

CA314 – Object Oriented Analysis & Design - 10

File name: CA314_Section_10_Ver01

Author: L Tuohey

No. of pages: 17

Table of Contents

10. Software Verification, especially testing 3

10.1 Summary of general SW Testing Concepts 3

 10.1.1. Basic Concepts..... 3

 10.1.1.1 What should be in a test case? 3

 10.1.1.2 Behaviour (Function) versus Structure 3

 10.1.1.3 Identifying test cases..... 4

 10.1.2 FUNCTIONAL TESTING techniques 5

 10.1.2.1 Boundary Value and Related Methods 6

 10.1.2.1.1 Boundary Value Analysis 6

 10.1.2.1.2 Robustness Tests 6

 10.1.2.1.3 Worst Case Testing 6

 10.1.2.1.4 Special Value Testing 7

 10.1.2.1.5 Random Testing 7

 10.1.2.1.6 Concluding notes on boundary value analysis etc 7

 10.1.2.2 Equivalence Class Testing 7

 10.1.2.3 Decision Table-Based Testing 8

 10.1.3 STRUCTURAL TESTING – Minimal outline..... 10

10.2 A non-test Verification Approach: McCabe Complexity 11

10.3 Some issues in OO Testing – Outline list 12

 10.3.1 GENERAL..... 13

 10.3.2 LEVELS OF TESTING 13

 10.3.3 INHERITANCE..... 13

 10.3.4 POLYMORPHISM..... 13

 10.3.5 DATA FLOW TESTING 14

 10.3.6 Use of Interaction Diagrams 14

 10.3.7 Use case model 14

10.4 Example O-O Unit and Integration level testing 14

 10.4.1 The example system (o-oCalendar, from Jorgenson) 14

 10.4.2 o-oCalendar example: Features to be verified - Summary 16

10. Software Verification, especially testing

10.1 Summary of general SW Testing Concepts¹

10.1.1. Basic Concepts

10.1.1.1 What should be in a test case?

Headings for the information to be provided in a test case are

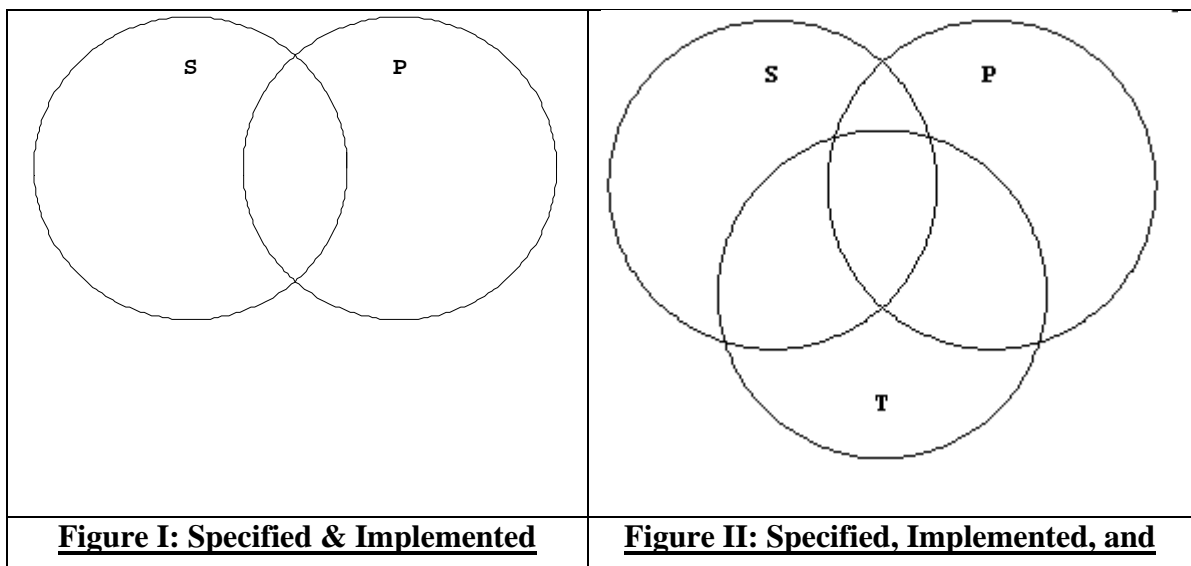
Test Case ID			
Purpose			
Preconditions			
Inputs			
Expected Outputs			
Postconditions			
Execution History			
Date	Result	Version	Run By
•	•	•	•
•	•	•	•

e.g. to verify a SW requirement

Identified by some testing method

10.1.1.2 Behaviour (Function) versus Structure

Structural & Behaviour views focus, respectively, on what software is and does, respectively.



