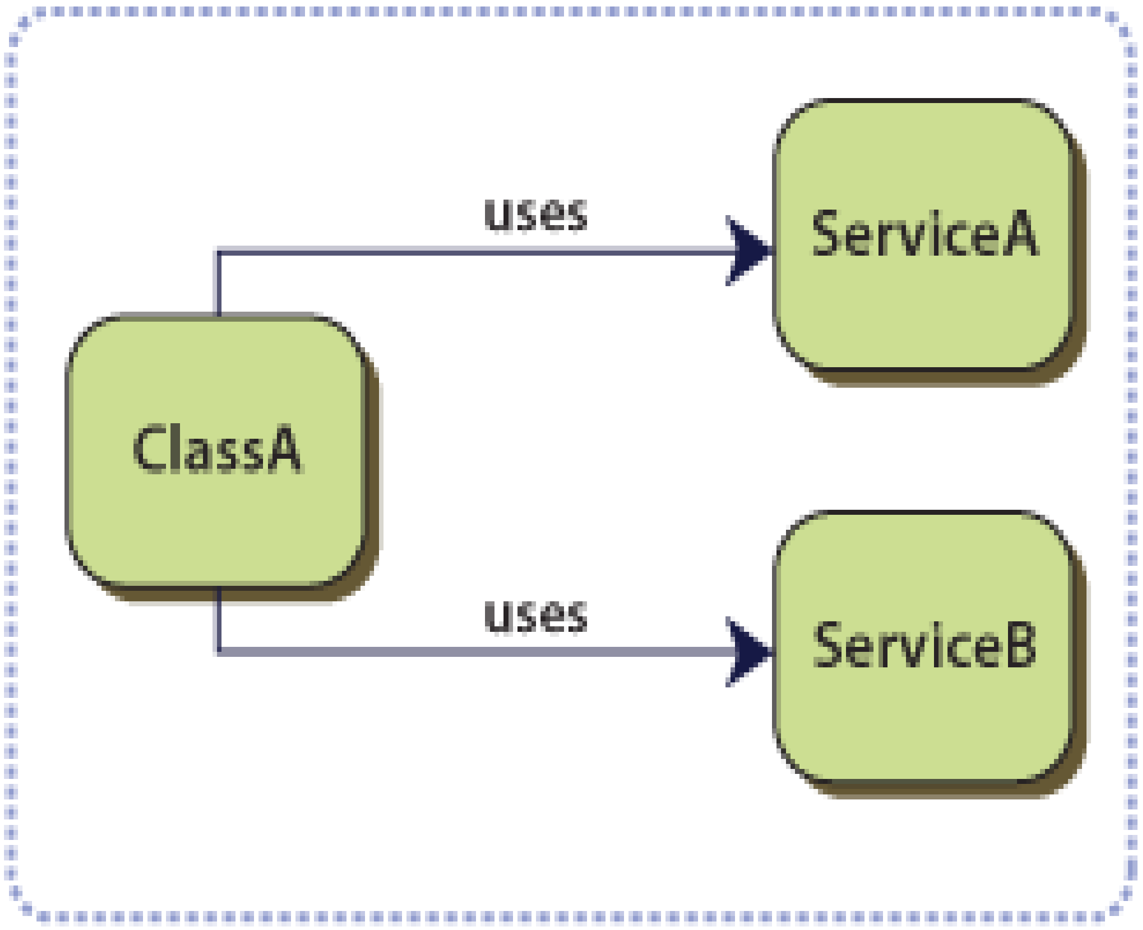
# Dependency Injection

- PROBLEM
  - You have classes that have dependencies on services or components whose CONCRETE type is specified at DESIGN time (OO sacrilege)
  - To replace or update dependencies, you must change your classes' source code
  - The concrete implementation of the dependencies must be available at compile time
  - Your classes are difficult to test in isolation and cannot be replaced with stubs or mocks
  - Your classes contain repetitive code for creating, locating, and managing their dependencies

