An Investigation into the Effects of Code Coupling on Team Dynamics and Productivity

James Westland Cain
Quantel Limited
Newbury, UK
james.cain@quantel.com

Rachel Jane McCrindle
Department of Computer Science
The University of Reading, UK
r.j.mccrindle@reading.ac.uk

Abstract

During the past three decades a number of theories have been proposed to explain the idiosyncrasies of software development as a team activity. These theories variously relate to: adding more programmers to a late project makes it later (Brooks); the structure of the system mirrors the structure of the organization that designed it (Conway); software modules are a responsibility assignment (Parnas) and one must consider stability and responsibility during dependency analysis (Martin). This paper compares and combines these theories into a coherent model of software development that links software coupling and dependency management with team productivity.

As a practical test of this model, the paper then investigates the effects of coupling in two large commercial systems (both measured in person decades of effort). It achieves this by using the VCML Views visualisation technique, developed by the authors, to expose the system wide coupling found in the code and how this coupling develops during the lifetime of a project. It then compares the resultant VCML views with simple attributes of the two projects, such as programmer numbers and programmer productivity to derive a set of important conclusions.

In particular, it finds that unmanaged coupling within the code is a good indicator of potential productivity bottlenecks; that the number of programmers on a project is not necessarily a good indicator of programmer productivity; and that the architecture of a software system can radically alter the number of programmers that can effectively work together on a system.

Keywords

Information Hiding, Dependency Management, Coupling, Large Scale Software, Visualisation, Object Oriented Systems.

1. Theories of Software Architecture

When Fred Brooks published his theories of project management in the Mythical Man Month, he forever linked the addition of developers to a late project with failure [2]. The basis for Brook's Law was that each programmer added to a team multiplied the communication burden that each other programmer would have. Further, he stated that the work on a software project is not easily partitioned into isolated tasks, and that this lack of parallelism means that programmers conflict with each other and impede each other's progress.

Three years prior to the publication of the Mythical Man Month, David Parnas had presented his ideas on Information Hiding [14]. In his paper he defined a software module as "a responsibility assignment rather than a subprogram," driving home the idea that modular design enables decisions about the internals of each module to be made independently. That is, the aim of structure in a program is to support co-ordination of the development work.

In the previous decade, Melvin Conway proposed what has since become known as Conway's Law —that the structure of the system mirrors the structure of the organization that designed it [6]. He states that architecture is about relationships between system parts and therefore is also about relationships between people. He further argued that the organisational arrangements could only be optimised with respect to "the system concept in effect at that time." Simply put, this means that if the system architecture is unstable the organisation cannot sensibly isolate software modules in order to allow parallel team development.

The views of Parnas and Conway are echoed by the findings of Herbsleb and Grinter who studied the effects of software architecture on multi-site development teams [10]. They noted that instability of the software architecture creates an enormous need for communication.

Proceedings of the 26 th Annual International Computer Software and Applications Conference (COMPSAC'02)
0730-3157/02 $17.00 © 2002 **IEEE**

They stated, “We believe that the qualitative evidence from our case study strongly supports Conway’s and Parnas’ positions that the essence of good design is facilitating co-ordination among developers.”

Twenty years after the Mythical Man Month, Brooks too admitted the following: “I dismissed Parnas's concept as a ‘recipe for disaster’ in Chapter 7. Parnas was right and I was wrong. I am now convinced that information hiding, today often embodied in object-oriented programming, is the only way of raising the level of software design” [3].

Bob Martin has worked for years to document the *principles of object oriented class design*, that he states are all driven by the need for dependency management [12,13]. He lists the symptoms of rotting design as: rigidity, fragility, immobility and viscosity and attributes these symptoms to improper dependencies between the modules of the software. He states that, in order to forestall the degradation of the dependency architecture, the dependencies between modules in an application must be managed. Martin further states that the way to measure dependencies between modules is to measure the coupling between modules. In this respect he distinguishes between *afferent coupling* & *efferent coupling*.

