Zusammenfassung: Pragmatic Programmer

Group Assignment SEP2
Luzia Kündig

June 1, 2022

Contents

1 Intro

Das Buch "Pragmatic Programmer" fokussiert sich neben verschiedensten technischen Best Practices vor allem auf auch den sogenannten "Common Sense", also den gesunden Menschenverstand. Es behandelt sich auf Themen, die jeden Programmierer beschäftigen, wie z.B.

- Mit heutigen Entscheidungen die Zukunft "schmerzfreier" gestalten
- Aufgaben und Dinge für sich selber und seine Teammitglieder einfacher machen
- Fehler zu machen und damit umzugehen
- Positive Gewohnheiten zu entwickeln
- Sein Toolset als Programmierer zu verstehen und zu beherrschen

Saron Yitbarek, der das Vorwort schreibt, vergleicht den Nutzen dieses Buches für Programmier-Einsteiger mit den "freundlichen Nachbarn", die dir in einer fremden Stadt Dinge zeigen, die wichtig sind: die effizientesten Pendlerstrecken, die besten Cafés, Kniffe und Tricks die man kennen sollte. Dieser Vergleich trifft für mich absolut ins Schwarze.

Dieses Buch präsentiert in 9 Kapiteln insgesamt 53 "Topics" quer durch den Alltag als Programmierer. Mit dieser Zusammenfassung versuche ich, die für mich hilfreichsten, interessantesten oder einfach amüsantesten Themen hervorzuheben.

2 Die Philosophie

Die Grundeigenschaften eines Pragmatic Programmers, beschrieben unter dem Abschnitt *Pragmatic Philosophy*:

- *Think about your work*: If this sounds like hard work, then you're exhibiting the realistic characteristic.
- *Care about your craft*: We who cut mere stones must always envision cathedrals.
- *Kaizen (japanisch)*: Das Konzept, jeden Tag ganz kleine Verbesserungen vorzunehmen, um schlussendlich durchgehend hohe Qualität zu erreichen.
- *Take responsibility*: It's your life. Du entscheidest, wie es aussieht.

2.1 Software Entropie

Entropie in Software wird als *Amount of Disorder* oder *Software Rot* beschrieben. Software altert. Um die gegebene Qualität über eine längere Zeit zu erhalten, sollten Probleme (sog. Broken Windows) wie zum Beispiel

- schlechtes Design
- falsche Entscheidungen
- unschöner Code

möglichst bald behoben werden. So kann dazu beigetragen werden, dass auch zukünftige Arbeiten am Code auf gewissenhafte und saubere Weise ausgeführt werden. Niemand will der erste sein, der eine gute Basis schlechter macht.

2.2 Veränderung

Veränderung und der richtige Umgang damit ist das zweite grundlegende Konzept in diesem Buch. Geschieht diese langsam und in kleinen Schritten, wird sie viel weniger stark wahrgenommen, als wenn auf einen Schlag etwas komplett anders ist. Im positiven kann man sich dies zu Nutze machen, indem man Verbesserungen langsam und schrittweise einführt, und den Menschen die nötige Zeit gibt.

Kleine, stetige Veränderungen in der Aussenwelt oder den Voraussetzungen in einem Projekt sind aber ebenso einfach zu verpassen, wie sie im positiven angenommen werden. Deshalb sollte man stets den Blick vom aktuellen, spezifischen Problem auch wieder aufs grosse Ganze richten und hinterfragen, ob man immernoch auf dem richtigen Weg ist.

3 Wissen, Technologien, Beherrschen der Tools

3.1 Knowledge Portfolio

Das beste und wichtigste Asset, das ein Programmierer in seinen Job mitbringt, ist Wissen. Unendlich viele verschiedene Technologien, die sich extrem schnell verändern machen es unumgänglich, dass man sich stetig weiterbildet. Auch wenn es nur einige Minuten am Tag oder in der Woche sind. Neue Technologien ausprobieren, Kurse besuchen, News und Publikationen lesen sind Tätigkeiten, die im Zeitplan einen fixen Platz haben sollten. Dies wird im Idealfall vom Arbeitgeber geschätzt und vergütet, da es vor allem auch diesem zu Gute kommt.

