness [8]. Some of these criteria do not guarantee statement coverage, while others will guarantee both statement and decision coverage [8].

Note that DCCC can exist in forms that the standard structural coverage criteria may not catch, as shown in figure 1. For example, business rules contained within the data access layer are not monitored by standard structural coverage criteria. Business rules can combine Structured Query Language (SQL) statements, as well as basic conditional statements, that return data that adheres to these rules. How these rules are interpreted will be monitored by standard structural coverage criteria, but at a level below the rules themselves.

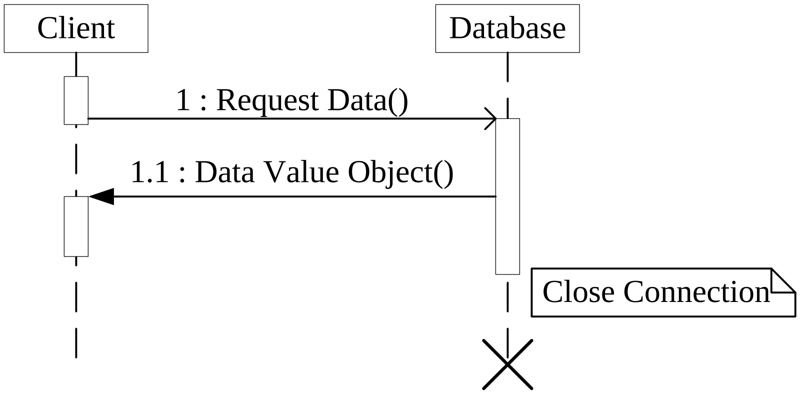


Figure 1. Data and Control Flow in a Client-Database Interaction

Figure 1 shows the control and data flow for a client-database interaction (e.g., between a flight management system and a navigation database) using the UML [7]. The business rules contained in the Request Data method are the control statements that determine what information is returned from the Database to the Client. Control flow criteria would be applied within this method, measuring how the method responds to the business rules that are run against the Database. The data flow is the movement of information from the Database to the Client. Data flow criteria would be applied to the methods that marshal the information from the Database to the Client. The exit criterion is when the database connection is closed by the client after the information has been successfully received. Within standard OOT practice, design patterns that encapsulate the data within objects are used to reduce the chance of external dependencies (e.g., Data Value Object, transfer object) [9].

Given collectively, one can conclude from the following that the intent of the structural coverage analyses (confirmation) of DCCC is to provide an objective assessment (measure) of the completeness of the requirements-based tests of the integrated components.

- Dependencies between software components are mentioned in the definitions for DCCC in DO-178B [2], the material provided in DO-248B [3] FAQ#67, Certification Authorities Software Team (CAST)-19 [10], and Struck [11].

- Interfaces between components are mentioned in DO-248B [3] FAQ#67, CAST-19 [10] and Struck [11].

- Interactions between components are mentioned in CAST-19 [10].

- DO-248B [3] FAQ#67 mentions reviews, analyses, and requirements-based testing against the software architecture being involved in DCCC verification.

- CAST-19 [10] suggests that DCCC measurement should be conducted against the requirements-based testing of integrated components.

- The placement of the confirmation of DCCC objective and its reference in DO-178B [2] is within the section on structural coverage analysis.

- CAST-19 [10] states "the intent of the structural coverage analyses of data coupling and control coupling is to provide a measurement and assurance of the correctness of these modules/components' interactions and dependencies."

- CAST-19 [10] states "the purpose of the structural coverage of the data coupling and control coupling is to evaluate the adequacy of the integrated testing and provides an analysis of the integration activities."

- CAST-19 [10] and Struck [11] propose that the purpose of DCCC verification is to be a completion check of the integration testing effort.

This means that DCCC verification helps to ensure the demonstration of the presence of intended interactions (function) between components and supports the demonstration of the absence of unintended interactions (function) between components. This indicates that the confirmation of DCCC is specifically targeted at the integration process and its tests.

2.2 INTENT OF INTEGRATION.

Integration is the process of combining and verifying multiple components together. The purpose of integration is to ensure that a collection of components operate with each other correctly so that the collection provides required functionality, performance, and reliability. This requires the verification of the interfaces (data) and relations (interactions) between the components.

The interface between components is any means by which they share information. This includes parameters passed during a call, or a series of calls, as well as any shared (global) objects. These shared objects may be accessed either directly, generally by name, or through an alias, generally through a pointer. This considers such things as:

- The size and structure of the data (e.g., Current_Altitude is a signed 32-bit Institute of Electrical and Electronics Engineers (IEEE) floating point number), along with the valid values (states) the data can assume and the valid operations that can be applied to the data. This is generally referred to as the type. The rules of strong-typing and type-safety require that both components have the same view of the size and structure of the data, its valid values and the valid operations.

- The meaning (i.e., semantics) behind the data (e.g., Current_Altitude is the number of feet above mean sea level). Note that the meaning goes beyond the type of the data, though modern languages supporting abstract data types can incorporate a great deal of the semantic information into the type system. Good practice requires that both components use the same semantics for the data (i.e., have the same understanding of the meaning of the data) [12].

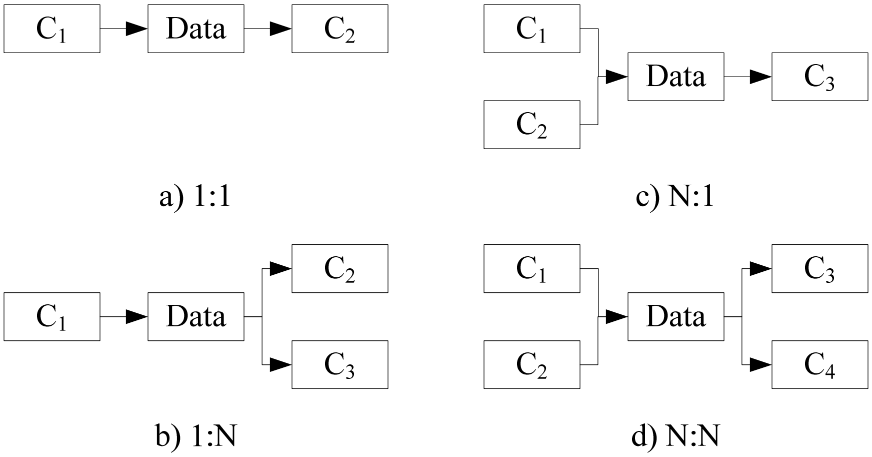
- The type of information flow interaction (data dependency) between the components. Figure 2 uses data dependency graphs (DDG) to illustrate the different types of information flow interactions. DDGs show a data item, the components that write to it (generally on the left) and the components that read from it (generally on the right). Directed edges connect writers to the data, and the data to the readers.

- In figures 2(a) and 2(b) there is one writer and one (1:1) or multiple (1:N) readers, respectively. The interoperation between the components in these two types of interactions is simple and straightforward—the writer needs to write the data before the readers need to read it (e.g., Current_Altitude is written by the altimeter and read by the Primary_Flight_Display).

- In figure 2(c), there are multiple writers and one reader (N:1). The interoperation between these components is not so simple as the previous cases with a single writer, since now the correct writer needs to write the data for the reader to read it under the correct circumstances (e.g., the Primary_Flight_Display will read and display the Current_Speed, which is provided by the Ground_Speed function when the aircraft is on the ground, and the airspeed function when the aircraft is in flight).

- In figure 2(d), there are multiple writers and multiple readers (N:N). This is the most complex interaction, since the correct writer needs to write the data before a particular reader reads that data under the correct circumstances.

Multiple writers providing data under the correct circumstances require logic to control when the appropriate writer is used. This logic could be neatly incorporated into the logic of a single component, or could be distributed across multiple components and be visible only at the system level. DO-178B requires verification of logic at the DC and MCDC levels for software at Level B and Level A, respectively [2]. However, the current practice of applying the criteria only measures the logic within a single component. Logic distributed across multiple components may require something beyond the current practice.



a) 1:1

c) N:1

b) 1:N

d) N:N

Figure 2. Data Dependency Types

The relations between components include such things as:

- Sequencing—a requirement on the execution order of components, e.g., component C_1 must execute before component C_2, component C_3 calls component C_4. Certain forms of call trees can be used to both express these sequencing requirements and to document the implementation. When the traditional input-process-output functional decomposition model is used for architecting a system and its components, sequencing requirements are generally due to some form of data dependency between the components. Earlier components prepare outputs that become inputs for later components. Traditional structure charts and data flow diagrams can be used to both express these sequencing requirements and to document the implementation. UML Sequence and Activity diagrams [7] can also be used to both express these sequencing requirements and to document the implementation.

- Timing—a requirement on time between components, e.g., component C_1 must execute 37 ms before component C_2, component C_4 must complete execution within 25 ms after being called by component C_3. For real-time embedded systems, timing requirements are generally due to the necessary interactions between the system and the rest of the world. Timing diagrams can be used to both express these timing requirements and to document the implementation.

- Dependency—statements in component C_1 influence the execution of statements in component C_2. In particular, for integration verification, faults in the statements of component C_1 affect the execution of statements in component C_2 [13].

Sequencing and timing are concerns that have well-established disciplines for their specification and verification. As identified previously, methods already exist to document both the requirements and the implementation. Checking that the implementation is in conformance with the requirements is part of the verification of these relationships. Dependency relations require further explanation, which is provided in the following sections.

2.3 DEPENDENCY—GENERAL.

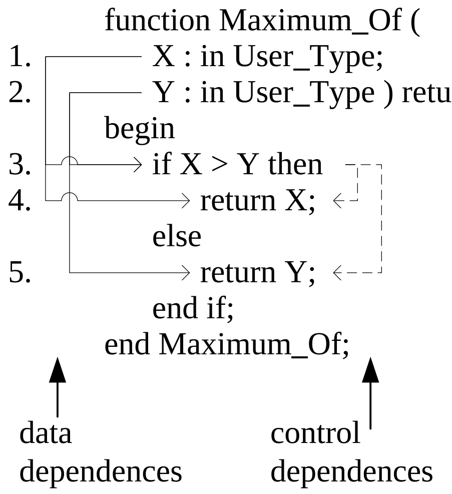
Dependency relations require an interface between the components. This interface can be direct between call-pairs (e.g., component C_1 calls component C_2, and both components access object A), or indirect (e.g., component C_1 calls components C_2 and C_3 at different times, either directly or indirectly, and both C_2 and C_3 access object A). Dependency relations are also known as coupling [13]. Many types of dependency relations exist [14]:

- Syntactic—one statement affects the execution of another based on the syntax of the programming language used.

- Semantic—one statement affects the execution behavior of another during execution. A semantic dependency always implies a syntactic one (i.e., a semantic dependency is always allowed by the syntax of the programming language used). A semantic dependency can be executed dynamically (i.e., witnessed) during some operational scenario of the system containing the software.

- Control—one statement controls the execution of another statement. This may be either syntactic or semantic. In figure 3, statements 4 and 5 are control dependent on statement 3, since the outcome of statement 3 determines which statement will be executed. All the statements in figure 3 are control dependent on statements in other components that call the function Maximum_Of, since none of the statements will execute unless a call is made.

- Data—one statement defines the value of an object used in another statement. This may be either syntactic or semantic. In figure 3, the statements labeled 3 and 4 are data dependent on the statement labeled 1 because X is defined in the statement labeled 1 and used in the statements labeled 3 and 4. Likewise, the statements labeled 3 and 5 are data dependent on the statement labeled 2 because of Y.



```
          function Maximum_Of (
1.            X : in User_Type;
2.            Y : in User_Type ) return User_Type is
          begin
3.            if X > Y then
4.                return X;
              else
5.                return Y;
              end if;
          end Maximum_Of;
```

Figure 3. Maximum_Of and Dependencies

Dependencies are generally represented in a program dependency graph (PDG) [15 and 16]. Figure 4 shows the source code for function Maximum_Of from figure 3 on the left along with the corresponding PDG on the right. A PDG is a directed graph consisting of a set of nodes representing either the entry points or computations within the design or code and a set of directed edges, expressed as ordered node pairs (n_{source}, $n_{destination}$), representing either control dependencies or data dependencies. The directed edge flows from the source node to the destination node. In figure 4, the entry node carries the name of the function.

```
    function Maximum_Of (
1.       X : in User_Type;
2.       Y : in User_Type ) return User_Type is
    begin
3.       if X > Y then
4.            return X;
         else
5.            return Y;
         end if;
    end Maximum_Of;
```

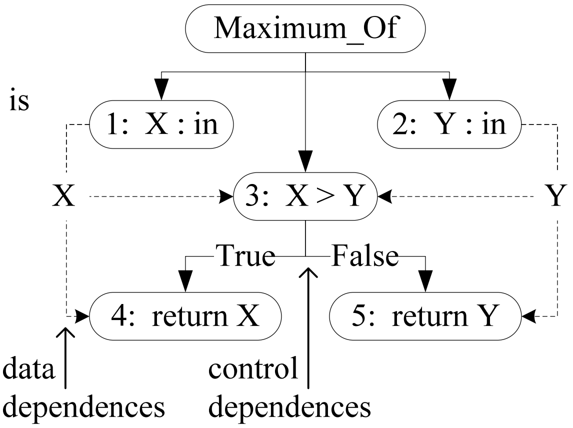


Figure 4. Maximum_Of Program Dependency Graph

A control dependency edge flows between two nodes and is labeled with the predicate outcome if traversal of the edge is conditional on some predicate in the source node. In figure 4, there are unconditional control dependency edges between the entry and statements/nodes 1, 2, and 3. This means that if function Maximum_Of is called and entered, these statements in the source code and the corresponding nodes in the model will be executed as long as no exceptions are raised. There are conditional control dependency edges between statement/node 3 and statements/nodes 4 and 5. If the predicate in statement/node 3 is True, then statement/node 4 will be executed, otherwise statement/node 5 will be executed. One significant difference between a control flow graph (CFG), or flowchart, and a PDG is that in a PDG, not all outcomes of a decision statement (branch point) need be represented with a control dependency edge [15 and 16]. Figure 5 depicts an example CFG on the left and the corresponding PDG with control dependency edges only on the right where not all outcomes of branch points in the CFG are represented with control dependency edges in the PDG. Note that statement/node 4 is control dependent on statement/node 1 being True, as statement/node 4 is reached when statement/node 2 is both True and False.

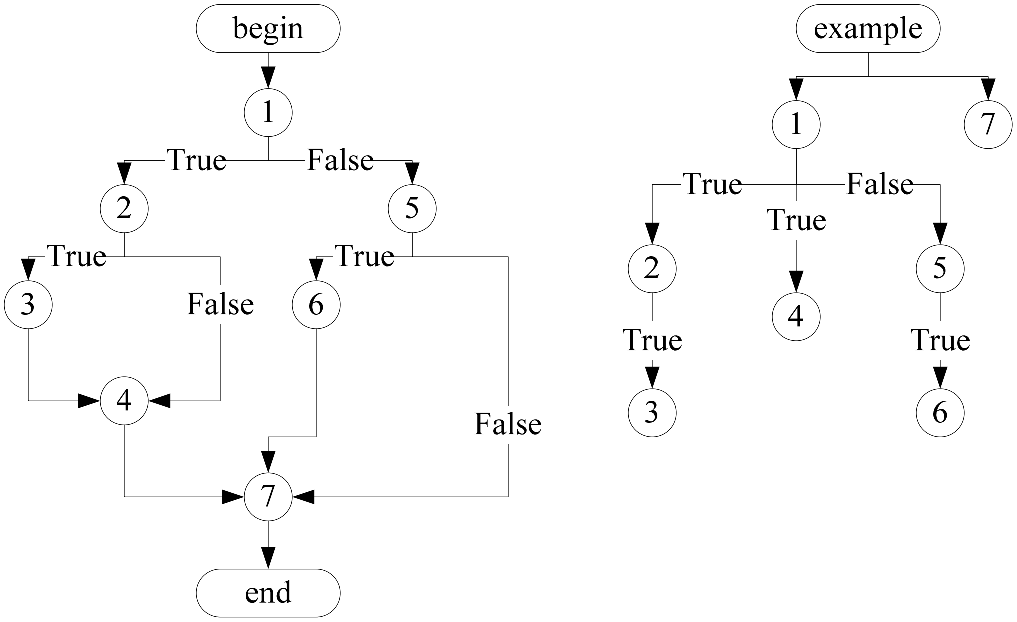


Figure 5. The CFG PDG Comparison

A data dependency edge flows between two nodes where the source node defines the value of some object that is used in the destination node and is labeled with the name of the object. In figure 4, there are data dependency edges between statement/node 1 and statements/nodes 3 and 4. These edges are labeled “X,” since X receives a value in statement/node 1 that is used in statements/nodes 3 and 4. A similar data dependency for Y is shown for statement/node 2 and statements/nodes 3 and 5.

