Abstract—Dependency Injection (DI) is a powerful design and implementation technique to create designs that result in loosely coupled classes in object-oriented environments. IOC containers such as Spring provide a framework to describe the DI constructs for classes that have been designed with dependency inversion. However the knowledge of classes and their dependency is now distributed between the code and context files where the wiring of the classes (beans) is specified.

While IOC containers are able to automatically inject dependent objects at run time based on wiring descriptions, programmers often have the tedious job of ensuring that the wiring descriptions are type safe and will not result in runtime exceptions.

We propose an object reflection driven approach to provide DI verification based on static analysis of the classes that have dependency inversion and their wiring descriptions.

Index Terms—Dependency inversion, inversion of control containers, auto-wiring of objects

I. INTRODUCTION

Object-oriented technology has lived up to its promise of providing not only a paradigm for system decomposition but also a technology for building scalable and maintainable systems. However, just using object-oriented designs and programs will not result in software that is maintainable and scalable. It is now established that an essential characteristic of scalable, maintainable, extensible and stable software systems is loose coupling between different modules and components. It is also established that modularization of a software system results in loose coupling between different components of the software. Essentially loose coupling reduces dependency between components and the variability of a component with time due to changes causes lower impacts on other components. Coupling between components or classes in an object-oriented program arises from dependency between classes and components.

While no coupling between components and classes is not only an impossible achievement in software design, it also undesirable because no-coupling indicates the lack of utility of the component or class, which has no coupling between itself and other classes. This indicates that there is no dependency between the class in question and other classes and neither does any other class depend on this class. The class in question has no role to play in such cases and is redundant. High degree of coupling arises from designs that create classes on which a number of classes depend on. This is usually a result of poor abstraction and decomposition. In such cases there a large number of classes that depend on the class in question, which introduces rigidity and fragility into the programs. Martin [1] defines rigidity as designs that are hard to change and fragility as designs that are easy to break. The solution to both of these problems is to reduce coupling between classes. As mentioned earlier, zero coupling is not the solution to this problem, but rather loose coupling. Loose coupling is obtained by creating rich abstraction [2]-[3] and following the principle of Single Responsibility Principle (SRP) [1]. SRP essentially requires a class to fulfill a single responsibility. Martin offers a simple test for SRP: if a class has to change for more than one reason, then it violates SRP. The most striking aspect of SRP and loose coupling is that even though these principles were stated decades ago, it is quite common to see these principles violated even in modern software. The key culprit is violation of modularity principles.

Nearly sixty years ago, Nobel laureate Herbert Simon and his collaborator Albert Ando analyzed various evolutionary systems and stated principles governing aggregation of components to achieve equilibrium of complex systems [4]. Software is also an evolutionary system and Simon-Ando principles are applicable. Essentially the principles stated by Simon-Ando are:

-Hierarchy] Frequently, complexity takes the form of hierarchy, where a complex system is composed of inter-related subsystems, that in turn have their own subsystem, and so on, until some lowest level of elementary components is reached.

-Local Equilibrium] In the short-term as a result of stronger internal bonds, subsystems tend to reach equilibrium “approximately” independent of each other.

-Global Equilibrium] In a long-term period the whole system evolves towards a state of global equilibrium under the influence of weak interactions among subsystems, the local equilibriums reached are maintained.

Simon-Ando principles are completely aligned with software decomposition principles stated by Parnas and advocated by modern architecture principles [5]. Weak interactions, which is key for stability of complex systems, result from loose coupling between subsystems. An effective way to achieve loose coupling between classes and components in addition to following SRP, is the Dependency Inversion Principle [1], which states that when a class must depend on another class, the dependency must be on abstraction rather than concrete implementation. An extension of this principle is Dependency Injection, where dependency relations are externalized and “injected” into the dependent classes; that is the dependent classes are materialized outside the classes that depend on each other. Dependency injection creates the problem of correctly injecting dependent classes. There are two parts to this

Manuscript received September 15, 2012; revised November 9, 2012.
Shrinidhi R. Hudli is with the M.S. Ramaiah Institute of Technology (e-mail: shrinidhi@hudli.com).
Raghu V. Hudli is with ObjectOrb Technologies Pvt. Ltd.

