Code Ownership Revisited

Jurgen Appelo
jurgen@noop.nl
www.noop.nl
January 27, 2008

Introduction

Among well-known methodologies for software development one can recognize two philosophies regarding the assignment of responsibilities to team members for the code that they produce: *collective code ownership* and *individual code ownership*. In this article I explain that there are not two but four ways of assigning responsibilities among team members. I also claim that the choice for either of these models should be made not by methodologies but by project managers, architects or team leaders, and I present a number of criteria which might be helpful while selecting the best model.

The Tragedy of the Commons

Christmas and New Year's Eve in 2007 I spent my holidays in Surinam, a small country in South America, and a former Dutch colony. For me as a Dutchman this meant that I enjoyed not a typically Dutch gourmet but a typically Surinamese buffet on Christmas Day, and great fireworks (*pagara's*) that would have been illegal in my home country and almost ripped my eardrums to pieces. The things I especially noticed in Surinam, in addition to the delicious food and the fantastic fireworks, were the plastic bottles on the banks of the river, the broken tiles of the sidewalks and the poorly maintained government buildings in the inner city. They are examples of the phenomenon that economist William Forster Lloyd in 1833 called "The Tragedy of the Commons" (and which was popularized in a 1968 essay by Garrett Hardin): a resource in joint possession of a group of people is destroyed when each owner individually benefits from short-term revenue while the long-term costs are spread over all other owners. Collectively owned properties in a country (including the environment, infrastructure and government buildings) deteriorate when people make good use of them while nobody takes responsibility for their maintenance. This problem, which basically amounts to non-ownership, can also be seen in software development activities. Any developer can name examples of quality problems in code, like inconsistent interfaces, mixed coding styles and poor documentation. When developers only deal with new features and immediate results, while at the same time nobody on the team feels directly responsible for maintaining the shared code base, the quality of the code will deteriorate, fully in line with The Tragedy of the Commons.

Four Methods of Artifact Assignment

To prevent non-ownership (and thus loss of quality) in a software project, it is necessary that we explicitly assign responsibilities to team members. In literature this is also called *code ownership*. Since it does not in the least deal with property rights issues, and not just with code either, this name seems like a neat continuation of the long tradition of poor naming in our field. Property rights in a software project mostly lie with the organization or with the customer and not with the team members in the project. And besides responsibility for code we must also assign responsibilities for other artifacts, such as models, documents, graphics files and test cases. I therefore prefer to speak of *artifact assignment* rather than code ownership. And following one other author (Martin E. Nordberg) I distinguish four policy principles, to which I have applied my own labels:

Local Artifact Assignment (LAA): Most systems we build today have interfaces with other systems. We divide major problems in smaller problems and we cut large systems into subsystems. Local Artifact Assignment is my label for delegating policy to subsystems (and subsystems within subsystems). Property laws in Surinam are different from those in the Netherlands, and within Surinam the rules applicable to the jungle are different from those in the city. LAA defines the levels and determines the territories to which we apply specifically tailored policies. Similarly, we might want to apply a policy for web pages that differs from the one dealing with database tables. And the code written for a major financial workflow possibly deserves a different approach than the code written for public web services. In short, using LAA we determine a policy per subsystem.

Authoritarian Artifact Assignment (AAA): When distributing work among software developers, the traditional approach is to have one authoritarian person (Chief Architect, Team Leader or Lead

Developer), who is responsible for the quality of his own results and those of the other team members in the project. We can compare this with the distinction magazines make between editors and writers. Several writers provide intermediate results, but responsibility for the final results lies exclusively with the editor. In Surinam we see such an implementation when considering landowners who have other people living and working on their lands, but who need to keep an eye on the quality of all the work that is performed. With this policy, the team leader holds the reins firmly in his hands, as if he were a benevolent dictator.

Figure 1: The Presidential Palace in Surinam, which is in fine condition

Collective Artifact Assignment (CAA): This approach is relatively new in our field and it has mainly received publicity thanks to the *Extreme Programming (XP)* methodology, under the name *Collective Code Ownership* (a term which, unfortunately, is only 33.3% correct). According to this principle, all team members have equal responsibility for all the results that they produce within the context of the project.