A data dependency edge that flows into a branching predicate is considered by some to be a part of the control dependency of the branched-to statements/nodes. In figure 4, the data dependency edges between statements/nodes 1 and 3 for X and statements/nodes 2 and 3 for Y are examples of this form of special data dependency. This makes sense as the execution of statements/nodes 4 and 5 is dependent on the values of X and Y in statement/node 3, which were defined in statements/nodes 1 and 3, respectively. This is also in conformance with the definition for control coupling in DO-178B [2] since the statement (component) defining the data is influencing the execution of the other statement (component).

Data dependencies are generally represented with a definition-use pair (du-pair) [17]. A du-pair is a pair of statement/node numbers that represent where an object is defined and where that definition can be used. For the definition to be used, at least one subpath (sequence of statements) between the definition and use must be definition clear (i.e., not contain another definition of the object). The two intracomponent du-pairs for X in figure 4 are (1,3) and (1,4), respectively, while those for Y are (2,3) and (2,5).

A du-pair can be extended into a definition-use association (DUA) by adding a predicate that defines when the du-pair exists. For a du-pair to exist, there must be at least one definition-clear subpath between the definition and the use. This means that the predicate in the DUA defines all the definition-clear subpaths for the du-pair. For the du-pair (1,3) for X in figure 4, this predicate would be that the function Maximum_Of is entered without an exception being raised. This would make the DUA (1, 3, Enter(Maximum_Of)), where the function "Enter" means that the identified subprogram must be entered without an exception being raised. For the du-pair (1,4) for X in figure 4, the predicate would be that the function Maximum_Of be entered without an exception and that the value of X at statement/node 3 be greater than the value of Y at statement/node 3. Since the values of X and Y at statement/node 3 are just the values on entry, the DUA would be (1, 4, Enter(Maximum_Of) and (X > Y)).

A syntactic dependency can be identified with a static analysis of the code. The analysis needs only to identify where objects are defined and where those same objects are used when a definition-clear control-flow path exists between the definition and the use. Figure 6, on the left-hand side, shows the syntactic dependencies for object A. Object A has definitions and uses on all lines. Note that there are definition-clear paths from both lines labeled 1 and 2 to both lines labeled 3 and 4, which leads to the following four syntactic intracomponent du-pairs for A are (1,3), (1,4), (2,3), and (2,4).

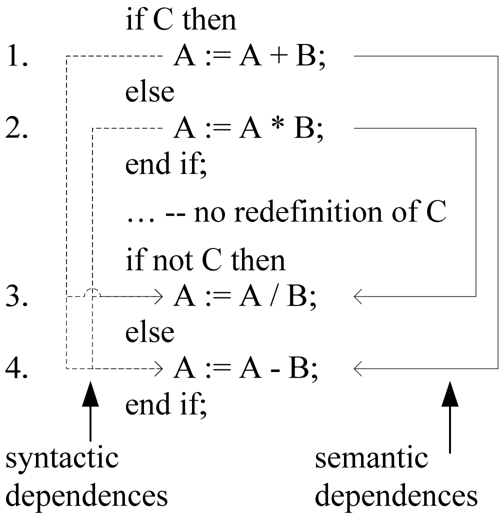


Figure 6. Syntactic vs Semantic Dependencies

However, a syntactic dependency may not be realizable with execution data (i.e., there is no data that would cause that dependency to be executed). Semantic dependency overcomes this problem by requiring that the dependency be executable. Figure 6 shows the difference between syntactic dependencies on the left and semantic dependencies on the right of the code. Since there is no redefinition of C between the two if-statements, when C is True, the statements labeled 1 and 4 will be executed, and when C is False, the statements labeled 2 and 3 will be executed. This means the only realizable dependencies and intracomponent du-pairs for A are (1,4) and (2,3).

Semantic dependency requires that one statement affects the execution behavior of another. This is demonstrated by either modifying one statement, or changing the value of one statement, to see if the execution of the other statement is changed [18 and 19]. In figure 6, if the statement labeled 1 is changed from a sum to a difference (i.e., changed from A: = A + B to A: = A – B), then for all nonzero values of B, the result at the statement labeled 4 will differ between the two executions. This establishes that there is a semantic dependency between the statements labeled 1 and 4. A corresponding analysis can be done to establish the semantic dependency between the statements labeled 2 and 3.

Unfortunately, determining if a dependency is semantic is generally indeterminable [14]. However, it has been shown that reachability over the PDG, that is, finding program data that will execute (witness) the dependency represented by the PDG edge (i.e., cover the PDG edge), is a conservative approximation of semantic dependency [14]. That is, semantic dependency implies PDG reachability, but PDG reachability does not imply semantic dependency.

Covering all PDG edges is the same as achieving four levels of coverage:

- Executing all statements for the unconditional control dependency edges and those conditional control dependency edges leading to statements (statement coverage, partial decision coverage, partial MCDC).

- Executing those branch point outcomes that lead to statements for the conditional control dependency edges (partial decision coverage, partial MCDC). Recall that not all branches of a branch point need be represented in a PDG.

- Executing all executable du-pairs for the data dependency edges. This leads to execution of all statements containing definitions and uses of objects (partial statement coverage, partial MCDC). This may lead to executing branch point outcomes that do not have a corresponding PDG control dependency edge but form a definition-clear subpath (partial decision coverage, partial MCDC).

- Executing all subprogram calls for both forms (control and data) of dependencies (a byproduct of MCDC, therefore, partial MCDC).

Table 1 compares the levels of coverage identified in Table A7 of DO-178B [2] along with those of PDG edges beyond that currently in DO-178B [2]. In table 1, the first column identifies the four levels of coverage. Note that only the branch point/branches portion of DC is considered. The second through fifth columns show which of the coverage criteria are required by software Levels A, B, and C, and PDG edges, respectively. The point where a particular coverage criterion is satisfied by a software level, or by PDG edges, is indicated by an X in the row/column intersection. Note that covering PDG edges will require coverage beyond that required for all software levels. Note also that covering PDG edges does not satisfy all coverage required for Levels A and B software.

Table 1. Coverage Comparisons

	Level A	Level B	Level C	PDG Edges
Statement coverage	X	X	X	X
Decision (branch point) coverage	X	X		
MCDC	X			
Coverage of calls	X[1]			X
Coverage of du-pairs				X

[1]Byproduct of MCDC.

Covering all of the realizable dependencies may lead to excessive testing. When the cost of discovering valid semantic dependencies is less than the cost of executing the additional test cases, analysis to identify the unnecessary tests is justified. The Propagation, Infection, Execution (PIE) model is one of the approaches for fault-based testing [18 and 19]. The PIE approach can also be used to determine semantic dependencies. During one part of the PIE analysis, statements are mutated and the changes in execution between the original program and the mutated one are identified. During another part of the PIE analysis, the results of computations are mutated and the changes in execution between the original program and the mutated one are identified. Statements that execute differently (e.g., branch points that branch differently, computations that compute different results) are semantically dependent on the statement that was either mutated or had its outcome mutated. Note that with all nonexhaustive techniques, the PIE technique is not guaranteed to identify all semantic dependencies.

2.4 DEPENDENCY—INTEGRATION.

This section examines the use of dependency for integration.

2.4.1 From Intracomponent to Intercomponent Analysis.

For integration verification, there is a concern with relations between components. To extend dependency relations from intracomponent to intercomponent usage, one of the statements in the definitions and examples given in section 2.3 must be in one component, while the other statement resides in a different component.

Note that reference to the object can be direct, generally by name, or indirect through an alias. An alias is an object name that can refer to other objects. The other objects do not need to have the same name, but generally need to have the same, or compatible, type. Examples of aliases are the formal parameter names for the actual parameters and pointers. Read-only formal parameters, also known as pass-by-copy or in-mode, will represent uses of the actual parameters. Write-only formal parameters, also known as out-mode, will represent definitions of the actual parameters. In some languages (e.g., Ada95), these parameters are allowed to be read after an initial assignment, in which case these reads will represent uses of the actual parameter. Read-write formal parameters, also known as pass-by reference or in-out-mode, will represent both definitions and uses of the actual parameters. Pointers are equivalent to read-write parameters.

The previously mentioned extension to intercomponent dependencies means that for a particular object, there will be a definition of that object in one component and the usage of that object definition in another. The object will now have an intercomponent du-pair. As mentioned previously, the two components can be involved in a direct call or invocation (e.g., tasking, interrupt) relationship (e.g., one calls the other, one rendezvous with the other) or an indirect one. Figure 7 gives some examples of the cross component dependencies.

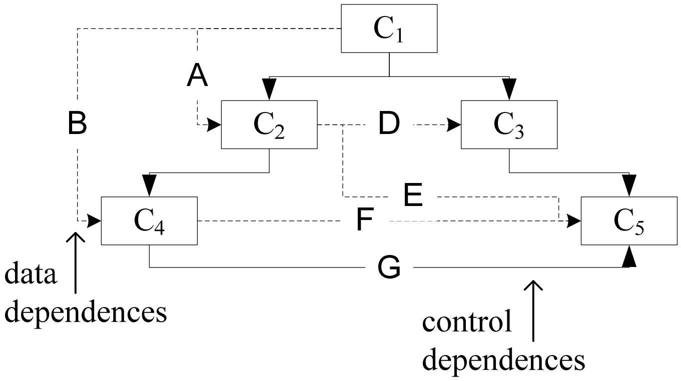


Figure 7. Intercomponent Dependencies

Within figure 7, components C_2 and C_3 are shown as control dependent on C_1 as C_1 calls C_2 and C_3. Similar control dependencies exist between components C_2/C_4 and C_3/C_5. There is also a control dependency between components C_4 and C_5 due to a shared object named G. This is an example of the special form of data dependency mentioned previously where G receives a definition in component C_4, and that definition is then used in the predicate of a branch point in component C_5. During this study, it was determined that handling this special form of data dependency as any other (i.e., normal) data dependency does not appear to compromise the verification in any way and is illustrated for completeness.

Several forms of normal data dependency are also shown in figure 7. Data dependency between direct call-pairs is shown by object A between components C_1/C_2 and between indirect call-pairs by object B between components C_1/C_4 (indirect because C_1 calls C_2 which in turn calls C_4). Data dependencies between components that are not part of the same calling subtree are shown by object D between components C_2/C_3, object E between components C_2/C_5, and object F between components C_4/C_5. The details behind these forms of dependencies will be discussed in sections 2.4.2, 2.4.3, 3.1, 3.2, 3.5, and 3.8.

Figure 7 represents a simplified view of the dependencies between components. PDGs are extended to system dependency graphs (SDG) to model multicomponent systems (i.e., programs with subprograms, generally known as procedures and functions) [16]. An SDG consists of a collection of individual PDGs connected together with additional nodes representing calls/returns and additional dependency edges based on the relationships that cross calls/returns. Strictly speaking, an SDG is a PDG for a larger component. In figure 7, the nodes and edges within the component PDGs have been suppressed, and only the dependency edges between the components have been shown. Note that this simplification results in a loss of information, specifically whether there are multiple accesses to the objects within the components and whether these accesses are conditional or not (i.e., nested within a conditional statement).

SDGs come in many forms and are known by many names. Some of these forms enhance CFGs, while others enhance call graphs. Representations of SDGs enhancing CFGs are given in references 13, 17, 20-24 and others not referenced in this report. One representation of SDGs enhancing call graphs is given in reference 25. Matrix versions of these graph representations are also employed by some [25].

One matrix representation, adapted from reference 25, consists of building a matrix where the rows identify the object, the columns identify the subprograms (methods in OOT), and the intersections are annotated with D if the subprogram defines the object, with U if the subprogram uses the object, and left blank otherwise. A version of this intercomponent definition-use matrix (DUM) for the dependencies expressed in figure 7 is given in table 2.

Table 2. Intercomponent Definition-Use Matrix

	C_1	C_2	C_3	C_4	C_5
A	D	U			
B	D			U	
D		D	U		
E		D		U	
F				D	U
G				D	U

Within table 2, six intercomponent du-pairs are identified. It appears that all that is needed to realize these intercomponent du-pairs is a component execution sequence where the definers are executed before the users. However, two types of information may be found in an SDG that are not found in DUM: (1) multiple occurrences of definitions and uses within a component and (2) whether these definitions and uses are conditional (i.e., nested within a conditional statement). Because of this, DUMs must be extended to capture this information when necessary. One extension is to list multiple definitions and uses in the matrix intersections and the predicates under which the definitions and uses are executed. This extension changes a DUM from listing the components of intercomponent du-pairs to listing the components of intercomponent DUAs.

Comparatively, all the previous dependency representations (e.g., PDG, SDG, DUM) connect definitions of objects in one component with uses of those definitions in another component. All of these representations are different variations on the DDG. In the DDG, components are shown linking to the data without regard to how many instances of definitions or uses there are in the components. In the PDG and SDG, statements containing definitions are shown linking to other statements containing uses with the links identifying the data. This captures the essential property of components having multiple definitions or uses and their conditionality. In the DUM, the intersections between subprograms and objects show the link(s). DUMs need to be extended to capture multiple definitions and uses, when present, and their conditionality. Any representation and analysis that captures all of the nonredundant intercomponent dependencies (i.e., intercomponent du-pairs) are considered acceptable.

In a previous study, information flow analysis was identified as providing support for DCCC verification [1]. In information flow analysis, the inputs and outputs of subprograms are specified during design and checked for conformance in the implementation [26]. Additionally, the dependencies of outputs on inputs can also be specified (e.g., X depends on Y and Z in the assignment X := Y + Z). Inputs and outputs include both parameters as well as globals. This analysis provides a list of the data items defined and used in each subprogram. When globals are used, they can be directly listed in a DUM (or other equivalent representation). When parameters are used, the actual parameters can be listed in the DUM. One benefit of the information flow approach is that information is available during the design phase, before code is available. Once code is available, it is confirmed to be in conformance to the design, and further details can be added to the analysis (i.e., actual parameters are used to refine the dependencies on formal parameters). To verify the linkages between components, further analysis and testing is required.

2.4.2 Non-OOT Intercomponent Dependency Development.

Coverage of intercomponent dependencies for integration testing was suggested by Harrold and Soffa [17 and 20]. This work was then expanded upon by Jin and Offutt [13, 21, and 22] who introduced the phrase "coupling-based integration testing" (CBIT). Both groups adapted standard data-flow coverage criteria to apply interprocedurally, allowing different levels of coverage criteria. Harrold and Soffa followed the more traditional data-flow-dependency analysis approach, which requires whole program analysis and tracking of aliases. The benefit of this approach is that all intercomponent du-pairs are identified, even when there is no calling relationship between the components. Jin and Offutt limited their analysis to direct call-pairs, which simplifies the analysis by only requiring the tracking of parameters. During the course of this research, no evidence was found to indicate which approach should be chosen.

Jin and Offutt suggested CBIT as partial satisfaction of the DO-178B [2] objectives for DCCC and showed how the criteria could be used when integrating atomic components, as well as collections of components (subsystems). They also demonstrated that in one study, CBIT detected more faults with fewer test cases than other methods for integration testing.

Support for coverage of dependencies is provided not only in research reports and papers, some of which are cited throughout this report, but in published books for practitioners as well. Coverage of intracomponent dependencies is discussed in many non-OOT testing books [12, 27-30]. These books point out that coverage of dependencies, specifically data-flow oriented testing, addresses a set of concerns that are missed by the control-flow approaches. Coverage of intercomponent dependencies for integration testing, though using dissimilar approaches, is discussed in three of those books by two different authors [12, 28, and 29]. All of these books follow the more traditional data-flow-dependency analysis approach, possibly because they were published before the work of Jin and Offutt.

Coverage of data dependencies by testing has been shown to be more effective, though costlier, than coverage of control dependencies [29]. The result concerning cost is in general agreement with the analysis presented previously in table 1. This project was unable to formally address either the cost or effectiveness issues.

