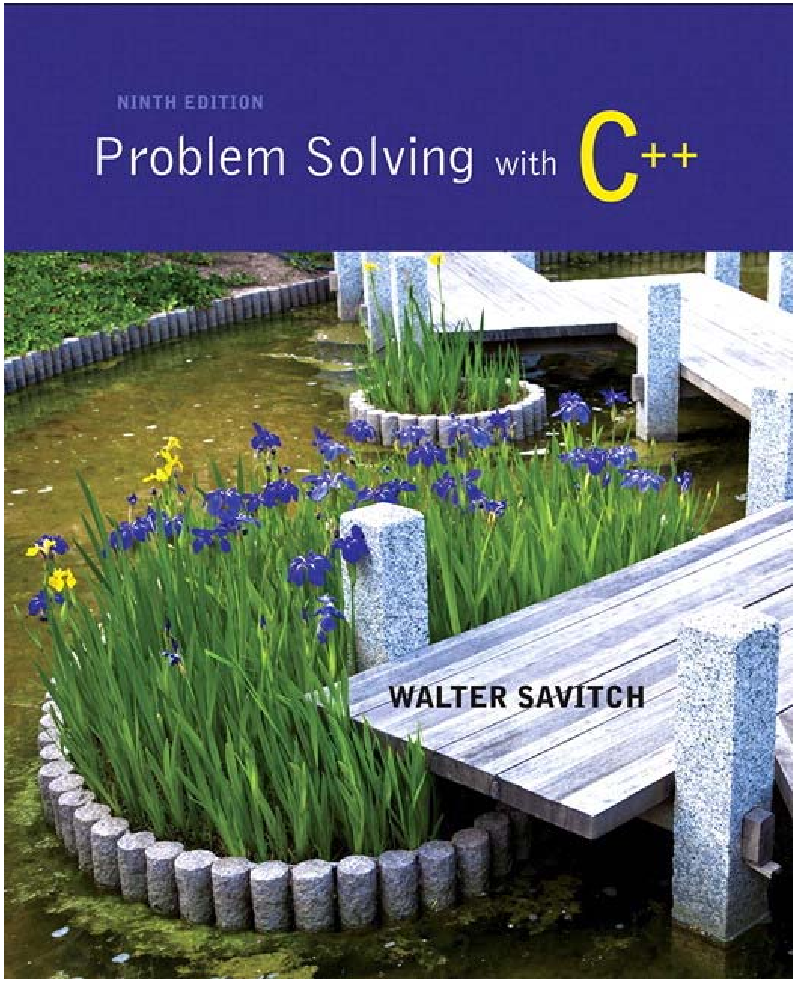
SFU SIMON FRASER UNIVERSITY
BURNABY

UML Diagrams and Software Testing

Instructor: Scott Kristjanson

CMPT 135

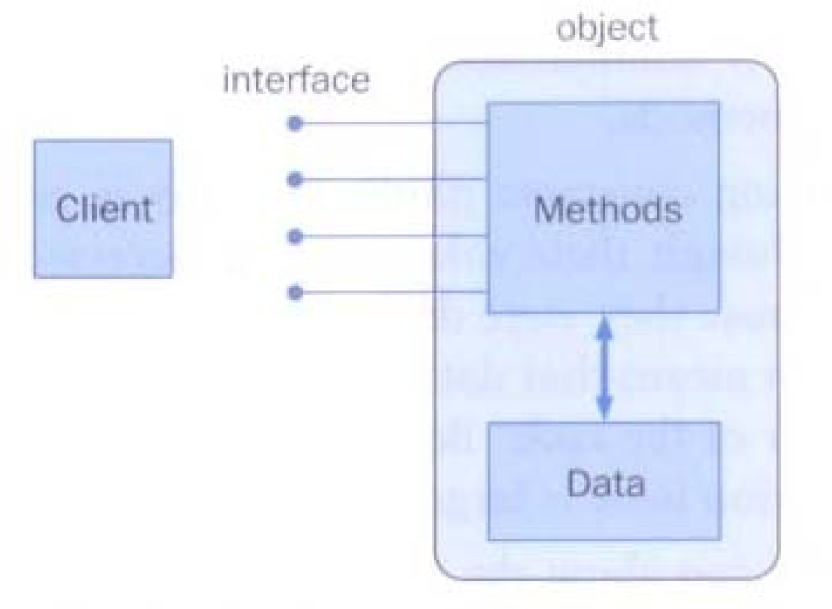
