International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2015): 6.391

# Understanding Dependencies and Analyze Dependencies with DDR Algorithm

**Vimal Don Bosco S[1], Dr. Latha R[2]**

[1]Research Scholar, Department of Computer Applications, St.Peters University, Avadi, Chennai, Tamil Nadu

[2]Professor & Head, Department of Computer Science and Applications, St.Peters University, Avadi, Chennai, Tamil Nadu

**Abstract:** ***In project source deployment to the application server, it is needed to analyze the dependencies of the source which is dependent on the defined process of the application. There are three categories of dependencies available. Dependencies can be order time, runtime, obvious, covered up, immediate, backhanded, relevant and so forth. A component that is to be reused across many different contexts should not have any context dependencies. Standard and customized interface dependencies are also involved with the depreciated method and interfaces. The DDR algorithm will help to find dependencies and resolving their dependencies for best utilization.***

**Keywords:** Dependencies, Dependency type, Dependency category, Analyzing dependencies, DDR Algorithm

## 1. Dependency

At whatever point class utilizations another class or interface B, then A relies on upon B. A cannot complete its work without B, and A cannot be reused without additionally reusing B. In such a condition the class A is known as the dependant and the class or interface B is known as the dependency. A dependant relies on upon its conditions. Two classes that utilization each other are called coupled. The coupling between classes can be free or tight, or some place in the middle. The snugness of a coupling is not parallel. It is not either free or tight. The degrees of snugness are nonstop, not discrete. Additionally describe conditions as solid or feeble. A tight coupling prompts solid dependencies, and a free coupling prompts frail dependencies, or even no dependencies in a few circumstances. The Dependencies, or couplings, are directional. That A relies on upon B doesn't imply that B additionally relies on upon A.

### 1. 1. Dependencies are bad to the Application

The Dependencies are awful in light of the fact that they diminish reuse. Diminished reuse is terrible for some reasons. Regularly reuse has a positive effect on improving speed, code quality, code coherence and so forth. Conditions can hurt reuse is best shown with beneath situation. Class Calendar Reader, that can read a timetable occasion list from an XML document. The usage of Calendar Reader is outlined beneath:

```
public class CalendarReader {
   public List readCalendarEvents
(File calendarEventFile){
    //open InputStream from File and read calendar events.
   }
}
```

The technique read Calendar Events takes a File protest as a parameter. Accordingly, this strategy relies on upon the File class. This dependency on the File class implies that the Calendar Reader is able just of perusing date-book occasions from neighborhood documents in the record framework. It cannot read schedule occasion records from a system association, a database or a from an asset on the classpath. The Calendar Reader is firmly coupled to the File class and in this way the neighborhood document framework.

A less firmly coupled execution is trade the File parameter with an InputStream parameter as beneath:

```
public class CalendarReader {
   public List readCalendarEvents(InputStream
calendarEventFile){
      //read calendar events from InputStream
   }
}
```

As you may know, an InputStream can be gotten from either a File question, a system Socket, a URL Connection class, a Class protest (Class.getResourceAsStream(String name)), a section in a database through JDBC and so on. Presently the CalendarReader has not coupled to the nearby document framework any longer. It can read schedule occasion records from a wide range of sources. With the InputStream rendition of the readCalendarEvents() technique the CalendarReader has turned out to be more reusable. The tight coupling to the neighborhood record framework has been expelled. Rather, it has been supplanted with a dependency on the InputStream class. The InputStream dependency is more adaptable than the File class dependency, however, that doesn't imply that the CalendarReader is 100% reusable. Despite everything, it cannot undoubtedly read information from an NIO Channel, for example.

### 1.2. Dependency Types

A dependency isn't just a dependency. There are a few sorts of conditions. Every sort prompts pretty much adaptability in the code. The dependency types are:

- Class Dependencies
- Interface Dependencies
- Method / Field Dependencies

Class dependencies are dependencies on classes. For example, the strategy in the underneath code box takes a String as a parameter. In this way, it relies on upon the String class.

```
public byte[] readFileContents(String fileName){
   //open the file and return the contents as a byte array.
}
```

Interface dependencies are dependencies on interfaces.

