4-27-1992

The Development and Application of Testing Techniques in an Object-Oriented Paradigm

Robert F. Kozal
Grand Valley State University

Follow this and additional works at: http://scholarworks.gvsu.edu/theses
Part of the Computer Sciences Commons

Recommended Citation
http://scholarworks.gvsu.edu/theses/137

This Thesis is brought to you for free and open access by the Graduate Research and Creative Practice at ScholarWorks@GVSU. It has been accepted for inclusion in Masters Theses by an authorized administrator of ScholarWorks@GVSU. For more information, please contact scholarworks@gvsu.edu.
THE DEVELOPMENT AND APPLICATION OF TESTING TECHNIQUES IN AN OBJECT-ORIENTED PARADIGM

by

Robert F. Kozal

April 27, 1992

A Thesis Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in Computer Information Systems

GRAND VALLEY STATE UNIVERSITY

This thesis is the study of testing object-oriented programs. Aspects of object-oriented programs for which testing techniques are developed include classes, methods, inheritance, polymorphism, and encapsulation. The application of these techniques are illustrated, and a comparison between traditional testing techniques and object-oriented techniques is also drawn.
# TABLE OF CONTENTS

1. Introduction ............................................. Page 3

2. Background in Design/Structured Design .......................... Page 5

3. Object-Oriented Design (OOD) .................................. Page 13

4. Testing in a non Object-Oriented Environment ................ Page 23
   Integration Testing ......................................... Page 24
   Functional Testing ......................................... Page 26
   Structural Testing ......................................... Page 29

5. Testing in an Object-Oriented Paradigm .......................... Page 32

6. Triangle Problem Designed and Tested .......................... Page 51
   Structured Design Test Cases ................................ Page 52
   Object-Oriented Design Test Cases .......................... Page 56

7. Packet Problem Designed and Tested ............................ Page 64
   Structured Design ......................................... Page 65
   Object-Oriented Design ..................................... Page 68

8. Comparison between Traditional Techniques and Object-Oriented Testing .......... Page 75

Appendices

A. Programs ................................................... Page 79

B. Output for Problem Solutions ................................ Page 96

C. Figures ..................................................... Page 102

D. Bibliography ............................................... Page 132
Chapter 1.

INTRODUCTION

The advent of software development created the need for software design methodologies. Without descriptive and authoritative methodologies the development process, from requirements to finished product (usually in the form of a program or collection of programs), could become a tangled web of program bugs, missed user requirements, and erroneously latent functionality. The traditional method of software development has been the Waterfall method, a rather linear approach to the design of software. Out of this life cycle has come a design method called structured analysis and design. This method offers a constructive structured process which is efficient in many cases and is relatively easy to learn and apply. Many other methods have been introduced as well: data-driven methods, operational specifications, and transformational implementations. To complement the framework which software design methodologies provide, testing techniques are used as a means of ensuring bug-free and failure-proof software. These testing techniques have been developed within the environment created by the methodology. Validation and verification techniques such as structural and functional testing
are two examples of testing components that accompany a particular design method.

Testing techniques are added primarily because the methodology governing the software process has flaws and cannot meet all requirements of the software developer. This lack of a "perfect" methodology is the reason for the evolution of software design methods. This evolution has seen the introduction of a design methodology, called Object-Oriented Design (OOD), back in the 1980’s. This particular methodology brings to the software design table new techniques that corrects issues that other techniques fail to address. Unfortunately, the testing techniques currently in use for other methodologies may not fully apply to this new design method. Since object orientation is based on different design building blocks, the leading testing techniques cannot fully realize the potential problems in an object-oriented environment. In addition, the development of testing techniques specifically for an object-oriented environment has not caught up with the growth and acceptance of object-oriented design.

Development of object-oriented testing techniques is an issue that needs to be addressed, especially since many object-oriented applications are now being developed. The object-oriented approach is a method that provides features which other methodologies do not; the testing technique appropriate for the object-oriented method needs to be realized in order to take full advantage of the design.
Chapter 2.

BACKGROUND IN DESIGN/STRUCTURED DESIGN

If an application developer were given the task of implementing a system, chances are the developer would follow a pre-determined design strategy in order to capture in the implemented system all of the stated system requirements. Design strategies have evolved considerably over the years; however, types of systems have evolved at a much faster rate. Systems have moved from simplistic (in today’s standards) to very complex, such as a retail 24 hour replenishment system.

This evolution from simple systems to complex systems has created many "opportunities" for developers. For many years systems were designed using techniques such as structured design and data-oriented design. Now that systems are becoming more complex, there is a need for the industry-wide adoption of alternative design methods, especially those which incorporate inheritance and information-hiding.

From Grady Booch comes a definition of a complex system. This definition includes the following five attributes:

1. Complexity takes the form of hierarchy.
2. Choice of primitive components is arbitrary.
3. Intracomponent links are stronger than
intercomponent links.

4. Hierarchic systems are usually composed of a few different subsystems in combinations/arrangements.

5. Complex systems evolved from simple systems. [BOOCH p.10]

For those systems which contain these five attributes, Booch follows up with a theory of system design success: "Most successful complex software systems are those whose designs encompass a well engineered class and object structure and embodies the 5 attributes of complex systems." [BOOCH p. 13]

Structured design and data-oriented design can handle designing complex systems to a certain extent; however, they lack inheritance and offer limited information-hiding. Mostow has postulated five principles on design:

1. Satisfies a given (perhaps informal) functional specification.

2. Conforms to limitations of the target medium.

3. Meets implicit or explicit requirements on performance and resource usage.

4. Satisfies implicit or explicit design criteria as the form of the artifact.

5. Satisfies restrictions on the design process itself. [BOOCH p. 20]

Design methods incorporate several related characteristics. For example, the notation that a design method employs usually expresses the system via a set of modeling templates. This notation falls within the specified process for that particular
design method. This differs from a design method's life cycle in that the process dictates how the notations and the tools work together to design a system. The tools of a system are automated techniques using the method's notation to enforce the rules of the design method [BOOCH p. 21].

Obviously there are differences between the various design methods. A more in-depth comparison will be made later after the overview on each method, but there are higher level disparities that can be illustrated here:

1. Procedure-oriented design is algorithms.
2. Object-oriented design is classes and objects.
3. Logic-oriented design has goals, often expressed in predicate calculus.
4. Rule-oriented design has if-then rules.
5. Constraints-oriented design has invariant rules.

[BOOCH p. 38]

A breakdown of tasks in software creation has resulted in several now familiar software life cycles, the most popular being the Waterfall method of system development. The Waterfall method includes the following phases:

1. System Requirements,
2. Software Requirements,
3. General Design,
4. Detail Design,
5. Coding,
6. Testing,
7. Implementation,
8. Maintenance

Variations have been offered on this basic theme; however, the method previously presented encapsulates enough of the original intent to suffice here. The Waterfall method provides an environment suitable for many design techniques, including Structured Design. Main proponents of structured design include Yourdon, Constantine, Meyers, and Page-Jones.
II. Structured Design

The objectives of structured techniques are listed below:

**PRIMARY OBJECTIVES**

1. Achieve high-quality programs of predictable behavior
2. Create programs that are easily modifiable
3. Simplify programs and the program development process
4. Achieve more predictability and control in the development process
5. Speed up systems development
6. Lower the cost of system development

**SECONDARY GOALS**

1. Decompose complex problems and constructs into successively simpler ones.
2. Achieve simplicity of design
3. Control complexity
4. Achieve clear thinking about systems and programs
5. Use diagramming techniques that are clear as possible
6. Improve the readability of diagrams and code
7. Improve communication with end users
8. Achieve unity of architecture
9. Employ consistent, teachable methods
10. Employ a standard set of control structures that can be converted into code with minimum effort
11. Achieve precise communication among people in a development team
12. Minimize the number of developers on a team, achieving one-person teams where possible.

13. Use techniques that work well for large systems as well as small systems.

14. Minimize errors

15. Catch errors as early as possible.

16. Achieve provably correct design where possible

17. Achieve rigorous interfaces between separately developed modules.

18. Achieve ever more powerful building blocks and libraries

19. Achieve sound data administration

20. Achieve sound data analysis

21. Provide an analyst’s and programmer’s workbench with which to maximize help from computers in achieving objectives

22. Achieve the maximum automation of systems design with techniques that make possible the automatic generation of code.

[MARTIN p. 5-7]

Martin also includes four basic principles of structured techniques: the principle of abstraction, the principle of formality, the principle of divide-and-conquer, and the principle of hierarchical ordering [MARTIN p. 16]. The principle of abstraction simplifies a problem to an extent where the dependent attributes of the problem are separated, allowing a developer to examine a system or a system’s requirements with a varying amount of detail. The principle of formality sets forth a strict attention to following a rigid method. The divide-and-conquer concept
proposes to break down a problem into smaller pieces, presenting a series of smaller and less complex problems than the original. The hierarchical ordering concept layers the solution into a tree structure in order to better understand the solution and the steps needed to achieve it. [MARTIN p. 16] Martin also outlined four basic steps in structured design:

1. Represent the design as a flow of data through a set of processes.

2. Represent the design as a hierarchy of functions (or procedural components).

3. Evaluate and improve the design.

4. Prepare the design for the implementation step. [MARTIN p. 423]

Structured design assumes systems have essentially three tasks:

1. Collection and transformation of input data into a ready form for processing.

2. Data processing with the purpose of transforming input data into output results.

3. Transformation and dispersement of the results into final output form. [MARTIN p. 446]

Structured design techniques are realized through the following tools: Data Flow Diagrams, process specifications, data dictionary, state transition diagrams, and Entity/Relationship diagrams. These diagrams are used to logically describe a system and its tasks.

Data Flow Diagrams (DFD’s) are composed of these components: data flows, processes, data stores, and terminators. Transform
analysis is used to design a program by identifying the primary functional components and the high-level inputs and outputs for these components. The Data Flow Diagram is the input to the transform analysis. Transaction analysis serves as an alternative to transform analysis. These two guide structured design through the above 3 tasks.

Figure 1 is an example of a Data Flow Diagram. The processes are denoted by bubbles, the data stores by two lines, the data flows by lines with arrows, and the terminators by a box. Data Flow Diagrams are then converted to structure charts. Structure charts are composed of boxes and arrows which relate to modules and sequence of modules (see figure 2). Entity Relationship Diagrams and State Transition Charts are other graphical means of representing data relationships and dynamic transformations.

Structured design incorporates other concepts as well. The principle of information hiding establishes a relationship between the user and the physical nature of the data. Martin details four other principles: the principle of localization, the principle of conceptual integrity, the principle of completeness, and the principle of logical independence [MARTIN p.33].
Chapter 3.

