

Web-Based Software Testing (of Systems and Sub-Systems)

Workshop

TARDEC – January 16, 2002 (13:00-17:00 Hrs)

Gautam B. Singh, Ph.D.

**Associate Professor of Computer Science & Engineering
Oakland University, Rochester, MI**

Verification & Validation (V&V)

Verification

“Are we building the product correctly”

Validation

“Are we building the right product”



Verification & Validation (V&V)

- Software Verification Plan:
 - Requirements, Design/Architecture, and Source Code Verification
- Activities
 - Walkthroughs/ Inspections (*Verification*)
 - Structure Tests (*Verification*)
 - Functional Tests (*Validation*)
 - Performance Tests (*Validation*)
 - Stress Tests (*Validation*)
 - Source Code Structure Tests (*Verification*)
 - Static Analysis
 - Code Level Metrics



Testing Process Perspective

- V & V Activities integrated into Software Development Process.
 - Requirements
 - Architecture
 - Design
 - Coding



Requirement Verification:

- Determine formal/informal verification strategy to use.
- Determine if the requirements are adequate.
- Begin development of test data



Design Verification

- Determine if the design is consistent with requirements.
- Determine if design is adequate.
- Generate functional (black box) and structural (white box) test data.

Construction/Code Verification

- Consistency with Design.
- Is the implementation adequate?
- More functional and structural test data is generated.

Definition of a Program

- **Program**: An object that may be conceptually or actually executed.
- A program is a representation of a function - relationship of input (*domain element*) to output element (*range element*).
- **Testing** is used to ensure that the program faithfully realizes this function.
- Design and requirement specifications can be *conceptually* executed...CONCLUSION?




Five Stages of Testing Process

1. Obtain a valid (invalid) value from (outside) the functional domain
2. Determine the expected behavior
3. Execute the program and observe its behavior
4. Test succeeds if the expected and observed behavior agree.
5. A bug is uncovered if they do not agree.



Defining Expected Behavior

- Is the hardest stage of the testing process!
- Ad-hoc methods utilized: hand-calculation, simulation, and alternate solutions.
- Static Vs Dynamic Testing
Dynamic test results utilize actual test results; static methods based on program analysis.
- **Exhaustive testing** requires observing output for the entire set of functional domain values.
- Automated generation of test cases that are consistent, valid and complete is not possible.



Functional Black Box Testing

- Boundary Value Analysis: Partition the program domain so that input data sets that span the partitions can be defined.
- Design Based Functional Testing: The requirements are subdivided into design blocks that implement these. Testing targets execution of sub-blocks.
- Cause Effect Graphing: A partitioning of the range is used to drive the testing process.



Structural Based Testing (White Box Testing)

- Coverage Based Testing: Control flow graph of the program is used for determining the effectiveness of testing.
- Complexity Based Testing: Analysis of the software complexity is used to drive the testing process.



Clean Room: Background

- Clean room approach increases the productivity and quality of the software.
- **Cleanroom**: A Modern approach.
- Traditional Approaches consider defects inevitable and defect removal is a part of development process.



Clean Room: Software

- Motto: Quick and Clean. ZERO DEFECT.
- Resource Allocation is better when time is spent on getting it right...
- As opposed to defect removal which is an inefficient activity that eats up resources.
- Clean Room teams (IBM and other organizations) have produced quality software.

Clean Room Quality

- Clean Room Software is based on foundations of formal methods.
- Typical software:
 - Unit Testing: 25-35 errors/kLOC
 - Functional Testing: 5 errors/kLOC
- Cleanroom:
 - 2.3 errors per kLOC

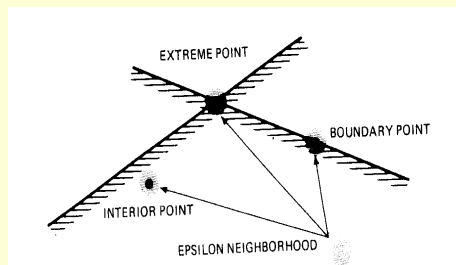
Incremental Development

- A development and certification pipeline.
- Correctness built into the development. Thus formal methods are often used.
- Software increments are developed and certified by small teams.
- System integration is continual.