Technique or field dependencies are dependencies on solid strategies or fields of a question. It doesn't make a difference what the class of the question is, or what interfaces it actualizes, the length of it has a strategy or field of the required sort. Technique or field dependencies are regular in API's that utilization reflection to acquire its objectives.

Hibernate (a comparable ORM API) can utilize either getters or setters, or get to the fields specifically, likewise by means of reflection. That way, Hibernate has either technique or field dependencies.

Method (or function) dependencies can likewise be found in dialects that support work pointers or technique pointers to be passed as parameters to different strategies.

### 1.3 Additional Dependency Characteristics

Dependencies have other essential qualities than simply the sort. Dependencies can be order time, runtime, obvious, covered up, immediate, backhanded, relevant and so forth. These extra dependency attributes will be secured in the accompanying segments.

#### 1.3.1. Interface Implementation Dependencies

In the event that class A relies on upon an interface I, then A does not rely on upon the solid usage of I. But, A relies on upon some execution of I. A cannot complete its work without some usage of I. In this manner, at whatever point a class relies on upon an interface, that class likewise relies on upon an execution.c. In this manner, the more strategies an interface has the bigger the likelihood is that developers will simply adhere to the default usage of that interface. As it were, the bigger and more complex an interface turns into, the more tightly it is coupled to its default usage. On account of interface usage conditions, you should not add usefulness to an interface aimlessly. On the off chance that the usefulness could be typified in its own particular segment, behind its own interface, the developer should do as such.

#### 1.3.2. Compile-Time and Runtime Dependencies

A dependency that can be determined at compile time is a compile-time dependency. A dependency that cannot be determined until runtime is a runtime dependency. Compile time dependencies have a tendency to be less demanding for designers to see than runtime dependencies; however, some of the time runtime dependencies can be more adaptable.

#### 1.3.3. Visible and Hidden Dependencies

An obvious dependency is a dependency that designers can see from a class interface. On the off chance that a dependency cannot be seen from the class interface, it is a concealed dependency.

In the prior cases, the String and CharSequence dependencies of the readFileContents() techniques are noticeable dependencies. They are visible from the strategy presentation, which is a part of the class' interface. The method dependencies of the readFileContents() technique that take an Object as parameter, are undetectable. You cannot see from the interface if the readFileContents() strategy calls thefileNameContainer.toString() to get the document name, or as it really does, calls the getFileName()method.

#### 1.3.4. Direct and Indirect Dependencies

A dependency can be either direct or indirect dependency. In the event that class A utilizations a class B then A has an immediate dependency on B. In the event that A relies on upon B, and B relies on upon C, then A has an aberrant dependency on C. In the event that you cannot utilize A without B, and can't utilize B without C, then you cannot utilize A without C either. Indirect dependencies are likewise called tied dependencies, or transitive dependencies (in "Better, Faster, Lighter Java" by Bruce A. Tate and Justin Gehtland).

##### 1.3.4.1. Unnecessarily Extensive Dependencies

Once in a while components depend on upon more data than they have to complete their occupation. For instance, envision a login part for a web application. The login component needs just a client name and a password and will give back the user object, assuming any, that matches these.
In any case, the login technique now has what I call an "unnecessarily extensive dependency" on the HttpServlet Request interface. It relies on upon more than it needs to do its work. The Login Manager just needs a client name and a password to query a client yet takes a HttpServletRequest as a parameter for the login technique. A HttpServletRequest contains significantly more data than the Login Manager needs.

The dependency on the HttpServletRequest interface causes two problems:

1) The LoginManager cannot be reused (called) without an HttpServletRequest occurrence. This can make unit testing of the LoginManager harder. You will require a taunt HttpServletRequest case, which could be a considerable measure of work.
2) The LoginManager requires the names of the username and password parameters to be called "user" and "password". This is also an unnecessary dependency.

#### 1.3.4.2. Local and Context Dependencies

At the point when creating applications it is typical to break the application into minor components. Some of these components are universally useful components, which could be valuable in different applications as well. Different components are application particular and are not of any utilization outside of the application. For a universally useful segment, any classes having a place with the component (or API) are nearby. The rest of the application is the "context". In the event that a universally useful component relies on upon application particular classes, this is known as a context dependency. Context dependencies are terrible on the grounds that it makes the universally useful component unusable outside of the application as well. It is enticing to surmise that lone a terrible OO originator would make context dependencies, yet this is not valid. Context dependencies frequently happen when developers attempt to improve the plan of their application. A decent case of this is demand preparing applications, similar to message line associated applications or web applications.