The potential of this approach is clear from the magnificent presidential palace in Surinam (Figure 1) which is collectively owned by all Surinamese people. But it can also go horribly wrong, given the deplorable state of most other government buildings in Surinam (Figure 2).

Figure 2: The Department of Justice and Police in Surinam, which is in bad condition

Individual Artifact Assignment (IAA): This is the third variant, promoted in a more specific form in the *Feature Driven Development (FDD)* methodology under the name *Individual Class Ownership* (a term which is also only 33.3% correct). It is the logical counterpart to the collective approach. Its philosophy says that for any artifact within the system only one team member is responsible. In Surinam you can experience the consequences of this approach when you eat at a *warung* (a Javanese eating-house). These warungs are often situated in the middle of a residential area because home owners can do whatever they want within the borders of their small territories. The opening of such a warung in somebody's backyard can quickly lead to the opening of a competing place a couple of houses away. As a tourist you can fully enjoy the fun, noise and smells in such a street. However, the neighbors might not appreciate it as much as the tourists do.

These four policy principles are applicable to teams of two or more persons. However, in many organizations small systems are built entirely by just one person (Figure 3a). In a team consisting of one person all policies are equal (i.e. LAA = AAA = CAA = IAA). But when this one-person team grows by adding an additional team member a choice needs to be made. There are four options:

1. De second team member can operate as an assistant or subordinate of the first (= AAA, Figure 3b);
2. The two team members can accept joint responsibility for each other's work and for the entire system (= CAA, Figure 3c);
3. The team members can divide responsibilities among each other, so that they don't carry any responsibilities for each other's work (= IAA, Figure 3d);
4. The team members can split the system in two subsystems (= LAA, Figure 3e). This results in two one-person teams en for each subsystem we arrive back at the original situation (i.e. LAA = AAA = CAA = IAA).

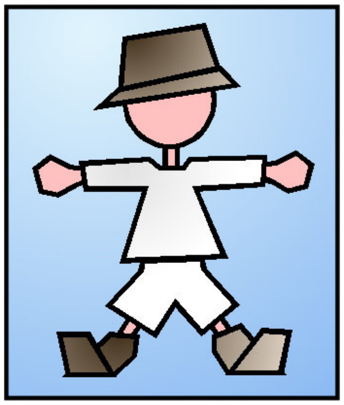

Figure 3a: A one-person team

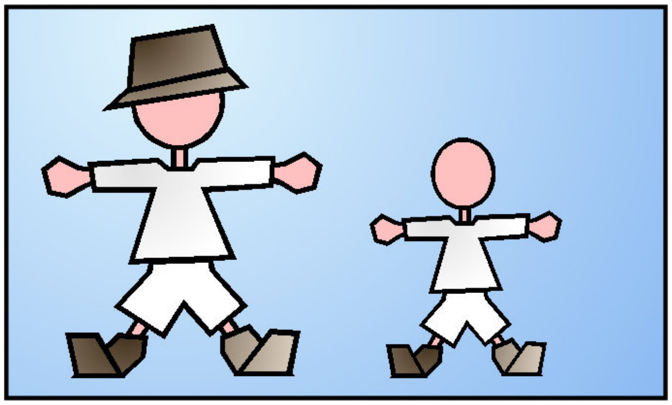

Figure 3b: A team with a leader and an assistant (AAA)

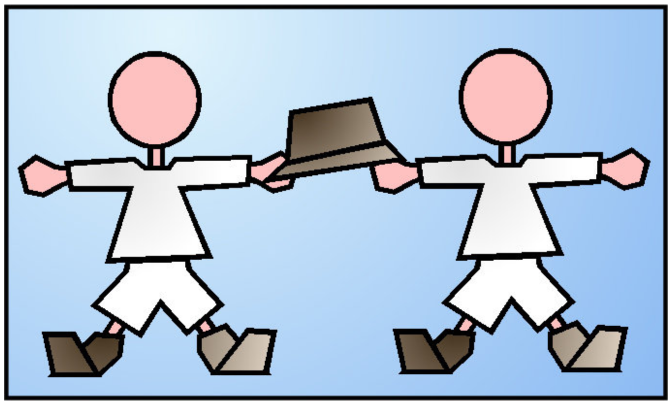

Figure 3c: A team with shared responsibilities (CAA)