¹ Notes in this section draw heavily on “Software Testing, a craftsman’s approach”, Jorgensen, P.C., CRC Press, 2002.

<u>Behaviours</u>	<u>Tested Behaviours</u>
S = Specification (expected), P = Program (observed), T = Tested	

Figure I is intended to highlight issues that a program tester should be aware of in devising tests. The correct portion is where sets **S** and **P** intersect, but

What if certain specified behaviours have not been programmed?

What if some implemented (i.e. programmed) behaviours have not been specified?

Figure II has an additional set **T**, representing the set of test cases. There are several possibilities:

a. Specified behaviours that are not tested	<i>Part of S not in T</i>
b. Specified behaviours that are tested	<i>Part of S that is also in T</i>
c. Test cases that correspond to unspecified behaviours	<i>Part of T not in S</i>
d. Programmed behaviours that are not tested	<i>Part of P not in T</i>
e. Programmed behaviours that are tested	<i>Part of P that is also in T</i>
f. Test cases that correspond to unprogrammed behaviours	<i>Part of T not in P</i>

Some questions that arise:

What can a tester do to maximise the region where **S**, **P** and **T** all intersect? [Not just up to the tester?]

How are the test cases in **T** identified?

10.1.1.3 Identifying test cases

Functional testing is based on the view that any program can be considered to be a function that maps values from its **input domain** to values in its **output range**. This leads to the term **black box testing**, in which the content (implementation) of a black box is not known, and the function of the black box is understood completely in terms of its inputs and outputs. In this functional approach to test case identification, the only information used is the software specification. In Figure II, set **T** would be contained within set **S**.

In **Structural testing**, the implementation is known and used to identify test cases. Hence, it is sometimes called **white box testing**. In Figure II, set **T** would be contained within **P**.

In fact, a combination of functional and structural testing is usually the best approach.

The following is a summary some of the issues:

Advantages of functional approach	(i) test cases are not affected by changes of implementation (ii) test case development can proceed in parallel with implementation.
Common difficulties of functional approach	(i) Significant redundancies may exist between test cases (ii) May be gaps in that some of the software may be untested (iii) Will never uncover unspecified behaviour (<i>e.g. a virus!</i>)
Two different positions on structural testing!	(i) “This tool has been wasting tester’s time since the 1970s ...” (ii) “Branch coverage [arising in structural testing], if attained at the 85% or better level, tends to identify twice the number of defects that would have been found by ‘intuitive’ [functional] testing”
One disadvantage of structural approach	Hard to see how unprogrammed behaviours could be identified.
Characterisation of functional approach	Establishes confidence
Characterisation of structural approach	Seeks faults

10.1.2 FUNCTIONAL TESTING techniques

It was indicated above that “**Functional testing** is based on the view that any program can be considered to be a function that maps values from its **input domain** to values in its **output range**”. Very often, because it is more straightforward, this form of testing has focused on the input domain but, if possible, tests based on the output range should also be developed.

10.1.2.1 Boundary Value and Related Methods

10.1.2.1.1 Boundary Value Analysis

Boundary value analysis focuses on the boundary of the input space to identify test cases. The rationale behind boundary value testing is that errors tend to occur near the extremes of an input variable. Loop conditions, for example, may test for $<$ when they should test for \leq , and counters are often “off by one”.

Normally, for each input variable, there are 5 tests cases corresponding to

- minimum value (min)
- value just above minimum (min+)
- a nominal or typical value
- value just below maximum (max-)
- maximum value (max)

If there are 2 input variables there will be a total of $4*2+1=9$ test cases (nominal is common). If there are n input variables there will be a total of $4*n+1$ test cases.

Strongly typed languages permit explicit definition of variable ranges but other languages, such as C, do not; boundary value testing is more appropriate for the latter.

10.1.2.1.2 Robustness Tests

At its simplest, robustness testing is an extension of boundary value analysis where we investigate what happens when the extrema are exceeded slightly.

More generally, the objective of robustness testing is to demonstrate the ability of software to respond to abnormal inputs and conditions. For example, for state transitions, test cases should be developed to provoke transitions that are not allowed by the software requirements.

10.1.2.1.3 Worst Case Testing

A limitation or criticism of boundary value analysis is that it only works well when the program to be tested is a function of several *independent* variables. However, there are often interesting dependencies between variables.

“Worst case analysis” focuses on what happens when more than one variable has an extreme value.