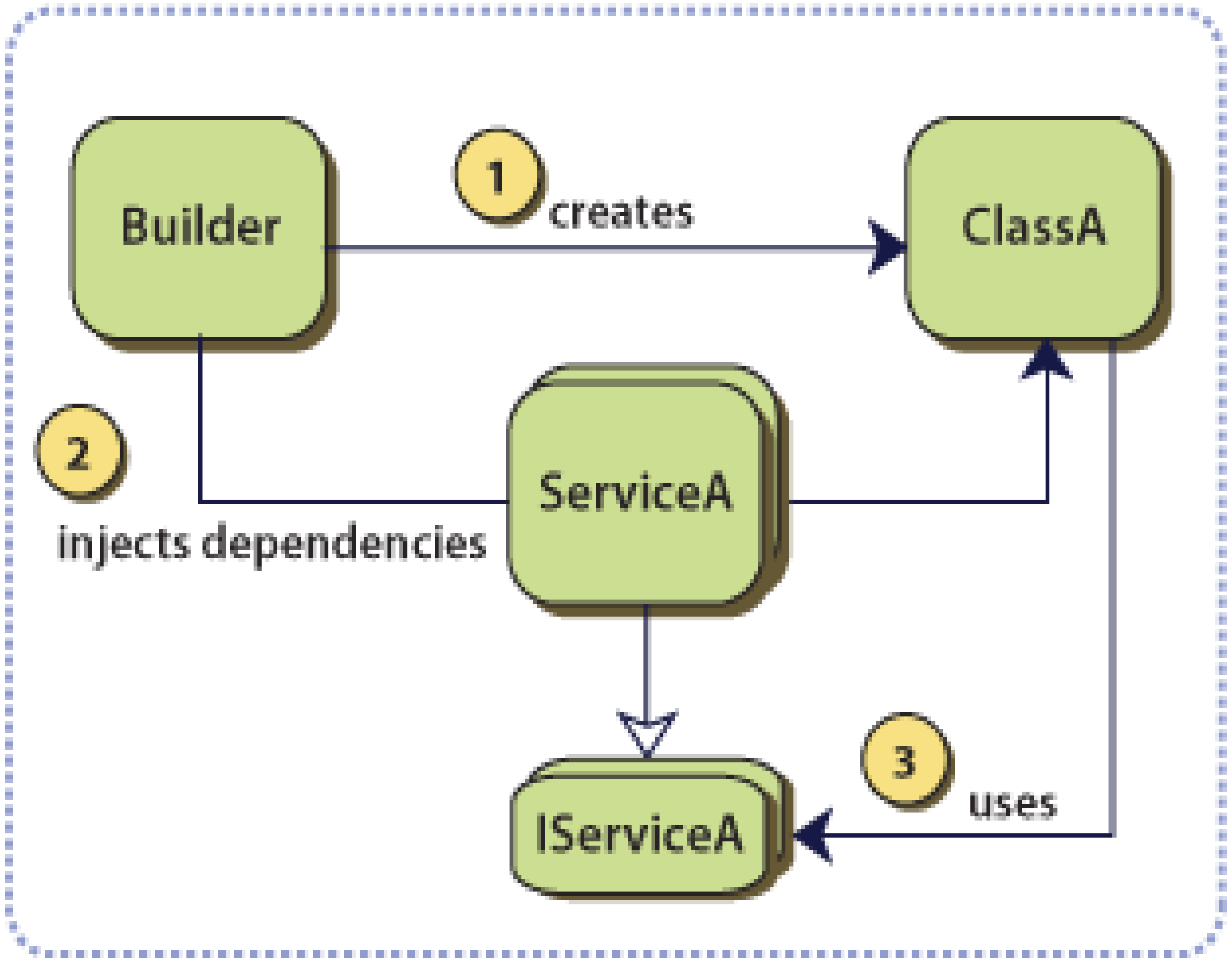
# Dependency Injection Problem UML



## Dependency Injection Solution

- Do not instantiate the dependencies explicitly in your class.
- Instead, declaratively express dependencies in your class definition.
- Use a Builder object to obtain valid instances of your object's dependencies and pass them to your object during the object's creation and/or initialization.
- Typically, you express dependencies on interfaces instead of concrete classes. This enables easy replacement of the dependency concrete implementation without modifying your classes' source code.

# Dependency Injection Solution UML



## Forces that justify using the pattern

- You want to decouple your classes from their dependencies so that these dependencies can be replaced or updated with minimal or no changes to your classes' source code.
- You want to be able to write classes that depend on classes whose concrete implementation is not known at compile time.
- You want to be able to test your classes in isolation, without using the dependencies.
- You want to decouple your classes from being responsible for locating and managing the lifetime of dependencies.

