Collective Code Ownership

Extreme Programming (XP) gave us the notion of *collective code ownership*, which I take to mean collective code *accountability*. With this mindset, individual Developers do not own individual modules, files, classes, or methods. All of those things are owned collectively—by the entire team. In other words, any Developer can make changes anywhere in the code base.

Consider the alternative to collective code ownership, where each Developer owns an assembly, a namespace, or a class. On the surface, that may seem like a good idea. That coder is the expert on that component, as well as its steward and gatekeeper for all changes. Strong code ownership like this can block productivity.

Consider the situation where two Developers (Art and Dave) are working on separate tasks that both need to touch a common component owned by a third Developer (Toni). Dave will have to wait while Art's functionality is coded and tested. A collective code ownership approach would allow Dave to code the feature himself. Rest assured, Azure Repos will track who made what changes to which files and enable a merge (or a rollback) to occur if any problems emerged. Another potential problem with strong code ownership surfaces when refactoring. Modern refactoring tools, like those in Visual Studio, can do this safely, but if the changes to those files cross ownership boundaries, the eventual merge operation will block productivity and may introduce instability.

Adopting a collective code ownership mentality can take time. This is especially true if the Developers are used to having strong code ownership. Pairing, mobbing, and shared learning are ways to break up the turf and the politics. Just as it takes time for the Product Owner and organization to trust the Developers' ability to self-manage, it also takes time for the individual Developers to trust one another.