SFU Surrey, Spring 2016



Encapsulation

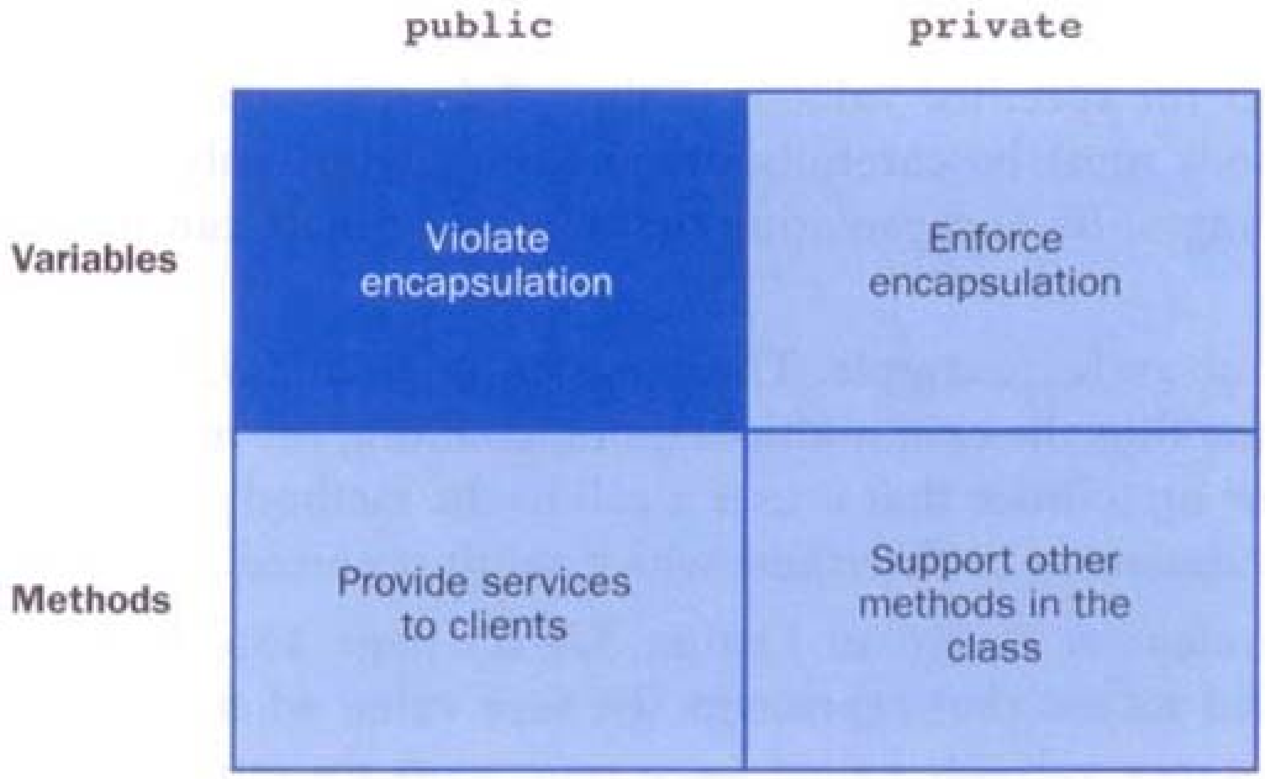
An encapsulated object can be thought of as a *black box* – its inner workings are hidden from the client