## Main forms of Dependency Injection

- Constructor Injection: use parameters of the object's constructor method to express dependencies and to have the builder inject it with its dependencies
- Setter Injection: the dependencies are expressed through setter properties that the builder uses to pass the dependencies to it during object initialization

## Liabilities

- You have to ensure that, before initializing an object, the dependency injection framework can resolve the dependencies that are required by the object
- There is added complexity to the source code; therefore, it is harder to understand
- There are more solution elements to manage

## Uses

•A simple way to load plugins dynamically or to choose stubs or mock objects in test environments versus real objects in production environments
•Injects the depended-on element (object or value, etc.) to the destination by 'automatically' knowing the requirement of the destination.
•Applies core principle of separation of behavior from dependency resolution

## Three Fundamental Elements

•A dependent consumer
•A declaration of a component's dependencies, defined as interface contracts
•An injection (sometimes referred to as a provider or container) that creates instances of classes that implement a given dependency interface on request
•The dependent object describes what software component it depends on to do its work. The injector decides what concrete classes satisfy the requirements of the dependent object, and provides them to the dependent

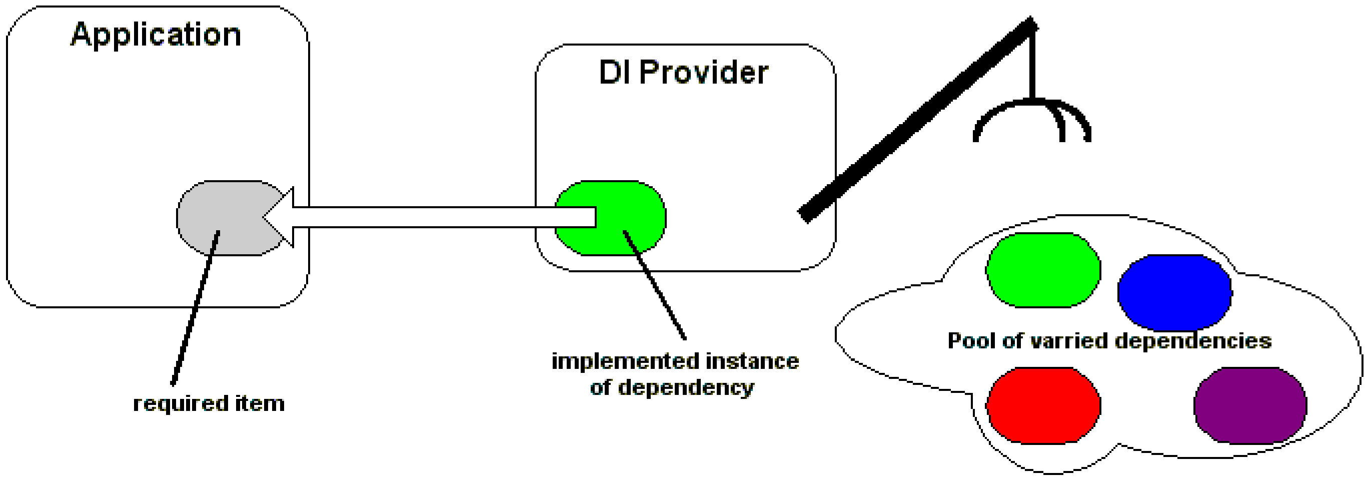
## Motivation for Use

- The primary purpose of the dependency injection pattern is to allow selection among multiple implementations of a given dependency interface at run time
- Multiple, different implementations of a single software component can be created at run-time and passed (injected) into the same test code
- The test code can then test each different software component without being aware that what has been injected is implemented differently

## Injector

- The Injector feels a bit like a Factory
- However, what separates Factory from Dependency Injection is the implementation details. Those details are the 'secret sauce' that tells the factory how to create the dependency
  - If a Factory by default knows how to creat the dependency, then it's just a factory
  - If by default the factory does not know how to create the dependency and has to read implementation details in order to know how to instantiate, configure, and wire together the object graph (dependency), then you have dependency injection

# Another Diagram


Application
DI Provider
required item
implemented instance of dependency
Pool of varried dependencies