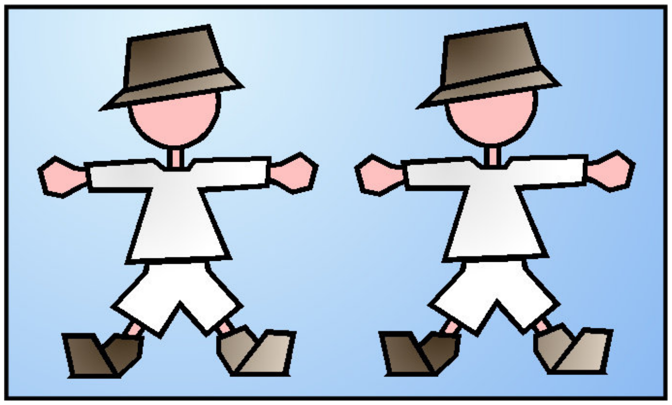

Figure 3d: A team with divided responsibilities (IAA)

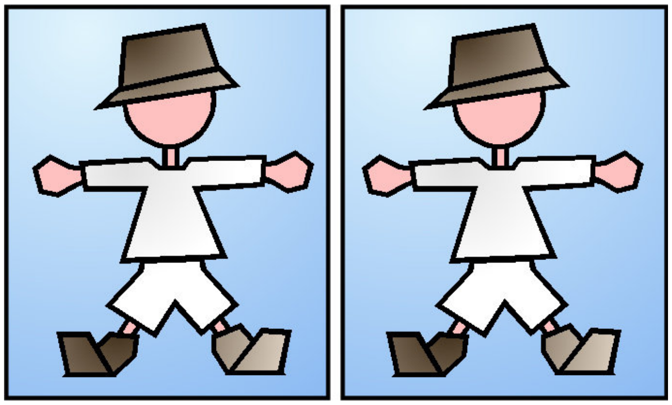

Figure 3e: Two subsystems each with its own one-person team (LAA)

Flexibility = Delegation of Policy Making

When software systems change over time their complexity usually increases. This means that the need to partition those systems into subsystems also increases. Ultimately, we are always dealing with programming interfaces and separation of responsibilities. The concept of *information hiding* must be implemented somewhere, not only (by definition) at the borders of a project but also (at our own initiative) within the project itself. If a system grows so large that too many developers are involved, we need to split it into subsystems (LAA). One of the reasons is that AAA, CAA and IAA are not scalable. Assignment of responsibilities to team members is not feasible when they are no longer able to oversee the overall system. But partitioning and information hiding come to our rescue. AAA, CAA and IAA are the solutions that we use on a local scale, and we use LAA for partitioning the entire system and our ownership policies.

A project manager, team leader or architect should be able to determine what policy (AAA, CAA or IAA) should apply to which subsystem. This choice depends on a number of environmental variables that can vary over time. Even within a release cycle the choice may vary per phase and per subsystem, maybe due to changes in team membership or because of a new agreement between customer and supplier. The need for flexibility and adaptability in our field is much bigger than it is for the Surinamese government. Choosing a rigid approach (for example, the adoption of either the collective approach of XP or the individual approach of FDD in all projects in an organization) does *not* indicate flexibility, in my opinion, even though both methods claim to belong to the agile category of methods. The most flexible approach is the delegation of policy to the lowest possible level at which people are still able to make informed choices. After all, at the project level people usually have the most reliable and up-to-date information on the context of the project. The choice made by a team leader can therefore be much better founded than that of an external process manager or the writers of a best-selling methodology. In this article, I attempt to provide the Surinamese policy makers among my readers with the right criteria to make their own informed choices.

Selection Criteria for Policy Making

There is no single answer to the question what the best policy is, but there are a number of criteria which I will list here.

The *Knowledge Criterion* concerns the *generalists-versus-specialists* argument. Individual responsibility can lead to specialization of team members, with fragmented but deep knowledge of parts of the system. Collective responsibility can have the effect that team members become generalists, with broad and equally divided knowledge of the entire system. Both alternatives have implications for architecture and applied techniques. Some authors claim that specialists are valuable in the construction phase because depth of knowledge is required for solving complex technical issues. Generalists are said to be valuable during the subsequent maintenance phase because they must be able to solve problems with broad but more superficial knowledge of the entire system. However, another line of thought claims that collective thinking in the construction phase leads to better solutions to complex problems, while individual responsibility for specific components would be useful in the later stable phases of the system. Conceptual integrity and consistency are always important in any architecture, but you have the choice to achieve this in either direction (breadth or depth) in the system. My conclusion is: choose IAA when the technologies used in individual components are complex; choose CAA if not the technologies but the problem domain is complex, and select AAA when the leader is the best person to understand both the technologies and the problem domain.

