Clean Code Development im Projekt

1. Isoliere Aspekte

In der Mitte der Entwicklung wurden die Geschäftsprozesse entzerrt, indem die Akteure, die Angestellte oder Kunden oder Lieferanten sind, als Interaktionsklassen 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. Minimiere Abhängigkeiten

Neben den unter Punkt eins genannten Strukturentscheidungen wurde auch auch darauf geachtet, dass Klassen sinnvolle Sichtbarkeiten besitzen. Das Privatisieren von Klassen, das zwar in Python nicht so gut implementiert werden kann wie Java, wurde zumindest als Prinzip berücksichtigt. Demzufolge sollen die Klassen in sich nicht von außen modifizierbar sein, können aber erweitert werden. Es gibt ein Fluent Interface in der Domain Verkaufsraum, mit dem Bestellungen für die Kunden gebaut werden können. Dieser Builder ist räumlich vom Kunden getrennt, wird jedoch per Definition nur von Kunden genutzt. Das zahlt auf das Interface Segregation Principle ein. Weiterhin wurde Tell, Don’t Ask umgesetzt, indem die Objekte ihre eigenen Daten halten, wie z.B. die Kasse ihren eigenen Betrag. Weiterhin haben die Klassen, die in der Interaktionsschicht agieren, fest zugewiesene Objekte, die sie als Werkzeuge nutzen können, wie z.B. der Bäcker mit seinen Rezepten und dem designierten Lagerbestand.

3. Tue nur das Nötigste

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. Durch die Nutzung von Interaktionsklassen, wird das Integration Operation Segregation Principle umgesetzt – die Integrationsklassen nutzen Objekte und deren Funktionalitäten, während die Klassen der genutzten Objekte nur Funktionalitäten für sich selbst beinhalten. Fachlich sind alle Klassen, die in der realen Welt von Menschen besetzt sind, in der Interaktionsschicht des Projekts zu verordnen, während alle Klassen, die in der realen Welt Objekte sind, nur operationale Methoden enthalten.

4. Versprechen halten

Der Entwurf ist mit den Erfahrungen im Projekt gewachsen, sodass das fertige Produkt auch wirklich umgesetzt werden kann. Die Funktionalitäten wurden dabei möglichst einfach und klar gehalten, Interaktionen sollen intuitiv sein (Principle of least astonishment). 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. Vererbungen werden sparsam und sinnvoll eingesetzt, was auf das Liskov Substitution Principle einzahlt. 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).

5. Halte Ordnung

Mit Hilfe von nützlichen Tastenkombinationen konnten Änderungen an Funktions- oder Variablennamen schnell für das ganze Projekt durchgeführt werden. Dazu wurde Refaktorisierung verwendet. Weiterhin wurde mit Hilfe von Sonarqube Metriken erstellt, mit Hilfe derer potenzielle Fehler erkannt und behoben werden konnten. Wurde neuer Code implementiert, wurde er zum Ende der Arbeitsphase hin aufgeräumt und mit nützlichen Anmerkungen versehen, sodass der Einstieg in der nächsten Arbeitsphase leichter verlief. Dies verlief nach Vorbild der Pfadfinderregel.