# Wikipedia Example

```
public interface IOnlineBrokerageService {
   String[] getStockSymbols();
   double getBidPrice(String stockSymbol);
   double getAskPrice(String stockSymbol);
   void putBuyOrder(String stockSymbol, int shares, double
buyPrice);
   void putSellOrder(String stockSymbol, int shares, double
sellPrice);
}

public interface IStockAnalysisService {
   double getEstimatedValue(String stockSymbol);
}

public interface IAutomatedStockTrader {
   void executeTrades();
}
```

# Wikipedia Example: Highly Coupled Dependency (No Injection Used)

```
public class VerySimpleStockTraderImpl implements IAutomatedStockTrader {

   private IStockAnalysisService analysisService = new
StockAnalysisServiceImpl();
   private IOnlineBrokerageService brokerageService = new
NewYorkStockExchangeBrokerageServiceImpl();

   public void executeTrades() {
      ….
   }
}

public class MyApplication {
   public static void main(String[] args) {
      IAutomatedStockTrader stockTrader = new VerySimpleStockTraderImpl();
      stockTrader.executeTrades();
   }
}
```

# Manually Injected Dependency

```
public class VerySimpleStockTraderImpl implements
IAutomatedStockTrader {

    private IStockAnalysisService analysisService;
    private IOnlineBrokerageService brokerageService;

    public VerySimpleStockTraderImpl(
            IStockAnalysisService analysisService,
            IOnlineBrokerageService brokerageService) {
        this.analysisService = analysisService;
        this.brokerageService = brokerageService;
    }
    public void executeTrades() {
        …
    }
}
```

# Manually Injected Dependency

```
public class MyApplication {
   public static void main(String[] args) {
       IStockAnalysisService analysisService = new
StockAnalysisServiceImpl();
       IOnlineBrokerageService brokerageService = new
NewYorkStockExchangeBrokerageServiceImpl();

       IAutomatedStockTrader stockTrader = new
VerySimpleStockTraderImpl(
           analysisService,
           brokerageService);
       stockTrader.executeTrades();
   }
}
```

# Unit Testing Using Injected Stub Implementations

```
public class VerySimpleStockBrokerTest {
    // Simplified stub implementation of IOnlineBrokerageService.
    public class StubBrokerageService implements IOnlineBrokerageService {
        public String[] getStockSymbols() {
            return new String[] {"ACME"};
        }
        public double getBidPrice(String stockSymbol) {
            return 100.0; // (just enough to complete the test)
        }
        public double getAskPrice(String stockSymbol) {
            return 100.25;
        }
        public void putBuyOrder(String stockSymbol, int shares, double buyPrice) {
             Assert.Fail("Should not buy ACME stock!");
        }
        public void putSellOrder(String stockSymbol, int shares, double sellPrice) {
             // not used in this test.
             throw new NotImplementedException();
        }
    }
```

## Unit Testing Using Injected Stub Implementations

```
public class StubAnalysisService implements
IStockAnalysisService {
        public double getEstimatedValue(String
stockSymbol) {
            if (stockSymbol.equals("ACME"))
                return 1.0;
            return 100.0;
        }
    }
```

# Unit Testing Using Injected Stub Implementations

```
public void TestVerySimpleStockTraderImpl() {
	// Direct the DependencyManager to use test
implementations.
	DependencyManager.register(
	   IOnlineBrokerageService.class,
	   StubBrokerageService.class);
	DependencyManager.register(
	   IStockAnalysisService.class,
	   StubAnalysisService.class);

	IAutomatedStockTrader stockTrader =
	   (IAutomatedStockTrader)
DependencyManager.create(IAutomatedStockTrader.class);
	stockTrader.executeTrades();
   }
}
```

