Optimizing Dependencies - A solution to Failure Proneness of software

1.Dr.G.AppaRao,

Professor and HOD, CSE,GITAM Institute of Technology,
Vishakapatnam, India.

2.P. V. N. Santhi Swaroop,

M.Tech(SE), GITAM Institute of Technology,
Vishakapatnam, India.

Abstract:

With the emerging technologies, large complex software isbeing built in terms of modules. Although the term 'module' is introduced to reduce the complexity in maintenance, but the dependencies among the modules is causing erroneous systems in existence.There is a need to architect/developer to look at these dependencies to be optimized. Predicting the dependencies is only possible by measuring the metrics of the system. Here in this paper we specify the different types of dependencies and notation of identified dependencies. A clear definition of these metrics is also provided.

Keywords: Dependency, Modules, Metrics, Coupling.

1. INTRODUCTION:

The term 'module' introduces a specific concept or a procedure that groups all the similar tasks as one. With this modularity large complex software is developed so that maintenance comes easy. But there are situations in demands that raise the problems over modules. There is a situation that one module depending on another module either physically or logically. The dependency may not be identified initially in developing stages. But later would cause an unpredictable failure to software.

However, the dependencies among modules would make solid problems in the system maintenance. We use UML diagrams to understand the dependencies among modules, but it failed in clear representation of these dependencies for complex software which is having more number of modules. Reducing the number of modules is not an appropriate solution to this problem. Here in this paper, we report and guide the architect/developer in identifying the dependencies, and to resolve them. Limited work has been done in identifying the relationship between the dependency and failure that caused. Within this aspect, we worked to identify and notify the dependency.

It has long been established that many software faults are caused by violated dependencies that are not recognized by developers designing and implementing a software system. The failure to recognize these dependencies could stem from technical properties of the dependencies themselves as well as from the way development work is organized. In other words, two dimensions are at play – technical and organizational. On the technical side, the software engineering literature has long recognized call and dataflow syntactic relationships as an important source of error. Research in the software evolution literature has introduced a new view on technical dependencies among software modules.

In software engineering measurement plays a major role in solving a problem. In order to predict the dependencies at the initial stage of development, we need to measure the different counts for understanding the attributes of the system. This phenomenon is simply termed as "Metrics". The detailed description of metrics is provided in the following sections.

2. Definition of dependencies:

Dependency is defined as the reflection on one module by the change of another module. This can be studied under 2 perspectives.

- Non-Technical dependency
- Technical Dependency

2.1 Non-Technical Dependency:
This Dependency comprises the terms workflow and coordination. Workflow represents the active sliding from one state to another state from elicitation to deployment. These are temporal basis and sequential basis. In a traditional view, the workflow transfers the work from developer 'i' to developer 'j' at some point in development process. Whereas coordination determines the additional work related dependencies. That is, it shows the impact on developer 'j' due to the changes made by developer 'i'.
2.2Technical Dependency:
Gall and colleagues introduced the idea of "logical" dependencies by showing that source code files that are changed together can uncover dependencies among thosefiles that are not explicitly identified by traditional syntactic approaches. Past work has also examined aspects of the relationship between logical dependencies and failures in software systems. Eick and colleagues used increases of such logical coupling as an indicator of "code decay". Graves and colleagues showed that past changes are good predictors of future faults, and Mockus and Weiss found that the spread of a change over subsystems and files is a strong indicator that the change will contain a defect.
We focus on technical dependency in 2 major perspectives.

- Logical
- Physical

Logical:
This perspective is a challenging one, where the code of software is being changed in one module and having impact on another module. For example, global data initialized in one module may need to be changed in another module. But as this is global data, the change in one module reflects in other module also.
Physical:
This perspective is drawn in terms of files, directories and supporting library files etc. That is, software is executable if and only if all or part of its components/files exists in right place (directory). If any file/component is misplaced or missing, then the remaining part of software is useless. Hence, the software and its execution are depending on the physical component's structure.