DOI: 10.7763/LNSE.2013.V1.16 71
problem:
1) The injected dependent classes must be type compatible with the abstractions.
2) The injected classes must fully implement the behavior expected by the dependent class.

In this paper we propose a strategy for verification of dependency injection that solves both parts of the problem.

In Section II we discuss Dependency Inversion and Dependency Injection. In Section III we discuss principles of Inversion of Control containers, which inject dependencies, with Spring as an example. In Section IV we discuss the strategy for verification of Dependency Injection and discuss results in Section V.

II. DEPENDENCY INVERSION AND DEPENDENCY INJECTION

As mentioned in Section I, in any object-oriented design, there will be dependencies between classes. For example, consider a design to implement remote method invocation. An excerpt of an example design is shown in Fig. 1, below. The Remote Method Invoker class needs a transport object to physically transport the method request. There is often a choice of transports implementing different protocols. In case the remote object is present on the same system, performance is optimized to use an IPC transport, if it is located on a different system, a TCP transport is needed, and if the remote object supports HTTP protocol (SOAP or Web Services) a HTTP transport is needed. In any case, the Remote Method Invoker depends on multiple transport objects and its association with such classes is shown in Fig. 1.

Fig. 1. Remote method invoker and multiple transport objects

There are other cases, where we may have similar designs. Consider the case of an email client that needs to send and receive email from different providers – Gmail, Yahoo Mail, Windows Live and other proprietary email providers. The email client object will have dependencies between the different email provider objects. A text-processing object that needs to spell check will have similar dependency structure between itself and spell checkers in English, German, French and other languages.

The implementation of such designs require the Remote Method Invoker, email client or the text-processing object to create and manage these dependent objects. It is typical that based on a parameter one of the dependent objects is utilized.

While on the surface this design looks all right, there is a serious problem with the design. The Remote Method Invoker object has too much knowledge of the dependent objects, creating a strong coupling. If a new transport object is needed the Remote Method Invoker object needs to be modified to use it. This makes extensibility hard. Further there is little or no abstraction of a “transport” that Remote Method Invoker uses. It relies on multiple concrete implementations of such transport objects – IPC, TCP, and HTTP.

The problem is solved using Dependency Inversion Principle (DIP) [1]-[6]. DIP looks at dependent objects as service providers. DIP emphasizes the principle of least knowledge about actual service provider – in our example, IPC Transport, TCP Transport and HTTP Transport. Since each transport is now viewed as a service provider of a service abstraction of a Transport service, the Method Invoker only needs to know that there is a Transport service and it depends on that instead of each service provider – IPC, TCP, and HTTP as shown in Fig. 2. below.

Application of DIP principle requires Remote Method Invoker to depend on abstractions and not on details (concrete objects). Fig. 2 correctly captures the relationship by introducing an abstraction called Transport. The concrete objects are merely subtypes of the Transport interface. Further each concrete transport object is viewed as a service provider in the design and the rest of the design has little or no knowledge of these service providers, but only of the high-level Transport service, which has methods to marshal and unmarshal method requests and responses.

The principle of least-knowledge provided by DIP makes designs and programs extensible. If a new Transport service, which is for example based on HTTPS to provide secure invocation is needed, it is just a service provider that plugs-in to the design as a service provider (subtype of Transport). Since Remote Method Invoker has no knowledge at the service provider level, this change can be done without making any changes to the Remote Method Invoker. The Remote Method Invoker is very loosely coupled to the service provider objects – IPC, TCP and HTTP – in Fig. 2. while it was strongly coupled to them in Fig. 1.

Another aspect of DIP is instantiation of the dependencies. If Remote Method Invoker were to manage its dependencies, then it would have knowledge of the service provider objects as well. The dependency is inverted out of the Remote Method Invoker and management of the dependencies is externalized. A simple method to set the dependent object is usually provided. In our example, setTransport method externalizes the dependency. The appropriate service provider transport object can be set on the Remote Method Invoker. The Remote Method Invoker can complete the remote method invocation, without knowing which particular concrete service provider it is using. It is indeed loosely
coupled with the service providers.

DIP is indeed a powerful design principle. DIP is used quite extensively in a large number of projects.