The *Resources Criterion* concerns the famous *truck factor* argument. The truck factor indicates how many team members would have to be run over by a truck before the project would be in serious danger. (In Surinam the term "taxi factor" would be more realistic.) Of course, the idea behind this is that staff turnover and absence (leave or vacations) come at the expense of a project when knowledge is either lost or temporarily unavailable. In the case of collective responsibility the truck factor is high (and thus the risk is low) because each team member has knowledge of the whole system and you would only have a problem when the entire team collectively decides to jump in front of a very large truck (or they decide to go on a vacation to Surinam). On the other hand, there is some communication overhead involved when knowledge must continuously be shared among all team members. The execution of any task with individual responsibility will cost on average slightly less time than in a situation with collective responsibility. When resources at any time must be able to take over each others work there is always a little loss of productivity. This applies to load balanced servers, cooks in Surinamese restaurants, as well as team members in software projects. It is in fact the premium paid every day to mitigate the risk of loss of knowledge. My conclusion is: choose IAA when daily productivity is of the utmost importance; choose CAA when the risk of (possibly temporary) loss of knowledge is unacceptably high; choose AAA when you want to take the middle road.

The *Management Criterion* deals with people management questions. You have to determine which people within a project team are able (or should be given the chance) of carrying responsibilities. This is a typical management problem and its outcome depends entirely on the types of employees in the team. You can choose IAA when a team member is suffering from the interference by colleagues who always think they know better. (A common argument against collective code ownership is that self-proclaimed "experts" are always “improving” the code of their colleagues.) You can also select IAA when people with strong personalities have respect for each other's work but also have a need for self-development and individual creativity, while they dislike submitting themselves to collective thinking. By contrast, you can choose CAA when project members have to learn to work together as a team. (A common argument against individual code ownership is that stubborn developers ignore any shared goals and simply go their own way.) You can also choose CAA when the team consists of professionals of sufficiently balanced experience levels who love to learn a lot from each other through cooperation on the same code. Finally, you can choose AAA when the team is not balanced and there is clearly one leader, though sometimes an informal one. This person may find the coaching of junior team members an interesting challenge.

Without a doubt I can say that there are more examples of people management that affect the choice of policy for artifact assignment. Also, some readers may be able to expand my three criteria to cover additional issues that I have not discovered yet. My suggestion is just to think about it, not to follow the standard prescription of any methodology, and to make your own well-founded choices.

Too Much of Anything Is No Good for You

As usual, too much of a good thing may not be good for you. Adhering to a policy that is too extreme may do more harm than good. A well known example of individual code ownership is that colleagues may need to write workarounds when the owner of a crucial piece of code refuses to cooperate, with the effect that the costs of the workarounds are higher than what they would have been had the owner made the proper adjustments himself. Another extreme example can be experienced when you no longer can improve your own code because its public interface is being used by a colleague who does not want to change her call into your code. With pure IAA you run the risk that developers become lazy and prefer not to make any changes to their own work. This means that they actually have a monopoly on their parts of the project.

Supporters of collective code ownership like to use these examples as arguments in favor of CAA, but they seem to ignore that application of CAA could lead to other problems. Opponents often present the argument that collective code ownership may lead to non-ownership. In such a case everyone owns the code, in theory, but in practice no one feels directly responsible. With CAA there is a dangerous tendency to drift towards the well-known problem of The Tragedy of the Commons. Poor quality of public services in comparison with private services is a well-known phenomenon worldwide, not just in Surinam. The only way to prevent this is to enforce strong discipline among team members. Extreme Programming does this effectively by combining collective code ownership with several other best practices (including refactoring, pair programming, unit testing, stand-up meetings and continuous integration). In fact, various authors rightly indicate that collective code ownership only works well in combination with these best practices. Unfortunately, this means that there is high coupling between these different concepts: you should better think twice before deciding to introduce one and not the others in your organization. But that means subjecting yourself to inflexibilities in implementing your own process improvement initiatives.

