



## **The Case for Class Ownership**

***By David J. Anderson***

### ***Introduction***

Feature Driven Development (FDD) [Coad 1999, Palmer 2002] believes in class ownership - the notion that a named developer (and perhaps a secondary deputy) is selected to "own" a class. Only that developer will write code in that class. This creates a constraint and a potential bottleneck. Other agile methods do not believe in class ownership. Class ownership is a deliberate choice. In the language of the Theory of Constraints [Goldratt 1990], it is a policy constraint. FDD has chosen to adopt class ownership as a constraint, whilst other methods have chosen to eliminate it by abolishing such a policy and allowing any developer to make the necessary changes in any class. On first observation, the elimination of a policy constraint such as class ownership would appear to allow development to happen faster and for development work to flow more easily. The decision to adopt class ownership was not made lightly and its continued use after almost 7 years of FDD is worth discussing. Class ownership is a choice. It is for individual managers to decide whether that choice is right for them and their organization. With this article, I hope to illuminate some of the reasons why FDD practitioners believe that class ownership makes sense.

### ***Team Work***

Class ownership necessitates the idea of a Feature Team. A Feature Team is a virtual team which forms to design and build a small batch of Features which touch the same classes. Class ownership forces the owners to work together to design sequence diagrams and to agree the interfaces/method calls between the classes. This team thinking activity improves the quality of the design. It also improves the social and communication skills of the participants and helps them to build a camaraderie and mutual respect.

Where XP uses pair programming to improve the quality of coding, FDD is using the enforced teamwork in the Feature Team to improve the quality of the design and to improve the knowledge of how the system works across team members.

The alternative in other agile methods, to FDD's team ownership of a Feature with individual ownership of methods which occur in owner's classes, is for one individual to own a Feature, Story or Task and to code that from start to finish. Individual

ownership of client-valued functionality encourages an individual to design alone. On the average, this leads to a poorer quality design than a team design and valuable opportunities for team working and building mutual respect amongst team members are missed. To compensate for this some form of review or design inspection would be needed. Few agile methods recommend the use of such reviews or inspections.

## ***Preventing Atrophy***

Paul Szego – one of the original FDD team – says that class ownership maintains “the Zen of the class” [Highsmith 2002]. What Paul is saying is that the quality of a class – how it works and what its responsibilities truly are – can be lost without a single owner. FDD uses class ownership to deliver the Lean principle of “building integrity in” [Poppendieck 2003] to a system design. Where other agile methods choose to insure integrity through refactoring, FDD prefers to avoid refactoring as much as possible. FDD pursues the Lean principle of “right first time” through use of domain modeling (out-of-scope for this article), Feature Team design and class owner implementation. A single owner for the class becomes the single architect for that class. In FDD each class is treated as its own component and special care is taken to maintain the architecture and public interface to that class. As Fred Brooks [1995] explained - the architectural integrity of any software system is best facilitated through the eyes of a single architect. A single architect is just another way of “building integrity in”.

The benefits of having a single class owner can be seen when reviewing code from a project which has been run under stressed circumstances. Class ownership introduces a constraint. When code needs to be written in a hurry, the class owner may not be available. There are two choices: wait for the class owner to free up; or allow a non-owner to check-out the class and make an update. When a team is against the gun, there is little choice - break the class ownership constraint.

However, there is a price to pay. This price is not immediately obvious. In fact the systems cycle before it becomes an issue is so delayed and far removed that the cause and effect are hard to associate. Nevertheless, the quality of a class will atrophy unless the code is peer reviewed by knowledgeable team members. I found a simple little example of this recently. It seems so trivial as to be hardly worth mentioning. However, class atrophy happens in small amounts and attention to detail is important.

A developer had opened another's class and added a very short method to complete a sequence. The method directly accessed a member variable (private attribute) of the class. Elsewhere there was a friendly scoped accessor method. All the rest of the class was using this method. Why? Naturally, the class owner had a reason - in a future release of the software, the attribute would not be accessed but calculated by calling out over the object model to other classes. The owner knew this from the domain modeling earlier in the project. By accessing the variable directly a potential future defect was introduced. This would inevitably cause some head-scratching later on and it is unlikely that enough information would have been captured in comments or design documents to really explain why some parts of the class were one way and others different. Were there not a class owner to debug the problem, a random developer may fix it the wrong way and cause yet more refactoring as a result.

So here is one small isolated incident. When you allow anyone on a team (particularly a bigger team) to edit any class for any purpose, you will quickly lose design integrity of the class unless other mechanisms are put in place to prevent it. FDD believes that the long term cost of other mechanisms is greater than the cost of introducing the policy constraint of class ownership.

### ***The Long Term Tradeoff***

Class ownership is a long term bargain often best made on medium to large projects. Why? Class ownership is a constraint which can slow the flow of development. The reduced speed may be costly on smaller projects with shorter life spans. Class ownership is all about building-in integrity. It is that integrity which will give the architecture longevity. If your project may need to be maintained for years and continued enhancements are foreseen beyond 12 months, and/or the project has more than a dozen developers, then it is important that it is built to scale and to last. Built-in integrity is a “must have” for longevity of larger scale systems. Other agile methods, offer to achieve this through refactoring. FDD offers to achieve it through the enhanced quality of class level integrity with class ownership coupled to team design preceded by preliminary domain modeling.

