The Dependable Dependencies Principle

Jason Gorman, Codemanship

April 2011

jason.gorman@codemanship.com

Abstract

For over 15 years we have known of design principles to help us manage dependencies in software to limit the impact of making changes. Another consequence of dependencies has been overlooked, namely the relationship between dependencies and risk of failure in our code. The more depended-upon code is, the greater the potential impact of its failure. We should therefore desire that code this is more depended-upon be more reliable. This paper explores the relationship between dependencies and reliability, and proposes a new design principle, with a first attempt at an accompanying set of metrics, to help us limit the impact of failure.

Introduction

In his papers on object oriented design principles[1], Robert C. Martin proposes that packages should depend on packages that are more "stable" than they are (the **Stable Dependencies Principle**). The more a package is depended upon, the greater the potential impact of making changes to it, and therefore the lower the likelihood that changes will be made, given finite time and resources and a need to prioritise changes that is the reality of all software development efforts.

Stable packages also depend less (or not at all, if said to be 100% stable) on other packages, which means that changes to other parts of a system are less likely to affect them.

For these reasons, Martin concludes that it is better to have packages depend on packages that are already depended upon by other packages but that have few, if any, outgoing dependencies themselves.

[1] OO Design Quality Metrics, 1994, Robert C. Martin

Observing this principle has the effect of limiting or localising the ripple effect in code[2], and lowering the overall cost of making changes to the software.

As a consequence of favouring stable dependencies, Martin also proposes another design principle. In order to accommodate change to stable packages, we must be able to extend them more easily so that behaviour can be added or modified without modifying the original package (the *Open-Closed Principle* – modules should be open to extension but closed to modification). In many OO programming languages, classes are easier to extend the more abstract they are. Hence he calls this the **Stable Abstractions Principle**.

Arguably, since the mechanics of dependencies and propagation of change through a network are ubiquitous and self-similar at any scale, these principles can be applied at multiple levels of code organisation and reuse ("units of code"): systems, packages, classes and functions/methods.

But change is not the only thing that can propagate through dependencies in our code. *Failures* can also propagate. By "failure", we mean when a function fails to execute correctly, satisfying its obligations in a client-supplier interaction.

A chain is only as strong as its weakest link. When any function fails in the call stack, usually the whole collaboration fails and the system behaviour fails. Therefore, failure of one function causes failure of a calling function, which causes failure of the functions that call them, and so on.

It may be that a function fails to execute successfully because of some hardware error or other unpredictable environmental factor. More often we find that functions fail because of programming errors. The code in the function is, in fact, incorrect.

The more depended-upon a function is, the wider the likely impact of failure will be. It follows that, as well as favouring dependencies on units of code that are more stable and more abstract, we should **favour dependencies on units of code that are less likely to fail**.

Here I propose a new design principle for any kind of software – be it object oriented or otherwise.

Dependable Dependencies

The Dependable Dependencies Principle
The more depended-upon a unit of code is, the less likely it must be to fail

Measuring Potential Impact of Failure (Rank)

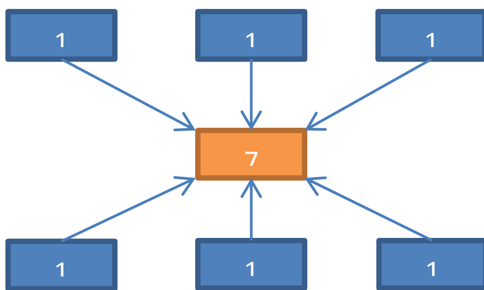
[2] Ripple Effect in Object Oriented Systems, Nashat Mansour & Hani Salem, 2007

Martin's principles deal with syntactic coupling between classes and packages – i.e., direct dependencies.