Box Structures

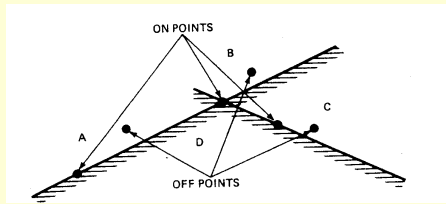
- All data is encapsulated in boxes.
- Black Box: Precise specification of external, user-visible behavior.
- State Box: Elements of stimulus history for achieving a desired behavior.
- Clear Box: Derived from state box and defines procedures that carry out state transitions.

Domain Partitioning



- Domain partitioning a powerful testing strategy.
- Definitions:
 - Interior Point
 - Boundary Point
 - Extreme Point

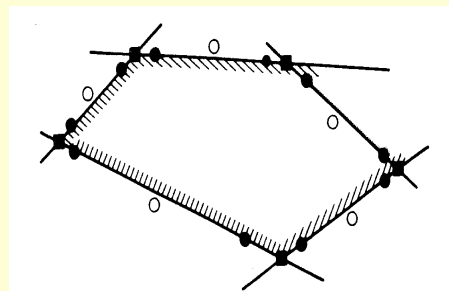
Closed and Open Domains



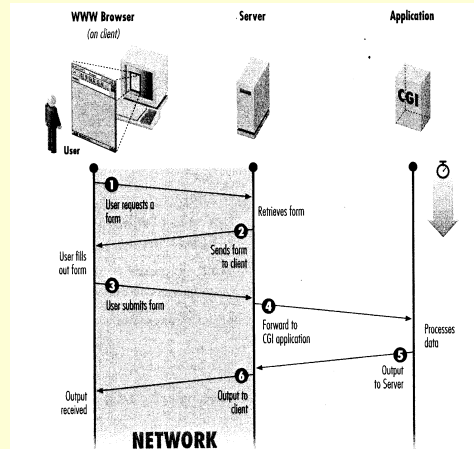
- Testing strategy is aimed at selecting the ON and the OFF points.
- OFF points are near domain boundary.
- OFF points in an open domain are on the inside of the boundary.

Example

- Strategy: Two ON points and one OFF points for each domain.
- Note that the OFF point for an Open domain is “inside” the boundary.
- Advisable to include the extreme points.



Domain Testing over Web



Web Form

- User Authentication
- Input specification

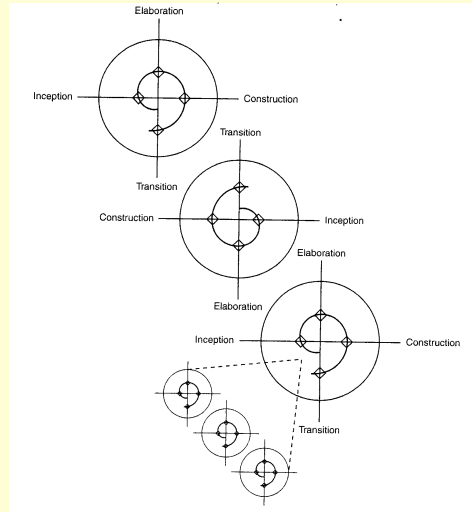
AUT (Application Under Test)

- program executed on the server using user specified data.

Domain Testing

- CGI.pm parses the user input.
- CGI invokes the background process on the server with user specified input.
- The output generated by the program is (possibly) displayed for user.
- Test vector (and user, scenario, etc. information) is logged into a database:
 - Statistical software quality estimation.

Iterative Incremental Process



- Functionality is gradually added to the software.
- Increments represent releases.
- Several iterations are done in each release.
- Underlying Philosophy: Continual Testing to reduce software failure rate.

Module Summary

- Domain partitioning a powerful strategy.
- Web based testing using CGI/Perl.
- Module 2:
 - CGI – Black box testing strategy
 - Example/Demo of a CGI-based testing.
- Module 3:
 - CORBA – White Box testing strategy
 - Example/Demo of CORBA-test bench.