Briand, Daly & Würst review coupling measures and define the terms *client class* and *server class* [1]. In the context of coupling, they see it as useful to distinguish the class that is using another class, and the class that is being used. They refer to the using class as the *client class*, and to the used class as the *server class*. Linking this to the work of Martin gives *afferent* coupling as being a measure of the number of references to a class and *efferent* coupling as a measure of the number of other class that the class uses. That is *afferent coupling* can be seen as the number of times the class is the server in a coupling relationship and *efferent coupling* as the number of times the class is the client in a coupling relationship.

Martin argues that high coupling is not always bad. Abstract interfaces can quite reasonably have a high number of references to them, making them responsible, as long as they are stable. The parts of a system that are unstable (i.e. those parts that are highly change prone) must be irresponsible, so that they can change without causing large ripples throughout the rest of the code. This is *information hiding* at work, and as long as all the shared system parts are not changing, separate programmers can check-out from the revision control system the parts of the system that require work without causing bottlenecks or code merges.

Thus to facilitate dependency management, it is not coupling that needs to be measured, but the direction of coupling. The combination of a high number of references (implying responsibility) and a high uses measure (implying instability) exposes poor information hiding, and identifies a bottleneck for parallel team development.

2. Combined Model: Unmanaged Coupling Impedes Team Development

Figure 1 shows a series of diagrams illustrating the theories that have been reported from the literature. These diagrams build into a combined model of dependency management and demonstrate the effect that module assignment and coupling can have on team dynamics and productivity.

Small C++ programs can be kept within one source file. However this strategy does not scale well. To facilitate team development (in the most general sense), program parts are assigned to separate modules. Figure 1.a shows two developers working on the same program. As the program is contained with one file, they cannot work independently from each other. This causes conflict within the revision control system and requires module changes to be merged. The resolution of this problem is shown in Figure 1.b. The program is broken into separate modules. The commonality between the two modules is assigned to a shared module, allowing the two developers to work on separate modules independently of each other. Thus module assignment facilitates teamwork. However there is a consequence to separating code into modules.

The shared module that contains the commonality defines an interface that dependant modules require for successful compilation. Figure 1.c shows that the shared module has become responsible to the dependant modules. Any change in the interface will affect the dependant modules, (by at least requiring a recompilation, but possibly requiring code modification as well). Therefore the responsibility of a module can be measured by counting the number of modules that depend upon it. The dependant modules that contain the variability can now be edited independently of each other. However if the shared module is edited then all the dependant modules are affected. Figure 1.d shows that the cost of editing a responsible module is higher than the cost of editing an irresponsible module. This implies that the responsible modules are required to be stable. Figure 1.e shows that the modules that it depends upon affect the stability of a module. Therefore the stability of a module can be measured by counting the modules that it depends upon. If the modules that a module depends upon are edited, then any edits in the module are affected. Figure 1.f shows that if the modules that a module depends upon are unstable, then team productivity is impaired. Therefore if a responsible module is also unstable, it implies that the system has not been assigned to modules correctly. By finding modules that are both responsible and unstable one has found a bottleneck to team development. By combining the theories of Parnas & Conway with Martin’s principles one can see that improper dependencies between software modules means poor information hiding and therefore the assignment of responsibility for each module has failed.

This model suggests that by measuring the stability and responsibility of modules, a prediction can be made of the ability of a team of software developers to work independently of each other.

Software development requires 'team players' that work in 'isolation'. Mackey notes that even though software engineering is thought of as a solitary activity that attracts a large number of introverts, most software engineering job adverts ask for 'team players' [11]. The 'isolated', introverted developers are dependant upon each other through the coupling of the common parts of their developments (shared modules). If an organisation tries to separate teams into parallel development efforts, the coupling between their separate efforts will limit their ability to work in parallel. Therefore there is a schism between different parts of the organisation caused by the software architecture.

Poor Information Hiding also conflicts with Conway's Law: the structure of the system mirrors the structure of the organization that designed it. If the organisation has a structure that does not mirror the module assignment and coupling of the software structure, then there will be excessive communication requirements between the development teams that are asked to work in parallel. If poor information hiding is allowed to dominate a project, then it will become late due to the inefficiencies of programmers competing for the same source module. If a project is in this state then one can see that adding more programmers will exacerbate the problem, leading to Brook's Law: adding more programmers to a late project makes it later. In conclusion, this model can be summed as follows: If software is improperly coupled then people are improperly coupled.