The client invokes the interface methods of the object, which manages the instance data



Slides based on Problem Solving with C++, 9th Edition, Walter Savitch

Visibility Modifiers



	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Method Declarations

A ***method declaration*** specifies the code that will be executed when the method is invoked (called)

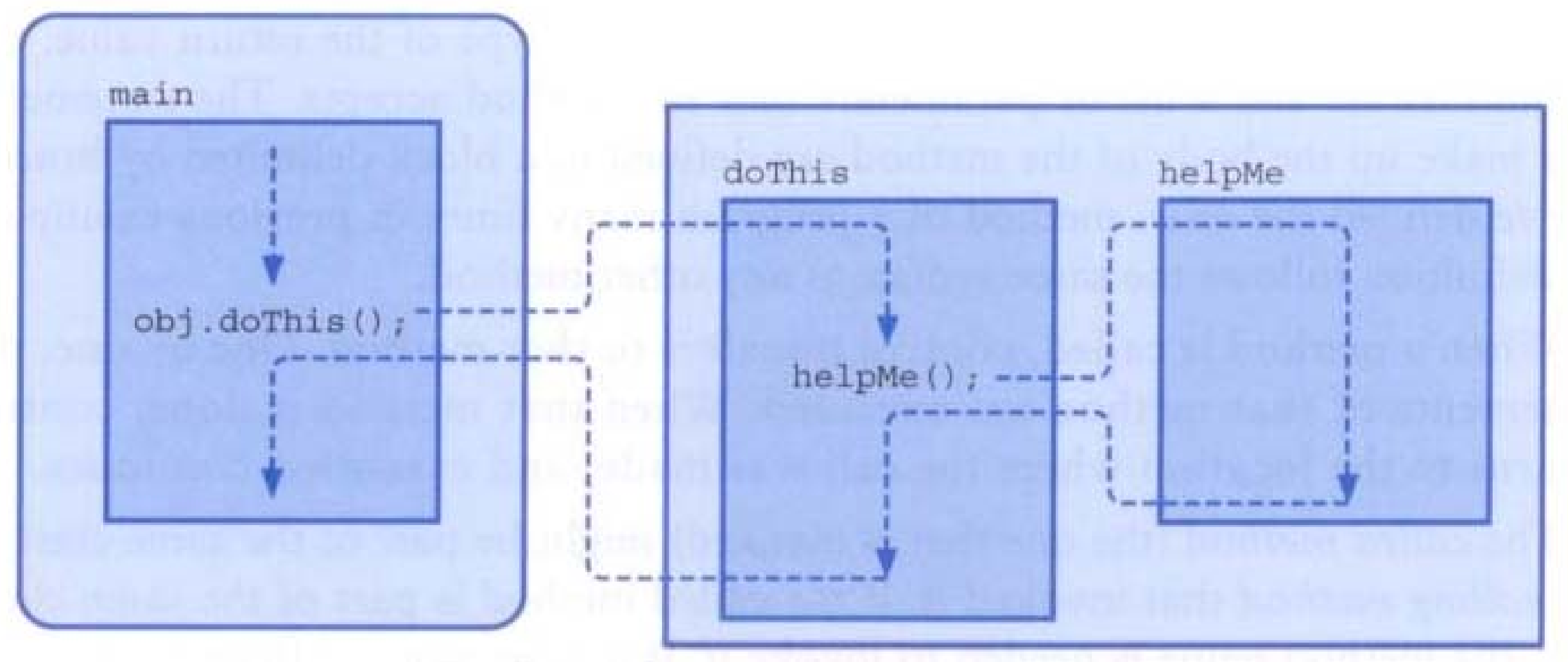
When a method is invoked, the flow of control jumps to the method and executes its code

When complete, the flow returns to the place where the method was called and continues

The invocation may or may not return a value, depending on how the method is defined