OBJECT-ORIENTED DESIGN

The theories behind object-orientation have existed for over twenty years, only recently have these approaches become more and more accepted. There are several reasons for this. First, the languages that support object-orientation are becoming more powerful. Second, there has been a noticeable shift from focusing on coding to concentrating on design and analysis. Third, systems developed today are more complicated and complex as well as more "domain-oriented." [COAD pg. 2]

An object can be defined as "an abstraction of something in the domain of a problem or its implementation, reflecting the capabilities of the system to keep information about it, interact with it, or both; an encapsulation of attribute values and their exclusive services." [COAD pg. 26] These objects can be abstractions or concepts for the real world "things" of which the system is concerned about. They usually have firm boundaries and serve two purposes: to "promote understanding, and provide practical basis for computer implementation." [RUMBAUGH p. 21]

A class can be defined as "a description of one or more objects, describable with a uniform set of attributes and services; in addition, it may describe how to create new objects in the class." [COAD pg. 27] A layperson might define a class as a group
of objects with common attributes, behaviors, and abstractions. Because an attribute is the property of the objects in the class, it is similar to an attribute in an Entity Relationship diagram. An object's set of behaviors is a set of all operations available to be applied to an object.

Common threads running through OOD include "abstraction, encapsulation, combining data and behavior, sharing, object structure emphasis, and synergy." [RUMBAUGH p. 17] Links and associations establish relationships among objects and classes. Links are relationships between two or more objects. Associations are groups of links with common structure and common semantics. Generalization is a concept that organizes the relationship among classes into hierarchies based on similarities and differences.

Inheritance is a "kind of" relationship, a specialization/generalization kind of relationship. Inheritance is the only unique difference between object-oriented programming and other types of programming. Inheritance states that classes share attributes and behaviors.

Polymorphism exists when the same behavior can be applied to many classes. For example, the behavior ROTATE is said to exhibit polymorphism because it can be applied to a cube, sphere, or any other object that fits within the class.

Encapsulation is a technique that separates the implementation part of the object from the specification of the object. This is closely related to structured design's information hiding.

Benefits of object-oriented design include:
1. Tackle more challenging problem domains,
2. Improve problem domain expert, analyst, designer and programmer interaction,
3. Increase the internal consistency across analysis, design, and programming,
4. Explicitly represent commonality,
5. Build systems resilient to change,
6. Reuse Object-oriented analysis, OOD, and Object-oriented programming results,
7. Provide a consistent underlying representation.

[COAD p. 17]

Problems in the software development life cycle have always existed, for a chasm lies between the analysis and design phases. For example, there exists a chasm between Data Flow Diagrams and the Entity Relationship Diagram in the structured analysis and design technique (see figure 3). Analyzing and designing in an object-oriented paradigm incorporates classes and objects in such a way that the transition from analysis to design is much more seamless than traditional methods. For object-oriented analysis, object-oriented analysis (OOA) creates objects and classes that map into the object-oriented design. In OOA, objects are descriptions of anything in the problem domain. In OOD, objects are software entities.

In the analysis phase, objects incorporate attributes, behavior and abstraction while objects in OOD incorporate data and methods. Mapping from analysis to design shows:

Data --- attributes
methods --- behaviors
OOA requires three models: the object model, the dynamic model, and the functional model. The object model defines the structure of "anythings" in the problem domain. This includes the objects identity, attributes, behaviors, and abstractions. During the analysis phase the object model must take on all of the real world happenings. Note that associations are also demonstrated in this diagram. The second model in OOA is the dynamic model. All real world timing and sequencing is described in the dynamic model (hence the name dynamic). All system related areas dealing with time, control, and events are mapped into state diagrams. This chart also shows the organization of the dynamic events. The third model is the functional model. This is more related to the structured techniques in that it describes all transformations and functions of the system. Using Data Flow Diagrams, the functional model shows all processes, stores, sinks, sources, and data mappings.

How do all these models relate? All three describe one area of a system, very specifically and partitionally, with each model referencing the other two models. Consequently, the functional model operates on data structures that the object model describes. The dynamic model uses the same structures as well. All events in the dynamic model are linked to the functions and processes in the functional model, as well as the behaviors described in the object model.
To accomplish object-oriented design, Rumbaugh suggests the following:

1. Combine the three models to obtain operations on classes.
2. Design algorithms to implement operations.
3. Optimize access paths to data.
4. Implement control for external interactions.
5. Adjust class structure to increase inheritance.
6. Design associations.
7. Determine object representations.
8. Package classes and associations into modules. [RUMBAUGH p. 228]

OOA and OOD consist of distinct activities that can either be applied in sequence or intertwined. [COAD p. 23]

Just as structured analysis and design fits within the construct of the Waterfall lifecycle, object oriented techniques fall within their own lifecycle. Rumbaugh describes the following methodology:

1. Analysis - Analyze develop the requirements into the functional, object, and dynamic models. A problem statement would also be developed.

2. Systems Design - Make decisions about the general structure of the system, including performance, security, and resources. This is based in part on the dynamic model.

3. Object Design - Develop an object design based on the object model developed in analysis. The operations for the objects can also be found in the dynamic and the functional models.
4. Implementation - Taking the object design, the developer now examines the functional model and produces the system via a object oriented language. This is where testing would also occur.

[RUMBAUGH p. 4]

The impact of object-oriented techniques can be summarized as follows:

1. Shifting of development effort into analysis
2. Emphasis on data structure before function
3. Seamless development process
4. Iterative rather than sequential
OOD EXAMPLE
This example is an employee inquiry. See figure 4 for the object design. The implementation of the object design follows this introduction. Note the private and public attributes and methods. These two characteristics play an important part in the development of testing techniques for object-oriented environments. Public attributes and methods can be accessed by external entities (other classes, modules, etc.). Designing in an object-oriented environment requires the continuous knowledge of what classes have what public and private components. Hidden data corruption can occur through the misuse of public attributes and methods.

// Class Declaration
#include <stdio.h> // C++ include functions
#include <string.h> // C++ include functions

// This is the only class for this example. // The employee class consists of three // private attributes and two public functions
class employee { // Here is the employee // class

    // The private attributes // come next

    private:
    int id;
    char name[80];
    float wage;

    // Here are the public // methods that can be // used

    public:
    employee(int i, char *n, float w);
    void print_payinfo(float hrs);
};
// This is where the functions are actually // coded

employee::employee(int i, char *n, float w) {
    id = i; strcpy(name, n); wage = w;
}

void employee::print_payinfo(float hrs) {
    printf("Employee #%d: %s\n", id, name);
    printf("Hours worked: %6.2f\n", hrs);
    printf("Amount paid: $%7.2f\n\n", hrs * wage);
}

// That is all for the employee class. Now the // code for the main section follows.

main() {

    // Initialize two employee objects:
    // Since the employee function is public we // can send a message to it from the main // section

    employee michael(1, "Michael Jackson", 15.00);
    employee oj(1, "O.J. Simpson", 22.00);

    // Now we are going to also send a message to // another public method: print_employee

    michael.print_payinfo(40.);
    oj.print_payinfo(52.0);

    return 0;
}
COMPARISON BETWEEN OOD AND STRUCTURED DESIGN

OOD and structured design both incorporate similar modeling components, which are the process modeling techniques. If one considers the Extended Systems Modeling Language (ESML) extension of structured design, then both incorporate some form of dynamic modeling. OOD, though, incorporates an object model as well.

The two techniques differ by the order emphasis on the various modeling components. On the one hand, OOD is dominated by the object model. OOD then builds the dynamic model, and then the functional model. Contrarily, structured design stresses functional decomposition. This means that the functional model is developed first, and then the dynamic model is built. In addition, structured designs incorporate an object model, and if this is the case this model is usually built last.

Using the OOD modeling technique, it is easier to extend boundaries of the OOD models than it is to do the same with the structured design models. Also, the points of view are different for each technique. Structured design is basically task-oriented; therefore, the system is a set of sequential processes. In contrast, OOD uses models to build the system, and the resulting implemented system is actually the problem domain described as a set of interacting entities.

The base set of building blocks are also different. For structured design, the developer uses procedures and functions to drive the build. In OOD the developer uses classes and objects to design an abstract of the desired system. The actual system is
often an extension of this abstract model.

Structured design groups related processes together as part of an higher level process. Again, the focus is primarily on the processes. In OOD classes and objects are grouped together so that they belong to a general-specific structure or whole-part structure.
TESTING IN A NON OBJECT-ORIENTED ENVIRONMENT

After the design and construction of a program or series of programs is complete, the next phase in the Waterfall life cycle and the object-oriented life cycle is the testing phase. Testing is a process designed to find faults in the constructed system, as well as to build confidence in the constructed system. Failures in a system are usually due to incorrect behavior in a program. This can be obvious erroneous behavior, or it can be latent functionality not incorporated in the original design. Howden believes that since threads unifying different testing techniques are nonexistent, testing has become a unreliable process that lacks order [HOWDEN p.4].

Five essential activities encompass verification and validation: technical reviews, testing, proofs of correctness, simulation and prototyping, and requirements tracing. Traditionally, testing has been split into three approaches:

1. Functional testing - confidence building
2. Structural testing - fault finding
3. Error Based
FUNCTIONAL TESTING

Functional testing attacks the inputs, outputs, and processes of a program. Thus, Functional testing is also called black box, and the developer must know all three to perform a valid functional test (see figure 7). This can involve the "testing of functions performed by functional synthesis over fault revealing test data" [Howden p.4]. Howden defines the following rules: expressions, conditional branching, iteration, and wrong variable faults [HOWDEN p.100]. Some functional testing techniques include random testing, cause-and-effect testing, and error guessing.

There are both advantages and disadvantages to functional testing. The analyst can begin writing testplans earlier, enabling the analyst time to carefully construct the testplan in such a way that it allows full test coverage. The second advantage is that functional testing is independent of implementation. The disadvantages of functional testing include the fact that the developer can never know how much of the program has been tested or how redundant the tests have been, and that the functional testing success relies upon the correctness of the specifications are.

Functional testing is based on three assumptions that Howden describes as the Functionality Principle, the Input-Output Oracles, and the Competent Programmer Assumption. The Functionality Principle, as Howden details it, asserts that "programs are collections of expressions, conditional functions, and iterative functions" [HOWDEN p. 48]. In reality, this inclusive statement
may or may not take into account functions such as file accesses. Second, the Input-Output Oracles necessitate that the developer find a means of determining if the output of a program was actually generated from the input. This could mean the developer must know at least in theory if not in practice, the algorithms the program uses to generate the output. The Competent Programmer Assumption states that the developer must be able to see that the program functions and the correct functions are equivalent.

Howden follows up on these principles with the following failures of the principles:

1. When the developer fails to realize all of the possible functions and/or structures of a given program,

2. When no oracle is available to test a specific function,

3. When the program functions are designed in such a way that a mapping to equivalent correct functions cannot be found.

[HOWDEN p. 51]

Specific functional tests can be broken down into the first of two categories, interface based testing. This technique consists of input domain testing, equivalence partitioning, and robustness testing. For input domain testing, the developer must generate test cases close to extreme domains of each input variable. The developer can then compare outputs to expected outputs. For equivalence partitioning, the developer looks for inputs that could be treated equally. Using robustness testing, the developer ensures
that the program doesn’t function in a way it wasn’t designed to, such as producing surprise outputs, like results outside of predetermined boundaries or ranges (negative results, results greater than upper bound).