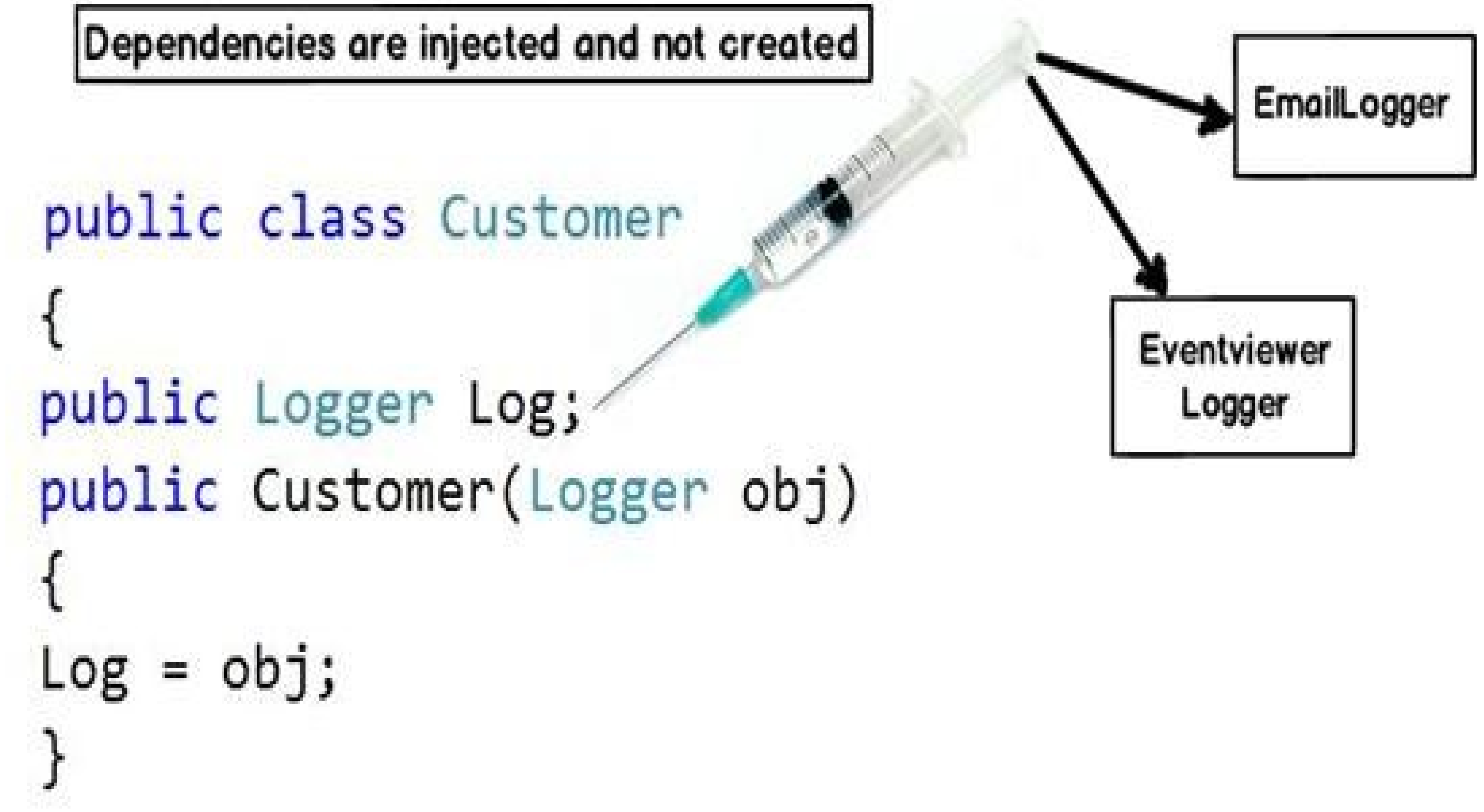
## Overall Benefits

- One benefit of using the dependency injection approach is the reduction of boilerplate code in the application objects since all work to initialize or set up dependencies is handled by a provider component
- Another benefit is that it offers configuration flexibility because alternative implementations of a given service can be used without recompiling code. This is useful in unit testing, as it is easy to inject a fake implementation of a service into the object being tested by changing the configuration file, or overriding component registrations at run-time.
- Furthermore, dependency injection facilitates the writing of testable code.

## Some Disadvantages

- Dependencies registered in the container are effectively black boxes as regards the rest of the system. This makes it harder to detect and recover from their errors, and may make the system as a whole less reliable
- Dependency injection hides the class' dependencies, causing run-time errors instead of compile-time errors when dependencies are missing or incompletely implemented
- The dependency injection container makes the code more difficult to maintain, because it becomes unclear when you would be introducing a breaking change
- If the dependency is injected at runtime, the performance of the application is reduced

# Dependencies are Injected, NOT Created



```
public class Customer
{
public Logger Log;
public Customer(Logger obj)
{
Log = obj;
}
```

## Oliver Klee of TDD Fame