For the purposes of testing this model two high quality and successful software projects have been studied over a period of between two and four years.

3. The Two Software Projects

Swift is a large-scale commercial piece of software developed by Softel Ltd. In 1999, Swift was a joint recipient of The Royal Television Society Award for Innovative Applications. It has well over 500,000 lines of original in-house code as well as incorporating third party COTS (Components-off-the-Shelf) such as MFC (Microsoft Foundation Classes). Development of Swift began in the spring of 1997 utilising a single developer and growing to seven fulltime personnel by the summer of 1999. It has continued with that level of support for the last two years. Therefore Swift has received (very approximately) twenty person years of effort over four years.

Quantel developed a new platform for video and audio editing in the 1990s. As part of this they have developed a software suite we shall call development 'B'. It has well over 800,000 lines of original in-house code and also uses many COTS such as the ACE networking toolkit and ImageMagick picture format library. For the purposes of this research we can say that Development B began in Autumn 1999 with six fulltime programmers. This figure has risen to 14 fulltime software engineers by the summer of 2001. Therefore development B has received (very approximately) twenty person years of effort over two years.

Some simple but useful attributes of the two projects are shown in Table 1 below:

Metrics Project (date)	Swift 2001-08-10	Development B 2001-07-17
Files *	1,930	3,540
Lines *	564,297	817,130
Approximate project start date	1997-05	1999-09
Approximate person years of effort	20 Years	20 Years

Table 1 Simple Attributes of the Two Projects.

* Metrics gathered with SourceMonitor [15]. Only source code written by the development teams was passed through the tool. Library headers (such as C++ Standard Library, Windows™ headers and third party headers) were excluded.

A simple analysis of these attributes shows that there are approximately 45% more lines of code in development B than in Swift. If the lines of code produced per programmer year are calculated the Swift team has averaged just over 28,000 LOC/Year whilst the development B team has averaged in excess of 40,000 LOC/Year.

It can also be seen that development B has double the number of programmers working in parallel These figures are therefore counter-intuitive, for the programmers on the development B team should have had more conflicting merges and compile ripples than the programmers on Swift. Based upon Brook's analysis of communication overhead, one would predict that the LOC/Year should be less on projects with a larger team of programmers.

The model presented earlier in the paper has suggested that the source code of the projects needs to be analysed to expose the level of responsibility and stability that the classes in the system exhibit. The model suggests that Swift will have more classes that are both responsible and unstable than development B.

Therefore, in order to explain the above figures, Section 4 presents a technique that can expose the physical coupling inherent in the source code within the two projects.

4. Visual Class Multiple Lens Views

Visual Class Multiple Lens (VCML) views are generated using Visual Class, a tool that implements the Lens visualisation technique developed by the authors. The Lens technique, the Visual Class tool suite and VCML

views have all been documented in detail elsewhere [4, 5]. In essence, Visual Class takes as its data source the Browser Database that can be generated by Microsoft Visual C++ during a compilation. Once the database is loaded into the Visual Class, each class within the code has its *reference* and *uses* measure calculated. All classes are then drawn, with *Lenses* affecting their visual attributes.

The *reference* measure is mapped to the size of each class and represented by a font size lens (the larger the font size the greater the number of references to that class). The *uses* measure is mapped to the colour of each class using a scale that runs from blue (low) to red (high). The images presented all have the same scaling applied to each lens. This scaling factor has been found iteratively by experimentation. It should be noted however, that the current layout algorithm in Visual Class only handles single inheritance, so in the case of multiple inheritance, classes will be drawn more than once. As shall be seen this does not detract from the usefulness of the VCML views since all that is important is the relative size and colour of each class.

In order to present the most relevant attributes of a piece of code, a threshold can be applied to remove all the classes that have a *uses* measure below 20 (or indeed any other specified number) other classes. This means that the views only have yellow and red classes visible. In effect the images have had all the stable classes removed, and the unstable classes have been exposed.