The second category of specific functional tests, function based testing, defines two techniques, special value testing and output domain testing. Special value testing relates to math functions, and can also be termed a worst case analysis test. The following are two examples:

1. Testing results by dividing a number by numbers close to 0.
2. Testing calculations with numbers differing by a large amount, or ranges of values differing by large magnitudes.

Output domain coverage, the second technique, essentially tests the complement of the input domain testing’s realm. In this method the developer picks the program inputs to produce the desired outputs and afterwards verifies that the results match the expected. One may easily compare this to cause-and-effect testing, which is essentially a mapping of inputs to outputs via Boolean operators and then produces a graph which can informally show the relationships between distinct outputs and distinct inputs.
STRUCTURAL TESTING

Structural testing looks for faults in the program by analyzing the program functions that are constructed into one program function. One technique is structural synthesis, which is a graph of program structures or functions. This is to describe the hierarchy or sequence that the functions are executed. The classifications for structural testing are vague. Some methods attack the program structure, others examine the program complexity. Basis path testing is a technique similar in style to DD path graphing [PRESSMAN p601]. The program is mapped to a diagram using bubbles as main program statement groupings. Grouping of statements should adhere to the preset definition, which is:

1. The grouping must have a unique identity,
2. The grouping must follow a unique functional concept,
3. The grouping should be relative to a joint existence.

The groupings are then mapped into a DD path graph diagram. Using this diagram, the developer can determine areas such as domains of change and domains of complexity.

Another structural test seeks out cyclomatic complexity, a software metric that provides a quantitative measure of the logical complexity of a program. This metric will define the number of independent paths in the basis set of a program, and provide an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once.

Condition testing tests all of the logical expressions in the
program. Obviously for this method to function correctly, the developer must know and recognize all of the possible logical conditions. One method of condition testing is to dissect the program, mapping all of the conditions to the specific program paths labeled in the DD path graph. Then the developer can take each expression and determine from the requirements if the expression is correct and the paths selected are correct as well.

With Data Flow Diagram testing the developer selects program test paths, driven by the Data Flow diagrams. The developer pays special attention to the variable specifications and definitions in the DFD’s.

One test similar to Data Flow Diagram testing is Data Flow path testing. In this test, the developer maps the flow of certain variables (data) through the program module. This technique can be easily combined with DD path graphing or condition testing to determine the exact point where the variable is initialized, changed, or read.

Loop testing is related to condition testing, as a condition usually dictates the type of loop used. There are essentially four kinds of loops:

1. Simple Loops,
2. Nested loops,
3. Concatenated loops,
4. Unstructured loops.

The developer should ensure that all loops in the program are verified to have a definite beginning and an ending.
All of the above tests could be performed on a given program, with usually the DD path graphing being the basis for the other tests. One particular order could be DD path graphing, condition testing, and loop testing. One of the key points in testing is to coordinate the functional testing with the structural testing. This ensures that the best possible test coverage is provided for the program.
INTEGRATION TESTING

Unit testing focuses on an individual program or module (see figure 5). For integration testing, the developer examines programs as a group (see figure 6). The developer moves beyond the functional, structural, and error-based tests and examines the group of programs for the following errors:

1. Import/Export Range Problems - These could be variable problems such as Module A uses $x < 40$ and Module B uses $x < 20$.

2. Import/Export Type Compatibilities - Problems that exist could have one program using $x$ as a alphanumeric type and passing $x$ to a program that uses $x$ as a numeric.

3. Representation/Interpretation - Module A might use 0 as a representation of "True" and Module B might use 1 for "True".

4. Parameter Access - Errors might occur because the developer is not aware of all the possible places where a variable might obtain a new value.

5. Transferred Control Domain - Problems might arise when a control variable is passed to a program that does not use it and then passed to a program that does.

This set of errors will be used to support some of the object-oriented testing concepts.

E.F. Miller proposed a set of Integration Level Test Coverage Metrics. The test coverage of the program group expands in direct proportion to the level of the test. The levels of the Integration Level Test Coverage Metrics include:

$I_1$ - Every module is invoked at least once.
I₂ = I₁ + "each major response" of a module.
I₃ = I₁ + "partitions of response variable".
Iₘ = Every invocation of all possible responses in every possible context.

There are also five integration strategies. The Top Down strategy decomposes the program group to a tree-like structure. The developer then tests each tree node under the assumption that the nodes under the tested node is correct. The developer can use stubs to facilitate this assumption. The second integration test strategy is the Bottom Up strategy. The developer tests the program group by building with proven components. The programs driving the test are located "under" the tested program. The Big Bang strategy eliminates the middleperson and dictates that the developer just throw the program group on the machine, turn the machine on, and have the maintainers enjoy the results. Thread testing strategies establish threads between modules. Each thread stub displays the module being tested, the initial settings, the next module to be called, and the parameters to be passed on to the next module. The last integration strategy is the Pair-Wise strategy. This strategy breaks down the relationships between modules. Invocations, "Uses", and "Shares Data" are examples of a Pair-Wise relationship.
Chapter 5.

TESTING IN AN OBJECT-ORIENTED PARADIGM

There are several questions one can ask concerning testing in an object-oriented paradigm, such as

"How can I prove that data is [sic] correct?"
"How does this function modify the data?"
"What assumptions are being made here?" [TOOKE p.36]

A result of Larry Constantine's paper "Object-Oriented and Structured Methods Toward Integration" has been the understanding that although object-orientation is a different view of data and processes, "structured analysis and design is sound enough to accommodate object-oriented adaptations of analysis and design." [CONSTANTINE p. 39]. This result is important in terms of testing in an object-oriented environment because many of the testing results obtained within the confines of structured design and data structured design still hold true.

Constantine determined that "an essential key to successful use of object-oriented organization is to use the well established principles of coupling and cohesion" [CONSTANTINE p. 39]. Coupling is a measure of interconnection among modules, and cohesion is the measure of functional relatedness in a module. There are seven
types of cohesion, listed in order of preferable to least preferable:

1. Functional cohesion,
2. Sequential cohesion,
3. Communicational cohesion,
4. Procedural cohesion,
5. Temporal cohesion,
6. Logical cohesion,
7. Coincidental cohesion.

There are also seven kinds of coupling:

1. No direct coupling,
2. Data coupling,
3. Stamp coupling,
4. Control coupling,
5. External coupling,
6. Common coupling,
7. Content coupling.

In the structured approach, it is desirable to have high cohesiveness and low coupling. This also carries through to object-orientation, especially in light of one of the most prominent benefits of object-orientation: reusability. Constantine asserts that reusability needs high cohesiveness and low coupling, as does structured design, and also well factored object modules [CONSTANTINE p. 39].

The structural and functional testing techniques listed
previously can also be modified to apply to the object-orientation paradigm. These can be considered to be two components of object-oriented testing. The following is a modified list of those testing techniques.

FUNCTIONAL TESTING

1. Input Domain Testing - generate test cases close to extreme domains of each input variable. Compare outputs to expected outputs. Apply a separate Input Domain test to each class.

2. Equivalence Partitioning - Look for inputs that could be treated the same. Again, perform this test per class.

3. Robustness Testing - Ensure program doesn’t do things it isn’t supposed to do. This will entail exercising the characteristics (specification and implementation) of each class.

4. Special Value Testing - relate to math functions. Since each class may have different math functions, the developer needs to test each one individually, in the same ways as in other design environments.

5. Output domain coverage - Pick inputs to get desired outputs. Each class could be set up to be individually tested and mapped to expected results.

6. Cause and effect testing - The use of graphs and decision tables of each class will be essentially the same as in other environments.

STRUCTURAL TESTING

1. Condition Testing - Exercise the various logical conditions in the methods.

2. Data Flow Diagram Testing - Although there isn’t a direct correlation between data flow diagrams and object-oriented design, sometimes DFD’s are used as part of the Functional Model, and if this is the case then the developer should select test paths of classes as relating to the classes attributes.

3. Loop Testing - Apply to each loop in each method in each class.

4. DD Path Graphing/analysis - Each class should have its
methods mapped out and analyzed. This will come in useful for maintenance as well as conditional testing.

5. Data flow path graphing - The same techniques used in the other design environments can be used in OOD. But object-oriented testing has a third component (see figure 8). The rest of this section will detail the third component.

Weyuker has developed a set of axioms on test data adequacy. These axioms will be used to support some of the theories on object-oriented testing that will be presented. These axioms are listed below:

1. Applicability: For every program there exists an adequate test set.

2. Non-exhaustive Applicability: There is a program \( P \) and test data set \( T \) such that \( P \) is adequately tested by \( T \), and \( T \) is not an exhaustive test set.

3. Monotonicity: If \( T \) is adequate for \( P \), and \( T \) is a subset of \( T' \) then \( T' \) is adequate for \( P \).

4. Inadequate Empty Set: The empty set is not an adequate test set for any program.

5. Renaming: Let \( P \) be a renaming of \( Q \); then \( T \) is adequate for \( P \) if and only if \( T \) is adequate for \( Q \).

6. Complexity: For every \( n \), there is a program \( P \), such that \( P \) is adequately tested by a size \( n \) test set, but not by any size \( n-1 \) test set.

7. Statement coverage: If \( T \) is adequate for \( P \), then \( T \) causes every executable statement of \( P \) to be executed.

8. Antiextensionality: There are programs \( P \) and \( Q \) such that \( P \) is equivalent to \( Q \), [test set] \( T \) is adequate for \( P \), but \( T \) is not adequate for \( Q \).

9. General Multiple Choice: There are
programs P and Q which are the same shape, and a test set T such that T is adequate for P, but T is not adequate for Q.

10. Antidecomposition: There exists a program P and component Q such that T is adequate for P, T' is the set of vectors of values that variables can assume on entrance to Q for some t of T, and T' is not adequate for Q.

11. Anticomposition: There exist programs P and Q, and test set T, such that T is adequate for P, and the set of vectors of values of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P;Q [P;Q is the composition of P and Q].

[PERRY pp. 13-14]

Most of these principles can be directly related to object-oriented design and the testing of object-oriented systems.

One must also take OOD design issues into consideration when testing in an object-oriented environment. These issues are decomposability, composability, understandability, and continuity. The breaking down of a problem into smaller interconnected problems, decomposability, provides many opportunities for error introduction; thus, the developer must be aware of how the solutions to the smaller problems connect and not simply assume that the composite of the solutions will equal a solution to the original problem. The problems incurred by the composing of solutions is related to the amount of coupling and cohesion a solution has. Another design issue is composability, the amount of reuse that a module offers. This is an inherent factor in OOD that presents some challenges. For instance, a particular class (or module) has, in the original design, been developed interfacing
with certain other classes. When the class is reused, the assumption may be that since the class has already been tested, reusing it means that it doesn’t have to be tested in its new program environment. This assumption may prove to be erroneous depending on how well the developer thought about the communications between classes. This complements Weyuker’s Anticomposability axiom. Understandability has been defined as "the ease with which a program component can be understood without reference to other information or other modules" [PRESSMAN p. 397]. Having a high degree of understandability can aid in the design or reuse of classes, as developers will have a limited assumption domain and a more robust knowledge domain. Continuity is "the ability to make small changes in a program and have these changes manifest themselves with corresponding changes in just one or a very few modules" [PRESSMAN p.398]. Relating this concept to testing, if a program has continuity, then the proposed changes can be mapped to the existing program and class (module) structure, and the developer should not need to test outside of the change domain. For example, the Ripple effect in structured design means that an error can propagate itself through the design, thereby extending the domain of change. This can happen in an object-oriented design as well, so ensuring the design has continuity means controlling the domain of change and limiting the effects of error propagation.