For example, if a program depends on 2 variables x and y then we have

(a) Boundary value analysis: $4*2+1=9$ test cases (see above)

(b) Worst case analysis: $5 \times 5 = 25$ test cases

[{(min_x, min_y), (min_x, min_{+y}), (min_x, nominal_y), (min_x, max_{-y}), (min_x, max_y), etc}]

If there were n variables then a total of 5^n tests cases would be needed for the worst case analysis, compared to $4n+1$ for the boundary value analysis.

10.1.2.1.4 Special Value Testing

This “ad hoc” approach occurs when testers use their domain knowledge or experience with similar programs or information about “soft spots” to devise test cases – while subjective, can be fruitful.

10.1.2.1.5 Random Testing

The basic idea is to use a random number generator to pick out test cases rather than using a deterministic approach, particularly the boundary value approach. While this avoids a bias in testing it raises the question of “how many random cases are sufficient?”.

10.1.2.1.6 Concluding notes on boundary value analysis etc

Each of the foregoing can, in principle, be applied to the output range of a program.

Another form of output-based testing is to devise test cases to check that error messages are generated when appropriate and are not falsely generated.

Domain analysis can also be used for internal variables (e.g. loop control variables, indices, etc). Of course, this brings up the issue of “visibility” of such variables to the testing software.

10.1.2.2 Equivalence Class Testing

Formally, in set theory, a *partition* of a set means a collection of mutually disjoint subsets whose union is the entire set. Equivalence classes form such a partition.

From a testing point of view there are 2 points to note:

- a) “Entire set is represented” provides a form of completeness in the test case.
- b) “Equivalence classes are disjoint” ensures a form of non-redundancy in test cases.

Example: Suppose a function F depends on two variables x and y and is implemented as a program. Suppose that input variables x and y have the following boundaries, and (meaningful) intervals within the boundaries:

$a \leq x \leq d$, with intervals $[a, b)$, $[b, c)$, $[c, d]$

$e \leq y \leq g$, with intervals $[e, f)$, $[f, g]$.

Note: “[“ denotes a closed (i.e. included) end-point, “(“ denotes an open (i.e. excluded) endpoint.

For completeness, we note the invalid intervals:

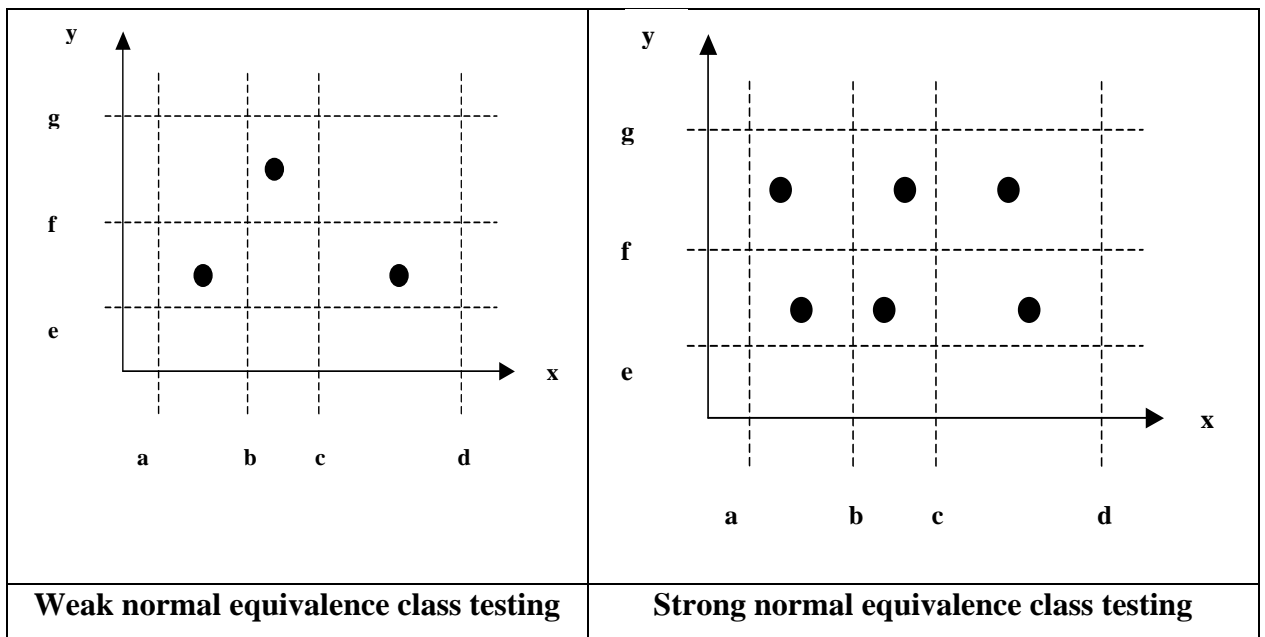
$$x < a \text{ or } (\infty, a), x > d \text{ or } (d, \infty)$$