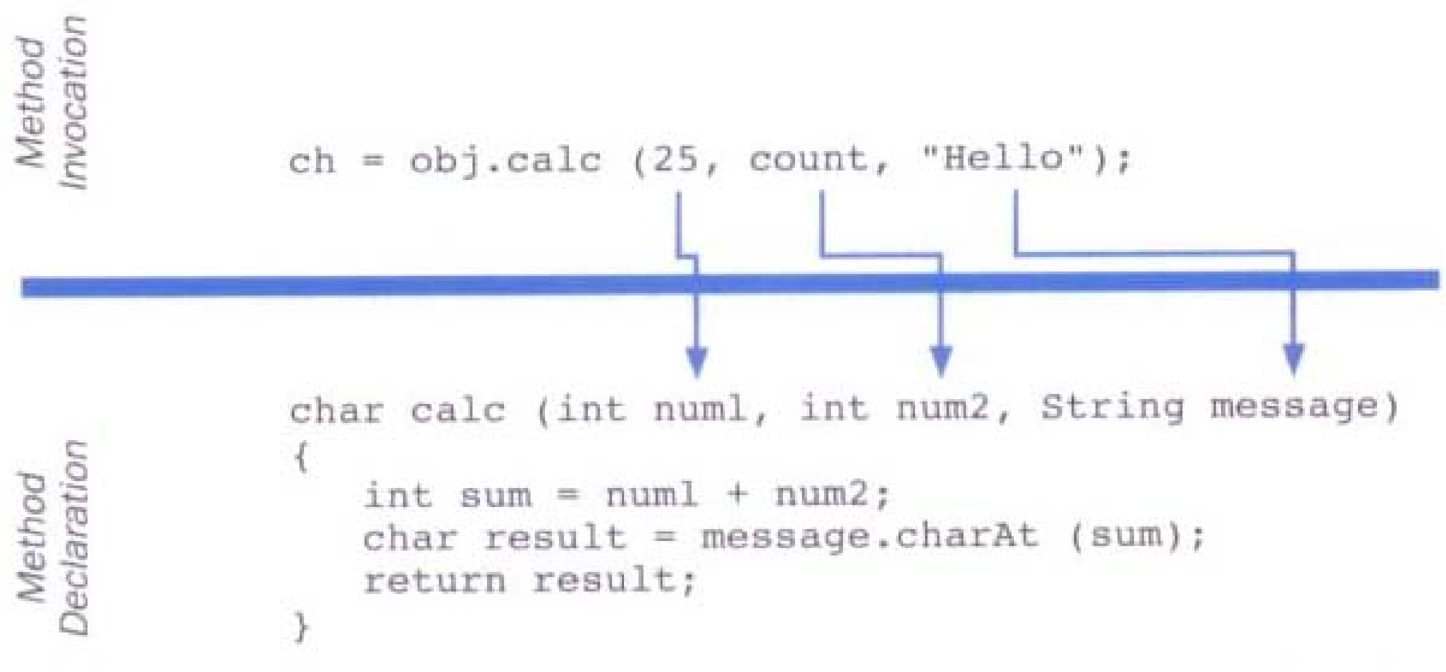
Methods

The flow of control through methods:



Pass by Value Parameters

When a method is called, the *actual parameters* in the invocation are copied into the *formal parameters* in the method header



```
ch = obj.calc (25, count, "Hello");
```