Classes should also offer protection from the propagation of errors. The classes should be designed to hide its information, and not allow corruption of its data from outside of the class.
For modular architectures, Pressman lists five design considerations, all of which impact testing in an object oriented environment:

1. Linguistic modular units - language should be "capable of supporting the modularity defined".

2. Few Interfaces - "the number of interfaces between modules should be minimized"

3. Small Interfaces - "amount of information should be minimized"

4. Explicit interfaces - "should communicate in direct and obvious ways"

5. Information hiding - "all information about a module is hidden from outside access" [PRESSMAN p.399]

The following is a summary of the object-oriented testing general techniques:

**FUNCTIONAL TESTING**

1. Input Domain Testing  
2. Equivalence Partitioning  
3. Robustness Testing  
4. Special Value Testing  
5. Output domain coverage  
6. Cause and effect testing

**STRUCTURAL TESTING**

1. Condition Testing  
2. Data Flow Diagram Testing  
3. Loop Testing  
4. DD Path Graphing/analysis  
5. Data flow path graphing

**OBJECT-ORIENTED DESIGN ISSUES AFFECTING TESTING**

1. Decomposability  
2. Composability  
3. Understandability
4. Continuity
5. Linguistic Modular Units
6. Few Interfaces
7. Small Interfaces
8. Explicit Interfaces
9. Information Hiding

An inquiry into specific object-oriented testing concepts will now be done. This inquiry will include the following concepts:

1. Classes,
2. Objects,
3. Attributes,
4. Methods,
5. Encapsulation,
6. Inheritance,
7. Polymorphism.
CLASSES

Testing classes is at the heart of testing in an object-oriented environment because all of OOD’s unique characteristics center around classes. Since for every class there exists an adequate test coverage set (referencing Weyuker’s Applicability axiom), one problem that a developer may be faced with is the question of "How do I generate the adequate test set?". There are several components to an adequate class test set. The first component details class structure test sets. Weyuker’s principle General Multiple Choice, centers around two programs that are shaped the same but have different needs in terms of test coverage. This can be transferred to classes, as two classes that have the same shape must have two separate test coverages developed for them. If the classes had the same structure as well, then one could possibly assume that the classes should be combined instead of remaining separate. This is reflected in a test called Class coverage.

The second component, the Class coverage test, ensures that a particular test coverage set $T$ is adequate for a particular class $C$, and that even if class $D$ is of identical shape to $C$, $T$ is not necessarily adequate for $D$. Again, this is an extension of Weyuker’s Multiple Choice axiom. Plus, if $D$ is actually a subset of $C$, then $T$ is not adequate for $D$ referencing a variation of Weyuker’s Montonicity axiom [PERRY pg 16].

The third component of the class test coverage set is the Class responsibility segment. Tooke asserts that "if the data is
wrong, then only a known portion of the program (the member functions) can be responsible." [TOOKE p. 42] Applying this to OOD, since a program in an OOD environment is primarily displaced into classes, the developer can easily pinpoint where the data is being tainted or corrupted. If the data corruption is occurring outside of the main module, then the Class responsibility test can be used to determine where this corruption of data is occurring. One procedure that has been developed to aid in class testing is the "assert(verify());" statement [TOOKE p.36]. This verification routine can help detect data corruption within classes. The strategic placement of this routine in various locations in the class structures could catch the corruption as soon as it occurs. The "assert(verify())" as defined by Tooke will also be used in defining Attribute testing sets.

The final component in the class test coverage set relates to Class method execution. Since a particular method execution order does not exist at class definition, the developer may not realize all the implications of the methods' functionality. For instance, two methods A and B in class C may use the same private data structure D. Method A may initialize all or part of D, while B may never attempt any initialization of D. The developer must ensure that the order of messages to A and B does not allow for B to access D without D being initialized.

Combining these four components into a class test coverage set, a developer can attack three views of classes: structure, class responsibility, and overall class method processing.
OBJECTS

To adequately test an object, one can refer to Weyuker's fifth principle, the Renaming principle, states that if the developer has an adequate test set $T$ for $P$, and $Q$ is a renaming for $P$, then $T$ is adequate for $Q$. When applying this principle to objects, one may assume that if the first object (and class) has an adequate test set for itself, all other objects within that class need not be tested. Another component to object testing is memory usage. Each object uses a certain part of memory. When an object is no longer needed, it needs to be deconstructed in order to release the memory that was used. Developers could also develop "garbage collectors", programs that clean up memory by comparing objects found in memory with the links and associations currently held by the program. If the object is no longer linked to a current activity, the "garbage collector program can "sweep" up the memory space and delete the object.

ATTRIBUTES

Tooke gives two functions to be used when debugging an OOD program, \texttt{dump()} and \texttt{verify()}. A developer could use these two functions to verify the attributes of a given class. [TOOKE p.38] These two functions will display data structures in a readable fashion. The functions that the developer uses to check data structures should be virtual member functions. This insures that the routine called at the start of the method is related to the "instance of derived class pointers accessed through base class pointers." [TOOKE p. 38]. For example, if upon calling the
verify() prior to method execution the developer finds a corrupted data structure, the developer then knows that an external action caused the corruption and not the method. If, on the other hand, the verify() failed at the end of method execution, then obviously the method corrupted the data. Tooke provides an example of verify() in his article "Object-Oriented DeBugging." [TOOKE p.38] This attribute testing component is directly linked to the Class responsibility test.

The developer can also use current testing techniques to check on the validity of the data structures designed. Functional testing often pinpoints data exception errors, and structural testing (specifically condition testing) can locate places in the program or class methods where the variables used won't produce the results that were the developer's original intentions.
METHODS

In order to test methods, as being part of a class, the first determination is deciding whether the method under question is a public method or a private method. Depending on the determination, the method could either be easily mapped into by the same class, or it could have a complex messaging by other classes. If a private method is being tested, then only the other class methods that use the method being tested could possibly affect that method’s performance. However, if a public method of the same class sends a message to the private method, erroneous data could leak into the private method depending on the message that the public method received. Also, the methods called by the method being tested could return erroneous data. The best way to adequately test a private method is via the private method roadmap.

A private method roadmap is similar to a data flow diagram in that it shows all of the inputs and outputs (vector messages) of a private method (see figure 10). The developer can use this diagram and examine all of the messages that may cause a problem. Combining this diagram with one of the attribute testing components, a set of test cases can be generated to cover potential pitfalls. A public method roadmap is similar to the private method roadmap except that it shows all of the inputs and outputs of a public method (see figure 11).

Weyuker’s Statement coverage principle is one key to adequate method structure testing. This principle states that if the developer has an adequate test set for the method, then the test
set causes all executable statements in the method to be executed and tested. Obviously, the developer must execute all possible paths within the method, and so DD path graphing would be an excellent choice for this test. Many other structural techniques can be applied to methods as well. Since methods are the implementations of a class (object), the resulting code falls within the data-oriented and structured approaches and therefore can be tested using the same techniques.

ENCAPSULATION

Encapsulation, as described previously, is a technique for information-hiding. This follows the abstract data type model, where the implementation is separated from the specification, thus allowing the design to be hidden within the implementation. When modules, or objects, call another module, encapsulation allows the called module to hide its implementation specifics from the calling module.

If we have a series of objects such as figure 9, the encapsulation of object Student hides its implementation from any object that would call it. This would suggest that if we changed the implementation for Student, we would not have to test any object that calls it. But, according to Weyuker's anticomposition axiom, all objects calling Student need to be tested because an object's implementation that is tested (as in unit testing) does not guarantee the same results when tested in combinations of object calls. This integration testing, according to Perry and Kaiser, is necessary in any programming situation.
INHERITANCE

Inheritance can be used with great success in object-oriented programs; however, the levels of errors incurred while inheriting can vary greatly. The following levels will be examined:

1. Data level,
2. Method level,

At the data level of inheritance, classes can inherit data structures that are not compatible with the methods already in place. For the method level of inheritance testing there are several key issues. One, the methods inherited by a class from the base class should not have any latent impact on the class's data or other methods. For instance, a method inherited should not communicate with an existing method, especially if the existing method already communicates with the inherited method. The inherited method may reference a data structure that is also inherited but overridden. This could lead to data corruption.

At the process level of inheritance, the class that inherits from a base class also inherits that base class's overall process. Although the inheriting class may have a similar process, there could exist situations where the two processes conflict. This could occur when the class that inherits from the base class overrides some of the inherited data structures or methods, thereby altering the overall process of the class.
POLYMORPHISM

Polymorphism deals with the concept of overridden inherited functions. An object is said to exhibit polymorphism when the base class’s functions that that object inherited are overridden in the object’s originating class. There are essentially two means of accomplishing polymorphism, and therefore two means of testing polymorphism. These two means are overloaded functions and virtual functions.

Overloaded functions are functions that have the same name but are designated for different classes. For example, class POINT and class BOX may have a function labeled DRAW (see figure 12). Now, if the class BOX inherited class POINT’s attributes and methods, then the DRAW function already existing for class BOX would overload any message to the DRAW function for the class BOX. Any messages for the DRAW method for the class POINT would still be received by class POINT. The compiler will handle any errors induced by the developer trying to use a function name twice with the same number of arguments for both functions. Therefore, the developer should concentrate testing polymorphic classes on the determination of which function the developer really wants to use in the particular point in the program. To ensure accuracy of overloaded functions, the programmer should diagram all inherited polymorphic calls. Using this chart, a comparison between the classes’ process and the program’s functionality map should point out any aberrant overloaded method calls.

The compiler will make decisions at compile time that the
developer may be unaware of. For instance, if a program uses a
pointer to a particular method, and if the method’s class also
inherited an overridden method by the same name, the compiler may
decide statically that the pointer points to the base class’s
method and not the intended method. With the graph mentioned
above, the developer may catch the incorrect compiler decision, and
fix this problem by using a virtual function instead.

Virtual functions are functions used to override a base class
method. Instead of being calculated statically at compile time, the
function is dynamically allocated at run-time. The only problem
that a developer would have to worry about would be designating a
method as a virtual method, and then losing track of the virtual
method and making design decisions assuming that it is a static
method.
SUMMARY OF SPECIALIZED OBJECT-ORIENTED TESTING TECHNIQUES

CLASSES

1. Class Structure Test
2. Class Coverage Test
3. Class Responsibility
4. Class Method Execution

OBJECTS

1. Retesting Axiom
2. Object Memory Usage

ATTRIBUTES

1. Dump()
2. Verify()

METHODS

1. Private Method Roadmap
2. Public Method Roadmap
3. DD path graphing

ENCAPSULATION

- Information Hiding Test

INHERITANCE

1. Data Level Test
2. Method Level Test
3. Process Level Test

POLYMORPHISM

1. Overloaded Functions
2. Virtual Functions

The intersection of traditional testing techniques and object-oriented techniques can be seen in figure 13.
Chapter 6.