$$y < e \text{ or } (\infty, e), y > g \text{ or } (g, \infty)$$

It is clear that the intervals are in fact equivalence classes – for example,

$$(\infty, a), [a, b), [b, c), [c, d], (d, \infty)$$

divides the whole x axis into 5 disjoint intervals.



10.1.2.3 Decision Table-Based Testing

Decision tables have the general form,

CASE	ID1	ID2	---
Condition 1			
Condition 2			
:			
Condition M			
Action 1			
Action 2			
:			
Action N			

The top row provides a reference for each column of the table. The "conditions" rows specify the variables or conditions to be evaluated while the lower rows specify the corresponding actions to be taken. A table entry of "x" represents "don't care", while

“u” represents “unchanged” (w.r.t. value before the table is executed). For example, the pseudocode

```
if A then
  Compute x
elseif B then
  Compute y
else
  Compute x
  Compute y
end if
```

would be represented by the decision table,

CASE	1	2	3
A	T	F	F
B	x	T	F
Compute x	T	F	T
Compute y	F	T	T

It is believed that, in the case of complex logic at least, decision tables provide a clearer less error prone representation than pseudocode or activity diagrams (or equivalent). Moreover, decision tables do *not* require that arbitrary and unnecessary sequential constraints be inferred.

Decision tables can be formed by analysis of a requirement specification, a use case description, pseudocode (as above) or actual source code. In fact, it could be decided at the outset to express requirements with the help of decision tables as illustrated by the following example from an actual project:

S-1.6.1-2

Eclipse_Imminent_Logic_T[IN:**Eclipse_Imminent**, INOUT: **IMU_On**[roll]] and **Eclipse_Imminent_Logic_F**[IN:**Eclipse_Imminent**, INOUT: **IMU_On**[roll]] are common functions that shall be used to switch the roll gyro on and off depending on whether or not an eclipse is imminent. They are, respectively, defined by the decision tables,

Eclipse_Imminent_Logic_T:

Case	1	2	3
Eclipse_Imminent	T	T	F
IMU_On [roll] (in)	T	F	x
IMU_On [roll] (out)	T	T	u
IMUturn on [roll]	F	T	F

(Remark: The value of “**IMU_On**[roll] (out)” in case 2 is the nominally expected output but whether it is achieved depends on the result of executing **IMUturn on**[roll])

and **Eclipse_Imminent_Logic_F:**

Case	1	2	3
Eclipse_Imminent	F	F	T
IMU_On [roll] (in)	T	F	x
IMU_On [roll] (out)	F	F	u
IMUturn off [roll]	T	F	F

TRACE: F3.1.2.1.2.3.4-12(part of sunlit), F3.1.2.1.2.3.5-1(part of sunlit),#

In terms of testing, the basic approach is that there will be a test case for each decision table column. Thus, there would be three test cases for the pseudocode example above.

10.1.3 STRUCTURAL TESTING – Minimal outline

This is presented very briefly.

According to Jorgensen (see above) “the distinguishing characteristic of structural testing methods is that they are all based on the source code of the program tested, and not on the definition”.

In fact, it is probably useful to bear in mind two aspects:

- a) Derivation of test cases based on analysis of the software design, whether at high level (e.g. object-object interactions described by UML sequence diagrams) or detailed level (e.g. the pseudocode for an individual method)
- b) Determination of the level of structural test coverage achieved (which can in principle be found automatically via automatic tools).

There are various “structural test coverage” metrics including

Metric	Description of Coverage
C_0	Execution of every statement
C_1	Execution of every decision-to-decision (DD) path
C_{MCC}	Execution of every condition in every decision
C_{ik}	Execution of every program path that contains up to k repetitions of a loop
C_{stat}	Execution of a “statistically significant” fraction of paths