3. Optimizing work dependency:
The coordination among the developer/architect is not in proper manner because of the following reasons

- Schedule busy
- Misunderstanding of system
- Personnel misunderstandings

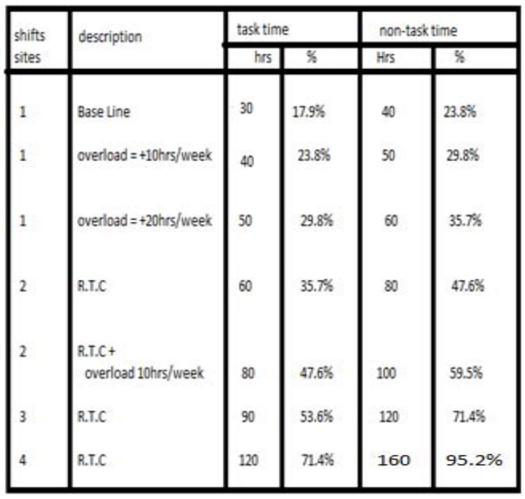
3.1 Schedule busy:
It is the place where coordination fails because; a developer is allotted for a specific time period to accomplish his/her work. But this time period is not for non-task activities which are for ex, unofficial meetings, formal and informal activities that take place before and after meetings, casual holidays, filling out sheets, etc. due to lack of schedule for these activities, the developer could get tired of working in tight schedule such that he/she may not communicate with co-developers or even customer.
3.2 Misunderstanding of system:
This is very often because non-uniform language in documentation makes developers misunderstanding.
3.3 Personnel misunderstandings
The irreducible inter-dependency among the software developers and tasks can be thought of a "Distributed Constrain Satisfaction Problem", where coordination is only the solution.

We are going to use the general concept "Round The Clock Development" (RTCD) for optimizing the work dependencies in terms of task activities and non-task activities.

"Round The Clock Development", is a general term which indicates that usage of 24 hours for working on a particular task.

Why we are going to RTCD? In order to have a large production within a short duration of period, we commit to work for complete day and night in shift procedure. But committing to 24Hrs work is not only a solution to fit the tight schedule. The reason is the employ/programmer is not assigned work in terms of 'task' and 'non-task' activities.

The following tabular form represents the increase of number of shifts or sites from top to bottom with respect to task time and non-task time. This phenomenon is state as "Calendar Efficiency".



shifts sites	description	task time		non-task time	
		hrs	%	Hrs	%
1	Base Line	30	17.9%	40	23.8%
1	overload = +10hrs/week	40	23.8%	50	29.8%
1	overload = +20hrs/week	50	29.8%	60	35.7%
2	R.T.C	60	35.7%	80	47.6%
2	R.T.C + overload 10hrs/week	80	47.6%	100	59.5%
3	R.T.C	90	53.6%	120	71.4%
4	R.T.C	120	71.4%	160	95.2%

Tabl Figure 1: Calendar Efficiency with RTCD

As shown in the above table, we presented a plan to utilize calendar time properly by performing following calculations.

Hours per week = 24*7 = 168hrs.

An employ can work for 8hrs and 5days for a week. So that totally 40hrs per week.

$$\text{Utilization} = \frac{\text{actual hours per week done}}{\text{Hours available per week}}$$

$$= \frac{40}{168} \implies 23.8\% \qquad \text{------(1)}$$

As 40hrs/week mentioned above is the time to accomplish the task activities. Here time is not mentioned for non-task activity.

Hence we state that 30 hrs/week is accomplished for only task activities and the rest of the 10 hrs is for completely non-task activities only. This time should be scheduled necessarily. Being considering the non-task activity, the time to complete task activity is reduced. But it is not sufficient to work with. Therefore as a first trial some overtime (overload) is added with another 10hrs per week. So that totally we get 50hrs per week in which we have 40hrs per week to proceed with task activity. But practically it is impossible for an employ/developer to work more than 8hrs per day. Hence we advice to increase number of working shifts or sites/locations such that parallel work is achieved in terms of modules.

In the table we kept an increase with in the number of sites/shifts up to 4 (say), while applying the R.T.C development pocess. Now we observe that totally 120hrs per week is being in utilization for task activites and 40 hrs/week for non-task activities.

