# Chapter V

## Construction versus sandbox

One process problem arising from the use of the Django framework is the issue of how to track time spent in the sandbox versus time tracked as construction. Typically, any time spent writing test code or production code would be counted as part of the construction phase. However, a significant amount of time in this project was spent trying to perform simple tasks in production code using Django idioms. Since many of these idioms were unknown to me at the start of the project, the typical “red-light, write production code, green-light” pattern of TDD was skewed. One of those cycles would normally take a small amount of time (measured in minutes). However, I had to learn how to idiomatically accomplish those unit-level tasks in many cases after writing the failing test in order just to produce the simple green-light production code. In some cases this meant that the length of a single unit test cycle could span an hour or more instead of two minutes.

Since I was *technically* writing production and test code during that time, I made the choice during iteration 1 to document this time as construction time. Within the scope of this project, that decision worked for estimating effort in subsequent iterations. A problem would arise if I were to use data from this project to estimate effort in a second Django project. Because my data tracking was not granular enough to distinguish “learning-Django-construction” from “normal-TDD-construction”, using this data again would produce skewed effort estimation because I would not be learning Django in a second project. I believe this would hold true for any project heavily based on a framework that was being used for the first time (or even a project in a new language). Once you know the framework and have added it to a conceptual toolbox, coding is smooth and the learning curve is low because you are merely applying things you already know in different combinations.

For this reason, if I were starting the project again today, I would track all time spent learning Django (even time when I was technically writing production code or tests) as sandbox time. That would still enable me to accurately estimate effort required for subsequent iterations by including sandbox time in those calculations. In a subsequent project, however (when I ideally would know Django and not require sandbox time) I could estimate effort based only on actual construction time.

## Replicating complex system behavior in the sandbox

The “sandbox” in PCSE is so named because you can climb in, build what you want, and destroy it when you’re through, all without worrying about affecting production code. For many tasks, this is adequate both as a metaphor and a technique. An example of a situation when the sandbox shines is familiarizing yourself with some aspect of how a language works – opening up a sandbox file, filling up a linked list and plucking the elements you think you want out of it, then going back to production code and rewriting that code to take advantage of your confidence.

When building a large system, however, there are limitations with the sandbox. Bugs were prone with Django (as in many systems) at the interfaces between multiple components. Furthermore, some bugs only manifested when components tried to access or iterate over entries in the database. Replicating these problems in the sandbox is difficult because the unit-level components or tests may appear to be working fine without the complexity of the entire system. During iteration 2, I reached a point at which the system was large and complex enough that I could no longer replicate problems in production code using from-scratch sandbox code.

There are at least a few ways to handle this. At first I attempted to make small changes to the production code in an attempt to tease out the nature of problems I was getting. However, even small changes to interfaces would break half or sometimes more of my test cases. This was problematic from an immediate standpoint (making it a pain to scroll through all of my expected failing test cases to find the one I actually cared about) and also from a process standpoint (I was losing confidence that failing test cases actually represented a problem or even that passing test cases meant something worked). I briefly tried replicating the system in the sandbox, but quickly discarded this idea as it mean prohibitive amounts of stubbing and recoding just to get simple test cases working again.

I eventually decided on a third method, made possible by my decision to track my code in a Git repository. At each stable point in the project, I made a commit to the repository, making sure I had test cases to match production code, etc. When beginning implementation of the next feature, I simply cloned the repository as a sandbox, keeping all the functionality of the system since the last stable commit but without the danger of breaking functional code. If I encountered a problem that needed to be solved in the sandbox, I did so. When I had solved it, I rewrote the code from scratch in the production codebase to prevent side effects, wiped out the sandbox, and committed the new stable version.

## Engineering and TDD concerns in Django

One of the most difficult aspects of forcing Django development into a software process is that big-picture decisions made during architecture and even iteration planning were often incompatible with Django implementation. The clearest example of this from the GWAAP project was the split between User and Applicant when referring to the two classes of users who needed access to the system. From an architectural standpoint, these classes were outlined as two separate subclasses of a joint parent class. The two distinct classes share some similar needs – the ability to be authenticated by the system, the need for a method of defining permissions and grouping, and the conceptual similarity that both classes represent people who need to use various aspects of the GWAAP system. For this reason, they needed to derive from the same base class. However, much of their functionality was also divergent: Applicants need Applicant objects and need to be jailed into only accessing information directly related to them and their application, while Users have much broader access to the system to say nothing of the fact that including an Application object for a faculty User makes no sense. Thus, during the architecture phase of the project, the two classes were given a hierarchy in keeping with the above.

During implementation, however, the reality of the way Django prefers to handle such things made it impossible to abide by the original architecture. Django includes a built-in User class (an unfortunate conflict with the GWAAP User class name). Naming conflicts were the least of the problems, however. Django “automagically” provides Manager objects go with all Django models, of which Django User is a distant subclass. Although this was resolved from a syntactic standpoint using special Python importing language (“import…from…as”), it made it impossible for the compiler to recognize special Django User methods when operating on GWAAP users. Specifically, it made it impossible to access the special User.create\_user() method, meaning all GWAAP Users and Applicants have to revert to the Manager.create() method, a less-powerful method that meant many user-specific details had to be set with multiple statements and an explicit save() call.

Worse still was the problem of checking authentication differences between GWAAP User and Applicant objects. While the Python interpreter recognizes the difference in the two classes, the Django framework does not. In practice, this means that Django treats all instances of GWAAP User and GWAAP Applicant as interchangeable with Django User. Thus, the built-in Django methods of authentication (like method decorators) gave the same access to Applicant as to User. This was an obvious semantic problem and required custom authentication checks for each View.

Ideally, the architecture of the system could have been rewritten to accommodate this limitation. However, at the point in construction this problem manifested itself, there were tens of test cases that depended upon the User/Applicant split. Changing that assumption would have necessitated not only a rewrite of production code but also an almost total rewrite of test code – in effect, it would have meant starting over from scratch. Do to the realities of time pressure, I elected to maintain the User/Applicant split from a test-case perspective, but to thereafter handle authentication differences by the inclusion of additional Django Permissions. This was yet another place in which Django automations were supplanted by hand-crafted code.

## Convention over configuration

An oft-cited tenet of the Ruby language is the idea of “convention over configuration”. The idea behind many of Ruby’s design decisions is to make minimal boilerplate code that accomplishes all standard behavior, and leave systems to define only behavior in which they differ from convention. [PERHAPS USE THE MAC METAPHOR HERE INSTEAD?] Django’s approach is similar in some ways. If what you want to accomplish is part of the built-in functionality of Django, it is simple, powerful, and secure to do so. However, designs that run counter to assumptions the framework makes are penalized heavily. They are difficult to implement, difficult to debug, and difficult to integrate with Django-friendly aspects of the system.

Obviously, this is difficult to reconcile with traditional method of architecture and design in software process. Although you may construct an accurate, well-planned design from a “green field” perspective, it breaks down as you try to reconcile your design with the preconceptions of Django.