This technique and its implementation expose the classes that have both a high reference measure and a high uses measure. These can be seen as the classes that are both *Big* and *Red*.

Section 5 describes the application of this technique to the two large-scale software systems.

5. Visualising Projects with VCML Views

Lenses 1 – 4 in Figure 2 show the interaction between the two directions of coupling within Swift over four years of development. The lenses show that over this time the Swift source code has become highly dependant upon several key classes (they are big) that have in the same period of time also become unstable (they are red).

`CWnd` is a class at the core of the Microsoft Foundation Classes COTS framework, and as such is not edited by the development team. Accordingly it does not change colour, it just grows over time. However all the other classes visible in the Lens view have been developed in house, and as such are indicators of unmanaged dependencies.

`CBlock` is a central abstraction in the Swift design, being the base class to all elements that can be displayed in its user interface. Over the four years of the visualisation the `CBlock` interface has become 'fat'. A fat interface is one that is all things to all people, with many of the derived classes not having a useful implementation of some of the inherited interfaces. Fowler in his catalogue of "bad smells in code", names this problem *Refused Interface Bequest* [8].

`CSwift32Doc` and `CMainFrame` are two other classes exposed as being both unstable and responsible. Both these classes inherit from key abstractions in the Microsoft Foundations Class hierarchy. They are both used heavily for routing messages around the system. `CMainFrame` is available globally through a variant of the 'singleton pattern' [9]. It is therefore used widely to dispatch messages from anywhere in the system to the user interface. It also delegates most of these messages to the main document class, `CSwift32Doc`. This leads to many of the types in the Swift system being required by both these classes to compile.

The Swift team undertook to re-factor the code to reduce the excessive compile times that the coupling was causing. The effects of this are shown in the difference between Lens 3 and Lens 4, the rate of growth in coupling has reduced, but the coupling problems have still worsened over the two-year period they represent.

On discussion of these results with the Swift team lead, he wants to add an implementation of the command pattern to distribute the dependencies caused by `CMainFrame` and `CSwift32Doc`. This would reduce the instability of the two classes, as they would no longer be dependant upon all the types that currently pass through their interfaces.

Lenses 5 – 8 in Figure 2 show the interaction between the two types of coupling within development B over the second 15 months of its development. During that time the development B code base went from 386,519 LOC to 1,016,512 LOC (calculated as before using SourceMonitor [15]). The lenses show that development B has a massive amount of instability (there is a lot of red). However the lenses also show that the instability is distributed through many hundreds of classes with very little responsibility (most of the red classes are small). Even so, there are two classes that are both unstable and responsible (both big and red), namely `iWidget` and `Image`.

The `Image` class is at the core of the ImageMagick COTS library, and like `CWnd` in the Swift lenses, it is never edited. The `iWidget` class is at the heart of the development B interface system. It is the base class to every displayable item in the user interface (much like the role of `CWnd` in MFC). It is also highly dependant upon the other classes in the system, due to its role as a drawable, clickable, sizable element etc.

The development B team used a central Object Factory [12], to register all class types using an abstract factory pattern [9]. This is the design that is used by both COM and CORBA to allow for distributed registration and creation of classes. Object Factories strongly mitigate the need for the unhealthy dependencies required for creating concrete objects. Instead of creating them directly, users create them through an abstract interface. The concrete

factory object gets created once at program initialisation time, and usually only in one place, thus rationalising dependencies at the cost of requiring casts [12].

Another pattern used by the development B team was a variant of the Visitor Pattern [9] called the Acyclic Visitor [13] to dispatch messages around the system, both to widgets and to worker threads. The suggested implementation of the Visitor Pattern causes a knot of dependencies where base classes are dependant upon all their derived classes. The Acyclic Visitor mitigates this dependency cycle by using forward declarations, multiple inheritance and run time type casting. So even though development B has real time constraints, the designers chose patterns that have runtime costs to avoid compile time dependencies and physical coupling.

6. Analysis

Over time, the Swift source code develops many classes that are both big and red, such as `CMainFrame`, `CBlock`, `CTransmit`, `CBound`, `CSwit32Doc` and `CSwift32App`. This is an indication of unmanaged dependencies, and indicates poor information hiding. According to Parnas, the assignment of source code to modules should be facilitating team development.

