# Writing Testable Software using Hierarchical Context Object Pattern

## What is a unit and what is a unit test?

A unit is a section of code encapsulated by a class or method.

A unit test exercises one or more facets of a unit in an automated fashion by simulating input, verifying expected output and reporting on the results.

## What are the benefits of unit testing?

The main benefit of unit testing is confidence. Confidence that when a change is made and after unit tests are executed that the system behaves as expected.

If a test does fail it will be found early in the development cycle before code is in production. The earlier a bug is found the cheaper it is to fix (sometimes by several orders of magnitude).

Unit testing allows for (constant) refactoring.

## How can software be written to facilitate unit testing?

Unit testing is easier if the unit being tested is loosely coupled with it's dependencies. Loose coupling allows dependencies to be easily mocked.

## Loose coupling strategies

### Abstract creation of objects

Problem: If class A constructs an instance of class B then class A has a tight binding with class B - it has intimate knowledge of how to create B and is limiting itself to only using instances of B and none of it's sub-classes.

The tight binding can be alleviated by abstracting the creation of objects.

### Abstract object lookup

Problem: If class A does not construct an instance of class B directly it can still be tightly coupled to an intermediary class or mechanism from which A obtains an instance of B

Examples of non-abstracted lookup:

* obtaining instances via static methods
* service lookup class whose instance can not be changed
* service lookup class is instantiated to get instance of another class
* instantiating a factory
* singletons

### Use objects via interface references rather than class references

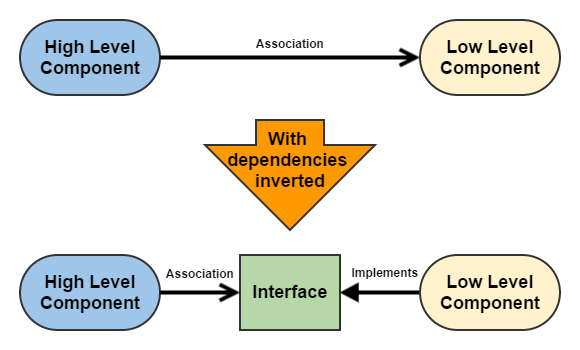
Problem: If class A uses an instance of class B via a reference to B it is still coupled with a concrete class. A looser coupling is if class A uses class B via a reference to an interface that class B implements.

## Introducing the dependency inversion principle

The dependency inversion principle states:

1. High-level components should not depend on low-level components. Both should depend on abstractions
2. Abstractions should not depend on details. Details should depend on abstractions

The principle inverts high and low level component dependencies so that high and low level components are dependent on a set of interfaces instead of high level components being dependent on low level components.



Software written using dependency inversion results in reusable high level components as they are not tied to low level components.

Abstracting the creation and lookup of objects and referencing them via interfaces help to facilitate an implementation of the dependency inversion principle known as dependency injection.

#### Example

**Before inversion:**

public class A {} // low level class

// High level class

public class B {

private A a = new A();

}

B b = new B();

In the above:

* class B is dependent on class A
* class B constructs the instance of class A

**After inversion:**

public interface X {}

public class A implements X {}

public class B {

private X x;

public B( X x ) {

this.x = x;

}

}

B b = new B( new A() );

In the above:

* class A implements interface X
* class B is passed and uses an instance of a class that implements interface X
* classes A and B are dependent on interface X
* B is not dependent on A

## Introducing dependency injection

Dependency injection is a pattern in which objects are passed by reference into a dependent object and are made part of the object's state.