With this calendar efficiency, both task and non-task activities are scheduled so that no overload or tight schedule for a developer. Now he/she can communicate with the people/customer/co-developers in time.

4. Optimizing technical dependency

With the earlier discussion about technical dependency we have 2 perspectives.

Logically we founded the following dependencies

4.1Content logical dependency:

It is when one module modifies or relies on the internal workig of another module.

Eg: accessing local data of aother module.

Solution:

- Use the read only interface design pattern.
- Make all the instance variables private.

4.2Common logical dependency:
It is when two modules share the sae global data changing all the modules using it.
Solution:

- Define the global variable in configuration files and use the global variable references in rest of the modules.

4.3External logical dependency:
They occur when two modules share an externally imposed data fromat, communication protocol or device interface. A form of logical dependency in which a software component has a dependency to applcation development using a third party or a particular type of hardware.
Solution:

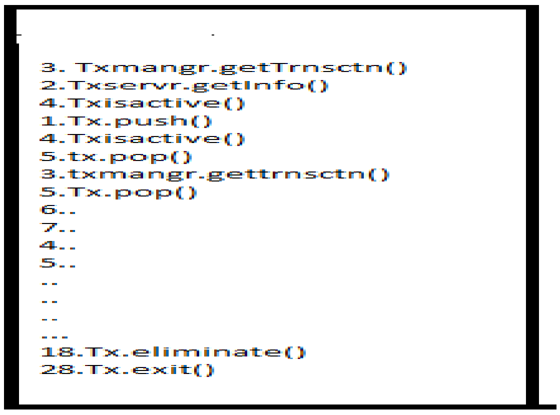
- Reduce the number of palces in code where such dependency exist.

4.4Data logical dependency:
It is when modules share the data through for eg parameters, each datumn is an elementary piece, and these are only data shared.
Solution:

- Pass a few arguments each containing more abstract information

5.Representation of dependencies syntactically.

Dependencies between the modules are identified by using the tracing tools, such that the sequence of execution of modules is traced.
Follow the steps to represent the dependencies syntactically:
Step 1: trace the program by using a tracing a tool.
Step 2: mark all the events that occurred in a sequence with unique numbers.
Step 3: cluster the similar events that perform on a single cohesive task.
Step 4: these clusters are said to be arranged mathematically in closure set forms.
Each closure set is termed as a 'concept'.
Step 5: each concept is again numbered so that it can be easy to show the relation ship between the modules.
Each concept may comprize of one or more repeated events since dependency occurs due to the usage of module references more times.
Step 6: identify the generators.
An event which impacts the other event is termed as generator.
Syntactically, when an event occurs then there must occur some set of events.



```
3. Txmangr.getTrnsctn()
2.Txservr.getInfo()
4.Txisactive()
1.Tx.push()
4.Txisactive()
5.tx.pop()
3.txmangr.gettrnsctn()
5.Tx.pop()
6..
7..
4..
5..
..
..
..
...
18.Tx.eliminate()
28.Tx.exit()
```

Figure:2 set of events with unique number.

A closure set for the above figure comprises
{ 1,2,3,4,5,6,………18, 28}
Say that whenever the event 2 is raised then the events 1,3,4,5,6,…18,28 also exists. Thus here 2 is a generator for all other events.

2 ---> {1,3,4,5,6,......18,28.} ----------(2)

If two or more events raising some set of other events then we use logical AND.

$2 \wedge 3 \text{--->} \{1, 4,5,6,......18,28.\}$ ---------(3)

We also use logical OR to represent either one or the other event may raise the other set of events.

$2 \vee 3 \text{--->} \{1, 4,5,6,......18,28.\}$ --------(4)

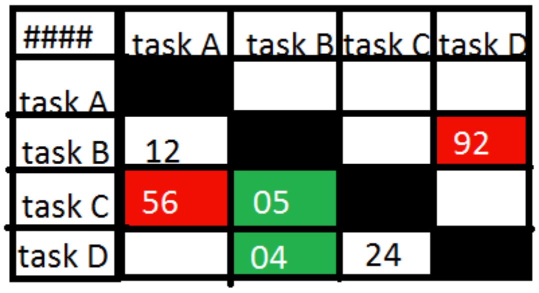
Causal dependency: it determines which event must precede in a sequence.
Let i, j, k be three events such that i should occur first then j and atlast k.
This constraint is termed as causal dependency. This concept involves time.
This is syntactically represented as

$i < j < k.$

Plausible dependency: it states a constraint that event i precedes j, but j does not precede i under any circumstances.