The classes that are both big and red will be required by many programmers at the same time in order to implement changes that should be independent of each other. Thus one can see that the when the software is improperly coupled the programmers become improperly coupled, leading to lower productivity and frustration.

This issue can be exacerbated by the use of private workspaces in revision control systems. If the programmers develop with separate, private versions of key header files, at some point their work needs to be checked in. This will lead to source code merging. Estublier and Casallas [7] have reported that merge tools in revision control systems often fail. They point out that merging modules is not a perfect mechanism. Inconsistencies may arise from a merge; the probability of problematic merges rapidly increases with the number of changes performed in both copies. Therefore they recommend that frequent merges are needed to keep cooperating workspaces in synch.

In development B, `iWidget` is definitely showing signs of being a problem. By Lenses 7 & 8, `iWidget` is both large and red. This is a sign that an increasing number of features are being added to iWidget by the different programmers on the development B team. Accordingly the development B team are re-factoring `iWidget` into smaller interfaces, avoiding the risk of allowing the code to become rigid, fragile and immobile.

However when comparing the productivity of the two software teams, the development B team has been able to deliver higher programmer productivity whilst having double the number of programmers working on the same source code.

7. Conclusion

This paper has presented a model of software development that can be summarised as follows: If software is improperly coupled then people are improperly coupled. It can be predicted from the model that unmanaged coupling within a software project will slow programmers due to the inability of each programmer to work in isolation. It has been shown that the Swift project, with less programmers and unmanaged coupling has experienced less programmer productivity than the development B project with double the programmers and partially managed coupling.

Martin has reported that the use of the architectural patterns can aid team development, and this has been shown to be true of the development B project. It has also been shown that Swift could benefit from the application of patterns to alleviate some of the unmanaged dependencies with its source code.

Whilst there are bound to be many other variables that affect the teams' productivity, one can see that dependency management and therefore information hiding is a key concept to facilitate team development. If poor information hiding is allowed to dominate a project, then it will become late due to the inefficiencies of programmers competing for the same source module.

In the future the authors plan to assess how the application of patterns affect the dependencies in Swift and analyse how iWidget can be re-factored to reduce its impact in development B. Visual Class needs modification so that it can draw multiply inherited classes sensibly. Finally the information source that Visual Class uses (MS Visual C++ browser database) cannot be used to calculate the exact reference count and uses count of each class (for example it does not supply information about member initialisers in constructors, so some of the reference counts may be low). The authors would like to be able to generate exact metrics for these measures to allow for statistical analysis of the results.

8. Acknowledgements

We would like to acknowledge the financial and project support provided by Quantel Limited and Softel Limited, without which this work would not have been possible.

9. References

[1] Briand, L., Daly, J., Wüst, J., **A Unified Framework for Coupling Measurement in Object-Oriented Systems**, IEEE Transactions On Software Engineering, Vol. 25, No. 1, January/February 1999.

[2] Brooks, F.P., **The Mythical Man-Month**, Addison Wesley Longman, Reading, Mass., 1975.

[3] Brooks, F.P., **The Mythical Man-Month: AFTER 20 YEARS**, IEEE Software, Vol. 12, No. 5; September 1995, pp. 57-60.

[4] Cain J., McCrindle R., **Program Visualisation using C++ Lenses**, Proceedings 7th International Workshop on Program Comprehension, Pennsylvania, May 1999.

[5] Cain J., McCrindle R., **Making Movies: Watching Software Evolve through Visualisation**, International Conference on Computation Science, San Francisco, May 2001, published in Springer Verlag Lecture Notes in Computer Science.

[6] Conway, M., **How Do Committees Invent?** Datamation, Vol.14, No. 4, Apr. 1968.

[7] Estublier, J., Casallas, R., The Adele configuration manager. In Tichy, W., F., (ed.), Configuration Management. Trends in Software. John Wiley & Sons, 1994.

[8] Fowler, M., **Refactoring Improving the Design of Existing Code**, Addsion Wesley, 1999.