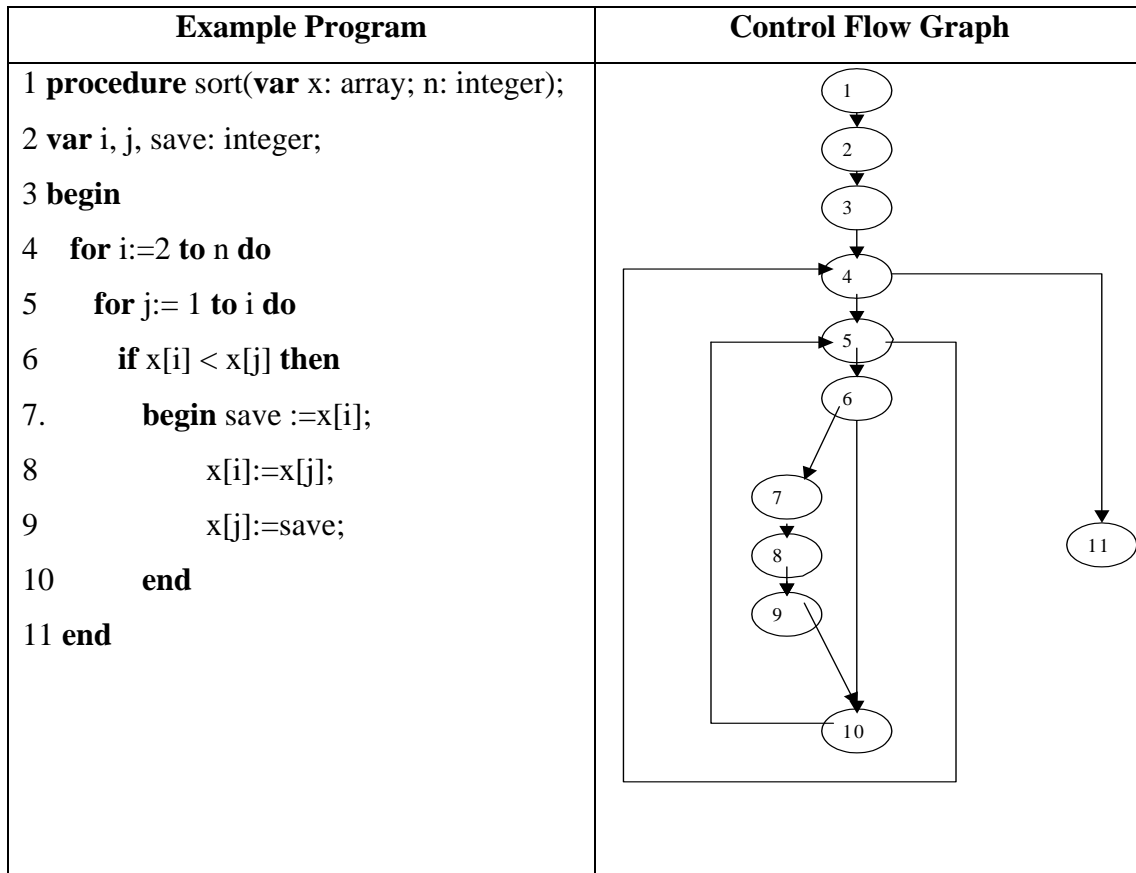
Particularly in terms of integration, it should be an objective to achieve adequate test coverage of data and control coupling between code components (e.g. classes).

10.2 A non-test Verification Approach: McCabe Complexity

Among software verification techniques, the importance of peer reviews, including particularly code inspections, is pointed out. There are various static analysis methods also, for which automated tools are available. Commonly, such methods involve calculation of a metric which is a measure of some aspect of an item’s quality.

An example is the calculation of McCabe’s “Complexity”. Note that, in this context, “Complexity” relates to attributes of software that affect the effort needed to construct or change it. [It has nothing to do with *computational complexity*].

- Illustration²:



- **Definition:** $CV = e - n + p + 1$, where

	CV	e	n	p
No. of	decisions + 1	edges	nodes	connected components

For the example, $CV = 13 - 11 + 1 + 1 = 4$

- **Use:** In design guidelines – for example, require $CV \leq 10$

In verification, focus on items with high CV

- **Alternative metric:** CV/LOC where LOC = lines of code

[focussed on *DENSITY* of decisions]

10.3 Some issues in OO Testing – Outline list

- The following list is not claimed to be complete.

- It is intended to give a summary indication of points to be aware of when testing object oriented software.

² Van Vliet, H., Software Engineering, principles and practice, Wiley 2004.

10.3.1 GENERAL

While one of the original hopes for object-oriented software was that objects could be re-used without additional testing, many people now think that OO software has potentially more severe testing problems than “traditional” software.