TRIANGLE PROBLEM

The triangle problem will be one empirical study designated to highlight testing in an object-oriented paradigm. The triangle problem can be summarized as follows: Given three inputs i, j, and k: determine if the three inputs form a triangle and if so, determine the type of triangle (equilateral, isosceles, or scalene).

The structured solution was first arrived at through the data flow diagram shown in figure 14. From the data flow diagram, a structure chart was developed. This is seen in figure 15. After coding and implementation, the structured solution is listed in program #1.
TRADITIONAL TESTING TECHNIQUES APPLIED TO THE TRIANGLE PROBLEM

Analyzing the structured solution, the following tests will be applied:

FUNCTIONAL TESTING

1. Input Domain Testing
2. Equivalence Partitioning
3. Robustness Testing
4. Special Value Testing
5. Output Domain Coverage
6. Cause and Effect Testing

STRUCTURAL TESTING

1. Condition Testing
2. Data Flow Testing
3. Loop Testing
4. DD Path Graphing/Analysis
5. Data Flow Path Graphing

INPUT DOMAIN TESTING

Applying input domain testing, the following test cases were generated:

i,j,k --> close to 0 : Received expected correct results. If i,j,k were real numbers, then there may have been errors.

i,j,k = 0 : This will calculate correctly but result is actually incorrect since a side of a triangle cannot be equal to 0.

i,j,k --> close to oo : This is dependent upon the capabilities of the machine and not upon the program.

i,j,k --> negative : if all three are negative, there is a possibility of receiving results. Specification should be changed to check for negative numbers since a side of a triangle cannot be < 0.
EQUIVALENCE PARTITIONING

Applying Equivalence partitioning, there were six input partitions:

1. All inputs are equal,
2. Two out of three inputs are equal,
3. Two out of three inputs are equal, but sum is less than the third,
4. None of the inputs are equal,
5. None of the inputs are equal and one side is greater than the sum of the other two,
6. One or more of the inputs is erroneous (negative, zero...).

The test cases generated were:

1. \text{i=3, j=3, k=3} \rightarrow \text{equilateral},
2. \text{i=6, j=6, k=10} \rightarrow \text{isosceles},
3. \text{i=100, j=6, k=6} \rightarrow \text{not a triangle},
4. \text{i=3, j=4, k=5} \rightarrow \text{scalene},
5. \text{i=3, j=13, k=50} \rightarrow \text{not a triangle},
6. \text{i=-24, j=23, k=1} \rightarrow \text{not a triangle}.

ROBUSTNESS TESTING

Applying the Robustness testing to this structured design, there appear to be few opportunities for latent functionality. All program functions in the requirements can attributed for by applying the input domain tests.

SPECIAL VALUE TESTING

There are several math functions that can be Special Value tested. For instance, the developer should note the numeric type of the variable \text{n}. The output from the program will report any
math discrepancies, such as math functions that can't handle negative inputs. In testing this program, all math functions could handle negative numbers, as well as very large numbers. There aren't any division equations, or equations where the result will be very large.

OUTPUT DOMAIN COVERAGE

There are four possible outputs, five including an abend. Program results can be "Not A Triangle", "Scalene", "Isosceles", and "Equilateral". Therefore, four inputs can be selected to generate the four outputs:

1. i=3, j=4, k=50 produces "Not A Triangle",
2. i=3, j=4, k=5 produces "Scalene",
3. i=6, j=3, k=6 produces "Isosceles",
4. i=7, j=7, k=7 produces "Equilateral".

CAUSE AND EFFECT TESTING

A few test cases were randomly selected. There were no anomalies.

CONDITION TESTING

Applying structural tests to the structured design, all conditions can be mapped and desk-checked for logic errors. Test cases generated included:

1. Testing the i=j, i=k, and j=k conditions,
2. Testing the n=0, n=1, n=2, and n=3 conditions,
3. Testing the CASE statement.

DATA FLOW DIAGRAM TESTING

The DFD was examined for the different processes, and since it
is a basic DFD the lowest level process were also the highest level processes. Therefore, it was easily mapped into the program. The Input process on the DFD was mapped to three input statements.

LOOPE TTESTING

There are no loops in this program.

DD PATH GRAPHING/ANALYSIS

The DD path graph for this program is shown in figure 16. The DD path graph was checked to ensure all executable statements have been tested. A testing tool capable of statement by statement execution is helpful for this test.

DATA FLOW PATH GRAPHING

Test cases tracing Triangle, n, i, j, and k were generated. n is initialized in one place in the program, and is updated in three. i, j, and k are never initialized but a value is inputed for them in the input process. This is the only place where they are updated. The variable Triangle is never initialized, but there exists no possible path through the program where Triangle is never updated.
For the OOD solution, see figure 17. The OOD program is listed in Program #2. Analyzing the object-oriented solution, the following tests will be applied:

**FUNCTIONAL TESTING**

1. Input Domain Testing  
2. Equivalence Partitioning  
3. Robustness Testing  
4. Special Value Testing  
5. Output domain coverage  
6. Cause and effect testing

**STRUCTURAL TESTING**

1. Condition Testing  
2. Data Flow Testing  
3. Loop Testing  
4. DD Path Graphing/Analysis

**OBJECT ORIENTED TESTING**

1. General Testing Concepts
   - Decomposability  
   - Composability  
   - Understandability  
   - Continuity  
   - Few Interfaces  
   - Small Interfaces  
   - Explicit Interfaces  
   - Information Hiding

2. Specific Testing Concepts
   - Classes  
     - Class Structure Test  
     - Class Coverage Test  
     - Class Responsibility  
     - Class Method Execution  
   - Object  
     - Retesting Axiom  
     - Object Memory Usage
Methods
  Public Method Roadmap
  DD Path Graphing

Encapsulation
  Information Hiding Test

Inheritance
  Data Level Test
  Method Level Test
  Process Level Test

INPUT DOMAIN TESTING

Applying input domain testing, the following test cases were generated:

\[
s_1, s_2, s_3 \rightarrow \text{close to 0} : \text{Received expected correct results. If } s_1, s_2, s_3 \text{ were real numbers, then there may have been errors.}
\]

\[
s_1, s_2, s_3 = 0 : \text{This will calculate correctly but result is actually incorrect since a side of a triangle cannot be equal to 0.}
\]

\[
s_1, s_2, s_3 \rightarrow \text{close to } \infty : \text{This is dependent upon the capabilities of the machine and not upon the program.}
\]

\[
s_1, s_2, s_3 \rightarrow \text{negative} : \text{if all three are negative, there is a possibility of receiving results. Specification should be changed to check for negative numbers since a side of a triangle cannot be } < 0.
\]

EQUIVALENCE PARTITIONING

Applying Equivalence partitioning, there were six input partitions:

1. All inputs are equal,
2. Two out of three inputs are equal,
3. Two out of three inputs are equal, but sum is less than the third,
4. None of the inputs are equal,
5. None of the inputs are equal and one side is greater than the sum of the other two,

6. One or more of the inputs is erroneous (negative, zero...).

The test cases generated were:

1. s1=3, s2=3, s3=3 --> equilateral,
2. s1=6, s2=6, s3=10 --> isosceles,
3. s1=100, s2=6, s3=6 --> not a triangle,
4. s1=3, s2=4, s3=5 --> scalene,
5. s1=3, s2=13, s3=50 --> not a triangle,
6. s1=-24, s2=23, s3=1 --> not a triangle.

ROBUSTNESS TESTING

With the OOD design, the developer would not necessarily know the methods used for the Side class. So, applying the Robustness testing, there appear to be few opportunities for latent functionality. All program functions in the requirements can attributed for by applying the input domain tests.

SPECIAL VALUE TESTING

There are several math functions that can be Special Value tested. In the main section, especially in the conditions, there exists several addition functions. The developer would not know this, though, in functional testing, and so the test cases generated may or may not exercise these functions. In testing this program, all math functions could handle negative numbers, as well as very large numbers. There aren’t any division equations, or equations where the result will be very large.

OUTPUT DOMAIN COVERAGE
There are four possible outputs, five including an abend. Program results can be "Not A Triangle", "Scalene", "Isosceles", and "Equilateral". Therefore, four inputs can be selected to generate the four outputs:

1. s1=3, s2=4, s3=50 produces "Not A Triangle",
2. s1=3, s2=4, s3=5 produces "Scalene",
3. s1=6, s2=3, s3=6 produces "Isosceles",
4. s1=7, s2=7, s3=7 produces "Equilateral".

CAUSE AND EFFECT TESTING

A few test cases were randomly selected. There were no anomalies.

CONDITION TESTING

Applying structural tests to the OOD design, the main section appears to be the best area for condition testing, since there are no conditions in the class methods. These can be mapped and desk-checked for logic errors. Test cases generated included:

1. Testing the s1=s2, s1=s3, and s2=s3 conditions,
2. Testing the n=0, n=1, n=2, and n=3 conditions,
3. Testing the s conditions where s determines the message displayed.

DATA FLOW DIAGRAM TESTING

A DFD was not generated as part of the OOA phase.

LOOP TESTING

There are no loops in this program.

DD PATH GRAPHING/ANALYSIS

The DD path graph for this program is shown in figure 19. The DD path graph was checked to ensure all executable statements have
been tested. The compiler used has a testing tool capable of statement by statement execution, and was used for this test.

**DATA FLOW PATH GRAPHING**

This is similar to the Attribute test, and was not done.

**DECOMPOSABILITY**

In order to apply a decomposability test, the OOD solution should be analyzed as to the extent of the decomposition. From the class diagrams, it is apparent that it was a straight split between side and triangle. There are no inheriting characteristics in either class. If the developer were to assume a part-whole relationship between the side class and the triangle class, then this would introduce the following test case:

In each of the solution areas (being the triangle and the side classes), trace the message communications relating to inheritance. Apply the specific functional tests to each data path and process path. For example, one functional test that could be used to test a inherited data path from side to triangle is the Equivalence Partitioning testing. This would outline the input (output) vectors that the side class could generate, and also outline the input vectors that the triangle class could receive.

**COMPOSABILITY**

Applying composability tests to the OOD triangle solution, the design and code were analyzed for the reuse capability. The Side class could be reused or inherited for classes such as Square, or Polygon, for example. The Triangle class could be reused, but probably only where another triangle class is needed.
UNDERSTANDABILITY

The implementation of the OOD is not very descriptive in terms of the developer's comments. Class demarcations are the only real comments included. In terms of testing, the maintenance developer may find it hard to map from original specification to the implementation. Another question that Understandability brings forward is "Can each class be understood without reference to other classes?". For this the answer would be "Yes". Now, if the specification called for more complex methods or attributes that were not self-descriptive, then the developer would need to be careful to address this issue.

CONTINUITY

The OOD triangle implementation hides information well, and so if a change were to be made in the determination of the triangle, then the developer would just need to change the Triangle class.

FEW INTERFACES

There are few interfaces between classes. This is due to the simplicity of the problem.

SMALL INTERFACES

The interface between the classes is small as well. The number of parameters being passed in the messages are small and easily tracked.