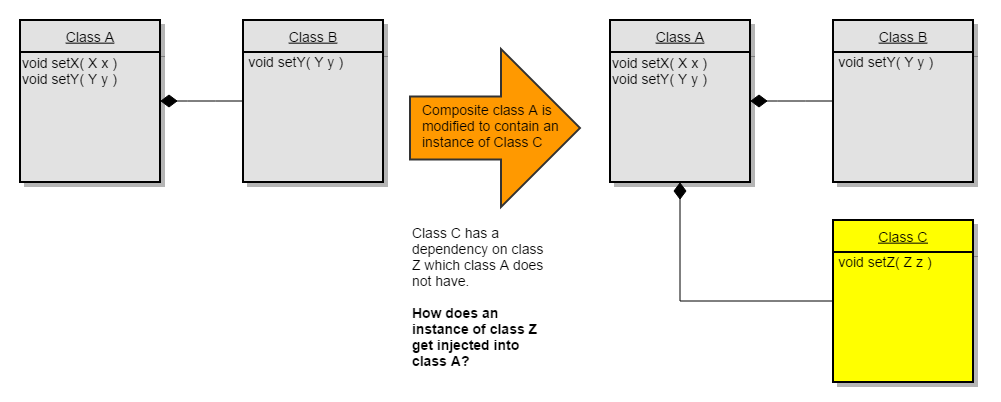
For example if an object A is dependent on an object that implements interface X then an instance of X will be injected into A usually via it's constructor or a "setX" method.

By injecting objects the three highlighted loose coupling strategies are employed:

* objects do not create their dependencies they are injected
* objects are not looked up they are injected
* dependencies are passed and used via an interface reference

## Problem with dependency injection

If a new object is introduced to a composite object and has dependencies that were not explicit dependencies of the parent object, then then the parent object has no way of injecting the new object's dependencies without (sometimes significant) refactoring.



It is possible to refactor code so that new dependencies are passed down to lower layers, however this is not always convenient. It also has the side effect of making classes in the middle dependent on a class they were otherwise not concerned with.

## Potential solution: context object pattern

Instead of injecting specific objects, a single context object is injected that contains references to services a (sub)object might depend on.

The implementation of a context is a hashmap where objects (services) are stored using key, value pairs. Objects are retrieved using a key.

public class Context

{

private HashMap<Object, Object> context = new HashMap<Object, Object>();

public Context() {}

public Context put( Object key, Object value ) {

context.put( key, value );

return this;

}

public Object get( Object key ) {

return context.get( key );

}

}

The context object is injected instead of specific objects which are now contained within the context.

Allows high level objects to inject objects to all parts of the system as long as the context is injected from parent to child classes.

A number of issues exist with the context object:

* if there is only one context then it behaves as a global memory space
  + services in the context can be replaced by any object in the system
* if there are multiple contexts then a context may have to be created for each context aware object

## Solution: Hierarchical context object pattern

Context object issues can be resolved by making it hierarchical.

In the hierarchical context pattern there are many context objects each with their own hashmap and a reference to it's parent. A hierachical context instance is a directed acyclic graph.

When an object is 'put' into the context it is put in it's hashmap. When an object is retrieved using a key via 'get' the context's hashmap is checked for the object and if it doesn't contain it then the parent context's 'get' method is called. Calling 'get' is recursive and up to and including the root context (the one with a null parent) may be queried for an object.

public class Context

{

private HashMap<Object, Object> context = new HashMap<Object, Object>();

private Context parent = null;

public Context() {}

private Context( Context parent ) {

this.parent = parent;

}

public Context createSubContext() {

return new Context( this );

}

public Context put( Object key, Object value ) {

context.put( key, value );

return this;

}

public Object get( Object key ) {

Object object = context.get( key );

if (( object == null ) && ( parent != null )) {

object = parent.get( key );

}

return object;

}

}

The hierarchical context pattern solves the issue of a global memory space as an object put in a child context can not overwrite the parent context.

It also solves the issue of instantiating new contexts as child contexts essentially contain what is contained in the parent context.

Child contexts are inexpensive and only need to be populated with new classes or classes that override the parent context.

## Context best practices

### Use class objects as context keys

public interface X {}

public class A implements X {}

X a = new A();

Context context = new Context();

context.put( X.class, a );

### Classes that can have context injected via a set method should implement ContextAware interface

public interface ContextAware

{

public void setContext( Context context );

public Context getContext();

}

### Create a sub context for any major subsystem

Context subContext = context.createSubContext();