III. INVERSION OF CONTROL CONTAINERS

To facilitate DIP and to remove knowledge of service provider objects completely out of code, Inversion of Control (IOC) containers are quite useful. IOC is simple inversion of dependency. IOC containers take up the responsibility of instantiating dependencies. The dependencies at the concrete level are specified in a declarative manner and the IOC containers are capable of injecting these dependencies. We show a simplified IOC controller structure in Fig. 3. below.

![IOC controller structure](image)

Fig. 3. IOC controller structure

The objects in the program use an IOC controller as a factory to instantiate dependent objects. For example, a program that uses the Remote Method Invoker object requests the IOC Container to create a Remote Method Invoker object. The IOC Container uses dependency declarations to create the Remote Method Invoker object and its dependent objects – IPC or TCP or HTTP. Often these declarations are externalized to files so that the dependency relations can be managed independently of the code. IOC containers also provide benefits for testing. During testing either stubbed out dependent objects or objects that have simplified behavior are used. Using IOC containers makes the code independent of the context in which it is used – testing or production. The only thing that needs to change is the externalized dependency file. In production environment the actual objects are declared while in testing the stub implementations are specified.

The most prevalent IOC controller, especially in the Java community is Spring [7]. Spring support is also now available for .NET and Python projects. Dependencies in Spring are defined in application context files which are XML files. Spring requires all objects to support Java bean like behavior – to have setter and getter methods for all the dependencies. The application context files define the dependencies between beans. An example declaration in XML is shown below:

```xml
<bean name="invoker_transport"
      class="IPCTransport">
</bean>

<bean name="method_invoker"
      class="RemoteMethodInvoker">
</bean>
```

The first bean definition is for the concrete transport implementation, and the second one simply refers to this. By just changing the invoker_transport bean class, we would be able to set different dependencies on the RemoteMethodInvoker. By using the <property> XML element, which refers to the invoker_transport, Spring automatically calls setTransport on the RemoteMethodInvoker object to inject the dependency.

This dynamic dependency injection, while flexible, comes with a problem. Any incompatibility or incompleteness of dependency injection has to be discovered at runtime. In the case of static dependency injection, where the dependencies are injected by user code compiled into the application, the compiler will ensure type compatibility, but cannot ensure completeness. In this case, an incompatible, invoker_transport definition such as the one below will cause runtime problems.

```xml
<bean name="invoker_transport"
      class="Foo">
</bean>
```

In this case when creating the setTransport method on the RemoteMethodInvoker object will have the wrong Transport type, and hence a runtime error will occur. Runtime errors are very expensive and cause disruption of programs and it is needless to say that they must be avoided.

In the next section we present a method to solve this problem so that static analysis of dependency declarations in application context files can be employed to detect not only incompatible dependency injections, but incomplete ones as well.

IV. VERIFICATION OF DEPENDENCY INJECTION

For verification of dependency injection, we require two requirements to be satisfied.

Compatibility: The type $T$ defined in reference bean must be compatible with the type of property $D$ of a bean.

Completeness: For all dependencies, $D_i$ of a bean, there must be a reference dependency bean that is injected into the bean.

A dependency injection is safe, if it satisfies both Compatibility and Completeness requirements. If a dependency injection is not safe, runtime problems will occur. Failure to comply with Compatibility requirement will most definitely cause a runtime problem while failure to satisfy Completeness requirement is likely to cause a runtime failure – it depends on the execution path that of the program. The path may not require invocation on a dependent object that has not been injected.

To our knowledge ours is the first attempt for verification of dependency injection. The closest tool we have seen is BeanOH [8]. Though beanoh claims to verify Spring Verification, they take a brute force approach of instantiating the beans and checking for failures. If there are no failures,
then they claim to have verified. Our approach is different. We check for safeties of dependency injection using the Compatibility and Completeness constraints defined above.