EXPLICIT INTERFACES

The program does not try to hide the messages with classes.

INFORMATION HIDING

As was stated before, the implementation hides information
well. The main section does not know how the determination of the triangle is done.

CLASS STRUCTURE TEST

The class structures are not alike, and so this test is not applicable.

CLASS COVERAGE TEST

There are also no subsets of classes, and so this test is not applicable.

CLASS RESPONSIBILITY

The Side class is responsible for the input of the three sides. If there is an interface error, or if the implementation would call for field edits on the side inputs, then this is the part of the implementation that would be examined. The Triangle class is much more involved. The Triangle class does not alter any of the sides, but it does return the triangle type. If during functional testing the developer discovered that the type returned was wrong, then the Triangle class would be the only section to error check for the triangle determination.

CLASS METHOD EXECUTION

Each side object and each triangle object is initialized before any use of that object. Three side objects are also initialized before any triangle object is created.

OBJECT MEMORY USAGE

The implementation does not contain any code to deinitialize an object; therefore, the implementation will cause memory problems further down the road. The program was run many times before such
a situation arose. The machine had to be rebooted because of the drain on memory.

PUBLIC METHOD ROADMAP

In figure 18, the Public Method Roadmap is shown. All of the messages to the methods are clearly labeled and easily recognized. There are no "looping" of services to be considered.

DD PATH GRAPHING

The DD path graph for the Side class is trivial. There is not a complex path situation for the Side class. The DD path graph for the Triangle class is shown in figure 20. Using the debugger provided by the compiler, each executable path was exercised and tested, fulfilling Weyuker’s Statement Coverage Axiom. The main section also has multiple paths, shown in figure 19. Again, each statement was exercised and tested.

INFORMATION HIDING TEST

This test tried to find places in the implementation where class information could be corrupted without sending messages. All of the class attributes were private, and when stubs were put in the implementation to reference these attributes, the attributes could not be changed.

DATA LEVEL TEST
METHOD LEVEL TEST
PROCESS LEVEL TEST

These three tests could not be tried because there was not any inheritance in the design.
Chapter 7.

PACKET SWITCHING PROBLEM

This problem was taken from William Swartout and Robert Balzer's article, "On the Inevitable Intertwining of Specification and Implementation".

The problem is summarized as follows:

"The package router is a system for distributing packages into destination bins. The packages arrive at a source station, which is connected to the bins via a series of pipes. A single pipe leaves the source station. The pipes are linked together by two-position switches. A switch enables a package sliding down its input pipe to be directed to either of its two output pipes. There is a unique path through the pipes from the source station to any particular bin.

Packages arriving at the source station are scanned by a reading device which determines a destination bin for the package. The package is then allowed to slide down the pipe leaving the source station. The package router must set its switches ahead of each package sliding through the pipes so that each package is routed to the bin determined for it by the source station." [SWARTOUT]

The bin map is shown in figure 21. The structured design solution is shown in figures 22 and 23. After coding and implementation, the structured solution is listed in program #3.
TRADITIONAL TESTING TECHNIQUES APPLIED TO THE PACKET PROBLEM

Analyzing the structured solution, the following tests will be applied:

FUNCTIONAL TESTING
1. Input Domain Testing
2. Equivalence Partitioning
3. Robustness Testing
4. Special Value Testing
5. Output Domain Coverage
6. Cause and Effect Testing

STRUCTURAL TESTING
1. Condition Testing
2. Data Flow Testing
3. Loop Testing
4. DD Path Graphing/Analysis
5. Data Flow Path Graphing

INPUT DOMAIN TESTING
Since there is only one input, and it is an integer, four tests were tried:
1. Destination → -oo
2. Destination → +oo
3. Destination = 0
4. Destination between -oo and +oo

EQUIVALENCE PARTITIONING
Applying input domain testing, the following test cases can be generated:
Destination equal to one of the bins → Program performed as expected,
Destination not equal to one of the bins → Program
routed package as expected.

**ROBUSTNESS TESTING**

Applying the Robustness testing to this structured design, there appear to be few opportunities for latent functionality. All program functions in the requirements can attributed for by applying the input domain tests.

**SPECIAL VALUE TESTING**

There weren't any math functions in this program.

**OUTPUT DOMAIN COVERAGE**

There were three outputs expected: an error, a correct bin, or an incorrect bin. Using this coverage, only two of the three were reached. A destination was selected to get the correct bin output and the incorrect bin output.

**CAUSE AND EFFECT TESTING**

After inputing many different bin destinations, no anomalies were found.

**CONDITION TESTING**

There are several condition statements that were tested, but none involved more than one condition. No logic errors were found.

**DATA FLOW DIAGRAM TESTING**

The implementation was mapped back to the DFD drawn for this problem. Since again the high level processes were the same as the lower level processes, the demarcations in the program processes matched that of the DFD.

**LOOP TESTING**

There is one While loop in the program, and the program was
executed in order to try and obtain an infinite loop. This was not accomplished. Since each pipe stream ends with a number greater than 100, the While loop always had an ending.

**DD PATH GRAPHING/ANALYSIS**

The DD path graph for this program is shown in figure 24. The DD path graph can be checked to ensure all executable statements have been tested.

**DATA FLOW PATH GRAPHING**

The variable Destination was mapped through the program. The variable is updated only once, and is referenced in several places. P and S were also mapped.
OBJECT-ORIENTED TESTING TECHNIQUES APPLIED TO THE PACKET PROBLEM

For the OOD solution, see figure 25. The OOD program is listed in program #4. Analyzing the object-oriented solution, the following tests will be applied:

FUNCTIONAL TESTING

1. Input Domain Testing
2. Equivalence Partitioning
3. Robustness Testing
4. Special Value Testing
5. Output domain coverage
6. Cause and effect testing

STRUCTURAL TESTING

1. Condition Testing
2. Data Flow Testing
3. Loop Testing
4. DD Path Graphing/Analysis

OBJECT ORIENTED TESTING

1. General Testing Concepts
   Decomposability
   Composability
   Understandability
   Continuity
   Few Interfaces
   Small Interfaces
   Explicit Interfaces
   Information Hiding

2. Specific Testing Concepts
   Classes
   Class Structure Test
   Class Coverage Test
   Class Responsibility
   Class Method Execution
   
   Object
   Retesting Axiom
   Object Memory Usage

   Methods
Public Method Roadmap
DD Path Graphing

Encapsulation
  Information Hiding Test

Inheritance
  Data Level Test
  Method Level Test
  Process Level Test

INPUT DOMAIN TESTING

Since there is only one input, and it is an integer, four tests were tried:

1. Destination $\rightarrow -\infty$
2. Destination $\rightarrow +\infty$
3. Destination $= 0$
4. Destination between $-\infty$ and $+\infty$

EQUIVALENCE PARTITIONING

Applying input domain testing, the following test cases can be generated:

Destination equal to one of the bins $\rightarrow$ Program performed as expected,

Destination not equal to one of the bins $\rightarrow$ Program routed package as expected.

ROBUSTNESS TESTING

The program was checked for robustness through the entering of varied inputs. No discrepancies were noticed.

SPECIAL VALUE TESTING

There were no math functions to be exercised.

CAUSE AND EFFECT TESTING

The program test plan called for the entering of a destination
that was not already tested in the previous test. For example, an
alphanumeric destination was entered. This obviously did not work.
Any other possible destination input was tried in the previous test
cases.

**CONDITION TESTING**

The While loop has the condition:

```c
while (local != destination && local < 99).
```

This was mapped to:

```c
while (1 <> d and 1 < 99).
```

In this condition statement, 1 must either be equal to d or be
greater than 99 to exit out of the program. 1 gets it's value from
the current_element that is returned from the pipe or switch
connector's next connection. Since the developer can see the pipe
and switch initialization, and also map out the pipes and switches,
it is apparent that sooner or later 1 will be greater than 99. The
only means that 1 can equal d is if the user enters in a
destination that is on the pipe/switch map.

Other conditions that are in this program include the
initialization of the left/right switches, and the checking to see
if the current_element is a pipe or a switch. There are no other
compound conditions to check.

**DATA FLOW DIAGRAM TESTING**

A Data Flow Diagram was not used as part of the object-
oriented design.

**LOOP TESTING**

The while loop is the only loop in the program. This was
checked in the Condition Test.

**DD PATH GRAPHING/ANALYSIS**

The DD path graph for the main section is in figure 26. Each executable statement was traced using the debugger option. There isn’t a DD path graph for the class methods, as they are trivial.

**DECOMPOSABILITY**

There is a measure of inheritance in this OOD packet solution. The class Connector serves as base class for the Pipe and Switch classes, and provides the attribute Number and the method return_connector_number. This method is not overridden by either class. The decomposition, then, primarily affects the Connector-Pipe-Switch arrangement. Another option for decomposability would be to let the Switch and Pipe classes inherit more attributes and methods from the Connector class. Test plans should be developed with the current inheritance structure in mind, though.

**COMPOSABILITY**

The Composability Test brings up the question of reusability. Can any of the packet problem classes be reused? If the Switch class were to be reused, the developer must be aware that Switch inherits an attribute and a method from the class Connector. If a developer wanted to reuse Packet, then the developer would not need to be concerned about any inheritance, only the possibility that the methods contained inside Packet reference another class’s methods.

**UNDERSTANDABILITY**

The code produced for this problem was commented in several
locations, enough to give a feel for the program but not nearly enough for a maintainer to grasp the best possible view of the program's functionality and sequence.

CONTINUITY

This program uses the main section to progress through the packet map of pipes and switches. If the program wanted to hide this implementation, then the developer could have constructed a class called Packet_Map, which inherited Pipe, Switch, and Connector. Then in the Packet_Map public methods a method called Deliver_Packet could have been written. This would have hidden the implementation of the delivery function from the main section.

FEW INTERFACES

The interfaces are few between the classes.

SMALL INTERFACES

The interfaces are small between the classes as well.

EXPLICIT INTERFACES

The program was not developed with the most explicit interfaces. For example, the initialization of the pipes and switches appears to be initializing an ordinary array, and not so much an array of objects.

CLASS STRUCTURE TEST

Class structures for this program are not alike, and so this test was not used.

CLASS COVERAGE TEST

There are also no subsets of classes, and so this test was not used.
CLASS RESPONSIBILITY

The Connector class is responsible for the number of each pipe and switch. Therefore, the area of responsibility for the Connector class extends into both classes as well as into the main section, as this is where the initialization happens. The other classes do not hold responsibility over any other class.

CLASS METHOD EXECUTION

The main section executes the class methods in the correct order. The objects are initialized before any create-read-update-delete method executes.

OBJECT MEMORY USAGE

The objects did not release their memory hold at any point in the program execution. Therefore, memory is being wasted and could cause problems in the future.

PUBLIC METHOD ROADMAP

The Public Method Roadmap is shown in figure 27.

DATA LEVEL TEST

The inheritance of the number attribute from the class Connector allows the use of the Data Level Test. This attribute was traced using the Class Responsibility Test and was type checked and value checked. No errors were detected.