2.4.3 Object Oriented Technology Intercomponent Dependency Development.

Object-oriented programming provides features not available in procedural software. OOT software centers on a class (fundamental concept); inheritance, aggregation and association (fundamental building blocks); encapsulation and information hiding, and polymorphism and dynamic binding (fundamental principles and building blocks) [5]. However, procedural programming controls the flow between the methods and operations applied to the objects of the classes. Therefore, methods useful for procedural integration can be used for object-oriented integration.

Several authors have extended CBIT to OOT [23, 24, 31-38]. These OOT CBIT approaches fall into two camps: those following the more thorough whole-program analysis approach first put forth by Harrold and Soffa [31, 33, 34, and 37]; and those following the call-pairs approach first put forth by Jin and Offutt [23, 24, 32, 35, 36, and 38]. Both of these camps had two independent sets of researchers following the respective approach. In addition, the OOT CBIT approach has been extended by other authors to a higher level of abstraction, using UML to define the dependencies based on messages and interactions.

In contrast to the non-OOT testing books where CBIT is rarely mentioned, OOT testing books discuss CBIT frequently [25, 30, 39, 40, and 41]. Each of these books uses different terminology and methodology to accomplish CBIT. Marick [30] and Siegel [39] use higher-level representations with a methodology close to the Harrold and Soffa approach. Note that these two books were published before Jin and Offutt. Bashir and Goel [25] use higher-level representations along with the Jin and Offutt call-pairs approach. Binder [40] uses higher-level representations and the Harrold and Soffa approach. McGregor and Sykes [41] use higher-level representations and a methodology close to the Harrold and Soffa approach (i.e., not restricted to call-pairs).

Inheritance and polymorphism has been shown to introduce new error classes and faults not present in non-OOT [42 and 43]. The fault detection capabilities of CBIT for OOT have been found to be effective against these OOT-specific faults, and more effective than control coverage [42 and 44]. CBIT for OOT has also been found to be more effective against integration/interface faults than other traditional approaches (i.e., smaller test sets and higher fault detection rates) [23 and 24]. Both of these results, the presence of CBIT in OOT testing books and its effectiveness, are not surprising as OOT is a data-flow paradigm.

Intracomponent dependencies (i.e., intracomponent du-pairs) are an unambiguous and automatable identification of coupling between statements. Many optimizing compilers already perform dependency analysis, and commercial tools performing this analysis (e.g., CodeSurfer® from GrammaTech) are available. Extension to coverage of intercomponent dependencies for integration verification has been well motivated in both the academic and practical literature for both non-OOT and OOT software. As previously mentioned, coverage of intercomponent dependencies for integration verification provides better results than traditional approaches for integration for both non-OOT and OOT software. Coverage of the feasible dependencies is an unambiguous, measurable, and automatable adequacy criterion for the integration testing process.

It is recommended that intercomponent dependencies (i.e., PDG/SDG edges or intercomponent du-pairs + calls) be covered during verification by a combination of reviews, analyses, and tests to satisfy the DO-178B objectives for confirmation of DCCC. Coverage to this level will ensure that all definitions for each object reach all feasible uses of that definition, all uses of each object are reached by all feasible definitions of that object, and all subprograms are called under some requirements-based test (operational scenario). This approach forms the foundation for the analyses performed in section 3.

3. OBJECT-ORIENTED FEATURES.

The specific OOT features investigated, solutions identified, and issues raised during the course of this study are documented within this section of the report. A previous study [5] identified DCCC concerns with the following OOT features:

- Class
- Encapsulation and information hiding
- Exceptions
- Implicit type conversions
- Inheritance
- Polymorphism with Dynamic Binding

During the course of this study, integration dependency concerns not identified in the previous study were discovered concerning the following OOT features:

- Inheritance (Note: listed twice due to two disjointed sets of concerns)
- Aggregation
- Association

Each of the OOT features is covered in its own following subsection. When appropriate, how dependency relations handle the DCCC concerns associated with the feature is discussed.

Note: The analysis performed in this report, as well as the references cited, assume subtype inheritance that conforms to the Liskov substitution principle [45]. This means that a subclass must accept all messages that its superclass will accept and it must produce appropriate results. As a result, “subclass objects can be substituted for superclass objects without causing failures or requiring special case code in clients” [40]. This ensures that “the objects of the subtype ought to behave the same as those of the supertype as far as anyone or any program using supertype objects can tell” [45].

3.1 CLASS.

A previous study [5] identified class helper functions, known as constructors, as having an effect on DCCC. Constructors are responsible for creating an object of a class at the beginning of its existence and can be used for initializing the attributes of the object necessary to establish its initial state. When constructors are used to set state variables, a strong coupling exists between the constructor's data input (parameters) and any method or class that employs state variables for control or data flow. Additional couplings to other classes in the hierarchy are possible through aggregation and association, with a ripple effect possible throughout the entire system. Aggregation is addressed in section 3.6 of this report, while association is addressed in section 3.7.

Errors can result due to all helper functions and their side effects, not just the constructors mentioned. Destructors and Finalizers are methods responsible for freeing memory or closing some resource when the object has completed its work at the end of its existence. Further discussion on helper functions is provided in the following sections. The first section discusses constructors, while the second discusses destructors and finalizers. Because the constructor coupling problem is recognized in the OOT community, patterns have been developed to deal with the problem.

3.1.1 Constructors.

Constructors set state variables by having the caller pass in the values for its attributes as arguments in the constructor. Figure 8 shows a class (TASCalculator) with two constructors (the TASCalculator methods) where the internal variables set in the constructors cause the class to acquire state information. These internal state variables effect (determine) both the data and control flow within the class when the airspeed calculation is made in method getSpeed as indicated in figure 9.

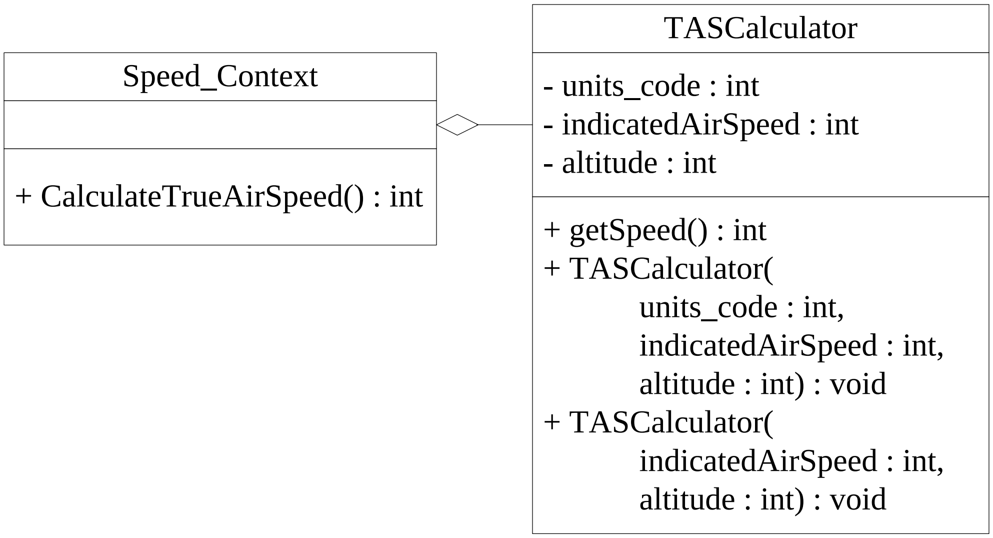


Figure 8. Open Class for Calculating Airspeed

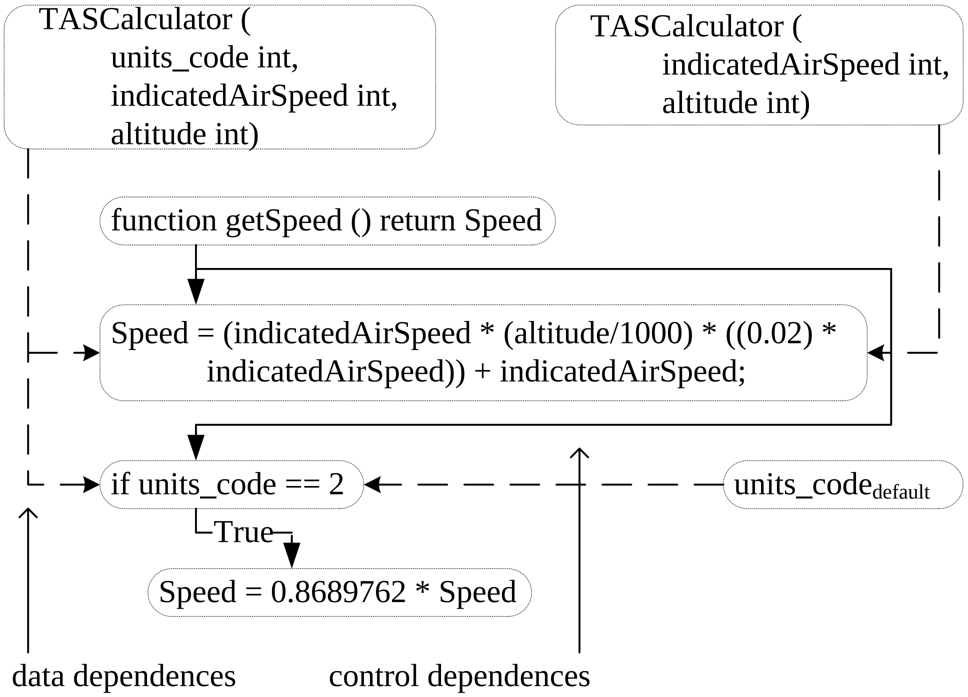


Figure 9. Constructor Dependencies

Figure 9 shows that the operation of the getSpeed method is dependent on which constructor is used to create the TASCalculator object. One of the constructors, depicted on the left of figure 9, allows the user to specify whether the returned airspeed will be returned in knots by specifying whether the units_code attribute is set to the value 2 (for knots), while the other constructor, depicted on the right of figure 9, will use a default value. Defining the dependency relations between the different constructors helps to identify those methods sensitive to state. These state-sensitive methods will generally have different dependency relations between the different constructors. As the analysis in figure 9 shows, the getSpeed method is one such method. The differences in dependency require verification.

Within standard OOT practice, there are design patterns that encapsulate the operations that set the internal state of an object into a separate helper class. These patterns also decouple the data and control flow. One such pattern, the Bridge pattern, separates the implementation from an abstraction for performing some function [9]. In figure 10, the Bridge Pattern is applied to the class for calculating airspeed shown in figure 8.

Within figure 10, all implementation is contained within the Airspeed Implementer class, while the data is now encapsulated within its own object (AirSpeedData). All data flow takes place within the private getAirspeedData method where the state values are assigned to the AirSpeedData object. The AirSpeedData object is an example of a J2EE design pattern, the Transfer Object [46].

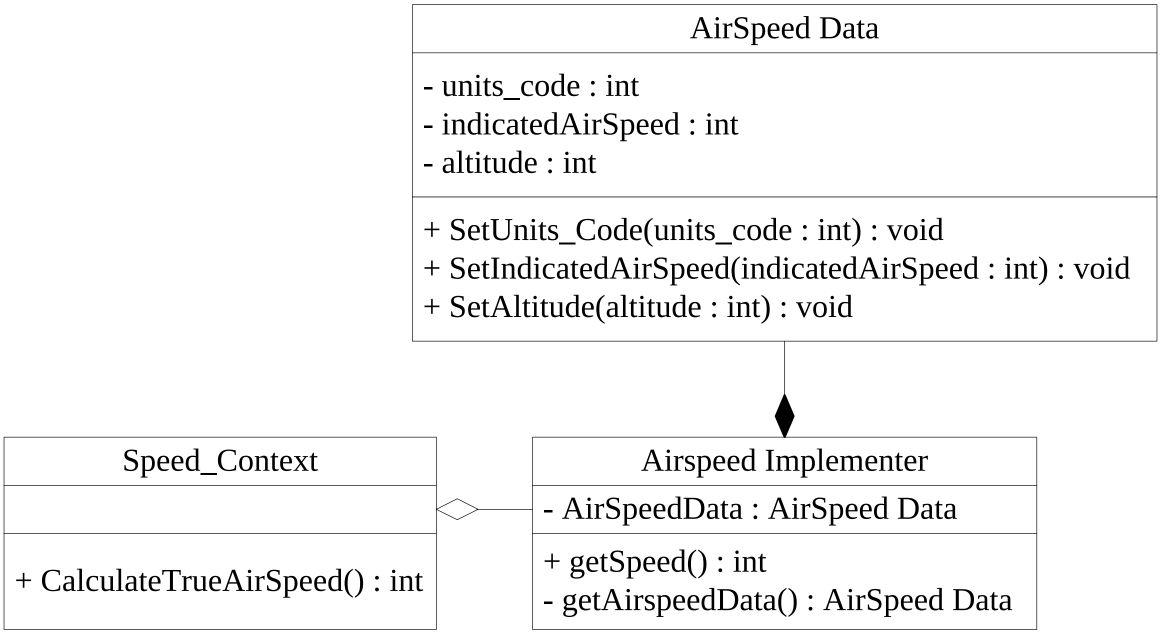

AirSpeed Data
- units_code : int
- indicatedAirSpeed : int
- altitude : int
+ SetUnits_Code(units_code : int) : void
+ SetIndicatedAirSpeed(indicatedAirSpeed : int) : void
+ SetAltitude(altitude : int) : void
Speed_Context
+ CalculateTrueAirSpeed() : int
Airspeed Implementer
- AirSpeedData : AirSpeed Data
+ getSpeed() : int
- getAirspeedData() : AirSpeed Data

Figure 10. Bridge Pattern Class for Calculating Airspeed

The difference in dependencies between the class designs given in figure 8 versus that given in figure 10 is shown in figures 11 and 12. Figure 11 shows the dependencies for the getSpeed method of figure 9 when that method is within the class given in figure 8, while figure 12 shows the dependencies for the getSpeed method of figure 9 when that method is within the class given in figure 10.

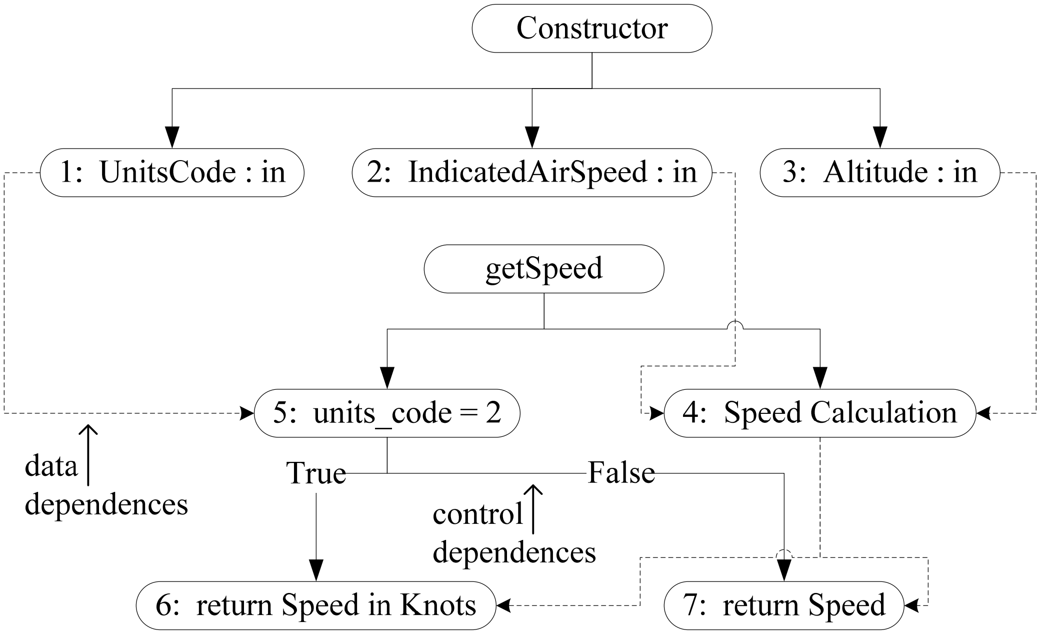


Figure 11. Dependency Graph for Open Class

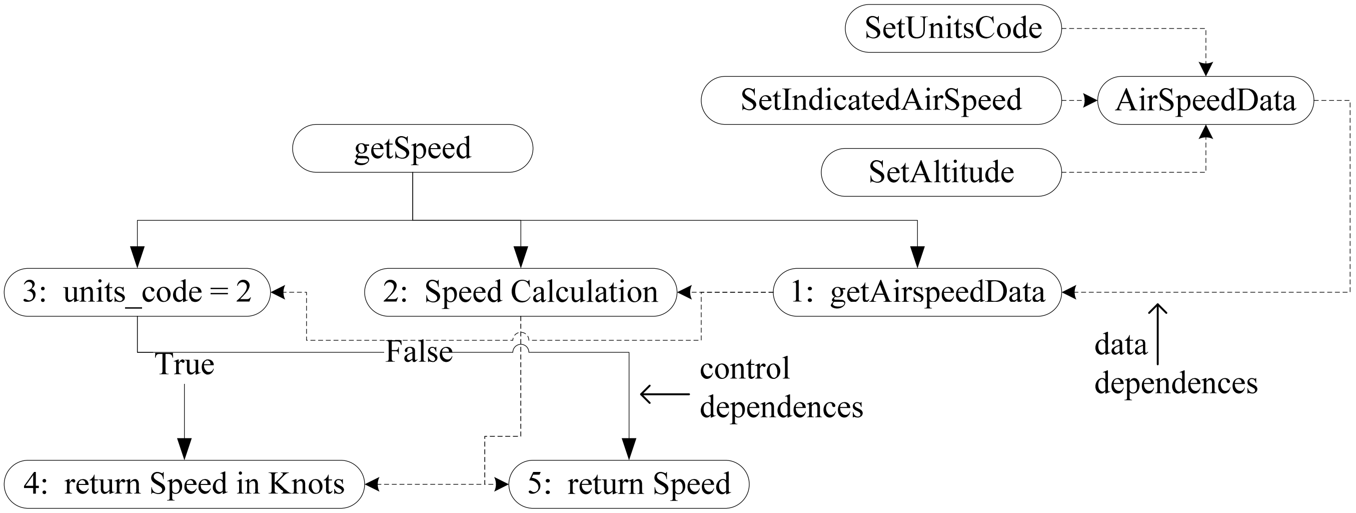


Figure 12. Dependency Graph for Bridge Pattern Class

The bridge pattern encapsulates business data and has the following purported advantages:

- State data is only set through methods that have the name of the state variable and the word “Set”—greatly lowering the chance of mixing up the state variables. With the TASCalculator constructor method of figure 8, there is a risk that the state variable values could be incorrectly assigned via transposition, since all three parameters have the same type (int).