10.3.2 LEVELS OF TESTING

(i) Traditionally, methods (functions) and procedures were the “units” in software unit testing. In OO testing, it is not so clear – often classes are taken as units. These, of course, are bigger than traditional units. In fact, one has then two levels of integration testing: (a) at unit test level one has “within class” integration and (b) integration testing between classes.

(ii) In a UML context, one often has a state diagram associated with “interesting” classes. Such a state diagram can be very useful as a basis for testing the class; essentially, one devises tests to exercise all the events and transitions of the state diagram. Nevertheless, it would, very probably, still be desirable to first test each method of the class separately.

(iii) In practice, some classes may be extremely large which may not be desirable at unit test level.

10.3.3 INHERITANCE

Although the choice of class as unit seems natural, the presence of inheritance complicates matters. If a class inherits attributes and/or operations from a super class then the desirable criterion that a unit should be compilable separately is lost. One remedy is to introduce the notion of flattened classes for testing purposes where a **flattened class** is the original class expanded to include all attributes and operations it inherits. However, the flattened class is not part of the final system which raises issues, including for configuration control.

10.3.4 POLYMORPHISM

The essence of polymorphism is that the same method applies to different objects. Ideally, one would want to prove that the method works for all possible arguments. But, in reality, may need to test for each type of argument.

10.3.5 DATA FLOW TESTING

Traditionally, this refers to forms of structural testing that focus on the points at which variables receive values and the points at which these values are used. This approach can be extended to data flow among object-oriented operations.

10.3.6 Use of Interaction Diagrams

UML sequence and collaboration diagrams can be very useful in defining integration tests between classes.

10.3.7 Use case model

We have already described how use cases may be a very sound approach for system level (function) testing.

