# Integration Testing

## Contents

( CITATION Kai08 \l 1033 )

# Integration Testing

Integration testing is a phase of software implementation where software modules are combined and tested to identify any defects (interface errors) that may occur when the modules interface with each other.

Its primary objective is to assemble a reasonably stable system in a laboratory environment such that the integrated system can withstand the rigor of a full-blown system testing in the actual environment of the system.

# Importance of Integration Testing

Interface errors occur because different modules are typically created by different groups of developers and unforeseen issues will occur.

Unit testing are performed using test drivers and test doubles. So the modules' interfaces were not exercised.

# Types of Interfaces

Software Modules typically interact with other using the following paradigms:

- **Procedure Call:** A calling module passes control to the called module. Data may be passed from the caller to the called module and vice versa.
- **Shared Memory**: A block of memory is shared between modules. The memory block may be allocated by one of the modules. Data are written into the memory block by one module and are read by another.
- **Message Passing**: One module prepares a message by initializing the fields of a data structure and sending the message to another module. This form of module interaction is common in client-server systems and web-based systems.

# Types of Interface Errors

| Error | Description |
|---|---|
| **Construction** | Inappropriate use of module inclusions (e.g. #include from C) may cause construction errors. |
| **Inadequate functionality** | Assuming that the called module will provide a service that is missing. |
| **Location of functionality** | Assuming that the called module will provide a service that is implemented in a different module. |
| **Changes of functionality** | Changing the interface of a provided service without correctly updated the related modules. |

| | |
|---|---|
| **Misuse of interface** | The calling module uses the interface incorrectly. E.g. wrong parameter type |
| **Misunderstanding of interface** | The calling module did not ensure the pre-conditions of the called module. |
| **Data structure alteration** | The data structure used in the called module is inadequate to handle the incoming data. |
| **Inadequate error processing** | The calling module fails to handle the error generated in the called module properly. |
| **Inadequate post-processing** | The called module failed to release resources no longer needed. E.g. memory |
| **Inadequate Interface Support** | The calling module and the called module do not use the same units. E.g. one module uses Celsius and the other uses Fahrenheit. |
| **Initialization/Value Errors** | One of the modules fail to initialize, or assign, the appropriate value to a variable. |
| **Timing/Performance Problems** | The communicating processes are not properly synchronized. E.g. race conditions |
| **Hardware/Software Interfaces** | These are caused by inadequate handling of hardware devices. E.g. Sending data at a high rate to a hardware device but not considering the buffer size of the device. |

# Granularity of Integration Testing

***Black-box testing*** techniques ignores the internal mechanisms of a system and focuses solely on the outputs generated in response to selected inputs and execution conditions.

***White-box testing*** techniques uses information about the structure of the system to test its correctness. The internal mechanisms and modules of the system are taken into account.

***Intrasystem testing*** examines the interactions of the modules that create a single cohesive system.

***Intersystem testing*** examines the interactions of higher level system where each system may be considered a standalone system.

***Pairwise testing*** examines only two interconnected systems in an overall system at a time. The purpose is to ensure that the two systems can function together.

# System Integration Techniques

Some common approaches to performing system integration are as follows:

- Incremental
- Top down
- Bottom up
- Sandwich
- Big bang

## Incremental

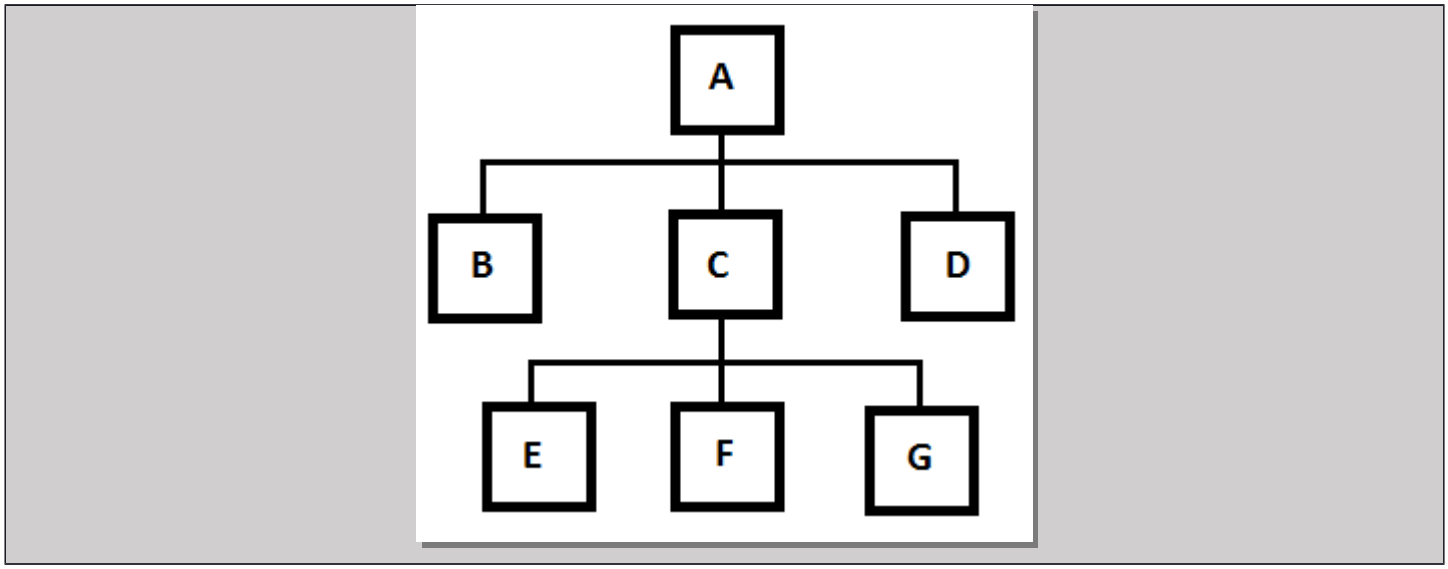
In each test cycle, a few more modules are integrated with an existing and tested ***build*** to generate a larger build. Effectively, the system is built incrementally, cycle by cycle, until the whole system is operational and ready for system testing.

A build is an interim compiled software used for internal testing within the organization. The final build will be a candidate for system testing, and such a tested system is released to the customers. The build is constructed from the following activities:

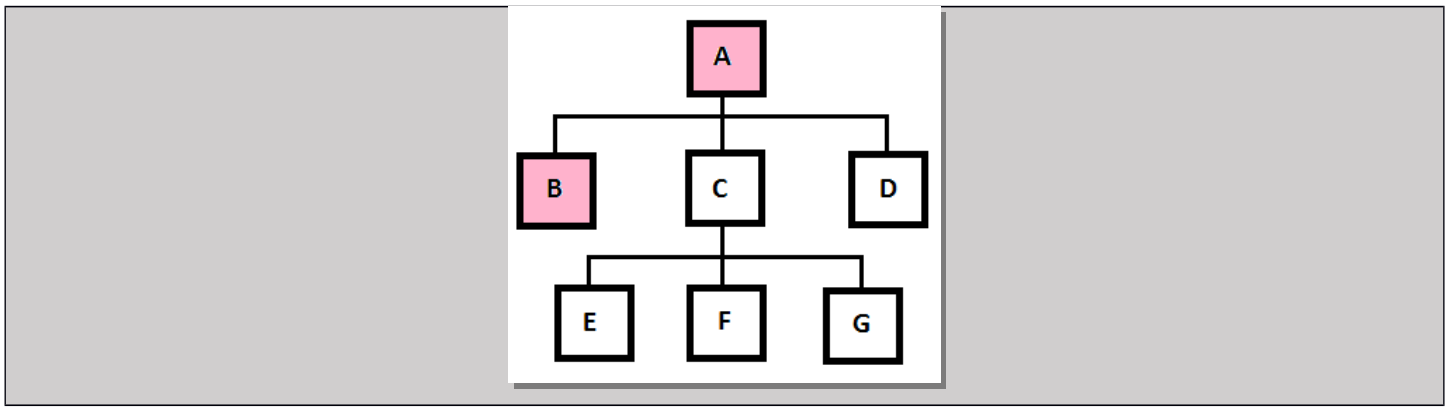
- Gathering the latest unit tested, authorized versions of the modules
- Checking in the code into the repository
- Compiling and linking the source code of those modules
- Verifying that the subassemblies are correct
- Exercising version control

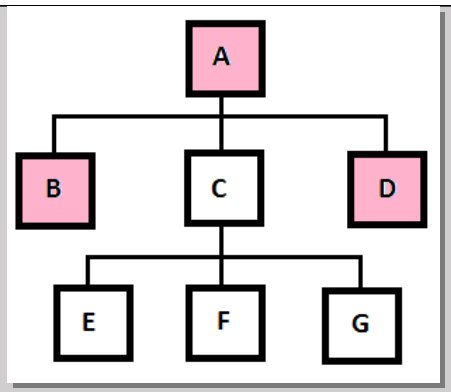
## Top-Down

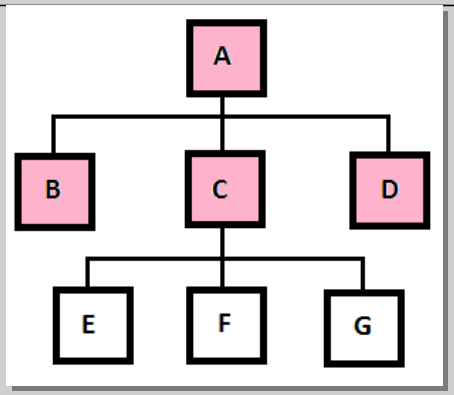
Consider the following module hierarchy.