- State data can be updated during the life of the object. As can be seen in figure 11, objects of the TASCalculator class in figure 8 are constant, while figure 12 shows that bridge pattern objects of the AirSpeedData class in figure 10 are not.

- Data can be passed throughout an application by sending a single object encapsulating multiple data items using the Visitor Design Pattern [9]. Encapsulation is addressed in section 3.2.

- Data flow criteria have to be applied to only one method that sets the state, and this is external to the object—the getAirspeedData method in the implementation class.

3.1.2 Destructors and Finalizers.

Destructors and Finalizers are methods responsible for freeing memory, closing some resource, cleaning up data structures, or performing whatever other activities are needed when an object has either completed its work or is no longer needed. They are supposed to be considered final, in that no other actions concerning the object will be possible after they run. Unfortunately, these actions can appear as side-effects, especially when other objects are impacted. As mentioned in reference 5, these methods can be employed by other features of the language (e.g., exceptions, implicit type conversions). However, these methods are not always invoked because some types of exceptions and other modifications in the control flow circumvent their calling.

Because of this, there may be one set of dependencies when the methods are invoked, and a different set when their invocation is circumvented. Analysis of both sets of dependencies is needed.

3.2 ENCAPSULATION AND INFORMATION HIDING.

Encapsulation and information hiding is the separation of the external (public) and internal (private) aspects of a class and its objects. A previous study [5] identified encapsulation and information hiding as having an effect on DCCC when intermediary objects are involved in the interactions between objects. For example, consider the three objects shown in the collaboration diagram presented in figure 13. In response to an event, Object_1 sends a message to Object_2 through Method_1. As a result of this message, Object_2 updates its attributes with values provided by Object_1. Later, in response to another event, Object_2 sends a message to Object_3 through Method_2. As a result of this message, Object_3 updates its attributes with values provided by Object_2. In this example, there is a dependency between Object_3 and Object_1, but that dependency may only be discovered through the code.

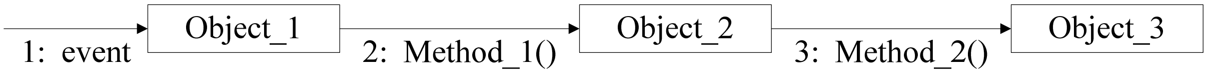


Figure 13. Collaboration Diagram

One solution to this problem is to chain dependencies together [12] to remove intermediaries. This chaining is simply the transitive closure over the dependency relations. For example, consider the dependency diagram in figure 14. In the upper portion of figure 14, object O_2 is shown updating its attribute B with object O_1's attribute A. Object O_3 is also shown updating its attribute C with object O_2's attribute B. The lower part of figure 14 shows the transitive closure where the intermediary object O_2 has been removed.

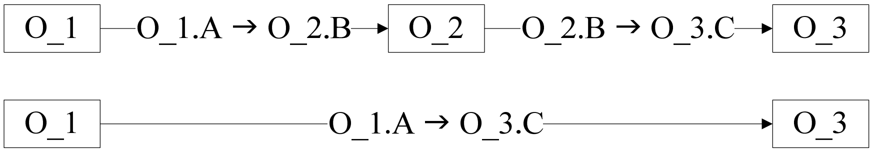


Figure 14. Transitive Closure

Note that this closure is only possible because the same attribute B of object O_2 appears on both sides of the transformation. If different attributes appeared on one side versus the other, the removal of object O_2 would not be possible. Figure 12 provides another example where transitive closure over the encapsulating AirSpeedData object will be needed to determine complete dependencies.

An example of coupling between classes on an aircraft's instrument panel is shown in figure 15. In figure 15, the individual display modules are tightly coupled to the sensors that provide basic information. Each display module is also tightly coupled to the next so that common information can be shared. Failure of one component, or an incorrect value, can cause others to fail or show incorrect data.

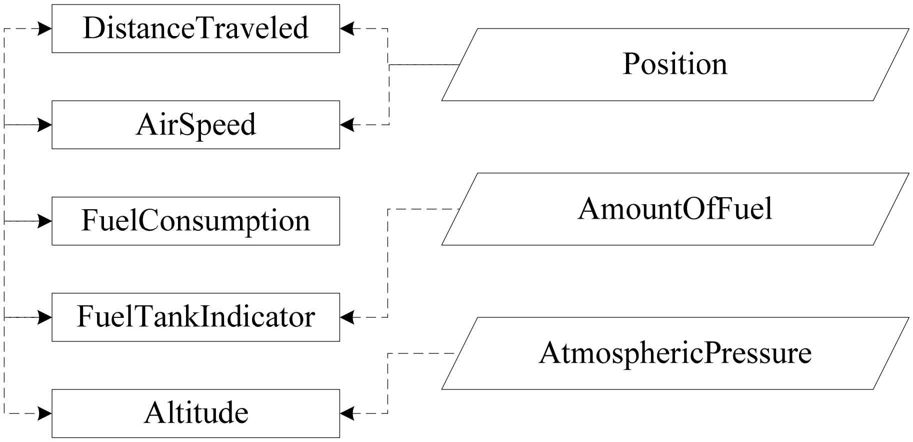


Figure 15. Instrument Panel Object Coupling

The Mediator design pattern provides a hub or central location for the interconnections between objects, while serving as a bridge to the data [9]. This pattern applied to the instrument panel objects of figure 15 is shown in figure 16. It converts many-to-many relationships to one to many relationships. This eliminates the possibility of errors that result from tightly coupling a number of classes together. The classes are then free to change independently of other classes, except the Mediator that records and monitors the changes.

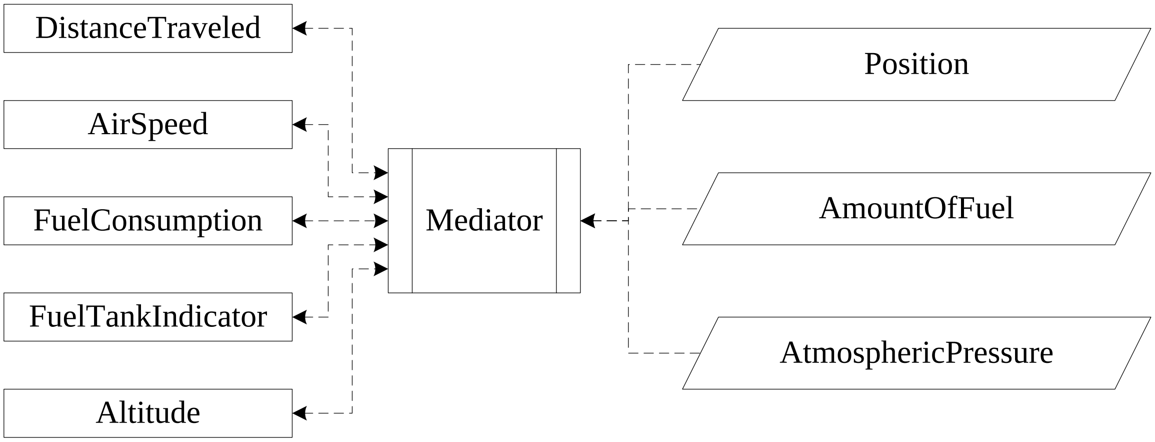


Figure 16. Instrument Panel Mediator Object Coupling

Another pattern, not shown, that decouples classes is the Observer pattern [9]. This pattern requires the classes to register with a subject or topic to receive event notice and any state change information from the topic. This is used mostly in queues and other asynchronous systems, but could also find application here.

Both the Mediator and Observer design patterns inject a new object into the system so that many-to-many dependency paths between multiple objects are reduced down to one-to-one paths with the central (intermediary) object and each of the other objects. This and other forms of indirection are used in the patterns in [9] to achieve flexibility and variability. However, this indirection may in turn complicate the design and its verification [9]. Taking the transitive closure of the dependency relations over the intermediary object is a necessary step to dealing with this complexity.

3.3 EXCEPTIONS.

A previous study [5] identified exceptions as having an effect on DCCC when destructors and finalizers invoked by the exception handler have side effects. As mentioned in section 3.1, identifying and verifying the dependency relations between all helper functions and the other class methods will address this concern.

3.4 IMPLICIT TYPE CONVERSIONS.

A previous study [5] identified implicit type conversions as having an effect on DCCC through the side effects of helper functions (i.e., constructors and destructors). As mentioned in section 3.1, identifying and verifying the dependency relations between all helper functions and the other class methods will address this concern. The previous study [5] mentioned that one complication is the need to perform some analysis at the object code level to detect implicit type conversions. The use of dependency relations has no impact on this need for supplemental analysis (i.e., it does not eliminate the need to perform some analysis at the object code level to detect implicit type conversions).

The previous study [5] also identified implicit type conversions as having an effect on DCCC by impacting the resolution of polymorphic references. Polymorphism is addressed in section 3.8.

3.5 INHERITANCE.

Inheritance is a mechanism whereby a class is defined in terms of other classes (its parents), adding the features of its parents to its own without disturbing either the relationships between its parents and their clients or the parent’s concrete implementations. Figure 17 contains a simple class hierarchy demonstrating inheritance.

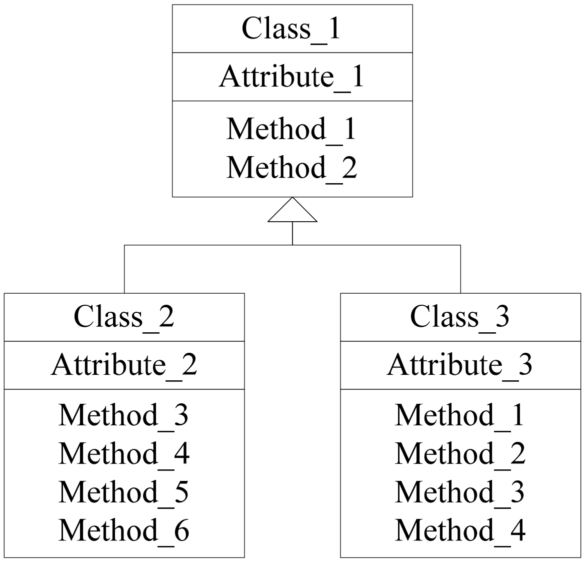


Figure 17. Inheritance Hierarchy

Inheritance is implemented by three basic mechanisms:

- Extension is the inclusion of the attributes and methods of ancestor classes (parents, their parents, etc.) in a subclass. In figure 17, Class_2 and Class_3 extend Class_1. They each inherit a copy of Attribute_1. Class_2 also inherits Method_1 and Method_2.

- Specialization is the definition of attributes and methods that are unique to that class. In figure 17, Class_2 specializes Class_1 by adding a new attribute, Attribute_2, and by adding four new methods, Method_3, Method_4, Method_5, and Method_6. Class_3 specializes Class_1 by also adding a new attribute, Attribute_3, and two new methods: Method_3 and Method_4. Note that the added methods, Method_3 and Method_4, in Class_2 and Class_3 are each distinct methods even though they have the same names. This occurs because there is no inheritance between these two classes.

- Overriding is the definition of either an attribute or method in a class with the same signature as in a parent class. A signature consists of the name of the feature, the type for attributes, parameter signatures for methods, and in some languages a return type for methods (e.g., Ada). In figure 16, Class_3 overrides Method_1 and Method_2 in Class_1 by providing its own methods. Overriding requires polymorphism with dynamic binding, which is discussed in section 3.8.

A previous study [5] identified inheritance as having an effect on DCCC through specialization and overriding changing the dependency relations between superclasses and subclasses. Table 3 presents an example set of definitions and uses of the class attributes by the class methods for the inheritance hierarchy in figure 17. In table 3, the first column identifies the class. The second column lists the methods. Note that Method_1 and Method_2 are inherited in Class_2, and are shown enclosed in angle brackets in table 3. The third column identifies the attributes defined by the method, while the fourth column identifies the attributes used by the method. Note that

Attribute_1 is inherited in Class_2 and Class_3 and is shown enclosed in angle brackets in table 3. Recall that the data in columns three and four are example data. This data was generated to support the analysis of the effects that inheritance mechanisms have on dependency relations.