CGI Programming

Module 2

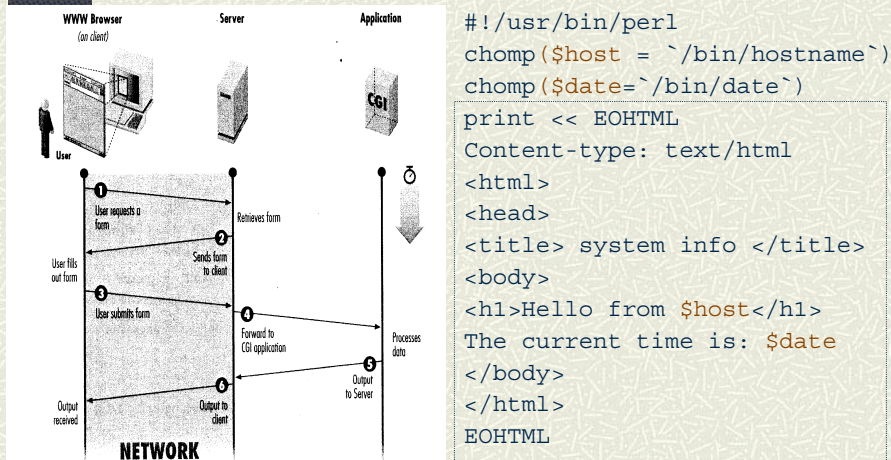
Why CGI?

- # Easy, Fast and a Lightweight method.
- # Will work with very little modification for the application.
- # Caution:
 - GUI for an application can not be tested using this method.
 - Partitioning will be required to separate the interface for processing logic.

What is CGI: Basic Architecture

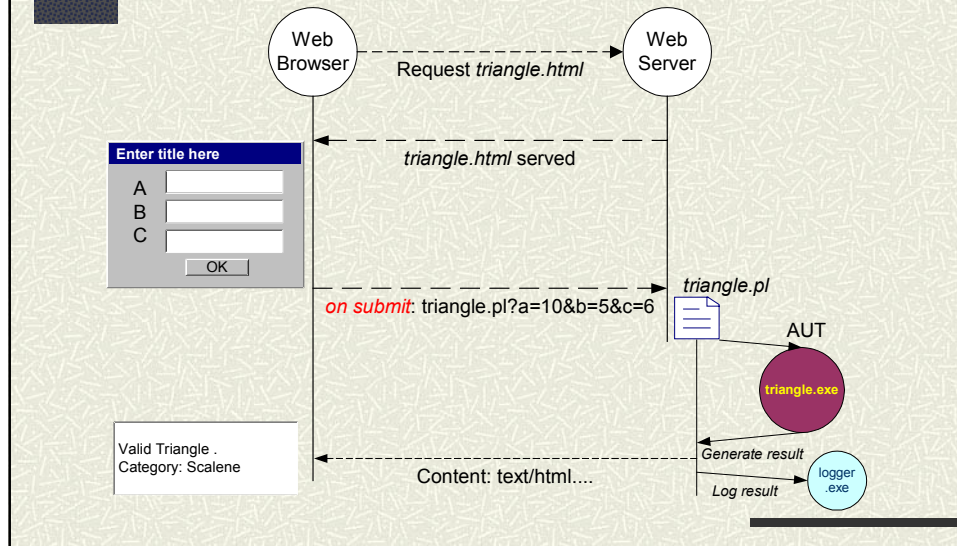
- # Data Entry Interface: CGI form
- # CGI.pm – CGI from Parser
- # Executive: CGI processor
- # Application: Not interactive execution
- # Logging: Input/Output stimulus logging

CGI: Common Gateway Interface



Here document

Control Flow: Object Interactions



Implementation

- # Form Definition: `triangle.html`
- # CGI file: `triangle.pl` (including a brief overview of perl, and `CGI.pm`)