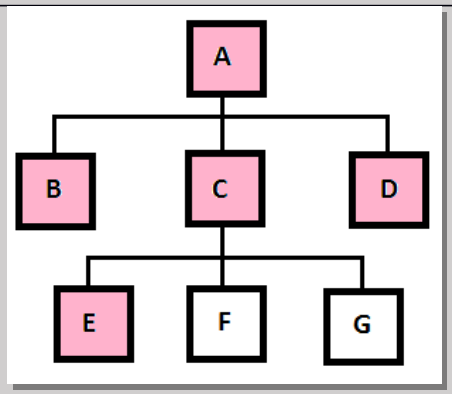


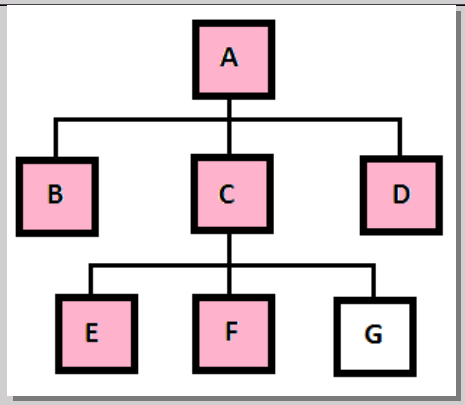

Top-Down Integration proceeds as follows:

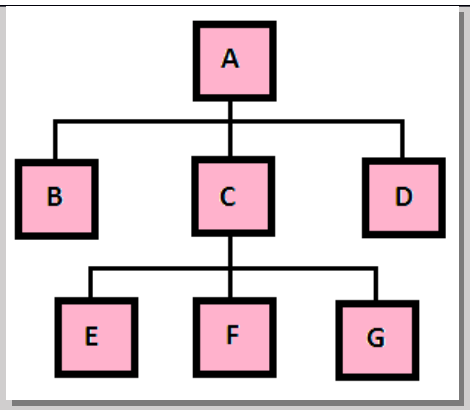



A
B
C
D
E
F
G

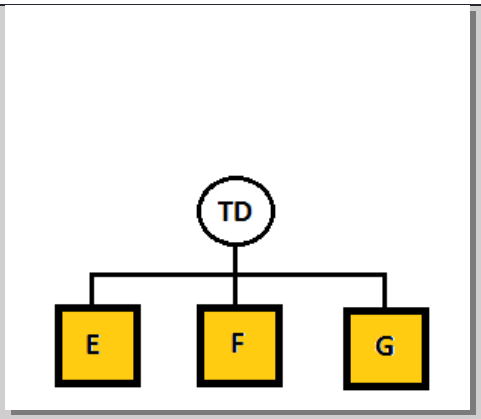

A
B
C
D
E
F
G

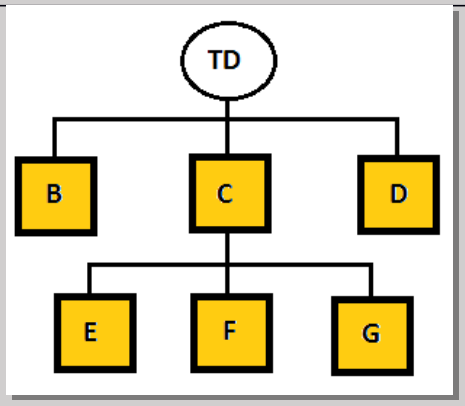

A
B
C
D
E
F
G

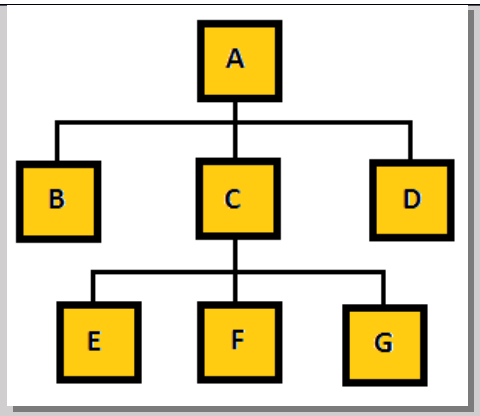

A
B
C
D
E
F
G




## Bottom Up






## Comparison of Top-Down and Bottom-Up

Faults in design decisions are detected early when using top-down design, these faults are detected later with the bottom-up approach.