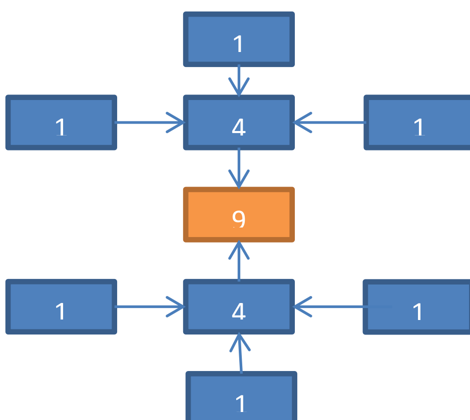
The DDP deals with *semantic coupling*, where failures can propagate along indirect dependencies all the way up the call stack. In DDP, a low-level function and a function on the system boundary can be indirectly coupled but directly affected by failure. Therefore, when assessing the structural importance of a unit of code, we must take into account direct and indirect dependencies on it.

This could be given by calculating the *rank* of that unit within the dependency network (in a similar way to calculating page rank within a network of linked web pages.)

Many algorithms exist for calculating the rank of a vertex within a network, but here I'll use a very simple algorithm to illustrate the principle. This technique is similar to the original published Google PageRank algorithm[3], only without a damping factor.

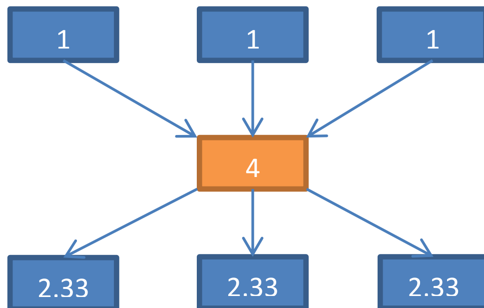


By default, every unit of code in the system is given an initial rank of 1. They convey a share of their rank on to units of code they depend upon. So a method called by 6 methods with rank 1 would end up with a rank of 1 + (6 x 1) = 7, for example.



[3] PageRank Citation Ranking: Bringing order to the web, Lawrence Page, Sergey Brin 1999

By this approach, a method that was only called directly by two other methods, but those methods had a rank of 4, could then attain a rank of 9. In this example, the method denoted by the orange box is just as depended upon – directly and indirectly – as the one in the previous example.



When a method calls more than one other method, it confers a proportional share of its own rank. For example, a method with rank 4 calling 3 other methods conveys rank of 4÷3 to each of those methods it depends upon.

By iteratively applying this calculation to the entire network of dependent units of code, we can arrive at a **rank (R)** for each. Some static code analysis tools like NDepend include a rank metric for methods, classes and packages, so we will not go into unnecessary detail about an implementation for this algorithm here.

The DDP states that units with a higher R would need to be more reliable.

Measuring Probability of Failure

Calculating reliability is not as straightforward. But we can draw some general conclusions from what we know about software, and about technology in general.

Statistically, code that is more complex is more likely to have defects. Studies that compare the distribution of defects in code show a string correlation with code complexity[4].

So one factor we are very likely to want to take into account is the complexity of a unit of code, since that has proven to be a reliable predictor of the likelihood of defects.

Cyclomatic Complexity [5](a measure of the number of executable paths in a program of function) has shown itself to be a decent measure of code complexity through several decades of research and application, which is why I propose to use it here.

[4] "The Role of Empiricism in Improving the Reliability of Future Software", Les Hatton 2008

If a unit of code has a higher cyclomatic complexity, we might desire that it have a lower rank within the system.

The higher the rank of a unit of code, the lower we might wish its cyclomatic complexity to be, such that we can strike a balance between the two such that high-ranked units with high cyclomatic complexity would present a high overall risk of failure (F) calculated as the product of rank and complexity.

Overall Risk Of Failure
$F = R \times C_C$

But this is not a complete picture. A method with CC = 10 but very high test assurance may present less of a risk of failure than a method with CC = 5 and no test assurance.