METHOD LEVEL TEST

The return_connector_number method does not update any of the inheriting classes' attributes, and so this test did not produce any error situations.

PROCESS LEVEL TEST
The Connector class was designed using the return_connector_number method as a means for returning the number of the connector. The classes that inherited this method, the Switch class and the Pipe class, following this process design.
Chapter 8.

COMPARISONS BETWEEN TESTING TECHNIQUES

The triangle problem illustrates similarities and differences between traditional testing techniques and object-oriented testing techniques. Recall figure 5, which shows the coverage provided by traditional unit tests. This coverage worked well for the structured triangle problem solution, but failed in adequately covering the OOD solution. The coverage in figure 8, extending the traditional techniques, adequately covers the OOD solution. The base of both coverages is the same, functional and structural, but the OOD solution needs the object-oriented extension.

This is especially true in testing the type_of_triangle method. If a developer did not use a public method graph, then errors on the message may be missed. Likewise, if the OOD solution incorporated more inheritance, then errors would be caught using the data level, method level, and process level inheritance tests. This same lack of coverage provided by the traditional testing techniques also applies to the packet problem test cases. The base of the coverage remains the same, with the functional and structural testing techniques, but again the areas of inheritance and message passing are not covered unless the object-oriented extensions are used.

Generally speaking, the two testing techniques, traditional
methods and object-oriented methods, are similar in that they both build from the same base of structural testing and functional testing. The two testing methods are conducted in the same fashion in both the structured and the object-oriented environments. Presumably any functional test that can be performed in a structured environment can be performed in an object-oriented environment.

The structural tests that can be performed in the two environments differ in the area of analysis tests. If an object-oriented design did not include Data Flow Diagrams, then a tester cannot perform a Data Flow Diagram test on the program. For the majority of the structural tests, however, they can be performed in either testing environment.

The major difference between testing in an object-oriented environment and a structured environment is the object-oriented concepts of messages, inheritance, and encapsulation. Since unit testing in a structured environment does not take into consideration the actual connecting of various modules, a developer does not need to test this aspect at the unit test level. However, since object-oriented design is based on the sending of messages, to avoid testing messages at the unit testing level would be disastrous for an OOD program.

OOD also incorporates classes, a structure that structured design does not have. Again, to miss testing the classes would mean missing a major part of an OOD program. Other data structure tests can be ported from one testing environment to the other. For
instance, variable type testing would be applicable in both arenas, as would parameter variable checking. Other test components in the object-oriented paradigm that is not in the structured paradigm are the inheritance tests, the polymorphism tests, and the encapsulation tests.

Test coverage provided by functional tests and structural tests are enough coverage for each section of code, but they are not enough to provide full object-oriented test coverage. Combining all of the techniques developed here will provide adequate coverage in the object-oriented paradigm.
APPENDICES
APPENDIX A

PROGRAMS
PROGRAM #1
STRUCTURED SOLUTION TO TRIANGLE PROBLEM

{TRIANGLE PROGRAM}
{WRITTEN 3/92}
{ROBERT KOZAL}

program TRIANGLE;
var
  i, j, k, n : integer;
  triangle : char;
begin
  n := 0;

  writeln ('Enter in the first side length:');
  readln (i);

  writeln ('Enter in the second side length:');
  readln (j);

  writeln ('Enter in the third side length:');
  readln (k);

  if (i=j) then
    n := n + 1;
  if (i=k) then
    n := n + 2;
  if (j=k) then
    n := n + 3;

  if (n=0) then
    if (i+j) <= k then
      triangle := 'N'
    else
      if (i+k) <= j then
        triangle := 'N'
      else
        if (j+k) <= i then
          triangle := 'N'
        else
          triangle := 'S'
  else
    if (n=1) then
      if (i+j) <= k then
        triangle := 'N'
      else
        if (i+k) <= j then
          triangle := 'N'
        else
          if (j+k) <= i then
            triangle := 'N'
          else
            triangle := 'S'
  else
    if (n=2) then
      if (i+j) <= k then
        triangle := 'N'
      else
        if (i+k) <= j then
          triangle := 'N'
        else
          if (j+k) <= i then
            triangle := 'N'
          else
            triangle := 'S'
  else
    if (n=3) then
      if (i+j) <= k then
        triangle := 'N'
      else
        if (i+k) <= j then
          triangle := 'N'
        else
          if (j+k) <= i then
            triangle := 'N'
          else
            triangle := 'S'

end.
else
    triangle := 'I'
else
    if (n=2) then
        if (i+k) <= j then
            triangle := 'N'
        else
            triangle := 'I'
    else
        if (n=3) then
            if (j+k) <= i then
                triangle := 'N'
            else
                triangle := 'I'
        else
            triangle := 'E'
    case triangle of
    'N' : writeln ('Not a triangle');
    'S' : writeln ('Scalene triangle');
    'I' : writeln ('Isosceles triangle');
    'E' : writeln ('Equilateral triangle');
end;
end.
// TRIANGLE PROGRAM
// WRITTEN 3/92
// ROBERT KOZAL

#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <iomanip.h>

// Class Declaration

class side {

private:
    int s;
    int length;

public:
    side(int s);
    return_side();
    void print_side(int s);
};

side::side(int s) {
    int l;
    printf("Enter side \%d length: ", s);
    cin >> l;
    length = l;
}

side::return_side() {
    int side;
    side = length;
    return (side);
}

void side::print_side(int side) {
    printf("Side Length \%d: \%d\n", side, length);
// Class Declaration

class triangle {

private:
  char triangletype;
  int s1;
  int s2;
  int s3;

public:
  triangle (int sd1, int sd2, int sd3);
  typeoftriangle ();
};

triangle::triangle(int sd1, int sd2, int sd3)
{
  s1 = sd1;
  s2 = sd2;
  s3 = sd3;
}

triangle::typeoftriangle()
{
  char s;
  int n;
  n=0;

  if (s1==s2)
    n=n+1;
  if (s1==s3)
    n=n+2;
  if (s2==s3)
    n=n+3;

  if (n==0)
    if ((s1+s2)<=s3)
      s='N';
    else
      if ((s2+s3)<=s1)
        s='N';
      else
        if ((s1+s3)<=s2)
          s='N';
        else
          s='S';
  else
    if (n==1)
      if ((s1+s2)<=s3)
        s='N';
      else
        s='N';

};
s='I';
else
  if (n==2)
    if ((s1+s3)<=s2)
      s='N';
    else
      s='I';
  else
    if (n==3)
      if ((s2+s3)<=s1)
        s='N';
      else
        s='I';
    else
      s='E';

  return (s);
}

main ()
{
  char s;
  int i;          
  int side1;
  int side2;
  int side3;

  side so(1);
  side st(2);
  side sh(3);

  so.print_side(1);
  st.print_side(2);
  sh.print_side(3);

  side1 = so.return_side();
  side2 = st.return_side();
  side3 = sh.return_side();

  triangle tr(side1, side2, side3);
  s=tr.typeoftriangle();

  if (s=='N')
    printf ("Triangle is actually not a real triangle\n");

  if (s=='S')
    printf ("Triangle is a scalene triangle\n");
if (s=='I')
    printf ("Triangle is an isosceles triangle\n");

if (s=='E')
    printf ("Triangle is an equilateral triangle\n");

return 0;
}
PROGRAM #3

STRUCTURED SOLUTION TO PACKET PROBLEM

{PACKET PROBLEM}
{WRITTEN 4/92}
{ROBERT KOZAL}

program PACKET;

var
    packet, destination : integer;
    pipe : array[1..13] of integer;
    switch : array[1..8] of array[1..2] of integer;
    switch_setting : array[1..8] of integer;
    flip : integer; {for alternating}
        {switch/pipe}
    element : integer;

begin

    {Initialize the pipes and switches}

    pipe[1] := 1;
    pipe[2] := 2;
    pipe[4] := 3;
    pipe[5] := 5;
    pipe[7] := 400;
    pipe[8] := 500;
    pipe[9] := 600;
    pipe[10] := 7;
    pipe[12] := 8;

    switch[1,1] := 2;
    switch[1,2] := 3;
    switch[2,1] := 100;
    switch[2,2] := 4;
    switch[3,1] := 200;
    switch[3,2] := 300;
    switch[4,1] := 5;
    switch[4,2] := 6;
    switch[5,1] := 7;
    switch[5,2] := 8;
    switch[6,1] := 9;
switch[6,2] := 10;
switch[7,1] := 11;
switch[7,2] := 12;
switch[8,1] := 800;
switch[8,2] := 13;

{initialize switch settings to the right}

switch_setting[1] := 1;
switch_setting[2] := 1;
switch_setting[3] := 1;
switch_setting[4] := 1;
switch_setting[5] := 1;
switch_setting[6] := 1;
switch_setting[7] := 1;
switch_setting[8] := 1;

{get destination for packet}

writeln ('Enter in packet destination: ');
readln (destination);

{initialize switch settings to the path for destination 100}

if (destination=100) then
  begin
    switch_setting[1] := 1;
    switch_setting[2] := 1;
  end;

{initialize switch settings to the path for destination 400}

if (destination=400) then
  begin
    switch_setting[1] := 2;
    switch_setting[4] := 1;
    switch_setting[5] := 1;
  end;

{initialize switch settings to the path for destination 700}

if (destination=700) then
  begin
    switch_setting[1] := 2;
    switch_setting[4] := 2;
    switch_setting[6] := 2;
    switch_setting[7] := 1;
  end;

{initialize switch settings to the path for destination 900}

if (destination=900) then
begin

switch_setting[1] := 2;
switch_setting[4] := 2;
switch_setting[6] := 2;
switch_setting[7] := 2;
switch_setting[8] := 2;
end;

flip := 1;  \{initialize to pipe\}
packet := 1;  \{initialize to entry station\}
element := 1;

while ((packet <> destination) and (packet <99)) do begin
  if (flip = 1) then begin
    packet := pipe[element];
    if (packet<99) then writeln ('Packet passed to switch ',element);
    flip := 2;
  end
  else begin
    if (switch_setting[element] = 1) then
      packet := switch[element,1]
    else
      packet := switch[element,2];
    if (packet<99) then writeln ('Packet passed to pipe ',element);
    flip := 1;
  end;
  element := packet;
end;

if (packet=destination) then begin
  writeln;
  writeln ('Packet arrived correctly at destination', destination);
end
else begin
  writeln;
  writeln ('Packet arrived incorrectly at destination', packet);
end;
end.
PROGRAM #4
OOD SOLUTION FOR PACKET PROBLEM

// PACKET SWITCHING PROGRAM
// WRITTEN 3/92
// ROBERT KOZAL

#include <stdio.h>
#include <string.h>
#include <iostream.h>

// Class Declaration
class packet {

private:
    int current_location;
    int packet_number;

public:
    packet(int packet_num);
    return_current_location();
    void update_current_location(int loc);
};

packet::packet(int packet_num)
{
    packet_number = packet_num;
    current_location = 0;
}

packet::return_current_location()
{
    int curr_loc;
    curr_loc = current_location;
    return (curr_loc);
}

void packet::update_current_location(int loc)
{
    current_location = loc;
}

// Class Declaration
class connector {

public:
    int number;
    int return_connector_number() {return(number);}
};