Of course, one can think of variations on the different policies that can remove some of their sharp edges. For example, we can use AAA as an intermediate form by appointing a passive authoritarian leader who only intervenes when IAA or CAA leads to conflicts or quality problems. (You can compare this with the Surinamese authorities normally keeping a distance and only becoming actively involved when food and fireworks come flying through the neighbor's windows.) One can also use tools and techniques to prevent potential problems. You can use versioning in a source control system to separate changes made by different team members, and you can let it generate notifications so that every owner knows who changed what and when. (Unfortunately, for binary formats such as graphics files such a policy of versioning and notifications will be of less practical use.) A special option (though not often considered) is to allow competition within a project. Team members can keep each other alert by allowing the possibility of building competing artifacts. With this idea, you can effectively counter any laziness and monopolistic behavior.

Exotic Arguments

Most arguments in this story, in favor of or against certain methods, are taken from books and on-line articles. They were not the only arguments that I have been able to find, but they were definitely among the strongest. Some other arguments I have read were more exotic than anything I have seen in tropical Surinam. There are writers who claim that team members will be more proud of their work when applying some specific policy. Other writers argue that the application of one method motivates the team much more than the application of another method. I have even seen the risk of someone suffering a burnout being used as an argument for one approach over another. Personally, I found these arguments weak or downright silly and therefore I chose to omit them from the criteria mentioned earlier, which has hopefully benefited the readability of my story. Pride, motivation and burnout, I believe, depend primarily on other circumstances in the organization and can be both fed and undermined with either of the policies in this article. However, if you are of another opinion then feel free to add additional criteria to the checklist I presented in this text.

One argument that regularly surfaces and which is inaccurate, according to my view, is that the collective approach is preferable because selfishness (in the individual approach) leads to anarchy. Several authors explain that team members should be serviceable towards the team, and public interest must be placed over private interest. I can say that many philosophers, economists, sociologists and biologists are of a different opinion on this matter. From Adam Smith to Ayn Rand and from Milton Friedman to Richard Dawkins, famous thinkers agree that self-interest and cooperation are respectively the purpose and the means that will trigger the emergence of fantastic systems. The challenge is to manage artifact assignment in such a way that self-interest and cooperation lead to optimal results for the owner of the project.

Conclusion

In this article I argued that there are not two but four ways of distributing responsibility for code and other artifacts among team members in a project. Three variants (authoritarian, collective and individual) correspond nicely to the three ways in which the granting of actual properties can take place in the real world. The fourth way (delegating policy making to the local level) is similar to the drawing of national and regional boundaries. I hope to have convinced readers that the selection of policies can best be done at the lowest possible level so that one can take into account the local circumstances. To those who would like to study the different variants some more, I would recommend to spend a vacation in Surinam so that one can see and experience the differences in practice, while enjoying good food and fantastic fireworks.

Sources and References

Coplien, J.O. and Harrison, N.B. (2005) *Organizational Patterns of Agile Software Development*. Upper Saddle River: Pearson Prentice Hall (p. 261-263)

Nordberg, M.E. (2003) "Managing Code Ownership". *IEEE Software*, March/April 2003 (p. 26-33)

Palmer, S.R. and Felsing, J.M. (2002) *A Practical Guide to Feature-Driven Development*. Prentice-Hall (p. 42-45)

On-line resources

Tragedy of the commons

http://en.wikipedia.org/wiki/Tragedy_of_the_commons

Collective code ownership is limiting software quality

http://weblogs.asp.net/ralfw/archive/2006/04/01/441639.aspx

Collective Code Ownership

http://www.extremeprogramming.org/rules/collective.html

Collective Code Ownership

http://www.xpexchange.net/english/intro/collectiveCodeOwnership.html

Collective Code Ownership and Text: A Conversation with Ward Cunningham, Part II

http://www.artima.com/intv/ownershipP.html

Design Principles and Code Ownership: A Conversation with Martin Fowler, Part II

http://www.artima.com/intv/principles4.html

Situational Code Ownership: Dynamically Balancing Individual -vs- Collective Ownership

http://www.cmcrossroads.com/index2.php?option=com_content&task=view&id=675