Form Interface: triangle.html

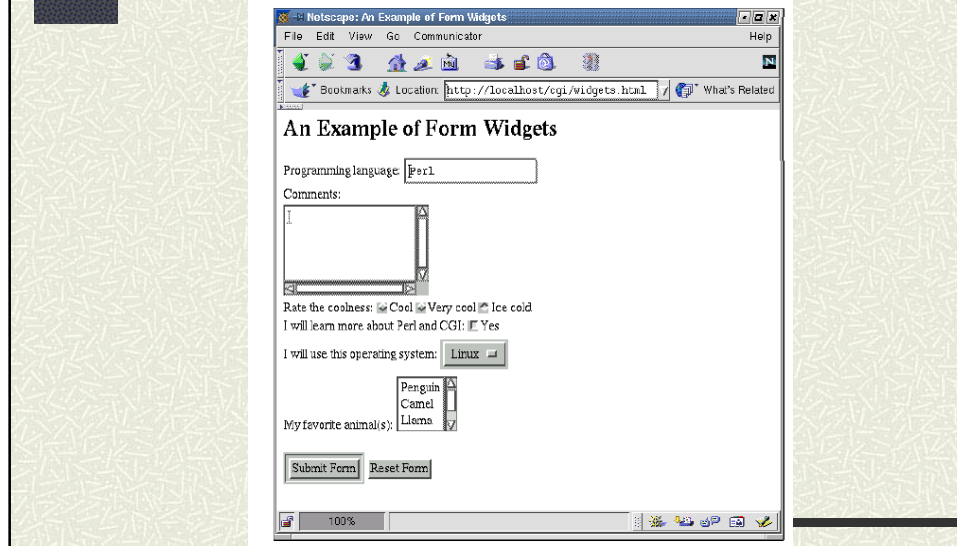
■ CGI forms allow many different components:

- Text boxes
- Combo boxes
- Radio buttons
- Option Groups
- Command Buttons

Example: Widget Specifications

```
<body bgcolor="#ffff">
<h1>An Example of Form Widgets</h1>
<form action="/cgi-bin/widgets.cgi" method="post">
Programming language: <input type="text" name="language" value="Perl">
<br>Comments:<br><textarea name="comments" cols="20" rows="5"></textarea><br>
Rate the coolness:
<input type="radio" name="coolness" value="cool"> cool
<input type="radio" name="coolness" value="very cool"> very cool
<input type="radio" name="coolness" value="ice cold" checked> ice cold<br>
I will learn more about Perl and CGI:<input type="checkbox" name="learnmore" value="yes" checked> Yes<br>
I will use this operating system:
<select name="operating_system" size="1">
<option>Linux</option>
<option>Solaris</option>
<option>HPUX</option>
</select> <br>
My favorite animal(s):
<select name="animal" size="3" multiple>
<option>Penguin</option>
<option>Camel</option>
<option>Llama</option>
<option>Panther</option>
</select> <br><br>
<input type="submit" value="Submit Form">
<input type="reset" value="Reset Form">
</form>
</body>
```

Example: Widget Display



Back to *triangle.html*

```
<html>
<body bgcolor="#ffffff">
<h1>Web Based Testing of Triangle package</h1>
<form action="/cgi-bin/triangle.pl" method="post">
```

```
Value for Side A: <input type="text" name="A" value="0">
```

```
Value for Side B: <input type="text" name="B" value="0">
```

```
Value for Side C: <input type="text" name="C" value="0">
```

```
<input type="submit" value="Submit Form">
```

```
<input type="reset" value="Reset Form">
```

```
</form>
```

```
</body>
```

```
</html>
```

POST and GET: Two methods for communicating data from form to the CGI module. POST is preferred for large data sets. GET sends the data as a part of the URL

HTML View

- # Note the FORM action defines what procedure needs to be invoked.
- # Function of the cgi-script *triangle.pl* is to act as a go between the input from and *triangle.exe* (I.e. the program being tested).
- # Configuration of APACHE – must have CGI enabled. [Apache Configuration File](#) (search for ScriptAlias)

PERL

- # Often called the “glue that holds the web”.
- # PERL has been ported to many operating systems.
- # Is extensible – adding modules is easy.
- # PERL is very powerful for text processing.
- # Meshes will into the services provided by the Operating System.

PERL – Basic Data Types

- # Scalar: \$a, \$b, \$c. **No type declaration.**
\$a = "Hello World";
\$b = 5;
- # Arrays: @x, @y.
@x = ('begin', 5, '+', 3);
@y = @x
- # Associative Arrays: %d
%d = (name => 'Joe',
age => 39);

Sample program

```
#!/C:/Perl/bin/perl.exe  
#hello.pl  
  
print "Content-type: text/plain\n";  
print "\n";  
print "hello, world!";
```

CGI.pm –

- # CGI.pm: Is a Perl Module specifically designed for use with the CGI scripts.
- # Perl Module implement OOP. A module is essentially a class that provides methods.
- # USE CGI ‘:standard’ : Passing the string ‘:standard’ to the USE pragma uses function that are standard for most browsers.
- # Also :standard option eliminates the need to create a CGI object .