**Compatibility Theorem:** For a bean B with a dependency D, a dependency injection with type T is compatible if and only if one of the following is satisfied:

1) T is a subtype of D
2) T is a subtype of D
3) T implements D, if D is an interface
4) T implements an interface D, and D is a subtype of D
5) T is a subtype of S, where S implements D and D is an interface.
6) T is a subtype of S, where S implements an interface D and D is a subtype of D

**Proof:** Condition 1, is valid. It T and D are the same, they are equivalent. Condition 2 is valid as subtypes are substitutable for super types. Condition 3 is valid as concrete implementations replace interface specification. Condition 4, is valid, as D, replaces D, which is replaceable by T which implements D. Condition 5 is valid; it is a combination of 2 and 3. Condition 6 is valid; it is combination of 2, and 4.

**Lemma 1:** Conditions stated in Compatibility Theorem is an exhaustive list of conditions that must be satisfied.

**Proof:** Dependencies can be injected by the following types of objects:
1) Injected objects are same as dependencies or subtypes. Conditions 1 and 2 cover this.
2) Injected objects implement dependent interfaces or subtypes of dependent interfaces. Conditions 3 and 4 cover this.
3) Injected objects are subtypes of objects that implement dependent interfaces or subtypes of dependent interfaces. Conditions 5 and 6 cover this.

It is trivial to see that Foo interface in Section III does not satisfy any of the 6 conditions stated in Compatibility Theorem and hence the dependency injection fails to satisfy Compatibility Theorem. In the case of IPCTransport class, it satisfies Condition 2. If Transport were to be an interface, then IPCTransport would satisfy Condition 3.

**Completeness Theorem:** Dependency injection for a class is complete, if and only if all dependent objects are injected and satisfy Compatibility Theorem.

**Proof:** The proof for this theorem is intuitive. All dependent objects must be materialized for an object to be useful. And the dependent objects that are materialized through dependency injection must satisfy Compatibility Theorem.

Our implementation of the verification method employs Java reflection. Though we use Java, our approach is applicable to all languages that support reflection. IOC containers will work for languages that support reflection. The algorithm for verification is given below:

**Step 1:** [Read Dependency Declarations] Read the application context file and using JAXB create the object graph representing the objects and their dependencies. JAXB traversal will yield relationships between objects and their dependencies.

**Step 2:** [Iterate] For each bean perform the Steps 3 to 5

**Step 3:** [Get Class] Load the class corresponding to the object (from the classpath)

**Step 4:** [Determine dependencies] Using Java reflection, get the attributes representing dependencies.

**Step 5:** [Check Compatibility] For each dependency determined in Step 4, load the dependency class and check for compatibility. Check Conditions 1 thru 6 till at least one Condition passes or all of them fail.

Step 4 implements Completeness Theorem and Step 5 implements Compatibility Theorem. Java Reflection API provides methods to obtain class objects and hence their definitions and API to query attributes and their types. Using Java Reflection, it is quite straightforward to implement the algorithm stated above.

The inputs to the implementation program are the set of compiled jar files or application classes and the Spring application context file containing the dependency declarations.

**V. CONCLUSION**

Our implementation has been tested on example designs in our projects. In all cases, it was observed that the Compability Theorem and Completeness Theorem satisfy the safety check for dependency injection.

The main contribution of our work is the ability to perform static analysis of dependency injection to verify safety properties. This avoids runtime errors and exceptions that programs using DIP and IOC controllers may run into.

**REFERENCES**


Shriniidhi R. Hudli is a senior-year undergraduate student in computer science at M. S. Ramaiah Institute of technology, Bangalore, India. He completed a summer project in 2010 in parallel programming on graphics cards in the Indian Institute of Science (IISc), Bangalore. In summer 2011 he was a cryptography intern at IISc. He presented a poster at the ATIPA/CRC workshop on Accelerator Technologies for High Performance Computing held in mid-2012 in Singapore. His research interests lie in parallel programming, algorithms, and artificial intelligence. Mr. Hudli has been a student member of ACM and IEEE since 2010.

Raghu V. Hudli has a Ph.D. in Computer Science from University of Nebraska, USA, 1990. He is the Managing Director of ObjectOrb Technologies Pvt. Ltd. Bangalore, India. Prior to that he worked in IBM US from 1990 to 1997. He is a guest faculty at the Indian Institute of Science. His research interests are in programming languages, distributed computing, and internet technologies. Dr. Hudli is a Senior Member of the IEEE, IEEE Computer Society and ACM.