6.Representation of dependencies graphically:

Representing the dependencies graphically improves the efficiency of identifying the dependencies for large complex software.
The phenomenon of representing graphically is defined as Inter-Dependency Matrix (IDM).
IDM focus on representing which module relies on which module. It also focus on architecture patterns and proposing constraints according to the patterns.
IDM determines whether the module can use the reference of another module or not.



####	task A	task B	task C	task D
task A	■			
task B	12	■		🟥 92
task C	🟥 56	🟩 05	■	
task D		🟩 04	24	■

Figure:3 Inter-Dependent Matrix

The above diagram is a depiction of IDM, in which 4 background colors are used for the following purposes.
Black: should not use the references of each other modules.
Green: can use the references of each other modules. But the references used must be less than 10 (threshold value). It indicates very less dependent.
White: can use the references of each other modules. But the references used must be greater than 10 and less than 30 (threshold value). This can be an optimal value for large projects.
Red: can use the references of each other modules. Here the references are more than 30 (threshold value). It indicates that the respective two modules are highly dependent among each other.
From the above diagram,
D→B, task D is highly dependent with using 92 references of task B.
A→B, task A is optimally dependent wih using 12 references of task B.
B→D, task B is less dependent with using 4 references of task D.

7. Predicting the dependency using Metrics:

The traditional syntactic view of software dependency had its origins in compiler optimizations, and focused on control and dataflow relationships. This approach extracts relational information between specific units of analysis such as statements, functions or methods, and source code files. Dependencies are discovered, typically, by analysis of source code or from an intermediate representation such as byte codes or abstract syntax trees. These relationships can be represented either by a data-related dependency (e.g. a particular data structure modified by a function and used in another function) or by a functional dependency (e.g. method A calls method B).
The work by Hutchens and Basili and Selby and Basili represents the first use of dependency data in the context of a system's propensity for failure. They found that routines and subsystems with lower coupling were less likely to exhibit defects than those with higher levels of coupling. Similar results have been reported in object-oriented systems. Chidamber and Kemerer proposed a set of measures that captures different aspects of the system of

relationships between classes. Briand and colleagues found that the measures of coupling proposed by Chidamber and Kemerer were positively associated with failure proneness of classes of objects.

7.1C.K. Metrics

C.K metric suite offers informative insight into whether developers are following object oriented principles in their design. They claim that using several of their metrics collectively helps managers and designers to make better design decision. CK metrics have generated a significant amount of interest and are currently the most well-known suite of measurements for OO software. Chidamber and Kemerer proposed six metrics; the following discussion shows their metrics.

a. Weighted Methods per Class (WMC)

The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods (method complexity is measured by cyclomatic complexity). The second measurement is difficult to implement because not all methods are accessible within the class hierarchy because of inheritance. Consider a Class C_1 with Methods $M_1 \ldots M_n$ that are defined in the class. Let $c_{1\ldots} c_n$ be the complexity of the methods. Then:

$$\mathrm{WMC} = \sum_{i=1}^{n} c_i$$

The number exity of the methods involved is a predictor of how much time and effort is required to (:lass. The larger the number of methods in a class, the greater the potential impact on c ıerit all of the methods defined in a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. These metric measures are understandability, reusability, and maintainability.

In figure 3 , WMC for Library is 3.