[9] Gamma, E., Helm, R., Johnson, R. & Vlissides, J., **Design Patterns, Elements of Reusable Object-Oriented Software**, Addison Wesley, 1994.

[10] Herbsleb, J.D., and Grinter, R.E., **Architectures, Coordination, and Distance: Conway's Law and Beyond**, IEEE Software, September/October 1999.

[11] Mackey, K., Stages of Team Development, IEEE Software July/August 1999.

[12] Martin, R.C., **Designing Object-Oriented C++ Applications Using The Booch Method**, Prentice-Hall, 1995.

[13] Martin, R., **Acyclic Visitor**, in Martin, R., Riehle, D., Buschmann F., (eds.), Pattern Languages of Program Design Vol. 3, Addsion Wesley, 1998.

[14] Parnas, D.L., **On the Criteria to Be Used in Decomposing Systems into Modules**, Comm. ACM, Vol. 15, No. 12, 1972.

[15] Wanner, J.F., SourceMonitor: **Expose Your Code**, pp 92-98, Doctor Dobbs Journal, March 2000. (Tool available, with source code, at www.ddj.com).

10. Figures

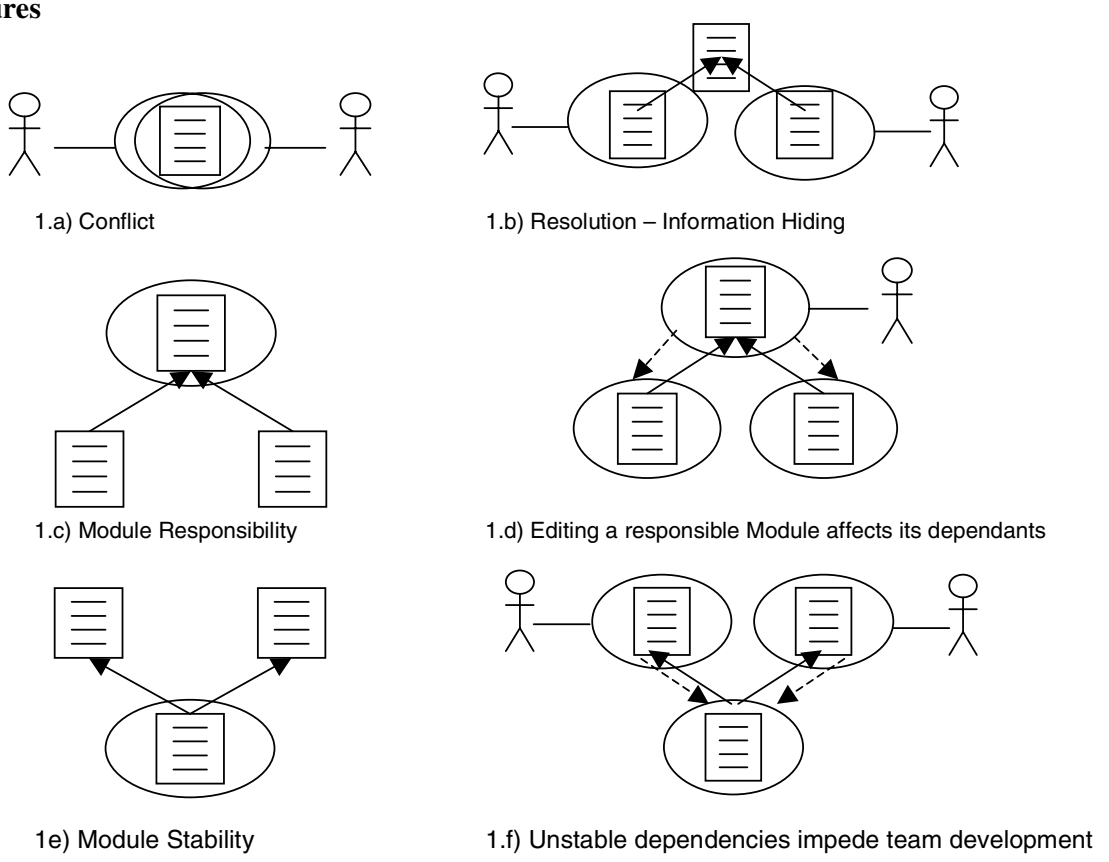
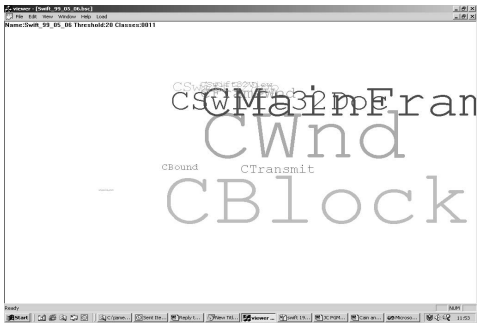