To present a more realistic picture of the risk a unit of code represents, we must combine test assurance and complexity. The developers of the CRAP4J[6] code analysis tool propose a measure called the C.R.A.P. ("Change Risk Anti-Patterns") metric, which combines cyclomatic complexity and test coverage at the method level to give an overall assessment of risk.

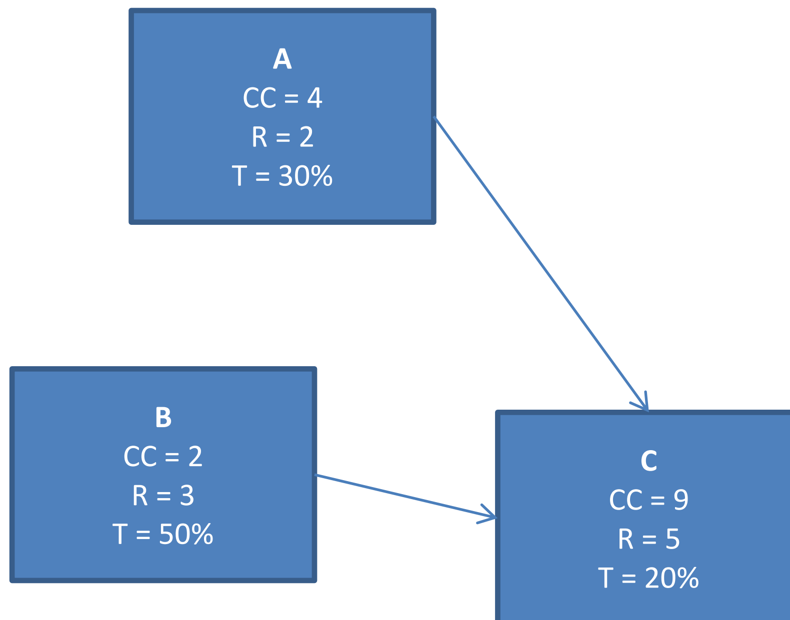
C.R.A.P.
$CRAP = (C_C^2 \times (1 - T)^3) + C_C$

Where T is the percentage of code covered by tests. (Which I acknowledge is not an ideal measure of test assurance, but is certainly an accurate measure of *lack* of test assurance, since code that is not executed in a test is not tested at all.)

We can update our formula for F to use the CRAP metric, such that:

Overall Risk of Failure
$F = R \times CRAP$

[5] "A Complexity Measure", McCabe 1976
[6] www.crap4j.org, Alberto Savoia, Bob Evans 2007



A has CRAP = 9.5, B has CRAP = 2.5 and C has CRAP = 50.5

Therefore:

F(A) = 19

F(B) = 7.5

F(C) = 252.4

Conclusion

Unit of code C has the highest rank in the system and is also the least dependable, presenting the highest risk to the system as a whole.

To fix this problem, we must either make C simpler (e.g., by refactoring) and/or improve test assurance for C.

In practice, C could be a method that plays a critical role in the system through direct or indirect reuse, or it could be a class in our system, or a package, or, indeed, a system in its own right that plays a critical role in a larger enterprise architecture.

Experience with dozens of software-intensive organisations suggests that they tend to be unaware of this picture, and often focus more time and effort on the quality of less critical components. This is often driven by an external value attached to a system or component by the business, and for a more complete picture this external value could also be taken into account by factoring in the relative importance the business places on features and then tracing that through the call stacks for well-defined usage scenarios, seeing which code gets executed in the process. But such a discussion

is beyond the scope of this paper.

Other factors for reliability could be considered, too. For example, a method may be short and simple but may rely on resources that are prone to failure, such as network connections or external peripheral devices like disk drives, sensors, barcode scanners and the like.

Finally, a more rounded view of test assurance could be given by factoring in other testing techniques, such as inspections, mutation testing and so on.

In all these cases, however, automated analysis of large, complex systems would prove difficult and costly, so it suits for the purposes of simplicity and cost constraints to end for now with the measures proposed in this paper, allowing for testing by their application, and for further work to be done in the future.