10.4 Example O-O Unit and Integration level testing

10.4.1 The example system (o-oCalendar, from Jorgenson)

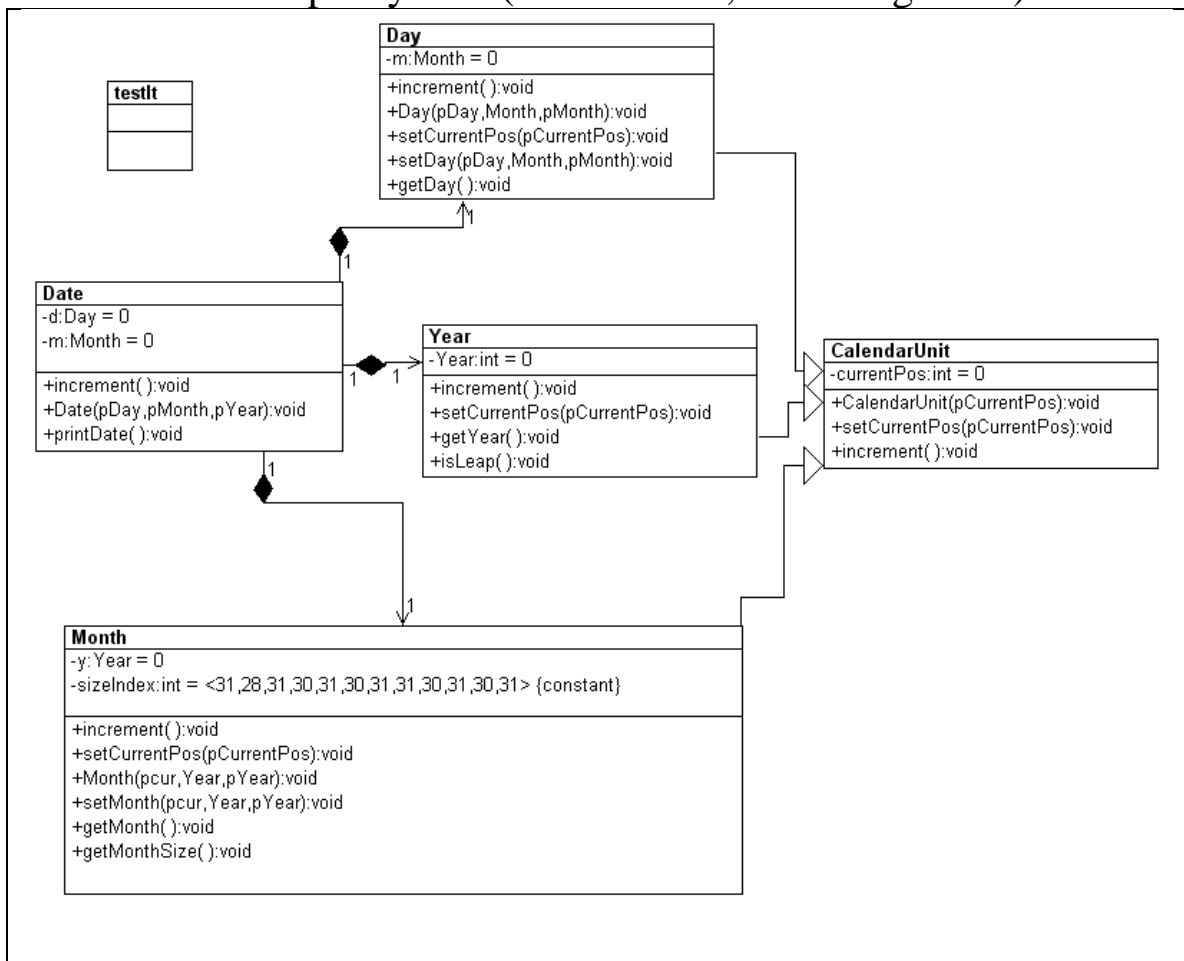


Figure 10.4.1-1: Class Model for o-oCalendar

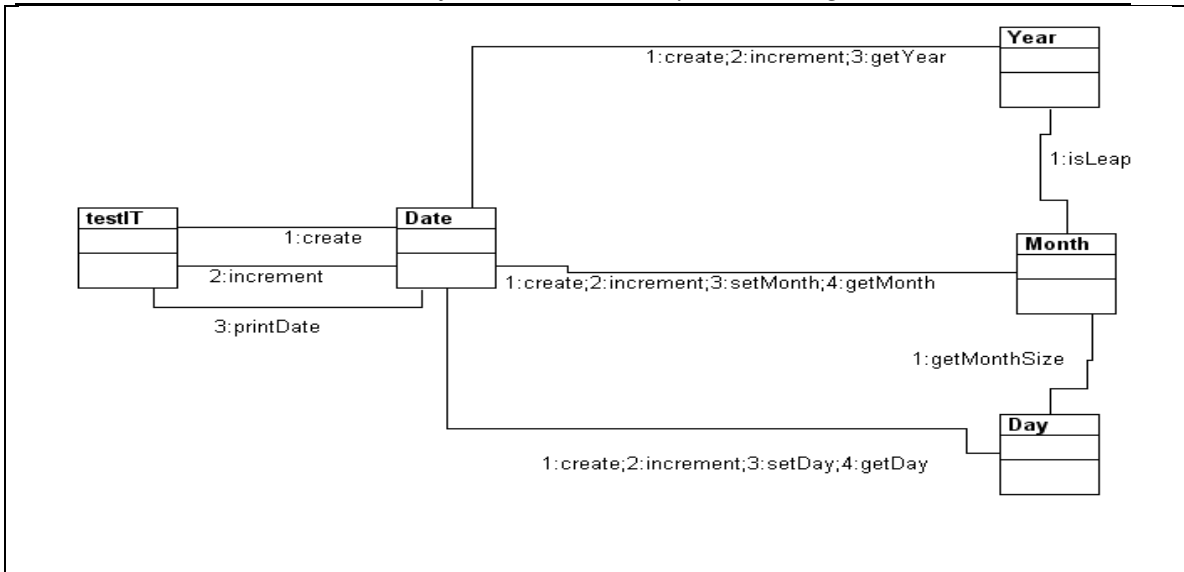


Figure 10.4.1-2: (“sort of”) Collaboration Diag showing (some of) message passing