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);
   return result;
}
```

Class Relationships

Classes in a software system can have various types of relationships:

Three of the most common relationships:

- Dependency: A *uses* B
- Aggregation: A *has-a* B
- Inheritance : A *is-a* B

UML Diagrams

UML stands for the *Unified Modeling Language*

UML diagrams show relationships among classes and objects

A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)

Lines between classes represent *associations*

A solid arrow shows that one class *uses* the other (calls its methods)

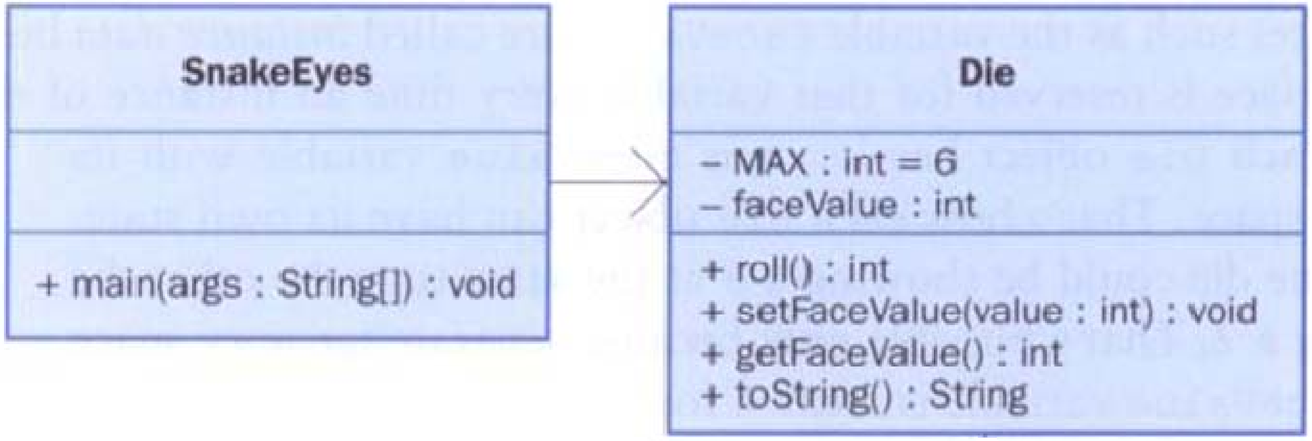
Dependency (A uses B)

A *dependency* exists when one class relies on another in some way, usually by invoking the methods of the other

We don't want numerous or complex dependencies among classes

Nor do we want complex classes that don't depend on others

A good design strikes the right balance



Aggregation (A has-a B)

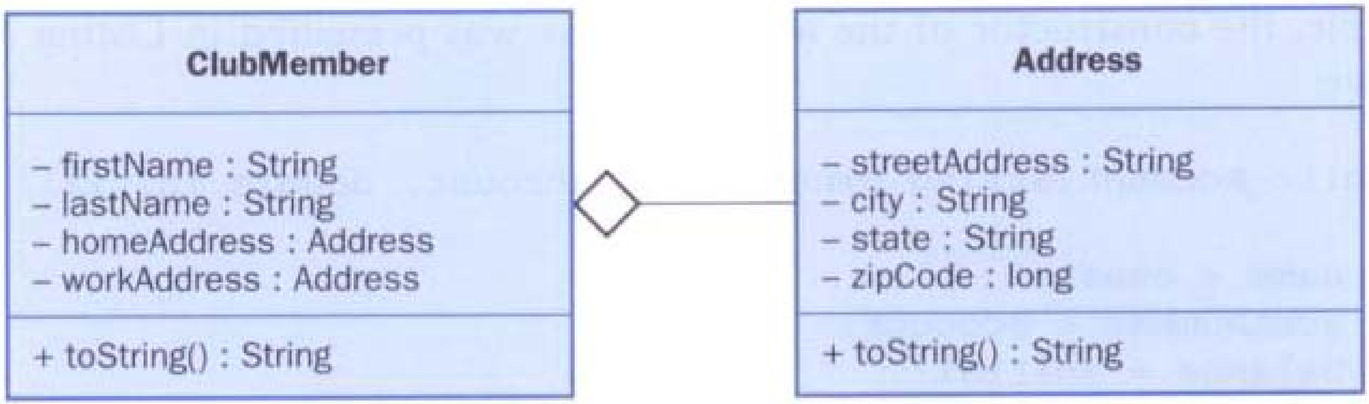
An *aggregate* is an object that is made up of other objects

Therefore aggregation is a *has-a* relationship

- A car *has a* chassis

The aggregate object is defined in part by the objects that make it up

This is a special kind of dependency – the aggregate relies on the objects that compose it



ClubMember

– firstName : String
– lastName : String
– homeAddress : Address
– workAddress : Address

+ toString() : String

Address

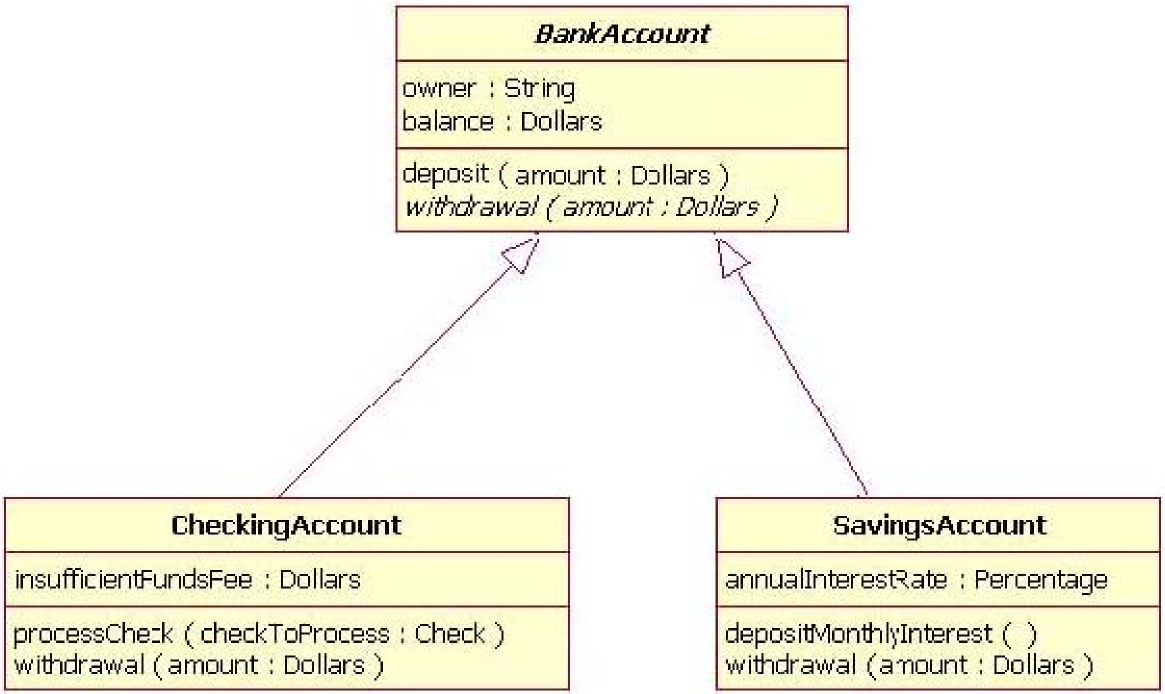
– streetAddress : String
– city : String
– state : String
– zipCode : long

+ toString() : String

Inheritance (A is-a B)

Derived class **IS-A** more specific version of its **Base** Class

Inheritance is indicated by a solid line with a closed, unfilled arrowhead pointing at the super class


BankAccount
owner : String
balance : Dollars
deposit (amount : Dollars)
withdrawal (amount : Dollars)
CheckingAccount
insufficientFundsFee : Dollars
processCheck (checkToProcess : Check)
withdrawal (amount : Dollars)
SavingsAccount
annualInterestRate : Percentage
depositMonthlyInterest ( )
withdrawal (amount : Dollars)

Software Testing - Design and Code Reviews

Review – meeting of several people designed to examine a design document or section of code

Presenting a design or code causes us to think carefully about our work and allows others to provide suggestions

Goal of a review is to identify problems