### ***The Regular Integration Tradeoff***

Class ownership and the agile practice of continuous integration don't mix! However, class ownership and ***regular integration*** might be a best of both worlds solution!

As mentioned earlier, class ownership forces team working. However, there is a side effect to team working and the use of most popular version control and continuous integration tools - it is impossible for two team members to check-in code simultaneously! If two (or more) people have been collaborating on a single Feature and one is adding methods which the other must use, or indeed modifying method signatures from existing code, then when the first developer checks-in, the build will break in a continuous integration environment. Not until the second (or subsequent) developer checks-in his updates will the continuous build recover.

The problem with continuous integration is that it overloads two concepts - check-in, a mechanism for safe storage, version history, and file sharing - with build, a mechanism for integration of the code. This becomes problematic when mixed with class ownership.

FDD has an explicit Promote-to-Build milestone for every Feature. What does this mean? It means simply that check-in can happen multiple times throughout the development of a Feature. Several owners can check-in and check-out as they see fit. In fact, check-in should happen every night. What happens if the developer is off sick tomorrow? Unless they have checked-in so that their secondary class owner can takeover, a Feature might get blocked. FDD relies on a 2-stage promotion process. Check-in happens to a revision labeled "dev" in the version control. The Chief Programmer or Development Manager has the right to "promote to build" which is a process of relabeling specific revisions. Only the revisions labeled "build" are built in the regular build process.

In FDD, the continuous integration process would run on the "build" revisions. As Work Packages of Features are only promoted to build (reabeled as "build" revisions)

when the Work Package is complete and reviewed, there is an inevitable delay between check-in and the basic integration test of compiling the build. This delayed process might better be called **regular integration**. By encouraging the development of small batches, there is still basic integrated compile checking happening frequently to highlight problems early. This meets the spirit of the continuous integration movement in agile methods, whilst delivering the long term quality and integrity benefits of class ownership. Hence, class ownership need not be seen as a choice which precludes continuous integration. Indeed class ownership with regular integration encourages frequent check-in – a good thing – when continuous integration discourages check-in until a developer is done testing everything related to the feature, story or task.

Finally, in a class ownership world with regular integration there is never a need to merge changes. There is never a possibility that two developers modified a single file. There is no scope to introduce bugs whilst changes are merged.

## **Summary**

Class ownership is a choice. It has tradeoffs. FDD chooses to use class ownership. This goes against the trend with many other agile methods. Class ownership requires discipline, teamwork and a little forethought and planning by Chief Programmers to make it work. Class ownership delivers class level component architecture integrity at the potential expense of smooth flow of coding. FDD believes that this pays dividends on projects with large numbers of developers and projects with a long expected lifespan and maintenance cycle. Class ownership is a deliberate policy constraint. FDD chooses to subordinate all other decisions to that policy constraint and to design the process around it. Other agile methods choose to eliminate the constraint by abolishing class ownership and compensating through refactoring (and in XP with pair programming).

I hope that this article has illuminated the issues around class ownership and allowed you – **the agile manager** – to make an informed choice as to what will work best in your organization.

## **About the author**

**David J. Anderson** is the author of the recent book, “Agile Management for Software Engineering – Applying the Theory of Constraints for Business Results” published in Peter Coad’s series by Prentice Hall PTR in September 2003. He is Principal Consultant with VA Systems Professional Services. David was one of the team which created the popular agile development method, Feature Driven Development. He has introduced FDD at two Fortune 100 companies Sprint (a telecommunications operator in the United States) and Motorola. He writes the regular **Agile Management** column at the Borland Developer Network website and publishes his weblog at <http://www.agilemanagement.net/>. He holds a degree in Computer Science and Electronics from the University of Strathclyde.

Email: [dja@agilemanagement.net](mailto:dja@agilemanagement.net)

## **References**

- Anderson, David J., *Agile Management for Software Engineering – Applying the Theory of Constraints for Business Results*, Prentice Hall, Upper Saddle River, NJ, 2003
- Brooks, Frederick P. Jr., *The Mythical Man Month*, Addison Wesley, New York, NY, 1995, Anniversary Edition, Chapter 4: Aristocracy, Democracy and System Design
- Goldratt, Eliyahu M., *What is this thing called The Theory of Constraints and how should it be implemented*, The North River Press, Great Barrington, MA 1990
- Highsmith, Jim, *Agile Software Development Ecosystems*, Addison Wesley, New York, NY 2002, page 279
- Palmer, Stephen R., John M. Felsing, *A Practical Guide to Feature Driven Development*, Prentice Hall PTR, Upper Saddle River, NJ 2002
- Poppendieck, Mary, and Tom Poppendieck, *Lean Software Development – an Agile Toolkit*, Addison Wesley, New York, NY 2003, Chapter 6: Build Integrity In