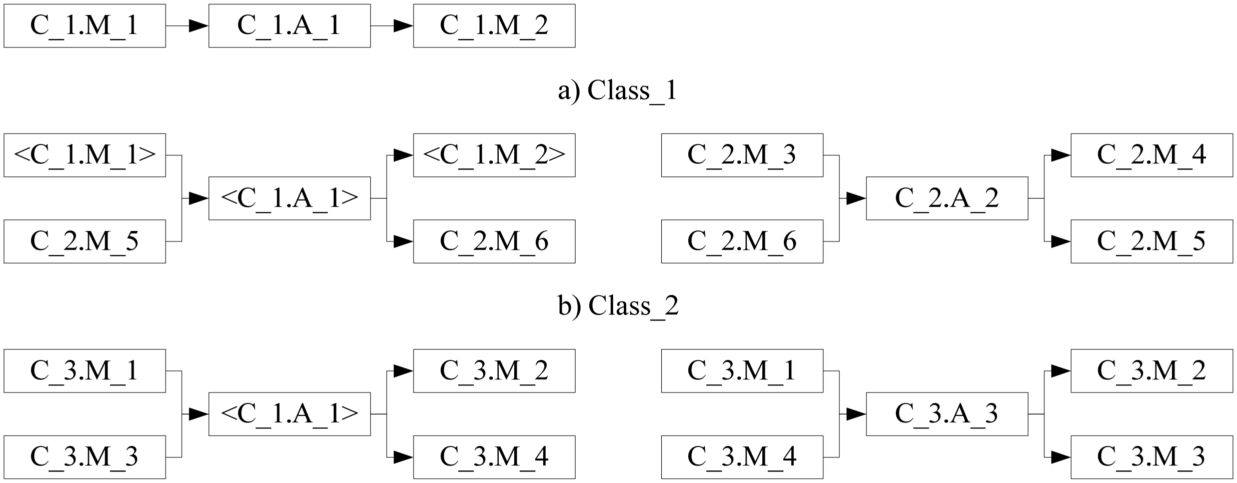
Table 3. Class Methods Definitions and Uses

Class	Method	Defines	Uses
Class_1	Class_1.Method_1	Class_1.Attribute_1	
	Class_1.Method_2		Class_1.Attribute_1
Class_2	<Class_1.Method_1>	<Class_1.Attribute_1>	
	<Class_1.Method_2>		<Class_1.Attribute_1>
	Class_2.Method_3	Class_2.Attribute_2	
	Class_2.Method_4		Class_2.Attribute_2
	Class_2.Method_5	<Class_1.Attribute_1>	Class_2.Attribute_2
	Class_2.Method_6	Class_2.Attribute_2	<Class_1.Attribute_1>
Class_3	Class_3.Method_1	<Class_1.Attribute_1>, Class_3.Attribute_3	
	Class_3.Method_2		<Class_1.Attribute_1>, Class_3.Attribute_3
	Class_3.Method_3	<Class_1.Attribute_1>	Class_3.Attribute_3
	Class_3.Method_4	Class_3.Attribute_3	<Class_1.Attribute_1>

Specialization adds new dependency relations for the added features and extends existing dependency relations if the added features interact with the inherited ones. Table 3 shows new dependency relations for Attribute_2 using Class_2.Method_3, Class_2.Method_4, Class_2.Method_5, and Class_2.Method_6. Table 3 also shows extended dependency relations for Class_1.Attribute_1 using Class_2.Method_5 and Class_2.Method_6. These changes are graphically depicted in the DDGs in figure 18. In figure 18, attribute is abbreviated as A, class is abbreviated as C, and method is abbreviated as M. The inherited methods and attributes are shown enclosed in angle brackets as in table 3. Figure 18(a) presents the DDG for Class_1's attribute: Attribute_1. Figure 18(b) presents the DDGs for Class_2's attributes: Attribute_1 and Attribute_2. Note that Attribute_1 is inherited. Figure 18(c) presents the DDGs for Class_3's attributes: Attribute_1 and Attribute_2. As with Class_2, Attribute_1 is inherited in Class_3. The extended dependencies for Attribute_1 are demonstrated by the differences between the left DDGs of figures 18(a) and 18(b). The new dependencies for Attribute_2 are demonstrated by the right DDG of figure 18(b).

Overriding can remove existing dependencies, replace existing dependencies, and add new ones. Table 3 shows removed dependencies for Attribute_1 in Class_3 in that the inherited Class_1.Method_1 and Class_1.Method_2 no longer access the attribute. Instead, Attribute_1 in Class_3 is now being accessed by all of Class_3's methods. Class_3.Method_1 and Class_3.Method_2 replace the inherited dependencies, and Class_3.Method_3 and Class_3.Method_4 add new dependencies. The removed and replaced dependencies for

Attribute_1 are demonstrated by the left DDGs of figures 18(a) and 18(c). The new dependencies for Attribute_3 are demonstrated by the right DDG of figure 18(c).



a) Class_1

b) Class_2

c) Class_3

Figure 18. Inheritance Data Dependency Changes

Each of the removed dependencies invalidates the verification associated with the dependency in the superclass (parents and ancestors), while each of the new dependencies will need new verification. Because of this, the testing that was performed in a superclass may no longer be valid for a subclass [47]. For structural coverage measures to perform their proper role in the lifecycle, inheritance requires different coverage measures [48]. These measures require the standard coverage data to be sensitive to the class context under which they are collected [5, 40, and 48].

One way to achieve this context sensitivity is to collect the coverage data in the flattened class [5, 40, and 48]. This has become the recommended practice in the standard object-oriented testing literature [39, 40, and 41]. Although this is always a conservative approach, there are circumstances where it is not necessary:

- If a subclass only extends the superclass (i.e., does not override any of its inherited features and does not add any of its own), then no retesting is necessary [47 and 49]. In this case, the dependency relations from the superclass remain unchanged in the subclass.

- If a subclass extends and specializes the superclass by only adding attributes and methods that do not interact with any inherited features, then only the added features require testing [49]. In this case, the dependency relations from the superclass also remain unchanged in the subclass. In addition, new dependency relations are added in the subclass.

- There are circumstances when complete testing of a subclass is not necessary even when inherited features are overridden and when new interacting features are added, such as those noted in reference 49.

The hierarchical integration testing (HIT) method can be used to determine when and how much retesting is required in a subclass [49]. Inherited tests that can be rerun as-is, tests that require modification, and newly-identified tests are also identified by the analysis [49]. The analysis treats inheritance as an integration problem and relies, in part, on changes in dependencies to make the identifications.

The effect on DCCC through specialization and overriding within subclasses is demonstrated by the class diagram in figure 19. At the top of the hierarchy is an EngineSensor class that wraps the actual transducers for the engine information. Extending the EngineSensor superclass produces a Display class. The Display class is further specialized to form individual classes for displaying the individual readings.

The more child classes, the more impact on DCCC can result at the lower (child) levels, where the implementation occurs. All classes in an inheritance hierarchy are tightly coupled. Changes to classes at the top of the hierarchy (parents) impact all those below (children). For example, in the hierarchy of figure 19, if the readValue method changes in the Display class, it will impact only the RPM class, as the Oil class has its own readValue method. DCCC errors can occur anywhere within the class hierarchy, especially if changes are made to those methods at the top. The more classes beneath the top level classes, the greater the chance for DCCC and the greater the difficulty in predicting behavior.

For example, if the readValue method on the Display class is changed, it could be expected that all the subclasses of Display will reflect the changes. However, the Oil and Temperature, and Pressure classes have their own ways to read data, with the Pressure class actually overriding the readValue method within the Oil class. Therefore, someone expecting to change all of the display values by changing a method in the Display class will have to look through all classes of the hierarchy to actually change the way all the classes read the sensor values.

Within standard OOT practice, the Builder design pattern separates the construction of a complex object from its representation, producing a complex object ready for use [9]. In the scenario where complex information displays are needed, figure 20 shows how the Builder pattern can produce the same objects as are in figure 19, but with less dependencies and chance of DCCC errors.

Instead of a number of dependencies spread throughout a class hierarchy, the Builder pattern yields ready-made objects. All control flow operations to create the objects are encapsulated in the DisplayFactory class, while the data flow is contained within the Display class. DCCC errors are reduced, since both data and control flow is limited to a single class.

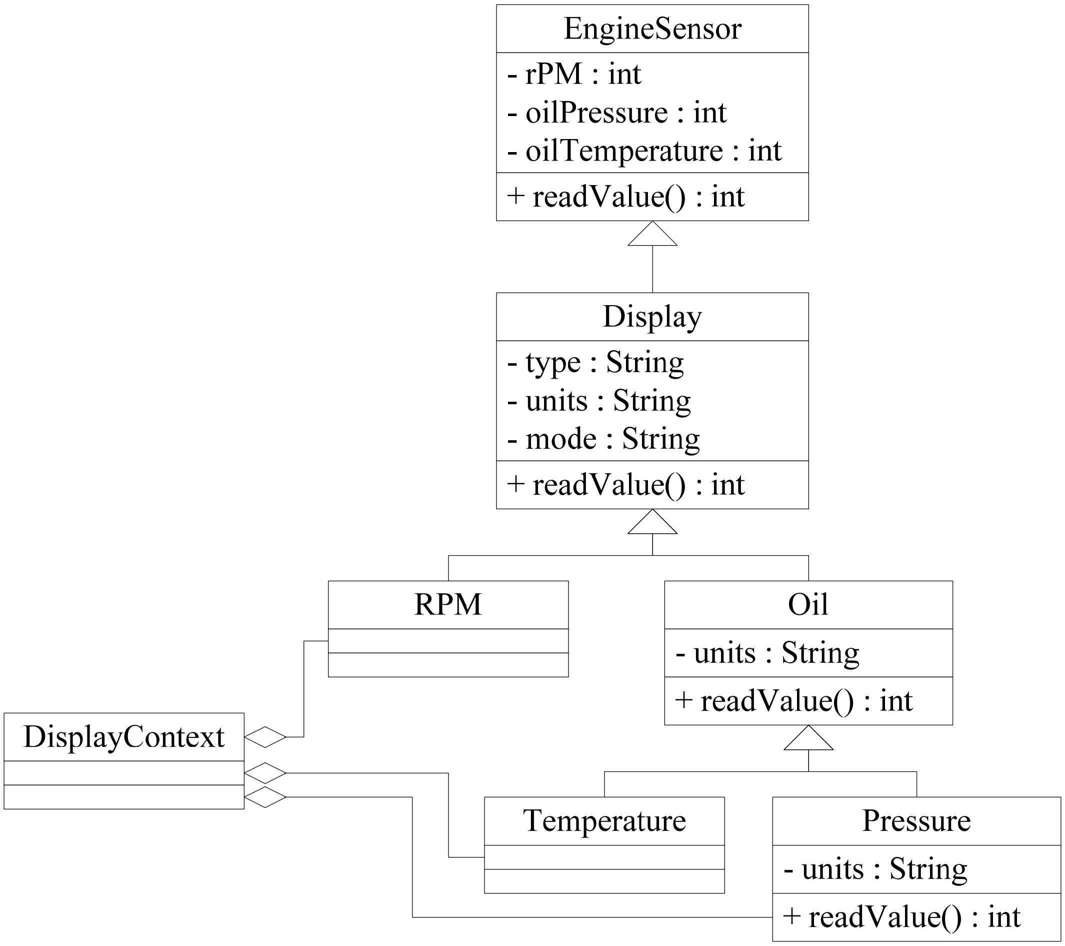


Figure 19. Inheritance Hierarchy for Engine Data Display

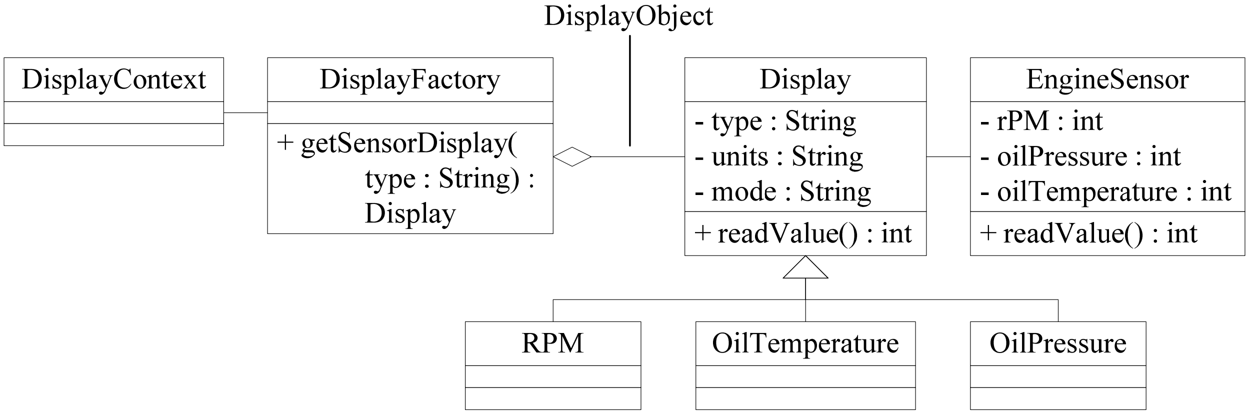


Figure 20. Builder Pattern for Engine Data Display

During the current study, it was discovered that inheritance, aggregation, and association add additional dependencies that need to be considered by the integration process for object-oriented software. These dependencies address the integration order of classes, the need for stubs and drivers, and how to generate an integration order that minimizes the number and complexity of stubs. Many publications have resulted to provide solutions [50 through 59] for these dependencies. While this is apparently a significant issue for object-oriented integration testing, this is, strictly speaking, not a DCCC issue, and appears to be out-of-scope of DO-178B [2].

3.6 AGGREGATION.

Aggregation occurs when one class is defined as a combination of other classes. It is generally implemented with objects of one class incorporating objects of other classes as attributes. Classes are coupled when a message is passed between objects of different (i.e., noninheritance related) classes. This means that methods declared in one class use methods or attributes of another class. Aggregation is one mechanism that enables this coupling. Dependency relations between the coupled classes require verification.

As noted in section 3.5 for inheritance, it was discovered that inheritance, aggregation, and association add additional dependencies that need to be considered by the integration process for object-oriented software. These dependencies address a number of items of concern for object-oriented integration testing, but these dependencies are not a DCCC issue and appear to be out-of-scope of DO-178B.

3.7 ASSOCIATION.

Association occurs when two classes exchange messages. This can occur as a result of:

- A call association—one of class A’s methods calls one of class B’s methods.
- An access association—one of class A’s methods accesses one of class B’s attributes.
- A parameter association—one of class A’s methods contains a parameter that is an object of class B.

Classes are coupled when a message is passed between objects of different (i.e., noninheritance related) classes. This means that methods declared in one class use methods or attributes of another class. Association is one mechanism which enables this coupling. Dependency relations between the coupled classes require verification.

Again, as noted in section 3.5 for inheritance, and restated in section 3.6 for aggregation, it was discovered that inheritance, aggregation, and association add additional dependencies that need to be considered by the integration process for object-oriented software; however, these dependencies are not DCCC issues.

3.8 POLYMORPHISM WITH DYNAMIC BINDING.

Polymorphism is the ability of a name in software text to denote, at run time, one or more possible entities. Polymorphism is required by inheritance as an object declared to be of a superclass can, at run time, be a member of any of the subclasses of that superclass. Which method to call and which attribute to access may depend on the class of which the object is a member at the time of dispatch. Polymorphism is generally supported by dynamic binding and dispatch. This has been compared to a goto statement with multiple variable destinations [60] or a case/switch statement [61]. A previous study [5] identified that most of the need for context coverage covered in section 3.5 is due to the altered dependencies caused by the use of polymorphism with dynamic binding and dispatch.

3.8.1 Static Binding and Dispatch.

Static binding and dispatch is the matching of attribute references to attributes and calls to methods at compile time or link time. With static binding and dispatch, each reference resolves to a single receiver entity (object, attribute, method). This creates a single set of dependency relations between the dispatch site entity and the receiver entity at each dispatch site. Static binding and dispatch present no new issues for traditional integration testing and can be tested with a single test set for the dependency relations. Note that both static and dynamic binding and dispatch can be present in object-oriented programming.

Figure 21 depicts a simple class hierarchy for demonstrating the differences between static and dynamic binding. Table 4 presents the definitions and uses of the class attributes by the class methods for the inheritance hierarchy in figure 21. Table 4 follows the same format as table 3.

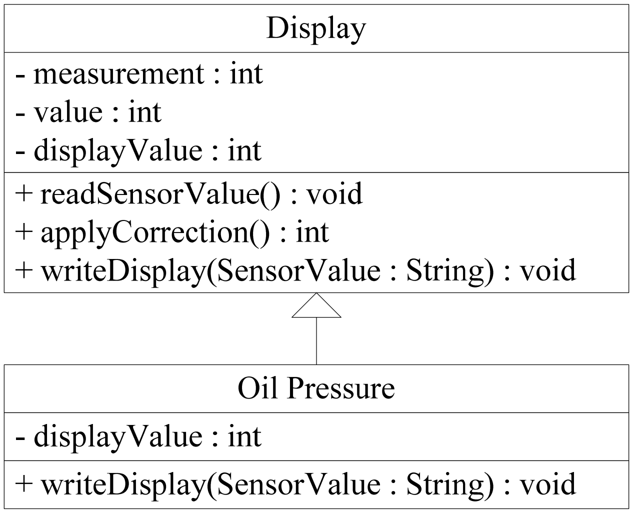

Display
- measurement : int
- value : int
- displayValue : int
+ readSensorValue() : void
+ applyCorrection() : int
+ writeDisplay(SensorValue : String) : void
Oil Pressure
- displayValue : int
+ writeDisplay(SensorValue : String) : void

Figure 21. Dispatch Class Diagram

Table 4. Dispatch Class Methods Definitions and Uses

Class	Method	Defines	Uses
Display	Display.readSensorValue Display.applyCorrection	Display.measurement Display.value	Display.measurement
	Display.writeDisplay	Display.displayValue	Display.value
OilPressure	<Display.readSensorValue> <Display.applyCorrection> OilPressure.writeDisplay	<Display.measurement> <Display.value> OilPressure.displayValue	<Display.measuremen>t <Display.value>

Figure 22 presents the source code for a method that uses the class hierarchy in figure 21.

```
Public void makeDisplay(Display display) {
    int sensorValue = 0
    display.readSensorValue();
    sensorValue = display.applyCorrection();
    display.writeDisplay(sensorValue);
}
```

Figure 22. Dispatch Source Code

Figure 23 depicts what happens in the code of figure 22 during a static binding and dispatch for an object of the class hierarchy in figure 21. In the center of figure 23, a CFG segment is depicted. For this example, the interest is in when the calls display.readSensorValue and display.applyCorrection are made. In figure 23, to the left of the CFG, the methods that are dispatched to when the actual type of display is Display are identified with dashed flow lines connecting them to the CFG (e.g., display.readSensorValue dispatches to Display.readSensorValue). Within the methods, the attribute accesses from table 4 are identified (e.g., Display.readSensorValue defines Display.measurement). To the right of the CFG in figure 23, the same analysis is depicted for when the actual type of display is OilPressure. Note that the methods invoked and the attribute references remain unchanged. The methods are the same because OilPressure inherits Display.readSensorValue and Display.applyCorrection from Display. The inheritance is depicted by the method name being enclosed in angle brackets. Figure 23 shows that there is one dependency relation set and one du-pair for measurement in this subset of code.