Design review should determine if the system requirements are satisfied

Defect Testing

Testing is also referred to as *defect testing*

Though we don’t want to have errors, they most certainly exist

A *test case* is a set of inputs, user actions, or initial conditions, and the expected output

It is not normally feasible to create test cases for all possible inputs

It is also not normally necessary to test every single situation

Defect Testing

Two approaches to defect testing

- *black-box*: treats the thing being tested as a black box
  - Test cases are developed without regard to the internal workings
  - Input data often selected by defining *equivalence categories* – collection of inputs that are expected to produce similar outputs
  - Example: input to a method that computes the square root can be divided into two categories: negative and non-negative

Defect Testing

Two approaches to defect testing

- *white-box*: exercises the internal structure and implementation of a method.
  - Test cases are based on the logic of the code under test.
  - Goal is to ensure that every path through a program is executed at least once
  - *Statement coverage* testing – test that maps the possible paths through the code and ensures that the test case causes every path to be executed

Other Testing Types

Unit Testing – creates a test case for each module of code that been authored. The goal is to ensure correctness of individual methods

Integration Testing – modules that were individually tested are now tested as a collection. This form of testing looks at the larger picture and determines if bugs are present when modules are brought together

System Testing – seeks to test the entire software system and how it adheres to the requirements (also known as *alpha* or *beta* tests)

Regression Testing – seeks to verify that recent changes have not broken existing functionality. Typically a small subset of test cases designed to cover key areas of functionality.

Test Driven Development

Developers should write test cases as they develop their source code

Some developers have adopted a style known as *test driven development*

- test cases are written first
- only enough source code is implemented such that the test case will pass

Test Driven Development

Test Driven Development Sequence

1. Create a test case that tests a specific method that has yet to be completed