Hello World – CGI.pm Style

```
#!/C:/Perl/bin/perl.exe
# simple.pl

use CGI ':standard';

print
  header(),
  start_html (-title => 'A simple page',
             -bgcolor => '#ffffff',
             -text => '#660099'),
  h1('Hello, world! again'),
  hr(),
  ' Time to go --',
  end_html();
```

If :standard is not defined ...

```
#!/C:/Perl/bin/perl.exe
# simple.pl

use CGI;
Sq = new CGI;

print
  $q->header(),
  $q->start_html (-title => 'A simple page',
                -bgcolor => '#ffffff',
                -text => '#660099'),
  $q->h1('Hello, world! again'),
  $q->hr(),
  ' Time to go --',
  $q->end_html();
```

Guts of the CGI program – triangle.pl

- # Gets the environment variables A, B and C that were set up by web server.
- # Executes the AUT with the parameters A, B and C. Redirects output to a result file.
- # Opens up the result file and displays it on the browser.

Reading the results.

- # All lines from the result file can be simply read:

Open RES, "result.txt"

@RES = <RES>;

- # The result file written to the browser by simply:

print @RES;

Show and Tell – triangle.pl

```
#!/C:/Perl/bin/Perl
use CGI 'standard';

$A = param ('A') || 10;
$B = param ('B') || 10;
$C = param ('C') || 10;
chomp($pwd = `pwd`);
$cmd = $pwd . '/../' . "AUT/Triangle/Debug/Triangle.exe SA SB SC >result.txt";
system ($cmd);
open (RES, "result.txt");
@res = <RES>;
print
    header(),
    start_html (-title => "Test Results..."),
    "The results are- ", br(),
    "@res",
    end_html;
```

The core program – AUT

- # Design: Domain validity – NOT implemented. (*Check each side > 0, and that triangle inequality holds*).
- # Testing conditions for classification.
- # Printing results
- # **For Web Testing:** The program does not need to be modified.

```

#include <iostream.h>
#include <stdlib.h>

int      main(int argc, char** argv)
{

    long      side[3];
    side[0] = atol(argv[1]);
    side[1] = atol(argv[2]);
    side[2] = atol(argv[3]);

    // Check for Equilateral
    if (side[0] == side[1] && side[0] == side[2])
    {
        cout << "It is an Equilateral Triangle " << endl;
        return 0;
    }

    // Check if any two sides are equal
    if (side[0] == side [1] || side[0] == side[2] || side[1] == side[2])
    {
        cout << "It is an Isoceless Triangle " << endl;
        return 1;
    }

    cout << "It is a Scalene Triangle " << endl;
    return 3;
}

```

Triangle.pl

- # Glues triangle.html with triangle.exe
- # Is the “Gateway” for sending the data from input to the application.
- # Other uses of the interface:
 - Logging activities...
 - Quality Assurance Database
 - Capture productivity of remote testers.
 - Generate statistical information on program correctness.

CORBA Based Testing

Module 3

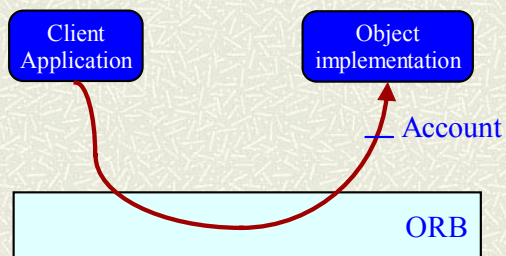
Why CORBA?

- # CORBA offers the ability to remotely run a set of functions defined in the interface file.
- # This allows us to remotely test individual functions for correctness.
- # Thus, a distributed execution of White Box testing is possible.

CORBA application design

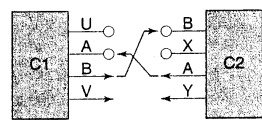
- # Interface declaration
 - Helps create the STUB and SKELETON
- # Server side programming
 - Implementation of the SKELETON
- # Client side programming
 - Invocation of functions defined in the STUB.
- # IIOP may be run over SSL for added security.