#### 1.3.4.3. Standard vs. Custom Class/Interface Dependencies

By and large, it is better for a component to rely on upon a class or interface from the standard Java (or C# and so on.) bundles. These classes and interfaces are constantly accessible to anybody, making it less demanding to fulfill these component dependencies. In addition, the classes are more averse to change and cause your application to fall flat accumulation. In a few circumstances, however, contingent upon JDK classes is not the best thing to do.

## 2. Dependencies Analyze with DDR Alogrithm

The Dependencies Detecting and Resolving (DDR) Algorithm will discover the dependencies and give indicate determining structure to the deployer.

There are five objects. They are depending on some of the other objects. The objects are called here as a software package, that relies on upon another package which must introduce first or it is the base hotspot for up and coming packages. The package should be installed in correct sequence.

Take, for instance, the following scenario: Software package A depends on B and *D*. B depends on C and *E*. *C* depends on *D* and *E*. D depends on nothing, nor does *E*.

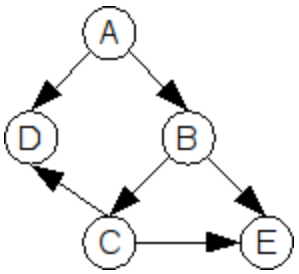


**Figure 1:** Dependency Graph

### 2.1. Representing data : Graph

So as to discover the appropriate arrangement of introducing the software package, represent data in the program. A is the simple data structure that consists of nodes (sometimes called vertices) and edges. Each software package is a node. Nodes are connected to each other by something called edges. An edge from one node to another signifies that the first node is dependent on the second. This relationship depends on is implicit in the program to resolving dependencies.

Let's define a class that can hold node information.

```
class Node:
  def __init__(self, name):
        self.name = name
        self.edges = []
def addEdge(self, node):
     self.edges.append(node)
```

This is class which can hold a name and a list of edges. We also have a method for adding edges to the node, which takes a node and adds it to the list of nodes which this node is dependent on.

Create bunch of nodes:

```
// Add vertices, e.g. equations.
g.addVertex("A");
g.addVertex("B");
g.addVertex("C");
g.addVertex("D");
g.addVertex("E");
```

Next, define the relationship between the nodes.

```
g.addEdge("A", "B"); # a depends on b
g.addEdge("A", "D"); # a depends on d
g.addEdge("B", "C"); # b depends on c
g.addEdge("B", "E"); # b depends on e
g.addEdge("C", "D"); # c depends on d
g.addEdge("C", "E"); # c depends on e
```

### 2.2. Algorithm

#### 2.2.1. Walking graph

Start with walking through the graph. For this, we need a starting point, which will be a node A. We start at A, and then we have to go through all the nodes that are connected to A. For each of those connected nodes, we have to go through that node's connected node, etc. So, we write a recursive function that calls itself for each node connected to the current node.

```
def dep_resolve(node):
  print node.name
  for edge in node.edges:
       dep_resolve(edge)
dep_resolve(a)
//cycle(s) detected
If detectCycles()//using  cycleDetector
     findCycles()  //using  cycleDetector
     while cycleVertices is not Empty
                iterate vertices from cycleVertices
                findCyclesContainingVertex(cycle)
//using  cycleDetector
                for subcycle from the cycle
                          print vertex
        remove vertex that cycle not encountered
```

The output of which is: A,B,C,D,E,E,D

#### 2.2.2. Dependency Resolution Order

We need to determine the resolution order of dependencies. Software A depends on B and D, so A can't install yet. D, however, doesn't depend on anything, so it can be installed. A software package can be installed when all of its dependencies have been installed, or when it doesn't have any dependencies at all.

Now take two arguments: node and resolved, which is the list of resolved nodes. So, they are always passed by reference. So when one of the iterations of the recursive function adds a node to the list, that change reflected in all the iterations.