Designing test data and stub behavior is more difficult with top-down integration.

## Sandwich

Integration is done using a combination of top-down and bottom-up integration.

## Big Bang

All modules are integrated and tested as a whole. This may be useful for small systems but is not recommended for large systems:

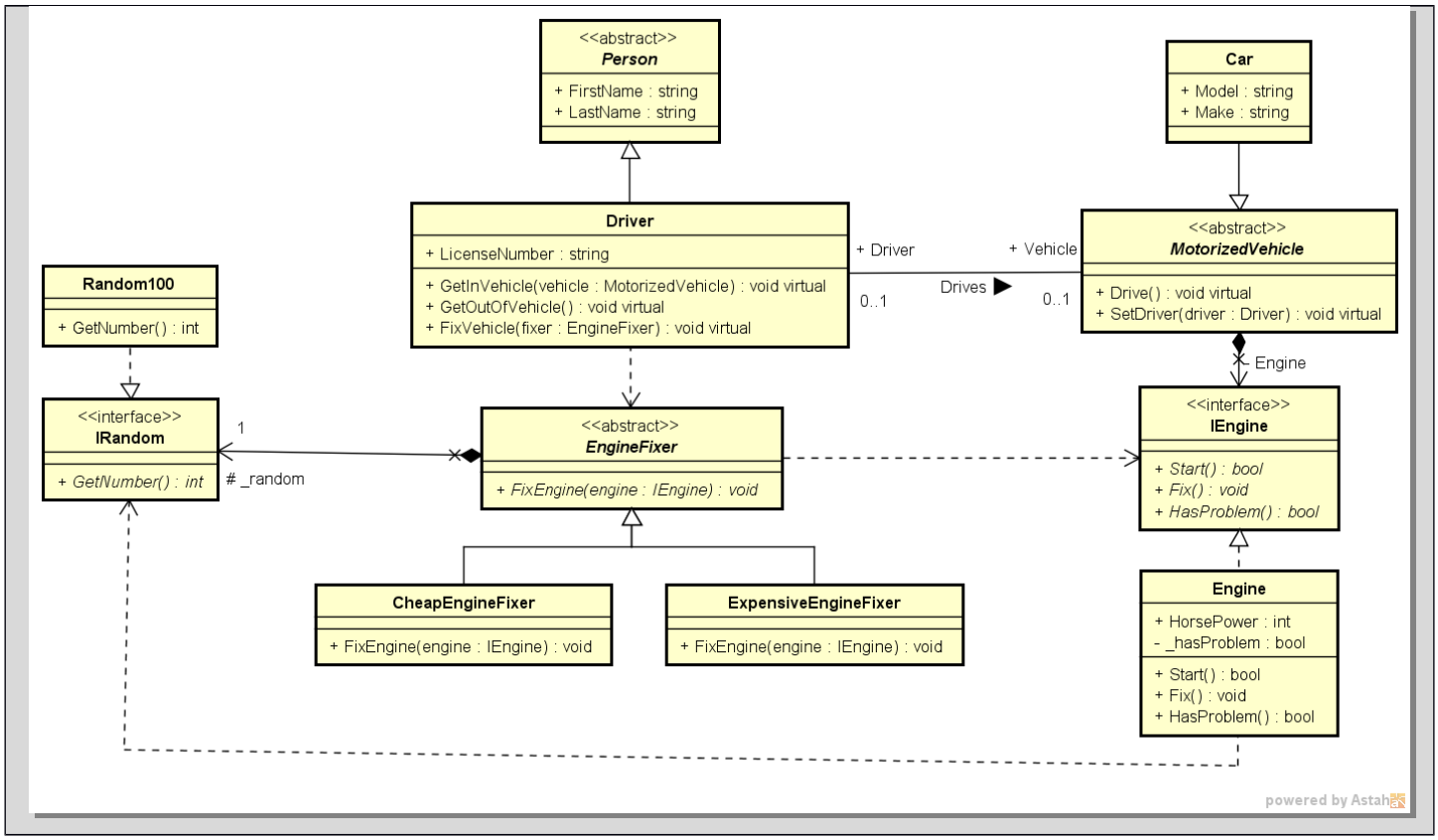
- It is more difficult to localize and potential defects.
- It is not cost effective.