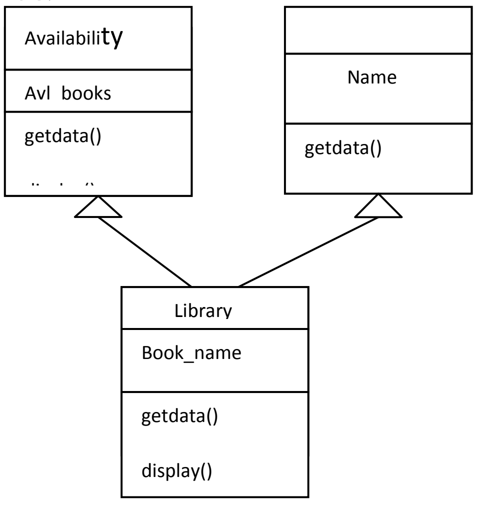


Figure4: Class Diagram of library information system

b. Depth of Inheritance (DIT)

The depth of a class within the inheritance hierarchy is maximum number of steps from the class node to the root of the tree and is measured by number of ancestor class. The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved. The deeper a particular class is in the hierarchy, the greater potential reuse of inherited methods. For languages that allow multiple inheritances, the longest path is usually taken.

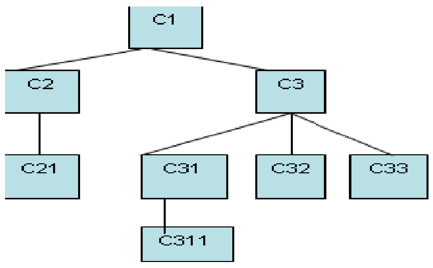


Figure 5: Depth of Inheritance

c. Number of Children (NOC)

The NOC metric equals to number of immediate subclasses subordinated to a class in the class hierarchy. Greater the number of children, greater the reuse, since inheritance is a form of reuse. The greater is the number of children the greater is the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of sub classing .The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.

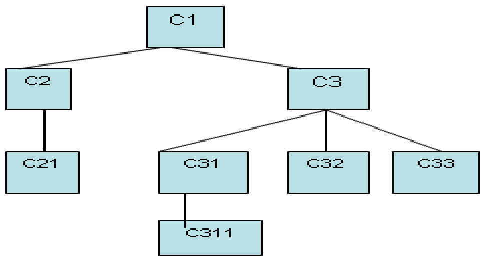


Figure 6: Number of Children

In the preceding example the NOC for C3 is 3 i.e. C31, C32, C33.

Coupling Between Object Classes (COB)

CBO for a class is a count of the number of other classes to which is coupled. CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one uses methods or instance variables of another. Excessive coupling between objectclasses is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. Direct access to foreign instance variable has generally been identified as the worst type of coupling. The value of
Metric CBO for class Library is 2 and for class Author and Availability is 0

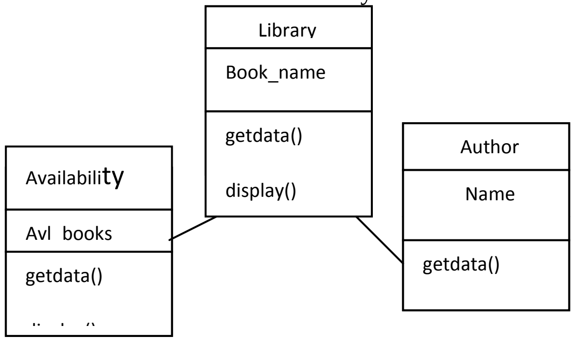


Figure7:Classdiagram of availability information system

d. Response for a Class (RFC)

The response set of a class is a set of methods that can potentially be executed in response to a message received by and object of that class. RFC measures both external and internal communication, but specifically it includes methods called from outside the class, so it is also a measure of the potential communication between the class and other classes.
It is given by
RFC=|RS|, where RS, the response set of the class, is given by

$$RS = M_i \cup_{all\ j} \{R_{ij}\}$$

whereMi= set of all methods in a class (total n) and Ri = {Rij} = set of methods called by Mi.
RFC is more sensitive measure of coupling than CBO since it considers methods instead of classes

e. Lake Of Cohesion in Methods (LCOM)