ORB

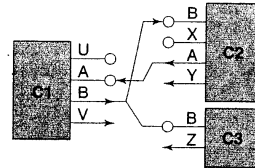


```
interface Account
{
    void deposit( in
unsigned long amount
);
    void withdraw( in
unsigned long amount
);
    long balance();
};
```

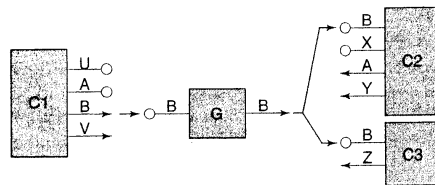
Connection-Based Software Architecture



Interface have input and output characteristics.

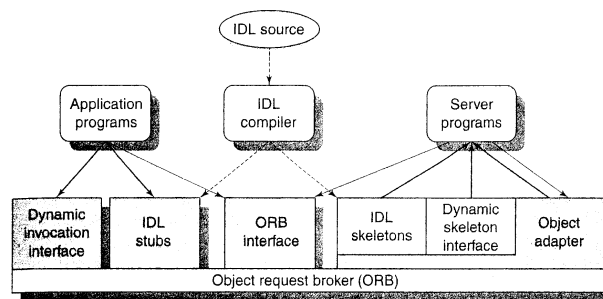


Information may be pushed to multiple outputs. (multiple-listeners)



Information distribution may be mediated through a common Gateway (adapters).

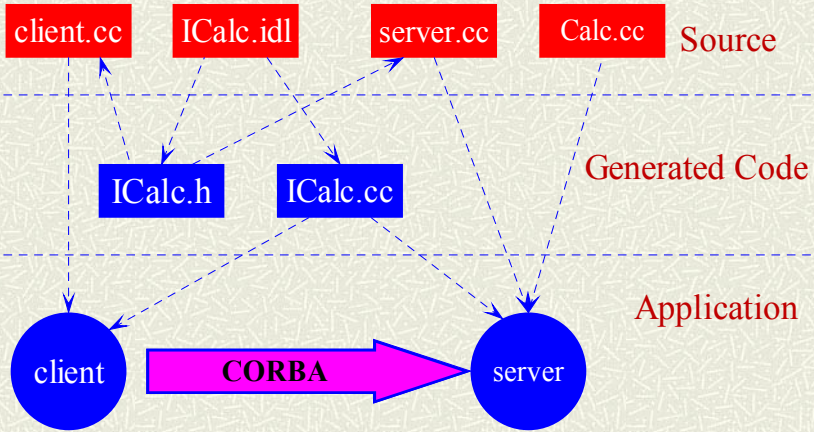
CORBA Architecture



- In CORBA (Common Object Request Broker Architecture), distribution of the information is mediated through a SOFTWARE BUS.
- Both the client and the server utilize an adapter to talk to the bus.
- The adapter on the client side is called a STUB, while the adapter on the server side is called a SKELETON.

Creation process of a CORBA application

*Executes WBT
test-bench*



Interface declaration

Decision:
What functions need testing?

CORBA interface

- # The services that an object provides are given by its *interface*.
- # Interfaces are defined in OMG's Interface Definition Language (IDL).
- # CORBA objects are described by IDL interface
- # An object interface indicates the operations the object supports, but **not** how they are implemented.

IDL Compiler

- # CORBA products provide an IDL compiler that converts IDL into a mapping language.
- # Example: ICalc.idl file:

```
interface ICalc {  
    float Add(in float input1, in float input2);  
    float Sub(in float input1, in float input2);  
    float Multiply(in float input1, in float input2);  
    float Divide(in float input1, in float input2);  
};
```