// Class Declaration
class switcher : connector {

private:
    int switch_setting;
    int right_pipe_connection;
    int left_pipe_connection;

public:
    switcher(int switch_num, int left, int right);
    void set_switcher(int switch_choice);
    return_pipe_connection();
};

switcher::switcher(int switch_num, int left, int right)
{
    right_pipe_connection = right;
    left_pipe_connection = left;
    number = switch_num;
}

void switcher::set_switcher(int switch_choice)
{
    switch_setting = switch_choice;
}

switcher::return_pipe_connection()
{
    int switch_set;

    if (switch_setting == 1)
        switch_set = left_pipe_connection;
    if (switch_setting == 2)
        switch_set = right_pipe_connection;

    return (switch_set);
}
// Class Declaration

class pipe : connector {

private:
    int switch_connection;
public:
    pipe(int pipe_num, int switch_connect);
    return_switch_connection();
};

pipe::pipe(int pipe_num, int switch_connect)
{
    number = pipe_num;
    switch_connection = switch_connect;
}

pipe::return_switch_connection()
{
    int switch_connect;
    switch_connect = switch_connection;
    return (switch_connect);
}

main ()
{
    // Current position of packet - local to main
    int local;

    // Current number of element (switch or pipe)
    int current_element;

    // Current number of element (switch or pipe)
    int curr_el_number;

    // Destination of packet
    int destination;

    // If s=1 then passing through switch
    int s;

    // If p=1 then passing through pipe
    int p;

    // Initialize series of switches and pipes. This is from diagram
switcher switchnum[8] = {
    switcher(0,1,2),
    switcher(1,100,3),
    switcher(2,200,300),
    switcher(3,4,5),
    switcher(4,6,7),
    switcher(5,8,9),
    switcher(6,10,11),
    switcher(7,800,12)
};

pipe pipenum[13] = {
    pipe(0,0),
    pipe(1,1),
    pipe(2,3),
    pipe(3,2),
    pipe(4,4),
    pipe(5,5),
    pipe(6,400),
    pipe(7,500),
    pipe(8,600),
    pipe(9,6),
    pipe(10,700),
    pipe(11,7),
    pipe(12,900)
};

// Initialize packet

packet_one(1);

// Initialize main station. Since we are starting at the
// first pipe, current element = 0 and p=1

current_element=0;
curr_el_number=0;
s=0;
p=1;

// Get destination of packet

printf ("What is the packet's destination: ");
 cin >> destination;

// Initialize selected delivery paths. Paths could also be // objects.
if (destination==100) {
    switchnum[0].set_switcher(1);
    switchnum[1].set_switcher(1); }

if (destination==400) {
    switchnum[0].set_switcher(2);
    switchnum[3].set_switcher(1);
    switchnum[4].set_switcher(1); }

if (destination==700) {
    switchnum[0].set_switcher(2);
    switchnum[3].set_switcher(2);
    switchnum[5].set_switcher(2);
    switchnum[6].set_switcher(1); }

if (destination==900) {
    switchnum[0].set_switcher(2);
    switchnum[3].set_switcher(2);
    switchnum[5].set_switcher(2);
    switchnum[6].set_switcher(2);
    switchnum[7].set_switcher(2); }

// Deliver to correct mail slot

// This will follow from switch to pipe... until destination
// is found. This also updates the current position of
// packet for future use, such as multiple packets and
// real-time handling of packets

printf ("\Nnow starting delivery...\n\n"");
printf ("Starting in pipe #0\n");

local = packet_one.return_current_location();

    while (local != destination && local < 99) {

        if (s==1) {
            s=0;
            p=1;

            current_element =
            switchnum[curr_el_number].return_pipe_connection();

            if (current_element <99)
                printf ("Passing into pipe
                #%d\n",current_element);
        }
    else
        if (p==1) {

}}
s=1;
p=0;

    current_element =
    pipenum[curr_el_number].return_switch_connection();

    if (current_element <99)
        printf ("Passing into switch
        #%d\n",current_element);
    }

    local = current_element;
    packet_one.update_current_location(local);
    if (local<99)
        curr_el_number = current_element;
    }

    if (local==destination)
        printf ("Arrived at correct destination %d
", destination);
    else
        printf ("Arrived at incorrect destination %d\n", local);
    return 0;
    }
APPENDIX B

OUTPUT FOR PROBLEM SOLUTIONS
TRIANGLE SOLUTION OUTPUTS

STRUCTURED SOLUTION OUTPUTS

OUTPUT #1
Enter in the first side length: 3
Enter in the second side length: 4
Enter in the third side length: 5
Scalene triangle

OUTPUT #2
Enter in the first side length: 4
Enter in the second side length: 4
Enter in the third side length: 4
Equilateral triangle

OUTPUT #3
Enter in the first side length: 3
Enter in the second side length: 4
Enter in the third side length: 76
Not a triangle

OUTPUT #4
Enter in the first side length: 8
Enter in the second side length: 4
Enter in the third side length: 4
Isosceles triangle

OOD SOLUTION OUTPUTS
OUTPUT #1

Enter side 1 length: 8
Enter side 2 length: 9
Enter side 3 length: 9
Side length 1: 8
Side length 2: 9
Side length 3: 9
Triangle is an isosceles triangle

OUTPUT #2

Enter side 1 length: 13
Enter side 2 length: 12
Enter side 3 length: 6
Side length 1: 13
Side length 2: 12
Side length 3: 6
Triangle is an scalene triangle

OUTPUT #3

Enter side 1 length: 100
Enter side 2 length: 4
Enter side 3 length: 5
Side length 1: 100
Side length 2: 4
Side length 3: 5
Triangle is actually not a real triangle

OUTPUT #4

Enter side 1 length: 17
Enter side 2 length: 17
Enter side 3 length: 17
Side length 1: 17
Side length 2: 17
Side length 3: 17
Triangle is an equilateral triangle
PACKET SOLUTION OUTPUTS

STRUCTURED SOLUTION OUTPUTS

OUTPUT #1

Enter in packet destination:
100
Packet passed to switch 1
Packet passed to pipe 2
Packet passed to switch 2

Packet arrived correctly at destination 100

OUTPUT #2

Enter in packet destination:
400
Packet passed to switch 1
Packet passed to pipe 3
Packet passed to switch 4
Packet passed to pipe 5
Packet passed to switch 5
Packet passed to pipe 7

Packet arrived correctly at destination 400

OUTPUT #3

Enter in packet destination:
700
Packet passed to switch 1
Packet passed to pipe 3
Packet passed to switch 4
Packet passed to pipe 6
Packet passed to switch 6
Packet passed to pipe 10
Packet passed to switch 7
Packet passed to pipe 11

Packet arrived correctly at destination 700

OUTPUT #4

Enter in packet destination:
900
Packet passed to switch 1
Packet passed to pipe 3
Packet passed to switch 4
Packet passed to pipe 6
Packet passed to switch 6
Packet passed to pipe 10
Packet passed to switch 7
Packet passed to pipe 12
Packet passed to switch 8
Packet passed to pipe 13

Packet arrived correctly at destination 900

OUTPUT #5

Enter in packet destination:
123
Packet passed to switch 1
Packet passed to pipe 2
Packet passed to switch 2

Packet arrived incorrectly at destination 100

OOD SOLUTION OUTPUTS

OUTPUT #1

What is the packet’s destination: 100
Now starting delivery...

Starting in pipe #0
Passing into switch #0
Passing into pipe #1
Passing into switch #1
Arrived at correct destination 100

OUTPUT #2

What is the packet’s destination: 400
Now starting delivery...

Starting in pipe #0
Passing into switch #0
Passing into pipe #2
Passing into switch #3
Passing into pipe #4
Passing into switch #4
Passing into pipe #6
Arrived at correct destination 400
OUTPUT #3

What is the packet’s destination: 700
Now starting delivery...

Starting in pipe #0
Passing into switch #0
Passing into pipe #2
Passing into switch #3
Passing into pipe #5
Passing into switch #5
Passing into pipe #9
Passing into switch #6
Passing into pipe #10
Arrived at correct destination 700

OUTPUT #4

What is the packet's destination: 900
Now starting delivery...

Starting in pipe #0
Passing into switch #0
Passing into pipe #2
Passing into switch #3
Passing into pipe #5
Passing into switch #5
Passing into pipe #9
Passing into switch #6
Passing into pipe #11
Passing into switch #7
Passing into pipe #12
Arrived at correct destination 900
Data Flow Diagram
Example

Figure 1
Structure Chart Example

Figure 2
Chasm Between
Structured Analysis
and Design

Figure 3
## OOD Employee Example

<table>
<thead>
<tr>
<th>Employee</th>
<th>id</th>
<th>name</th>
<th>wage</th>
<th>employee</th>
<th>print pay</th>
</tr>
</thead>
</table>

*Figure 4*
Structured Design
Unit Test Coverage

Figure 5
Integration Test Coverage

Program 1
Program 2
Program 3
Program 4
Program 5

Functional Tests

Structural Tests

Integration Tests

Figure 6
Functional Testing
(Black Box)

Figure 7
OOD Unit Test Coverage

Figure 8
OOD Student
Example

Figure 9
Private Method
Roadmap

These are Student class
Public methods

Figure 10
Public Method
Roadmap

Figure 11
Overloaded Functions

Figure 12
Comparison Between Traditional Testing and OO Testing

Figure 13
Structured Design for Triangle Problem

Figure 14
Structured Solution
Structure Chart
for Triangle Problem

Figure 15
Structured Solution
Triangle Problem
DD Path Graph

Figure 16
OOD Solution
For Triangle Problem

Figure 17
Triangle Problem
Public Method
Roadmap

Main

Create Side Object

<s, length>

Return Side

<s>

Create Triangle Object

<type, s1,s2,s3>

Return Type

<type,s1,s2,s3>

<3,4,5>

s1,s2,s3

<2>

side

Main

Main

Figure 18
OOD Design
Triangle Problem
DD Path Graph for
Main

Figure 19
OOD Design
Triangle Problem
DD Path Graph for Method typeoftriangle()
Packet Delivery
Map

Figure 21
Data Flow Diagram for Packet Problem

Figure 22
Structure Chart for Packet Problem

Figure 23
Structured Solution
Packet Problem
DD Path Graph

Figure 24
OOD Solution
to Packet Problem

Figure 25
Figure 26
Packet Problem
Public Method
Roadmap

Figure 27
Packet Problem
Public Method
Roadmap (cont.)

Create Switch Object

Main

Set Switcher

<2> choice

<2,6,7> switch_num
right
left

num, r, l

switch_set

Main

Return Connection

connection

Main

connection

<2> choice

Figure 27 (cont.)
Packet Problem
Public Method
Roadmap (cont.)

<2,6>
pipe_num
connection

Main

Create Pipe Object
num_connection

Return Connection
connection

Figure 27 (cont.)


[Cox] Cox, Brad J. "There is a Silver Bullet: A Software Industrial Revolution Based on Reusable and Interchangeable Parts Will Alter the Software Universe". *Byte*. October 1990 pp 209-215.