2. Execute all of the tests cases present and verify that all test cases will pass except for the most recently implemented test case
3. Develop the method that the test case targets so that the test case will pass without errors
4. Re-execute all of the test cases and verify that every test case passes, including the most recently created test case
5. Clean up the code to eliminate redundant portions (refactoring)
6. Repeat the process starting with Step #1

Debugging

Debugging is the act of locating and correcting run-time and logic errors in programs

Errors can be located in programs in a number of ways

- you may notice a run-time error (program termination)
- you may notice a logic error during execution

Through rigorous testing, we hope to discover all possible errors. However, typically a few errors slip through into the final program

A *debugger* is a software application that aids us in our debugging efforts

Simple Debugging using `cout`

Simple debugging during execution can involve the use of strategic **`cout`** statements indicating

- the value of variables and the state of objects at various locations in the code
- the path of execution, usually performed through a series of “it got here” statements

Consider the case of calling a method

- it may be useful to print the value of each parameter after the method starts
- this is particularly helpful with recursive methods

Debugging Concepts

Formal debuggers generally allow us to

- set one or more *breakpoints* in the program. This allows to pause the program at a given point
- print the value of a variable or object
- step into or over a method
- execute the next single statement
- resume execution of the program

References:

1. Walter Savitch, Problem Solving with C++. Pearson, 9th Edition, 2014, ISBN 978-0-13-359174-3
2. T. DeMarco, *Structured Analysis and System Specification*, 1979, ISBN 978-0-13-8543808
3. T DeMarco, Structured Analysis, Structural Design and Materials Conference 2001, Software Pioneers, Eds.: M. Broy, E. Denert, Springer 2002 http://cs.txstate.edu/~rp31/papersSP/TDMSpringer2002.pdf
4. Stevens, W., G. Meyers, and L. Constantine, *Structured Design*, IBM Systems Journal, Vol 13, No 2. 1974
5. Fairley, Richard E., *Software Engineering Concepts*, McGraw-Hill, 1985, ISBN 0-07-019902-7
6. IBM DeveloperWorks website, *UML basics: The class diagram*, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/