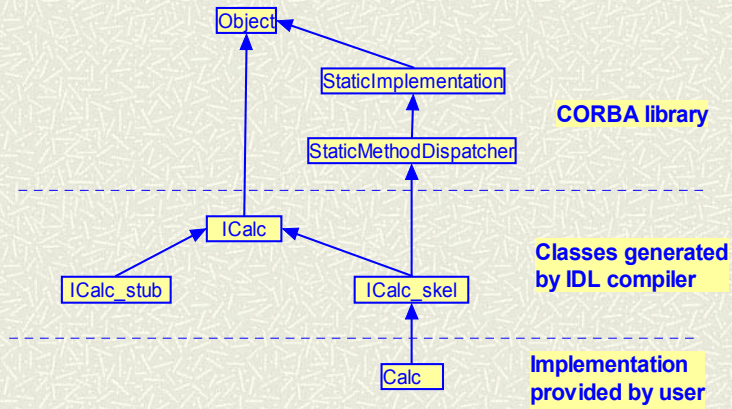
IDL data type mapping (marshalling)

| IDL Type | Java Type | C++ Type |
|---------------------------------|-----------------|-----------------|
| boolean | boolean | |
| char wchar | char | char |
| octet | byte | byte |
| short unsigned short | short | short |
| long unsigned long | int | int |
| long long unsigned long long | long | long |
| float double | float double | float double |
| string wstring | string | |

Generated code: stub & skel

```
class ICalc : virtual public CORBA::Object
{
    .....
}
class ICalc_stub: virtual public ICalc
{
    .....
}
class ICalc_skel :
    virtual public StaticMethodDispatcher,
    virtual public ICalc
{
    .....
}
```

Inheritance relationship



Server Side Programming

Provide code to forward the calls to the skeleton to appropriate methods in the AUT.

Calc Implement

```
#include "ICalc.h"
class Calc : virtual public ICalc_skel
{
public:
    Calc(){}
    CORBA::Float
        Add( CORBA::Float input1, CORBA::Float input2 );
    CORBA::Float
        Sub( CORBA::Float input1, CORBA::Float input2 );
    CORBA::Float
        Multiply( CORBA::Float input1, CORBA::Float input2 );
    CORBA::Float
        Divide( CORBA::Float input1, CORBA::Float input2 );
};
```

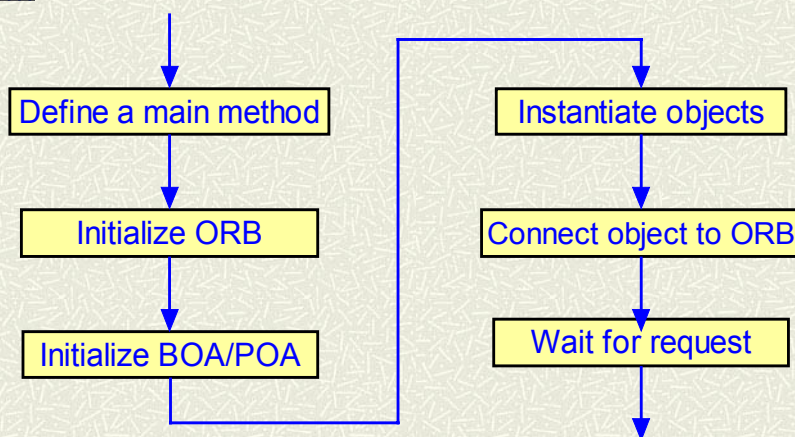
Object Adapter

- # Generation and interpretation of object references
- # Method invocation
- # Security of interactions
- # Object and implementation activation and deactivation
- # Mapping object references to the corresponding object implementations
- # Registration of implementations

Object Adapter

- # BOA: Basic Object Adapter
 - **Generating and interpreting object reference**
 - **Authenticating a principal process making a call**
 - **Activating and deactivating an implementation**
 - **Activating and deactivating individual objects**
 - **Invoking methods through the skeletons**

Elements of server-side program



server.cpp

```
#include "Calc.h"
int main( int argc, char *argv[])
{
    // ORB initialization
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
    CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa");

    // Create the server -- note down its reference
    Calc* server = new Calc();
    // Write out the object reference -- to be used by the client

    CORBA::String_var ref = orb->object_to_string( server );
    ofstream out("Calc.objid");
    out << ref << endl; out.close();

    // Keep running the server
    boa->impl_is_ready(CORBA::ImplementationDef::_nil());
    orb->run();
}
```

