Clean Code Development im Projekt

1. Law of Demeter

In der Mitte der Entwicklung wurden die Geschäftsprozesse entzerrt, indem die Akteure, die Angestellte oder Kunden oder Lieferanten sind, als Interkationsklassen umfunktioniert wurden, die mit Objekten in ihren Domänen bzw. mit anderen Akteuren agieren. Zu Beginn war es ein großes Durcheinander, weil ich zuerst rumprobieren wollte und dazu gerne alles nahe beieinanderhaben wollte. Eine Domain war dabei eine py-Datei, wo die Klassen erstmal gesammelt worden sind. Durch das Trennen der Klassen und eine Zuordnung in Domänen nach Fachlichkeit wurde erreicht, dass die wichtigsten Funktionalitäten der Akteure die sind, die in ihrer Domaine definiert sind. Die Lesbarkeit und Struktur wurde entzerrt und verbessert. Es wurde einfacher, Änderungen an Klassen vorzunehmen, da auch die Tests dieser Struktur folgen und somit abhängige Klassen direkt angepasst werden konnten, ohne mühsam komplette Prozesse nachprüfen zu müssen.

2. Kompositionen Vererbungen vorziehen

Es wurde nur dann Vererbung genutzt, wenn es vollständig sicher ist, dass die Funktionen der Superklasse für alle Kindklassen immer sinnvoll und nötig sind. Weiterhin halten die Superklassen nur generische Funktionen (Getter-Funktionen), die rollenunspezifisch sind. Gleichzeitig sind zum Beispiel in der Superklasse „Angestellte“ die Attribute definiert, die sicher von allen Kindklassen geteilt werden. Alle Angestellten haben einen Namen und einen Lohn. Die Kindklassen haben sehr verschieden Attribute und Funktionen, die von ihren Aufgaben geprägt sind. So wäre es zum Beispiel nicht nötig, den Bäckern Zugang zur Kasse zu geben, da sie nicht mit ihr interagieren. Ebenso müssen Verkäufer nicht die Rezepte der Bäckerei kennen, da es nicht ihre Aufgabe ist, Backwaren zu backen. Weiterhin kann die Geschäftshierarchie mit Führungskraft und untergeordneten Angestellten abgebildet werden, indem die Chef-Klasse Änderungen an den Angestellten vollziehen kann (wie z.B. die Lohnänderung, Einstellung und Entlassung).

3. Funktionen so simpel wie möglich gestalten

Ein wichtiges Ziel war die Gestaltung verständlicher und möglichst einfacher Funktionen. Da es sich um ein Management-System handelt, waren keine aufwändigen Prozesse und Funktionen nötig, um Geschäftsprozesse abzubilden. Vor allem bei den Klassen, die viel mit anderen Objekten interagieren, werden relevante Objekte direkt bei der Initialisierung zugeordnet. So wird bei der Klasse Verkäufer_in direkt die Auslage, Kasse und Kaffeemaschine bei der Initialisierung übergeben. Dies verhindert, dass für jeden Geschäftsprozess nötige Objekte in den Funktionen übergeben werden müssen, was schnell unübersichtlich und nervig wird – vor allem dann, wenn eine Funktion aus mehreren Teilfunktionen besteht, in denen die Objekte wieder und wieder übergeben werden müssen. Dies war erst eine spätere Entwicklung und demzufolge für einen Teil des Projekts ein Aspekt von Coding-Chaos.

4. Sinnvolles Benennen von Klassen und Funktionen

Passend zum Domänensystem wurden fachliche Bezeichnungen verwendet, die zu den Akteuren, Objekten und Funktionen passen. Um dies umzusetzen, wurde davon ausgegangen, dass die modellierten Prozesse bekannten Prozessen in Bäckereien nah sind. Natürlich sind Bäckereien in der Wirklichkeit nicht vollends durch Prozesse bestimmt. Es wurde versucht,

einen goldenen Pfad zwischen intuitiven Bezeichnungen und einer ordentlichen Struktur zu finden, die eine Basis nach modernen Entwicklungsstandards erfüllt. Dies wurde von Beginn an umgesetzt und später nur nochmal zum Wohle eines uniformen Benennungsschemas für Variablen und Funktionen bearbeitet.

5. Implementierung, die dem Design folgt

Wie schon in vorherigen Abschnitten erwähnt, wurden die verschiedenen Klassen jeweils in eine Ordner-Struktur getrennt, die den zugrunde liegenden Domänen entspricht. Jede Klasse verfügt über eine eigene Datei, in der nur ihre eigenen Funktionalitäten definiert sind. Akteur-Klassen können innerhalb ihrer eigenen Domänen und entsprechend den Beziehungen zu anderen Domänen handeln, indem sie auf die Funktionen der entsprechenden Klassen zugreifen können. Das Ergebnis ist ein modulares Design, das flexibel und gleichzeitig (mit Hilfe von Tests) zuverlässig ist. Auch die Tests folgen diesem Muster, das macht eine Übersicht leicht, welche Tests bereits vorliegen und welche noch implementiert werden müssen - Abdeckung nach Augenmaß sozusagen.