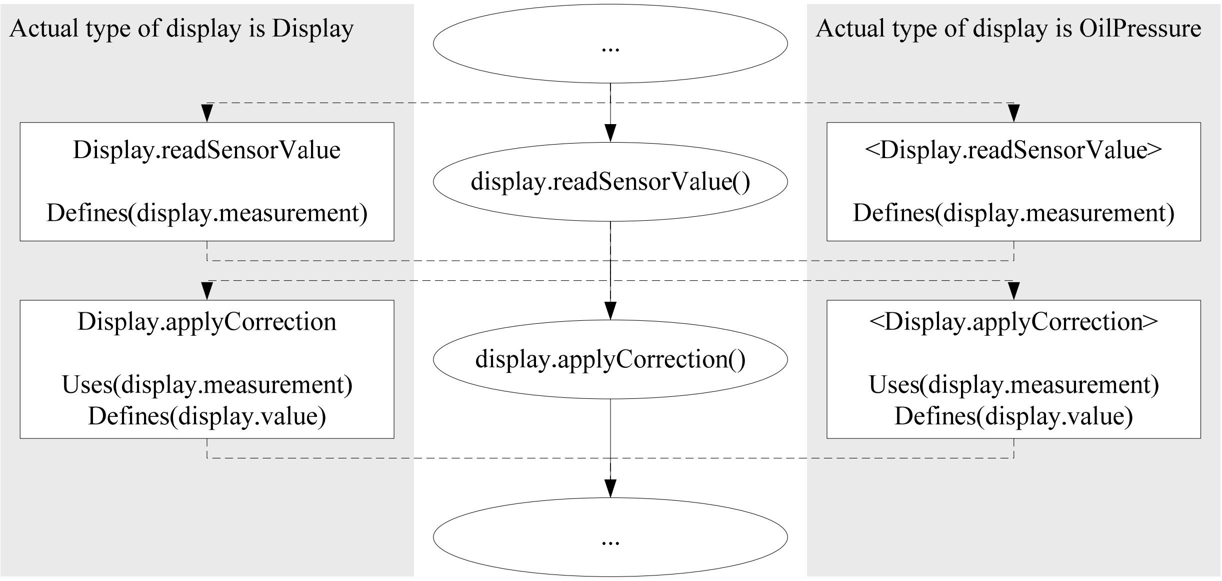

Actual type of display is Display
...
Actual type of display is OilPressure
Display.readSensorValue
Defines(display.measurement)
display.readSensorValue()
<Display.readSensorValue>
Defines(display.measurement)
Display.applyCorrection
Uses(display.measurement)
Defines(display.value)
display.applyCorrection()
<Display.applyCorrection>
Uses(display.measurement)
Defines(display.value)
...

Figure 23. Static Binding and Dispatch

3.8.2 Dynamic Binding and Dispatch.

Dynamic binding and dispatch is the matching of attribute references to attributes and calls to methods at run time as opposed to compile time or link time. With dynamic binding and dispatch, each (polymorphic) reference resolves to a set of possible receiver entities instead of a single unchanging (invariant) entity. This creates a set of possible dependency relation sets, one for each possible receiver entity. Covering each of these different contexts is needed, as mentioned in section 3.5.

Figure 24 depicts what happens in the code of figure 22 during a dynamic binding and dispatch for an object of the class hierarchy in figure 21. Figure 24 follows the same format as figure 23. In figure 24, the method display.writeDisplay is dynamically dispatched as OilPressure overrode the inherited method from Display. In figure 24, different methods are invoked and different attributes are referenced for the dynamic binding and dispatch case compared to the analysis for static binding and dispatch presented in figure 23. Figure 24 shows that there are two relation sets. One relation set is for the du-pair for display.value. The definitions for the two different intercomponent du-pairs occur in the same method: Display.applyCorrection. However, the uses appear in different methods: Display.writeDisplay, when display is of type Display, and OilPressure.writeDisplay, when display is of type OilPressure. The second relation set concerns the definition of display.displayValue within writeDisplay. Though this attribute has the same name and type in both classes, it is unique to each as OilPressure overrode the attribute it inherited from Display. Should displayValue be used in another method, and that use uses the definition from writeDisplay, two different intercomponent du-pairs will be possible (or none if there is a DCCC error).

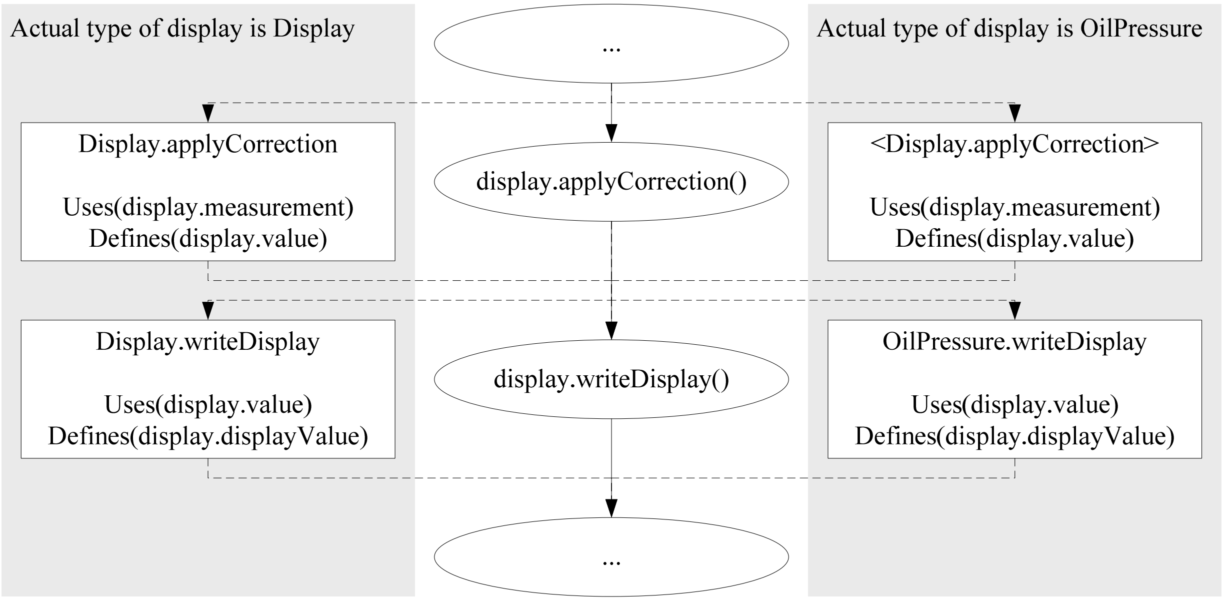

Actual type of display is Display
...
Actual type of display is OilPressure
Display.applyCorrection
Uses(display.measurement)
Defines(display.value)
display.applyCorrection()
<Display.applyCorrection>
Uses(display.measurement)
Defines(display.value)
Display.writeDisplay
Uses(display.value)
Defines(display.displayValue)
display.writeDisplay()
OilPressure.writeDisplay
Uses(display.value)
Defines(display.displayValue)
...

Figure 24. Dynamic Binding and Dispatch

The efficiency of dynamic binding and the reduction of code makes its use widespread in traditional OOT applications. Two surveys of the OOTiA community show that dynamic binding is being avoided by some and employed by others [1 and 4]. Best practice should be to document polymorphic methods well and to mark them as possible places to check for errors. The Command Design Pattern could be used to encapsulate requests within an object, while lessening the chance for errors that can arise from dynamic binding [9]. The Command Design Pattern applied to the Display class of figure 21 is presented in figure 25.

The Command Pattern does add some extra classes, as shown in figure 25. The Command classes encapsulate the action, forcing the proper object to be used. The DisplayCommand class, for example, requires a Display object be passed into its constructor. Within its execute method—shown in figure 26—the overloaded method is called. Strong typing within the constructor of the Command object ensures that the proper object gets loaded and used.

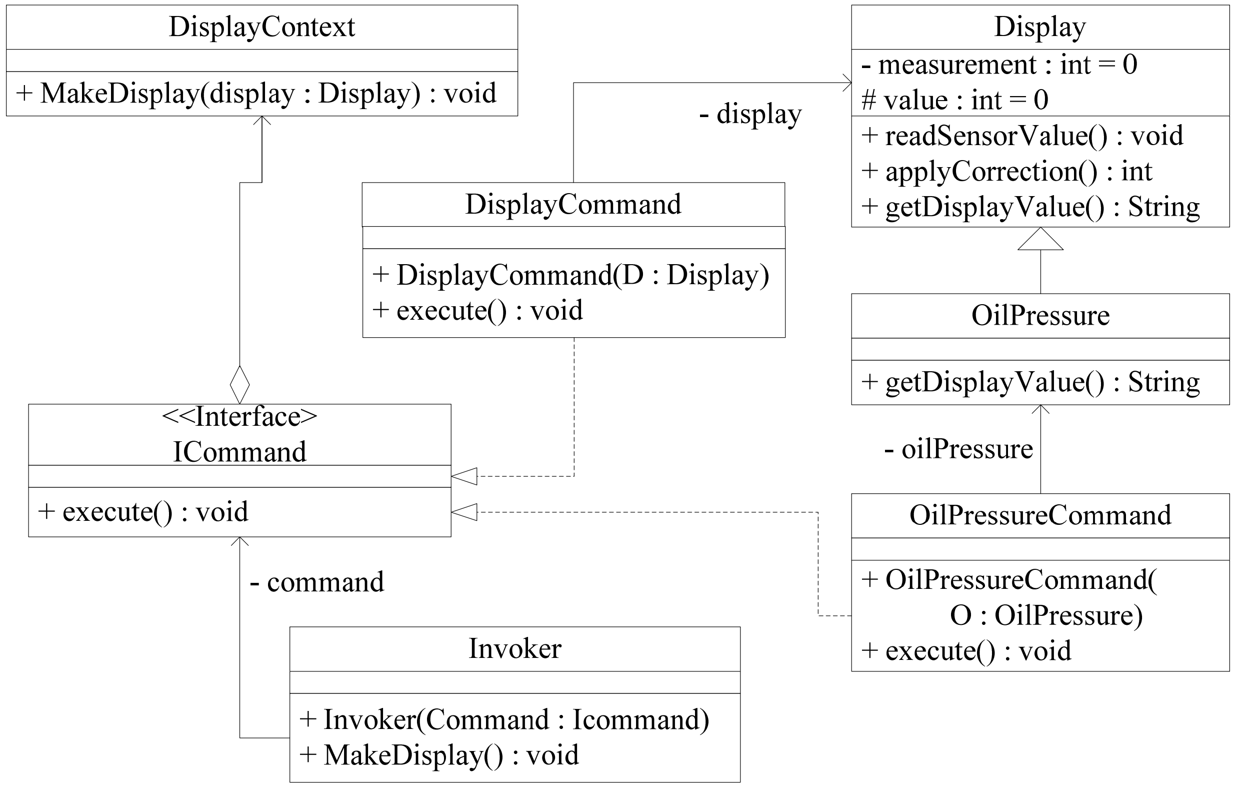


Figure 25. Command Design Pattern for Display

Figure 26 shows the invoking of the method shown in figure 22. While the data and control flow are still tightly coupled, the strong typing in the constructors reduces the chance of error due to the wrong type of object. Object errors will be caught at compile time and not run time, as the dynamically bound object is encapsulated by the Command object. Another benefit is that the execution of the method that uses dynamic binding is done within a command object, making it easier to maintain and extend.

```
public class DisplayCommand implements ICommand {
    private Display display;

    public DisplayCommand(Display D) {  ←————————  strongly typed constructor
        this.display = D;
    }
    /**
      * Required method
      */
    public void execute() {
        DisplayContext context = new DisplayContext();
        context.MakeDisplay(display);
    }
}


public class DisplayCommand implements ICommand {
    private OilPressure oilPressure;

    public OilPressureCommand(OilPressure O) {  ←————  strongly typed constructor
        this.oilPressure = O;
    }

    /**
      * Required method
      */
    public void execute() {
        DisplayContext context = new DisplayContext();
        context.MakeDisplay(oilPressure);
    }
}
```

Figure 26. Command Pattern Source Code

3.8.3 Polymorphism Verification.

Adequate testing of polymorphism with dynamic binding and dispatch is an issue with OOT [40, 62, 63-65]. Defining adequate testing of polymorphism with dynamic binding and dispatch is an active research area with no definitive answer yet. Many approaches have been suggested by both academia and industry [5, 6, 25, 32-42, 50, 61, 66-69]. Most of these approaches treat polymorphism as an integration issue.

Most proposals for the adequate testing of polymorphism with dynamic binding and dispatch require testing that covers some subset of all possible polymorphic bindings for a polymorphic reference (note that a subset can be a complete set). Two levels of coverage have been proposed. Figure 27 is a simple class hierarchy that will be used to illustrate the difference between the two coverage criteria. For the following examples, an object named sensor is used, which is declared to be of type Sensor from the hierarchy in figure 27, and a dispatch site where the polymorphic reference is sensor.applyCorrection(measurement).

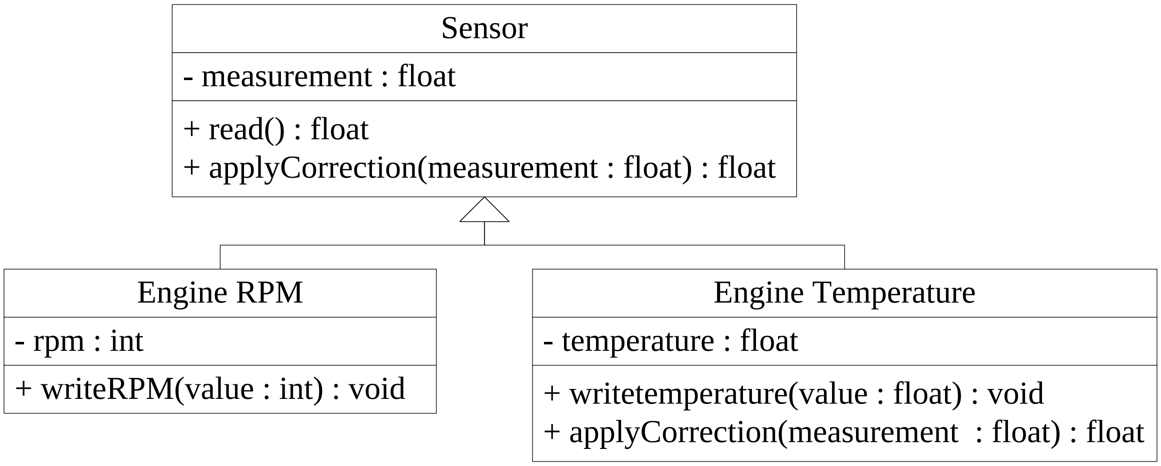

Sensor
- measurement : float
+ read() : float
+ applyCorrection(measurement : float) : float
Engine RPM
- rpm : int
+ writeRPM(value : int) : void
Engine Temperature
- temperature : float
+ writetemperature(value : float) : void
+ applyCorrection(measurement : float) : float

Figure 27. Class Diagram

The first criterion is known as the receiver-classes criterion (RCC) in reference 70 and inheritance coverage in reference 69. This coverage criterion requires exercising all possible classes at a dispatch site: the base class plus all of its subclasses. This criterion would require three tests for the example polymorphic reference. For the first test, the actual type of sensor is Sensor. For the second test, the actual type of sensor is EngineRPM. Note that the first and second tests will dispatch to Sensor.applyCorrection, since EngineRPM inherits that method from Sensor. For the third test, the actual type of sensor is EngineTemperature. This test will dispatch to EngineTemperature.applyCorrection as EngineTemperature overrides the inherited applyCorrection from Sensor.