The LCOM is a count of the number of method pairs whose similarity is 0 minus the count of method pairs whose similarity is not zero. The larger the number of similar methods, the more cohesive the class, which is consistent with traditional notions of cohesion that measure the inter-relatedness between portions of a program. If none of the methods of a class display any instance behavior, i.e., do not use any instance variables, they have no similarity and the LCOM value for the class will be zero.
Consider a class C1 with n methods $M_1, M_2,\ldots,M_n$. Let (I_j) = set of all instance variables used by method Mi. There as n such sets $\{I_1\},\ldots\{I_n\}$. Let $P = \{(I_i, I_j) \mid I_i \cap I_j = 0\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq 0\}$. If all n sets $\{(I_i),\ldots(I_n)\}$ are 0 then P=0

LCOM=|P| - |Q|, if |P|>|Q|
= 0 otherwise

8. CONCLUSION:

A clear understanding of dependencies is an essential part of system maintenance. In this paper we presented the definitions of dependencies which are most suitable to many applications. The metrics provided in this paper makes the designer to understand the complexity of the system. Supporting solutions are proposed for the dependencies just to optimize them but not to remove them. Representation of identified dependencies also helps the developer/designer to find the physical and logical dependencies among the software modules.

REFERENCES

[1] Curtis, B., Kransner, H. and Iscoe, N. A field study of software design process for large systems. Comm. of ACM, 31, pp. 1268-1287, 1988.
[2] Crowston, K.C. Toward a Coordination Cookbook: Recipes for Multi-Agent Action. Ph.D. Dissertation, Sloan School of Management, MIT,1991.
[3] Burt, R.S. Structural Holes: The Social Structure of Competition. Harvard University Press, 1992
[4] Chidamber, S.R. and Kemerer, C.F. A Metrics Suite for Object-Oriented Design.IEEE Trans. on Soft.Eng., 20, pp. 476-493, 1994.
[5] Baldwin, C.Y. and Clark, K.B. Design Rules: The Power of Modularity. MIT Press, 2000.
[6] Briand, L.C., Wust, J., Daly, J.W. and Porter, D.V. Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems.The Journal of Systems and Software, 51, pp. 245-273, 2000.
[7] Atkins, D. Ball, T., Graves, T. and Mockus, A. Using version control data to evaluate the impact of software tools: A case study of the version editor. IEEE Trans. on Soft.Eng., 28, pp. 625-637, 2002.
[8] Eick, S.G., Graves, T.L., Karr, A.F., Mockus, A. and Schuster, P. Visualizing Software Changes. IEEE Trans. on Soft. Eng., 28, pp. 396-412, 2002
[9] de Souza, C.R.B. On the Relationship between Software Dependencies and Coordination: Field Studies and Tool Support. Ph.D.dissertation, Donald Bren School of Information and Computer Sciences, University of California, Irvine, 2005.
[10] Cataldo, M., Wagstrom, P, Herbsleb, J.D. and Carley, K.M. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools.In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'06), 2006, pp.353-362.
[11] Cataldo, M. Dependencies in Geographically Distributed Software Development: Overcoming the Limits of Modularity. Ph.D. dissertation,Institute for Software Research, School of Computer Sciences, Carnegie Mellon University, 2007.
[12] Cataldo, M., Bass, M, Herbsleb, J.D. and Bass, L. On Coordination Mechanism in Global Software Development. In Proceedings of the International Conference on Global Software Engineering (ICGSE '07), 2007, pp. 71-80.
[13] Eaddy, M., Zimmermannn, T., Sherwood, K.D., Garg, V., Murphy, G.C., Nagappan, N., Aho, A.V. 2008. Do Crosscutting Concerns Cause Defects?IEEE Trans. on Soft.Eng., 34, pp. 497-515, 2008.

About Authors

G. Apparao is working as a Professor and Head, Department of Computer Science and Engineering at GITAM Institute of Technology, Vishakapatnam, India.. He has published and presented good number of technical and Research papers in National , International Conferences and International Journals like Ontologies of Machine Intelligence in Fraud Detection – A Legal Modeling Approach. His main research interests are Datamining, Artificial Intelligence, Operating Systems.

Santhi Swaroop P. V. N is currently pursuing M.Tech in Software Engineering from GITAM Institute of Technology, Vishakapatnam, India. He pursued B.Tech from CR Engineering collage, Tirupathi. His research interests are Software Reliability, Software Design Patterns and DataMining