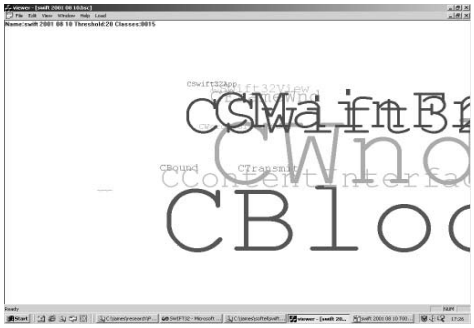
Figure 1 Consequences of Module Assignment

Lens 1. ‘Swift’ (1997-08-01)

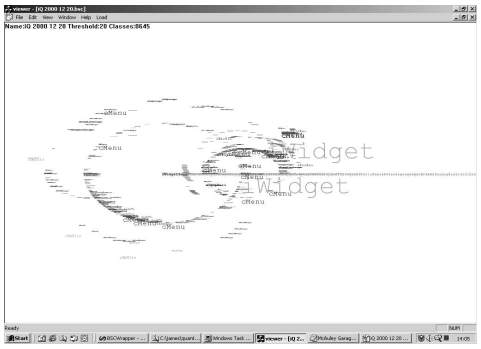
Lens 2. (1998-06-10)



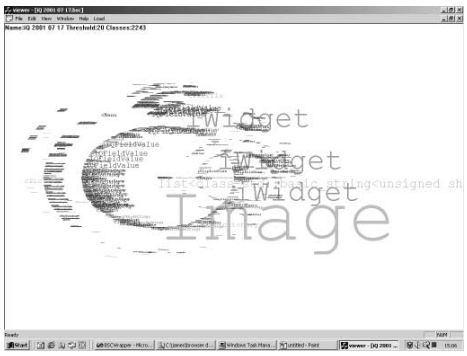
Lens 3. (1999-05-06)



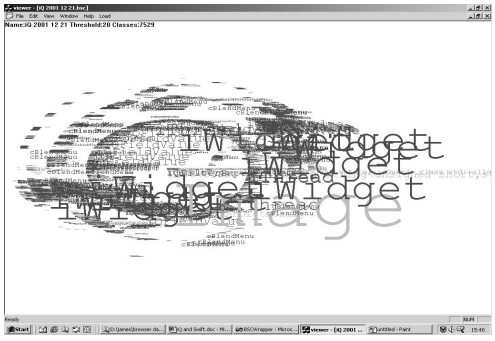
Lens 4. (2001-08-10, 564,297 LOC)



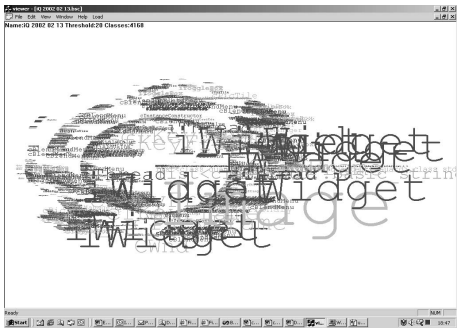
Lens 5. ‘B’ (2000-12-20, 386,519 LOC)



Lens 6. ‘B’ (2001-07-17, 817,130 LOC)



Lens 7. ‘B’ (2001-12-21, 996,358 LOC)



Lens 8. ‘B’ (2002-02-13, 1,016,512 LOC)

Figure 2 Lens Views of Swiftand development ‘B’