#### 2.2.3. Detecting Circular Dependencies

Suppose we add the following to the dependencies.
g.add Edge("D", "B"); # D depends on B. This makes the graph as below:

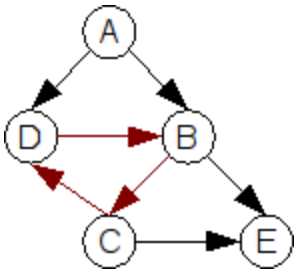


**Figure 2:** Dependency Graph: Circular Dependency

Now, Node D depends on B. But B depends on C and C depends on D and D depends on B and etc., we have now got a circular dependency which can never be solved.

#### 2.2.4. Optimization

Keep a list of all the nodes, we have seen in the program. We have previously determined that a circular reference is occurring when we see a software package more than once unless that software package has all its dependencies resolved. This means we do not need to remember the node we have seen if they are already resolved. This can save us some memory and processing time; since we only have to check a maximum of n (where n is the number of nodes in the graph) time each iteration. The approach to this is to just expel the hub from the seen list once it has been determined.

```
def dep_resolve(node, resolved, unresolved):
  unresolved.append(node)
  for an edge in node.edges:
      if edge not in resolved:
          if the edge in unresolved:
              raise Exception('Circular reference
detected: %s -> %s' % (node.name, edge.name))
          dep_resolve(edge, resolved, unresolved)
  resolved.append(node)
  unresolved.remove(node)
```

## 3. Conclusion

On the off chance that dependency found in a project or application, the user can characterize the dependency with utilization dependency categories. They are defined for resolving the dependency. The above discussion is utilized to know a few unique sorts and attributes of dependencies. In general interface, dependencies are preferable over class dependencies. Method and field dependencies can be very useful, but remember that they are also typically hidden dependencies, and hidden dependencies make it harder for users of your component to detect it, and thereby satisfy it. A component that is to be reused across many different contexts should not have any context dependencies. Meaning it should not depend on any other components in the context in which it is initially developed and integrated. The algorithm determines the dependencies and for best utilization, need to follow below rules.

- A software package can be installed when all of its dependencies have been installed, or when it doesn't have any dependencies at all.
- When a package has already been resolved, we don't need to visit it again.
- A circular dependency is occurring when we see a software package more than once unless that software package has all its dependencies resolved.

## References

[1] Vincenzo Musco, Martin Monperrus, and Philippe Preux, "Generative Model of Software Dependency Graphs to Better Understand software Evolution," Cornel University Library, arXiv.org, Peper Id: 1410.7921, Vol. 2 pp. 1-10, 2015.
[2] Pei Wang, Jinqiu Yang, Lin Tan, Robert Kroeger and J. David Morgenthaler, "Generating Precise Dependencies for Large Software", IEEE, pp. 47-50, 2013.
[3] Rantneshwer and Anil Tripathi, "Dependence Analysis of Component Based Software through Assumption", International Journal of Computer Science Issues, Vol. 8, pp. 149-159, 2011.
[4] Neeraj Sangal, Ev Jordan, Vineet Sinha and Daniel Jackson, "Using Dependency Model to Manage Complex Software Architecture", Object-Oriented Programming, System, Languages and Applications (OOPSLA), pp. 1-10, 2005.
[5] Martin P. Robillard, "Topology Analysis of Software Dependencies", ACM Transactions on Software Engineering and Methodology, Vol. 17, No.4, Article 18, 2008.
[6] Matthias Blume, "Dependency Analysis for Standard ML", ACM Transactions on Software Engineering and Methodology, Vol. 21, No.4, pp.790-812, 1999.
[7] Indumathi C P and Selvamani K, "Test Cases Prioritization using Open Dependency Structure Algorithm", International Conference on Intelligent Computing, Communication and Convergence, pp. 250-255, 2015 .
[8] Pradip S. Devan and R.K. Kamat, " A Review – LOOP Dependence Analysis for Parallelizing Compiler", International Journal of Computer Science and Information Technologies, Vol.5(3), pp. 4038-4046, 2014.

## Author Profile

**Vimal Don Bosco S,** Research Scholar, St. Peters University, Avadi, Chennai.