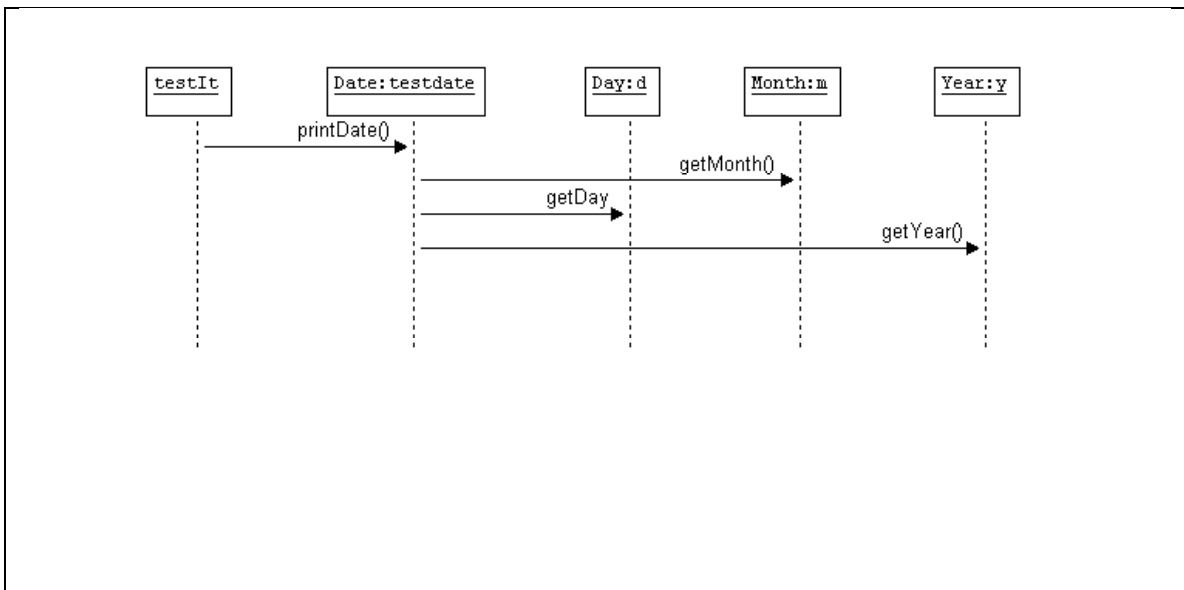


Figure 10.4.1-3: Sequence Diagram for printDate

10.4.2 o-oCalendar example: Features to be verified - Summary

Feature		Type	Method of Verification	
1	CalendarUnit	Class (abstract)	Unit tests jointly with each of <i>Year</i> , <i>Month</i> and <i>Day</i> . Cannot test an abstract class in isolation.	
2	Year	Class	Unit test each of its operations (with <i>CalendarUnit</i>), monitor “currentPos	
	Year.Year	Operation	Unit tests, driver only, no stubs needed	
	Year.getYear	Operation	Unit tests, driver only, no stubs needed	
	Year.increment	Operation	Unit tests, driver only, no stubs needed	
	Year.isleap	Operation	Unit tests, driver only, no stubs needed	
3	Month	Class	Unit test each of its operations (with <i>CalendarUnit</i>), monitor “currentPos & y	
	Month.Month	Operation	Unit tests, driver, stub of <i>Year</i> needed	alternatively assume <i>Year</i> is correct, after being already unit tested, and use it in testing <i>Month</i> . This is a form of <i>integration testing</i> which may be more efficient in terms of time & effort.
	Month.setMonth	Operation	Unit tests, driver, stub of <i>Year</i> needed	
	Month.getMonth	Operation	Unit tests, driver, stub of <i>Year</i> needed	
	Month.getMonthSize	Operation	Unit tests, driver, stub of <i>Year</i> needed	
	Month increment	Operation	Unit tests, driver, stub of <i>Year</i> needed	
4	Day	Class	Unit test each of its operations (with <i>CalendarUnit</i>), monitor “currentPos & m	
	Day.Day	Operation	Unit tests, driver, stub of <i>Month</i> needed	alternatively assume <i>Month</i> is correct, after being already unit tested (or integrated tested with <i>Year</i>), and use it in testing <i>Day</i> . This is a form of <i>integration testing</i> which may be more efficient in terms of time & effort.
	Day.setDay	Operation	Unit tests, driver, stub of <i>Month</i> needed	
	Day.getDay	Operation	Unit tests, driver, stub of <i>Month</i> needed	
	Day increment	Operation	Unit tests, driver, stub of <i>Month</i> needed	
5	Date	Class	Unit test each of its operations, monitor d, m, y	
	Date.Date, Date.increment, Date.printDate	Operations	Unit tests, driver (<i>testIt</i>), stubs of <i>Day</i> , <i>Month</i> , and <i>Year</i> needed	It’s probably better to check at least the basic logic of these operations, especially of <i>increment</i> , in an isolated unit test.
6	Date, Day, Month, Year, CalendarUnit	Classes	Integration tests of each operation of Date with all classes complete. Need a driver (<i>testIt</i>)	Should be able to re-use driver data of the corresponding unit test (see 5, above)

Note: Under 3 and 4, particularly, there is a suggestion to use previously tested items in testing another item. This and related aspects are discussed by Jorgenson (see points about “neighbourhood integration” and “expense of diagnostic precision”). Also, often, there is a decision to be made as to top-down or bottom-up integration.

If we had decided to unit test each class in isolation, then we would have the following additional test features to be tested at integration time: Month & Year, Day & Month, Day & Month & Year, Date & Year, Date and Month, Date and Day (with CalendarUnit being verified implicitly or as a side effect).