The second criterion is known as the target-methods criterion (TMC) in reference 70 and overriding method coverage in reference 69. This coverage criterion requires exercising all possible concrete methods at a dispatch site: the base class method plus all overriding methods in the subclasses. This criterion would require two tests for the example polymorphic reference. For the first test, the actual type of sensor is either Sensor or EngineRPM. Either actual type is acceptable since both will dispatch to Sensor.applyCorrection. For the second test, the actual type of sensor is EngineTemperature. This test will dispatch to the overriding EngineTemperature.applyCorrection.

A summary of the approaches taken for the adequate testing of polymorphism with dynamic binding is presented in table 5. The approaches are arranged in order of first publication. In table 5, the first column gives an identifying number for the approach. This number is used to identify the brief explanations of each approach following the table. The second column identifies the authors, while the third column lists their publications referenced by this report. The fourth column identifies what form of RCC, if any, is used by the proposed approach. The fifth column identifies what form of TMC, if any, is used by the proposed approach. The sixth column identifies if dependencies are used. Each of the approaches is briefly explained after the table. The following entries are used in table 5 for the entries concerning RCC and TMC.

- E—exhaustive. Every possible binding at every polymorphic reference.
- S—subset. A subset of every possible binding at every polymorphic reference.
- D—distributed. Every possible binding over all the equivalent polymorphic references.

If dependencies are used, an X appears in the last column. Note that the majority of approaches only use data dependencies.

Table 5. Polymorphism Coverage Approaches

ID No.	Authors	Publications	RCC	TMC	Dependencies
1	Kirani	66			X
2	McDaniel and McGregor	67	S		
3	Overbeck	68	S		
4	Paradkar	50	S		
5	Siegel	39	S		X
6	Watson and McCabe	61	D + E[1]		
7	Alexander and Offutt	32, 35, 36, 38		E	X
8	Bashir and Goel	25		S	X
9	Chen and Kao	42	E		X
10	Orso, Pezzè, and Martena	33, 34, 37	S		X
11	Binder	40	D + E[1]		X
12	McGregor and Sykes	41	S		
13	Chilenski, Timberlake, and Masalskis	5	D		
14	OOTiA	6	D		
15	Supavita and Suwannasart	69	E	E	

[1] The one exhaustive case may be in a test driver.

1. Kirani [66] proposes that testing covers message-method interactions and message sequences based on the specifications for the classes and methods. Guidance is not provided for selecting bindings for the polymorphic calls in the exercised combinations. The interactions and sequences are augmented with data-flow information to perform a certain form of dependency testing.

2. McDaniel and McGregor [67] propose that testing covers a subset of RCC. Additionally, this approach includes covering the states of the objects involved (i.e., message/state combinations). This approach uses a bottom-up pairwise integration test strategy using orthogonal arrays (latin squares) [71] to ensure coverage of all pairwise combinations of polymorphic resolutions with states. This approach does not address DCCC.

3. Overbeck [68] proposes that testing covers a subset of RCC. This approach uses a bottom-up pairwise integration test strategy using interface constraint specification

patterns extended from reference 49 to cover interactions between classes. This approach does not address DCCC.

4. Paradkar [50] proposes that testing covers a subset of RCC. Additionally, this approach includes covering the states of the objects involved (i.e., message/state combinations). This approach uses a bottom-up pairwise integration test strategy using a set of heuristics based on reference 47 to choose from feasible message/state combinations only. This approach proposes using orthogonal arrays (latin squares) [71] to further reduce the test set. This approach does not address DCCC, since data and control dependencies between methods are specifically identified as details that are not required for integration testing, though inheritance and polymorphism are identified as affecting data and control dependencies.

5. Siegel [39] proposes that testing covers a subset of RCC. This approach uses a bottom-up pairwise integration test strategy using HIT [49] and dependencies to ensure the test set is sufficient to cover differences in use case behavior.

6. Watson and McCabe [61] propose three possible approaches (optimistic, pessimistic, and balanced). Two of the approaches (optimistic and balanced) treat polymorphism with dynamic dispatch as a case statement based on the class of the object. None of the approaches address DCCC.

   - The optimistic approach proposes that testing executes each polymorphic reference at least once and each polymorphic resolution (class binding) at least once across the entire testing effort. This approach could be considered a distributed RCC approach.

   - The pessimistic approach proposes that testing covers RCC.

   - The balanced approach proposes that testing execute each polymorphic reference at least once, each polymorphic resolution at least once (distributed RCC), and for one polymorphic reference that forms an equivalence class with all others dispatching on the same base class, cover RCC. It is suggested that the RCC site could be in a test driver instead of the application code. This is the approach reported in table 5.

7. Alexander and Offutt [32, 35, 36, and 38] propose that testing cover TMC. They extend coupling-based testing [13, 21, and 22] (which extended data-flow testing) to cover the additional dependencies polymorphism allows. This approach uses a bottom-up pairwise integration strategy where data coupling is restricted to parameters only (i.e., no global data), and direct call-pairs only.

8. Bashir and Goel [25] propose that testing covers a representative subset of invocations of interfaces via use cases to cover intermethod dependencies. The approach is very data and state centric, since classes are sliced into individual attributes and the methods that access them. They point out that polymorphism compels one to many invocations of the

same interface to catch differences. The coverage of these dependency differences will achieve a subset of TMC.

9. Chen and Kao [42] propose that testing covers RCC. When multiple objects are involved, every possible combination of bindings must be tested (strong inheritance coverage in reference 69). Dependencies are covered to the all-du-pairs data flow testing criterion (i.e., every definition reaches every use at least once, every use is reached by every definition at least once).

10. Orso, Pezzè, and Martena [33, 34, and 37] propose that testing covers a subset of RCC. They extend traditional data flow testing to consider the additional dependencies polymorphism allows [31], which allows for all data flow coverage criteria to be applied. This approach uses a bottom-up pairwise integration test strategy using compiler optimization techniques to reduce infeasible bindings.

11. Binder [40] proposes that testing covers RCC at one polymorphic reference for each class, all polymorphic references must be executed at least once and all dependencies must be covered at least once. It is suggested that the RCC site would generally be in a test driver instead of the application code. This approach is similar to the balanced approach of [61].

12. McGregor and Sykes [41] propose that testing covers a representative sample of life cycle scenarios and interactions based on the specifications for the classes. This approach uses a bottom-up pairwise integration test strategy using orthogonal arrays (latin squares) [71] and HIT to achieve representative coverage of pre- and postconditions, invariants, states, state transitions, and sequences of operations and messages. This approach achieves a subset of RCC.

13. Chilenski, Timberlake, and Masalskis [5] propose that testing covers every polymorphic reference and every entry in every method table. This approach achieves distributed RCC. DCCC is identified as an issue in need of further study and resolution.

14. The OOTiA Handbook [6] also proposes that testing covers every polymorphic reference and every entry in every method table just as in reference 5, with the observation that DCCC requires something more. This approach achieves distributed RCC.

15. Supavita and Suwannasart [69] define a set of five coverage criteria to test polymorphism. One of the criteria, base class coverage, requires that each dispatch be covered. They provide an analysis showing that this is an unacceptable level of coverage (i.e., insufficient testing). One criterion, overriding method coverage, is TMC, while another, strong overriding method coverage, adds all combinations in multiobject dispatches to TMC. One criterion, inheritance coverage, is RCC, and the final one, strong inheritance coverage, adds all combinations in multiobject dispatches to RCC. This is the same criterion used in reference 42. They show that changes in dependencies

can make TMC inadequate for even purely inherited methods. They also show that these criteria are insufficient adequacy criteria and that further work is required.

The majority of the approaches listed in table 5 use coverage of dependencies as one of the adequacy criteria for the adequate testing of polymorphism with dynamic binding and dispatch. As mentioned previously, the majority of the approaches also treat polymorphism as an integration issue. Only the approaches of Siegel [39]; Alexander and Offutt [32, 35, 36, and 38]; and Orso, Pezzè, and Martena [33, 34, and 37] employ dependencies and integration. These last two approaches focus on the dependencies between the classes formed by polymorphic references and how those dependencies should be covered. They have been referred to as coupling-based testing.

The coverage of intercomponent dependencies (i.e., PDG/SDG edges or intercomponent du-pairs + calls) recommended in section 2.4.3 will fall somewhere between TMC and RCC and between the approaches of Alexander and Offutt and Orso, Pezzè, and Martena. As discussed in reference 69, inherited methods involved in new dependency relations require testing beyond that required by TMC. These new dependencies will be covered by the recommended approach. Dependency relations beyond direct call-pairs and involving globals require testing beyond that required by the approach of Alexander and Offutt. These dependencies will be covered by the recommended approach. Inherited methods that are not involved in any new dependency relations do not require the additional testing required by RCC and the approach of Orso, Pezzè, and Martena. The recommended approach will also not require this additional testing. Finally, the recommended approach achieves a subset of distributed RCC since exhaustive coverage is not required at each polymorphic reference, but over the collection of polymorphic references dispatching on the same base class.

4. RESULTS AND FURTHER WORK.

This report presents the results of an investigation into issues and acceptance criteria for the verification (confirmation) of DCCC within OOT in commercial aviation as required by Objective 8 of Table A-7 in DO-178B. The intent of the structural coverage analyses (confirmation) of DCCC is to provide an objective assessment (measure) of the completeness of the requirements-based tests of the integrated components (i.e., objectively measure integration testing). Currently, DO-178B does not impose an objective measure for the confirmation of DCCC between the code components called out in Section 6.4.4.3c of DO-178B [2].

This report recommends that intercomponent dependencies (i.e., PDG/SDG edges or intercomponent du-pairs + calls) be covered during verification by a combination of reviews, analyses, and tests to satisfy the DO-178B objectives for confirmation of DCCC. Coverage to this level will ensure that all definitions for each object reach all feasible uses of that definition, all uses of each object are reached by all feasible definitions of that object, and all subprograms are called under some requirements-based test (operational scenario). This coverage provides an objective measure for the confirmation of DCCC. As adequate testing of polymorphism with dynamic binding and dispatch is still an active research area, this recommendation may only be considered an interim solution.

Dependency analysis is well-established within the computer science and software engineering disciplines. Current compilers perform this analysis in support of optimization, and commercial tools performing this analysis in support of a number of activities (e.g., maintenance, change-impact analysis, and reverse engineering) are currently available. The use of dependency analysis as an adequacy criterion for testing, particularly integration testing, is also well-motivated in both the non-OOT and OOT testing literature and is known as CBIT.

The study, "Issues Concerning the Structural Coverage of Object-Oriented Software," identified a number of OOT features with DCCC concerns. This study showed how the DCCC aspects of each of these concerns is addressed by coverage of intercomponent dependencies. Some of these concerns resulted in additional dependencies that must be considered by the integration process for object-oriented software. These dependencies concern the determination of a proper integration order for classes and the minimal generation of stubs and drivers. However, these dependencies are, strictly speaking, not a DCCC issue, and appear to be out of scope of DO-178B.

Four open issues requiring further work remain. Three of these issues relate to the guidance provided by Objective 8 in Table A-7 of DO-178B. Within this table, test coverage (confirmation) of DCCC is required for Software Levels A through C. The only difference identified for this objective in the table is that Level C software does not require independency. This is in contrast to the control-flow adequacy criteria of objectives 5 through 7, where there are software-level-dependent differences.

The first issue concerns the level of dependency coverage required by the two major CBIT approaches: Harrold and Soffa, and Jin and Offutt. Both approaches adapted standard data-flow coverage criteria to apply interprocedurally, thus allowing for different levels of coverage. Both approaches were further adapted to apply to OOT. Orso, Pezzè, and Martena adapted Harrold and Soffa while Alexander and Offutt adapted Jin and Offutt. A follow-up study is recommended to determine if different levels of dependency coverage should be applicable to different software levels, just as different levels of control-flow coverage are applicable to different software levels.

The second issue concerns the level of dependency tracing required by the two major CBIT approaches. The Jin and Offutt/Alexander and Offutt approach only requires coverage for dependencies concerning parameters between direct call-pairs. The Harrold and Soffa/Orso, Pezzè, and Martena approach requires coverage for all interprocedural dependencies. The recommended approach in this report conforms to the more thorough analysis required by Harrold and Soffa/Orso, Pezzè, and Martena. A follow-up study is recommended to determine if the alternate approach of Jin and Offutt/Alexander and Offutt should be considered acceptable, and at what level. For example, the Jin and Offutt/Alexander and Offutt approach could be acceptable for software Levels B and C, where B requires independency, while the Harrold and Soffa/Orso, Pezzè, and Martena approach with independency could be required only for software Level A.

The third issue, considered major, concerns the level of coverage required for the adequate testing of polymorphism with dynamic binding and dispatch. Defining adequate testing of polymorphism with dynamic binding and dispatch is an active research area with no definitive answer yet. As such, the recommendation in this report may only be considered an interim solution where polymorphism with dynamic binding and dispatch is concerned. Two major approaches emerged during the course of this study: the TMC and the RCC. The majority of proposals studied in this report achieve a subset of RCC. The recommended approach in this report also achieves a subset of RCC, however, it is closer to TMC in application. A follow-up study is recommended to determine the acceptability of any of these approaches, and at what software level.

The fourth issue concerns the cost and effectiveness of the coverage of intercomponent dependencies. Though studies have shown the cost and effectiveness of other dependency approaches, the specific approach recommended in this report has not undergone such an analysis. A follow-up study is recommended to determine the cost and effectiveness of the coverage of intercomponent dependencies as recommended in this report.

5. REFERENCES.

Note that links were known to be correct when this report was published.

1. Chilenski, J.J., Heck, D., Hunt, R., and Philippon, D., "Object-Oriented Technology (OOT) Verification Phase 1 Report—Survey Results," The Boeing Company, August 2004.

2. "Software Considerations in Airborne Systems and Equipment Certification," RTCA Document No. RTCA/DO-178B, December 1, 1992.

3. "Final Report for Clarification of DO-178B 'Software Considerations in Airborne Systems and Equipment Certification'," RTCA Document No. RTCA/DO-248B, October 12, 2001.

4. Knickerbocker, J., "Object-Oriented Software—Object-Oriented Technology in Aviation (OOTiA) Survey," a presentation to the *2005 FAA Software/CEH Conference*, July 2005.

5. Chilenski, J.J., Timberlake, T.C., and Masalskis, J.M., "Issues Concerning the Structural Coverage of Object-Oriented Software," FAA report DOT/FAA/AR-02/113, November 2002.

6. "Handbook for Object-Oriented Technology in Aviation (OOTiA)," Revision 0, October 26, 2004, available at: http://faa.gov/aircraft/air_cert/design_approvals/air_software/oot/, last visited January 31, 2006.

7. The Object Management Group document, "Unified Modeling Language Specification," available at http://www.omg.org/technology/documents/modeling_spec_catalog.htm, last visited January 31, 2006.

8. Clarke, L.A., Podgurski, A., Richardson, D.J., and Zeil, S.J., "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Transactions on Software Engineering*, Vol. 15, No. 11, November 1989, pp. 1318-1332.

9. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, 1995.

10. Certification Authorities Software Team (CAST), Position Paper CAST-19, "Clarification of Structural Coverage Analyses of Data Coupling and Control Coupling," Completed January 2004, (Rev 2), available at: http://faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/, last visited January 31, 2006.

11. Struck, W., "Data Coupling and Control Coupling—Integration Verification Completion Criteria," a presentation to the *2005 FAA Software/CEH Conference*, July 2005.

12. Howden, W.E., *Functional Program Testing and Analysis*, McGraw-Hill, New York, NY, 1987.

13. Jin, Z. and Offutt, A.J., "Integration Testing Based on Software Couplings," *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, June 1995, pp. 13-23.

14. Podgurski, A. and Clarke, L.A., "A Formal Model of Program Dependencies and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transactions on Software Engineering*, Vol. 16, No. 9, September 1990, pp. 965-979.

15. Ferrante, J., Ottenstein, K.J., and Warren, J.D., "The Program Dependency Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, July 1987, pp. 319-349.