# Detailed Design

The detailed design is the internal design of the application. It elaborates on the internal details of the system components and interfaces. It specifies the structure and relationships of data elements along with associated algorithms.

## Detail Design Elements

- **Detailed Data Model**
  - Describes all data elements and how they are related to each other
- **Detailed Functional Specifications**
  - Describes how the data is transformed to produce information
  - Describes the contracts of each operation
- **Detailed Behavioral Specifications**
  - Describes the states and state transitions of the data objects




## CheapEngineFixer Implementation

```
public abstract class EngineFixer
{
    protected IRandom _random;

    public EngineFixer(IRandom random = null)
    {
        if(random == null)
        {
            random = new Random100();
        }
        _random = random;
    }

    public abstract void FixEngine(IEngine engine);
}

public class CheapEngineFixer : EngineFixer
{
    public CheapEngineFixer(IRandom random = null) : base(random)
    {
    }

    public override void FixEngine(IEngine engine)
    {
        var number = _random.GetNumber();
        if(number >= 80)
```

```
        {
            engine.Fix();
        }
    }
}
```

```
public interface IRandom
{
    int GetNumber();
}

public class Random100 : IRandom
{
    private Random _random = new Random();

    public int GetNumber()
    {
        return _random.Next(1, 101);
    }
}
```

```
public interface IEngine
{
    void Fix();
}
```

## CheapEngineFixer Unit Testing

```
[Category("Unit Tests")]
[Category("A CheapEngineFixture")]
[TestFixture]
public class ACheapEngineFixer
{
    [Test]
    public void Has20PercentChanceOfFixingAnEngine()
    {
        var randomMock = new Mock<IRandom>();
        randomMock.Setup(r => r.GetNumber()).Returns(80);

        var engineMock = new Mock<IEngine>();

        var sut = new CheapEngineFixer(randomMock.Object);
        sut.FixEngine(engineMock.Object);
        engineMock.Verify(e => e.Fix());
    }

    [Test]
    public void Has80PercentChanceOfNotFixingAnEngine()
    {
        var randomMock = new Mock<IRandom>();
        randomMock.Setup(r => r.GetNumber()).Returns(79);

        var engineMock = new Mock<IEngine>();

        var sut = new CheapEngineFixer(randomMock.Object);
```

```
        sut.FixEngine(engineMock.Object);
        engineMock.Verify(e => e.Fix(), Times.Never);
    }
}
```

## Engine Implementation

```
public interface IEngine
{
    void Fix();
    bool HasProblem();
}
```

```
public class Engine : IEngine
{
    private bool _hasProblem;

    public Engine(IRandom random = null)
    {
        if(random == null)
        {
            random = new Random100();
        }
        _hasProblem = false;
        if(random.GetNumber() >= 90)
        {
            _hasProblem = true;
        }
    }

    public void Fix()
    {
        _hasProblem = false;
    }

    public bool HasProblem()
    {
        return _hasProblem;
    }
}
```

## CheapEngineFixer / Engine Integration Testing

```
[Category("Integration Tests")]
[Category("A CheapEngineFixture")]
[TestFixture]
public class ACheapEngineFixer
{
    [Test]
    public void Has20PercentChanceOfFixingAnEngine()
    {
        var fixerRandomMock = new Mock<IRandom>();
        fixerRandomMock.Setup(r => r.GetNumber()).Returns(80);

        var engineRandomMock = new Mock<IRandom>();
```

```
        engineRandomMock.Setup(r => r.GetNumber()).Returns(90);

        var engine = new Engine(engineRandomMock.Object);
        Assert.That(engine.HasProblem(), Is.True);

        var sut = new CheapEngineFixer(fixerRandomMock.Object);
        sut.FixEngine(engine);
        Assert.That(engine.HasProblem(), Is.False);
    }

    [Test]
    public void Has80PercentChanceOfNotFixingAnEngine()
    {
        var fixerRandomMock = new Mock<IRandom>();
        fixerRandomMock.Setup(r => r.GetNumber()).Returns(79);

        var engineRandomMock = new Mock<IRandom>();
        engineRandomMock.Setup(r => r.GetNumber()).Returns(90);

        var engine = new Engine(engineRandomMock.Object);
        Assert.That(engine.HasProblem(), Is.True);

        var sut = new CheapEngineFixer(fixerRandomMock.Object);
        sut.FixEngine(engine);
        Assert.That(engine.HasProblem(), Is.True);
    }
}
```

# References


Naik, K., & Tripathy, P. (2008). *Software Testing and Quality Assurance Theory and Practice.* Wiley.