Server Reference

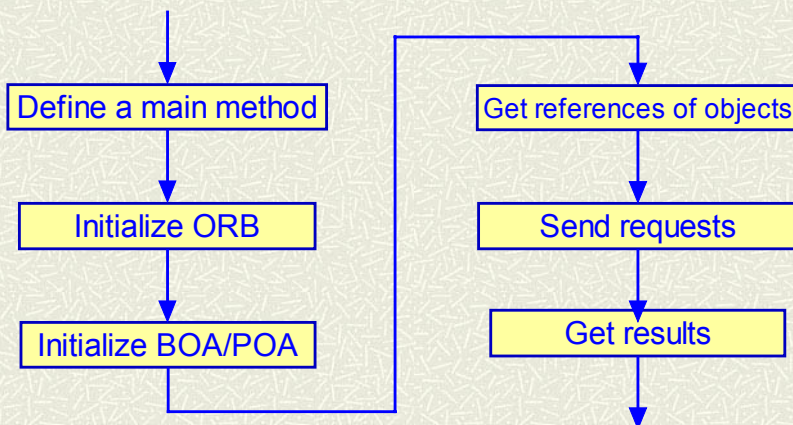
- # Generated by considering the IP/Network Interface address.
- # Global Identifier for a CORBA object.
- # This server object may be connected to by any client.
- # Calc object reference on my machine:

```
IOR:01000000e00000049444c3a4943616c633a312e3000000002000000
000000028000000010100000c0000003139322e3136382e312e3300a40a
00000c000000424f41c0a80103000007080301000000240000010000001d
0100000010000001400000001000000100010000000000901010000a
```

How -- Client side programming

Remotely located testers invoke methods on the CORBA server to execute a WBT test plan.

Elements of client-side program



Client.cpp

```
int main(int argc, char *argv[])
{
    CORBA::ORB_var orb=CORBA::ORB_init(argc, argv, "mico-local-orb");
    CORBA::BOA_var boa=orb->BOA_init(argc, argv, "mico-local-boa");
    ifstream in ("Calc.objid");
    char ref[1000];
    in >> ref;
    CORBA::Object_var obj = orb->string_to_object(ref);
    ICalc_var client = ICalc::_narrow( obj );
    int operation; float x, y;
    cout<<"1:Add\n2:Subtract\n3:Multiply\n4:Divide\n";
    cin>>operation; cout<<"X Y\n"; cin>>x>>y;
    if (operation == 1)
    {
        cout<<x<<" + "<<y<<" = "<<client->Add(x,y)<<endl;
    }
    .....
}
```

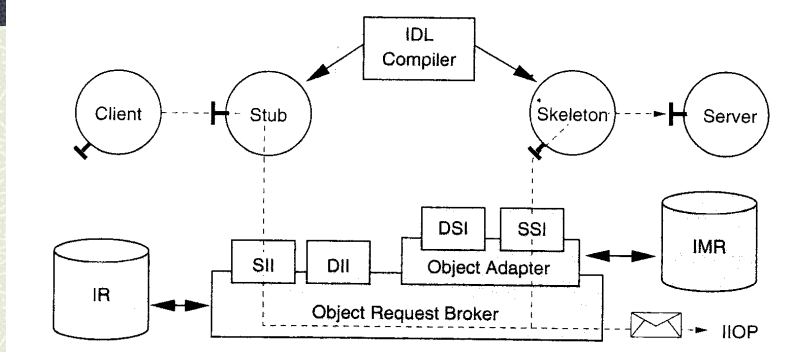
CORBA Interoperation

- # The ORB implements programming language independence for the request. The client issuing the request can be written in a different programming language from the implementation of the CORBA object
- # IIOP makes the interoperability possible

Client.java

```
import java.io.*;
import Account;
public class Client {
    public static void main( String[] args ) {
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init( args, null );
        String ref = null;
        try {
            BufferedReader in =
                new BufferedReader(new FileReader( "calc.objid" ));
            ref = in.readLine();
        } catch( IOException ex ) {
            System.out.println( "Could not open object reference file" );
            System.exit( -1 );
        }
        org.omg.CORBA.Object obj = orb.string_to_object( ref );
        Calc calc = ICalcHelper.narrow( obj );
        System.out.println("Add Test: " + calc.add(700,300 ));
    }
}
```

CORBA Objects may be searched for by names!



- CORBA also uses two run-time databases.
 - The Interface Repository (IR) can be queried at run-time.
 - IMR or Implementation Repository contains details on server.