16. Horwitz, S. and Reps, T., "The Use of Program Dependency Graphs in Software Engineering," *Proceedings of the 14th International Conference on Software Engineering (ICSE '92)*, May 1992, pp. 392-411.

17. Harrold, M.J. and Soffa, M.L., "Interprocedural Data Flow Testing," *SIGSOFT Software Engineering Notes,* Vol. 14, No. 8, December 1989, pp. 158-167.

18. Friedman, M.A. and Voas, J.M., *Software Assessment: Reliability, Safety, Testability*, John Wiley and Sons, Inc., Hoboken, NJ, 1995.

19. Voas, J.M. and McGraw, G.M., *Software Fault Injection: Innoculating Programs Against Errors*, John Wiley and Sons, Inc., Hoboken, NJ, 1998.

20. Harrold, M.J. and Soffa, M.L., "Selecting and Using Data for Integration Testing," *IEEE Software*, Vol. 8, No. 2, March 1991, pp. 58-65.

21. Jin, Z. and Offutt, A.J., "Coupling-Based Integration Testing," *Proceedings of ICECCS '96: 2nd IEEE International Conference on Engineering of Complex Computer Systems (held jointly with 6th CSESAW and 4th IEEE RTAW)*, October 1996, pp. 10-17.

22. Jin, Z. and Offutt, A.J., "Coupling-Based Criteria for Integration Testing," *Journal of Software Testing Verification and Reliability*, Vol. 8, No. 3, September 1998, pp. 133-154.

23. Briand, L.C., Labiche, Y., and Wang, Y., "Toward a Comprehensive and Systematic Methodology for Class Integration Testing," Technical Report SCE-03-02, Carleton University, March 2003.

24. Briand, L.C., Labiche, Y., and Wang, Y., "A Comprehensive and Systematic Methodology for Client-Server Class Integration Testing," *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, November 2003, pp. 14-25.

25. Bashir, I. and Goel, A.L., *Testing Object-Oriented Software: Life Cycle Solutions*, Springer-Verlag, New York, NY, 1999.

26. Barnes, J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, Boston, MA, 2003.

27. Myers, G.J., *The Art of Software Testing*, John Wiley and Sons, Hoboken, NJ, 1979.

28. Beizer, B., *Software Testing Techniques*, 2nd edition, Van Nostrand Reinhold, New York, NY, 1990.

29. Beizer, B., *Black-Box Testing—Techniques for Functional Testing of Software and Systems*, John Wiley and Sons, Hobocken, NJ, 1995.

30. Marick, B., *The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing*, Prentice Hall, Upper Saddle River, NJ, 1995.

31. Harrold, M.J. and Rothermel, G., "Performing Data Flow Testing on Classes," *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, December 1994, pp. 154-163.

32. Alexander, R.T. and Offutt, A.J., "Analysis Techniques for Testing Polymorphic Relationships," *Proceedings Technology of Object-Oriented Languages and Systems, 1999 (TOOLS 30)*, August 1999, pp. 104-114.

33. Orso, A. and Pezze, M., "Integration Testing of Procedural Object-Oriented Programs with Polymorphism," *Proceedings of the 16th International Conference on Testing Computer Software: Future Trends in Testing (TCS 1999)*, June 1999.

34. Orso, A., "Integration Testing of Object-Oriented Software," Ph.D. Thesis, Politechnico di Milano, 1999.

35. Alexander, R.T. and Offutt, A.J., "Criteria for Testing Polymorphic Relationships," *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, October 2000, pp. 15-23.

36. Alexander, R.T., "Testing the Polymorphic Relationships of Object-Oriented Programs," Ph.D. Thesis, George Mason University, 2001.

37. Martena, V., Orso, A., and Pezzé, M., "Interclass Testing of Object-Oriented Software," *Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems (ICECCS)*, December 2002, pp. 135-144.

38. Alexander, R.T. and Offutt, J., "Coupling-Based Testing of O-O Programs," *Journal of Universal Computer Science: Special Issue on Breakthroughs and Challenges in Software Engineering*, Vol. 10, No. 4, April 2004, pp. 391-427.

39. Siegel, S., *Object-Oriented Software Testing: A Hierarchical Approach*, John Wiley and Sons, Hoboken, NJ, 1996.

40. Binder, R.V., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, Boston, MA, 2000.

41. McGregor, J.D. and Sykes, D.A., *A Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, Boston, MA, 2001.

42. Chen, M-H. and Kao, H.M., "Testing Object-Oriented Programs—An Integrated Approach," *Proceedings 10th International Symposium on Software Reliability Engineering (ISSRE'99)*, November 1999, pp. 73-82.

43. Offutt, J., Alexander, R., Wu, Y., Xiao, Q., and Hutchinson, C., "A Fault Model for Subtype Inheritance and Polymorphism," *Proceedings 12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, November 2001, pp. 84-93.

44. Alexander, R.T., Offutt, J., and Bieman, J.M., "Fault Detection Capabilities of Coupling-Based OO Testing," *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, November 2002, pp. 207-218.

45. Liskov, B.H. and Wing, J.M., "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 16, No. 6, November 1994, pp. 1811-1841.

46. Core J2EE Pattern Catalog: Core J2EE Patterns—Transfer Object, available at: http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html, last visited January 31, 2006.

47. Perry, D.E. and Kaiser, G.E., "Adequate Testing and Object-Oriented Programming," *Journal of Object-Oriented Programming*, Vol. 2, January/February 1990, pp. 13-19.

48. Information Processing, Ltd. white paper, “Advanced Coverage Metrics for Object-Oriented Software,” available at: http://www.ipl.com/pdf/p0833.pdf , last visited January 31, 2006.

49. Harrold, M.J., McGregor, J.D., and Fitzpatrick, K.J, “Incremental Testing of Object-Oriented Class Structures,” *Proceedings of the 14th International Conference on Software Engineering*, 1992, pp. 68-80.

50. Paradkar, A., “Inter-Class Testing of OO Software in the Presence of Polymorphism,” *Proceedings of the 1996 Conference of the Centre For Advanced Studies on Collaborative Research*, November 1996, pp. 137-146.

51. Tai, K.C. and Daniels, F.J., “Test Order for Inter-Class Integration Testing of Object-Oriented Software,” *Proceedings of the 1997 21st Annual International Computer Software and Applications Conference, COMPSAC'97*, August 1997, pp. 602-607.

52. Labiche, Y., Thevenod-Fosse, P., Waeselynck, H., and Durand, M.H., “Testing Levels for Object-Oriented Software,” *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, June 2000, pp. 136-145.

53. Briand, L.C., Labiche, Y., and Wang, Y., “An Investigation of Graph-Based Class Integration Test Order Strategies,” technical report SCE-01-02, Carleton University, 2001.

54. Briand, L.C., Labiche, Y., and Wang, Y., “Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles,” *Proceedings 12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, November 2001, pp. 287-296.

55. Briand, L.C., Feng, J., and Labiche, Y., “Using Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders,” *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE '02)*, July 2002, pp. 43-50.

56. Milanova, A., Rountev, A., and Ryder, B., “Constructing Precise Object Relation Diagrams,” *Proceedings 18th IEEE International Conference on Software Maintenance (ICSM'02)*, October 2002, pp. 586-595.

57. Briand, L.C., Labiche, Y., and Wang, Y., “An Investigation of Graph-Based Class Integration Test Order Strategies,” *IEEE Transactions on Software Engineering*, Vol. 29, No. 7, July 2003, pp. 594-607.

58. Badri, L., Badri, M., and Ble, V.S., “Object-Oriented Integration Testing: A Method Level Based Approach,” *Proceedings of the Eighth IASTED International Conference on Software Engineering and Applications*, November 2004, pp. 324-330.

59. Chen, Q. and Li, X., "An Order-Assigned Strategy of Classes Integration Testing Based on Test Level," *Eighth International Conference on Computer Supported Cooperative Work in Design*, Vol. 1, May 2004, pp.653-657.

60. Ponder, C. and Bush, B., "Polymorphism Considered Harmful," *Software Engineering Notes*, Vol. 19, No. 2, April 1994, pp. 35-37.

61. Watson, A.H. and McCabe, T.J., "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," *NIST Special Publication*, 1996, pp. 500-235.

62. Smith, M.D. and Robson, D.J., "Object-Oriented Programming—The Problems of Validation," *Proceedings IEEE Conference on Software Maintenance*, November 1990, pp. 272-281.

63. Barbey, S. and Strohmeier, A., "The Problematics of Testing Object-Oriented Software," *Software Quality Management II, Building Quality into Software*, July 1994, pp. 411-426.

64. Barbey, S., Ammann, M.M., and Strohmeier, A., "Open Issues in Testing Object-Oriented Software," *Proceedings of 4th European Conference on Software Quality*, October 1994, pp. 257-267.

65. Orso, A. and Silva, S., "Open Issues and Research Directions in Object-Oriented Testing," *Proceedings of the 4th International Conference on Achieving Quality in Software: Software Quality in the Communication Society (AQUIS'98)*, April 1998.

66. Kirani, S., "Specification and Verification of Object-Oriented Programs," Ph.D. thesis, University of Minnesota, 1994.

67. McDaniel, R. and McGregor, J.D., "Testing the Polymorphic Interactions Between Classes," technical report: TR94-103, Clemson University, 1994.

68. Overbeck, J., "Integration Testing for Object-Oriented Software," Ph.D. thesis, Vienna University of Technology, 1994.

69. Supavita, S. and Suwannasart, T, "Adequacy Criteria for Testing Polymorphism in the Context of Interactions," *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'04)*, June 2004.

70. Rountev, A., Milanova, A., and Ryder, B.G., "Fragment Class Analysis for Testing of Polymorphism in Java Software," *IEEE Transactions on Software Engineering*, Vol. 30, No. 6, June 2004, pp. 372-387.

71. Mandl, R., "Orthogonal Latin Squares: an Application of Experiment Design to Compiler Testing," *Communications of the ACM*, Vol. 28, No. 10, October 1985, pp. 1054-1058.

6. RELATED ACTIVITIES AND DOCUMENTS.

The five activities and documentations that relate directly to the issues addressed herein and provided an important springboard for this study are:

- The joint FAA/NASA Object-Oriented Technology in Aviation project workshops and the associated documentation at http://shemesh.larc.nasa.gov/foot/.

- "Handbook for Object Oriented Technology in Aviation (OOTiA)," Revision 0, October 26, 2004, available at: http://faa.gov/aircraft/air_cert/design_approvals/air_software/oot/, [6].

- Chilenski, J.J., Timberlake, T.C., and Masalskis, J.M., "Issues Concerning the Structural Coverage of Object-Oriented Software," FAA report DOT/FAA/AR-02/113, November 2002 [5].

- Chilenski, J.J., Heck, D., Hunt, R., and Philippon, D., "Object-Oriented Technology (OOT) Verification Phase I Report—Survey Results," The Boeing Company, August 2004 [1].

- Certification Authorities Software Team (CAST), Position Paper CAST-19, "Clarification of Structural Coverage Analyses of Data Coupling and Control Coupling," Completed January 2004, (Rev 2), available at: http://faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/ [7].

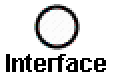
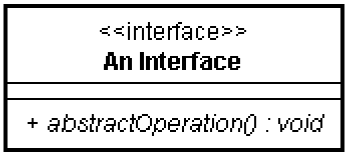
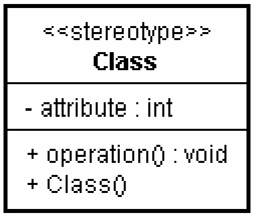
APPENDIX A—UML 2.0 CONVENTIONS USED

This appendix contains a brief overview of the Unified Modeling Language (UML) 2.0 used in this report. More detailed information can be found in the UML 2.0 standard.

The first table defines the basic components of object-oriented technology (OOT) used in this report. The first column gives the name of the component, the second column gives the symbol used, and the third column gives an overview of the use of the component.

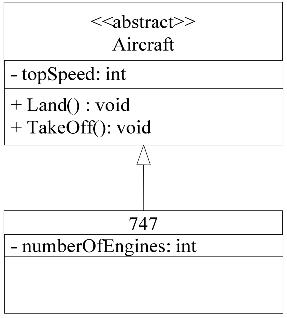
The second table defines the relationships of OOT used in this report. The first three columns are just like the three columns of the first table. The last column gives an example of the use of the relationship.

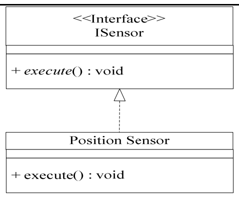
Basic Components

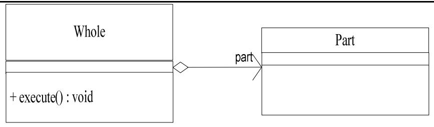
Name	Symbol	Use
Interface	  <<interface>> An Interface + abstractOperation() : void	An interface is a type, just as a class is a type. Both define methods, but only classes define attributes. An interface never implements methods, only defines their method signature to enforce uniformity across a number of classes. A class can implement multiple interfaces.
Class	 <<stereotype>> Class - attribute : int + operation() : void + Class()	A class is the specification for objects, specifying each object's behavior (operations or methods) and state variables (attributes or properties). The constructor for the class has the same name as the class, but there is no return type. Visibility of attributes and methods, when depicted, is either public ("+") or private ("-").

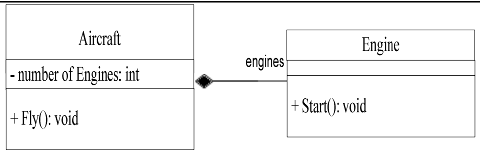
Relationships

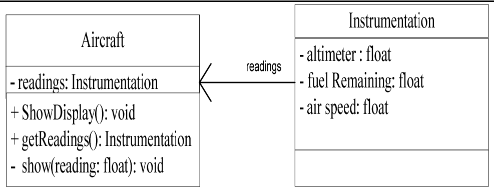
Name	Symbol	Use	Example

Inheritance	◁—	**Implementation Inheritance (Generalize/Specialize)**
		Object-oriented systems define classes in terms of other classes. For example, The 737, 747, 767, 777, and 787 are all types of Aircraft. In object-oriented terminology, 737, 747, 767, 777, and 787 are all subclasses of the aircraft class. Similarly, the aircraft class is the superclass of the others.
			 <<abstract>>
			Aircraft
			- topSpeed: int
			+ Land() : void
			+ TakeOff(): void
			747
			- numberOfEngines: int
			`public class 747 extends Aircraft {`
			`private int numberOfEngines;`
			`}`
			`public abstract class Aircraft {`
			`private int topSpeed;`
			`public void Land() {`
			`}`
			`public void TakeOff() {`
			`}`
			`}`

Implement-ation	———
	OR
	- - - - - - -▷
		Interface Inheritance (Specifies/Refines)
		An interface is the boundary between two systems where they interact. Within the Java programming language, an interface is a type, just as a class is a type. Both define methods. An interface never implements methods; this is left to the classes that implement the interface. By providing a common interface, classes of different types can interact.
			 <<Interface>>
			ISensor
			+ *execute*() : void
			Position Sensor
			+ execute() : void
			`public interface ISensor {`
			`public abstract void execute();`
			`}`
			`public class PositionSensor implements ISensor {`
			`public void execute() {`
			`}`
			`}`

Aggregation	◇———	**Aggregation**
		A part exists independently of the whole.
			
			public class Whole {
			public void execute(){
			Part part = new Part()
			}
			}

Composition	◆———	**Composition**
		A collection or other class that does not exist independently.
			 Aircraft
			- number of Engines: int
			+ Fly(): void
			engines
			Engine
			+ Start(): void
			public class Aircraft {
			public void Fly(){
			Collection Engines;
			Engine engine;
			for(i = 0; i < number of Engines; ++I)
			{
			engine = new Engine();
			engine.Start();
			}
			}

Dependency	⟶	**Dependency**
		Primary Object has another object as a typed attribute or variable.
			 Aircraft
			- readings: Instrumentation
			+ ShowDisplay(): void
			+ getReadings(): Instrumentation
			- show(reading: float): void
			readings
			Instrumentation
			- altimeter : float
			- fuel Remaining: float
			- air speed: float
			public class Aircraft {
			private Instrumentation readings;
			
			public void getReadings(){
			readings = Instrumentation.factory.create();
			
			}
			public void ShowDisplay(){
			show(readings.altimeter);
			show(readings.fuel);
			show(readings. air speed);
			}
			}