3.2 Shell Games, Power Editing, Text Manipulation

Wenn sich die grundlegenden, ständig wiederkehrenden Aufgaben der täglichen Arbeit auch nur ein wenig effizienter erledigen lassen, hat das eine grosse Auswirkung auf die Produktivität. Deshalb bringt es grosse Vorteile, sich mit Shell Commands oder Editing Shortcuts zu befassen. Eine kleine Investition an Zeit, die sich auszahlt.

3.3 Version Control

Egal ob bei der Arbeit im Team an einem längeren Projekt, oder nur zum sichern verschiedener Einstellungen auf dem eigenen Computer, bringt Version Control grosse Vorteile. Abgesehen von der offensichtlichen Funktionalität wie Branching oder dem einfachen Rollback auf einen früheren Stand, kann vor allem auch das automatisierte Builden, Testen oder Deployen von Code mittels Version Control Systemen automatisiert werden.

3.4 Debugging

Zum diesem Thema geben die Authoren verschiedene gute Tipps und Strategien.

- Fix the Problem, not the Blame: Wichtiger ist, dass das Problem behoben wird, nicht wer schuld ist.
- Don't Panic: Das Problem soll systematisch und überlegt angegangen werden, so wie auch die tägliche Arbeit erledigt wird. Im Panik-Modus werden wahrscheinlich nicht die besten Entscheidungen getroffen.

- Debugging Strategien: Abgesehen von den verschiedenen Vorgehensweisen beim Debugging (*Rubber Duck Debugging, Binary Chop, Elimination, Logging*) sollte grosser Wert auch darauf gelegt werden, dass der Bug mittels neuen Unit Tests zukünftig abgefangen wird, sodass dasselbe Problem nicht erneut gelöst werden muss.

4 The Essence of Good Design: Entscheidungen treffen

4.1 ETC - Easier to change

Ein System hat ein gutes Design, wenn es sich an die Menschen, die es nutzen, anpassen kann.

Easier to change ist deshalb das grundlegende Prinzip, auf dem fast alle weiteren Designprinzipien aufbauen. Es soll keine feste Vorgabe sein, sondern eine Hilfe. Immer wenn eine Entscheidung zwischen Vorgehen A oder B getroffen werden muss, kann man sich an diesem Grundsatz orientieren.

Um diese Denkweise zu verinnerlichen kann man sich diese Frage bewusst immer wieder stellen. Beim Speichern einer Datei, beim Schreiben eines Tests oder beim Beheben eines Bugs. Ist der Code, den ich soeben geschrieben habe easy to change?

4.2 DRY - Don't repeat yourself

Jedes Stück an Information braucht eine einzige, eindeutige und authoritative Repräsentation in einem System.

Das Duplizieren von Informationen führt unweigerlich zu Inkonsistenz. Sei dies im Code, in Dokumentationen oder anderswo. Weitere, nicht ganz so offensichtliche Arten von Duplikation sind die folgenden.

Duplikation der Repräsentation

Interne/Externe APIs: Das Wissen über die ausgetauschten Daten bzw. deren Struktur muss grundsätzlich auf beiden Seiten der Kommunikation vorhanden sein. Abhilfe schaffen können hier Sprachen zur neutralen Spezifikation von APIs. Solche Dokumente werden idealerweise zentral abgelegt und können zur Erstellung von automatisierten Tests und Test-Clients verwendet werden.

Datenquellen: Falls möglich sollten Objekte aus ihrer Dantenbankrepresentation automatisiert erstellt werden können. Andernfalls müssen Änderungen an der Struktur an verschiedenen Orten durchgeführt werden.

Duplikation zwischen Entwicklern

Diese Art von Duplikation ist wahrscheinlich am schwierigsten zu entdecken und adressieren. Ein wichtiges Mittel hierzu ist die Kommunikation von Teams zu stärken, innerhalb sowie übergreifend. Weiter muss es einfacher sein, bereits erstellte Funktionalität wiederzuverwenden, als sie neu zu implementieren.

4.3 Orthogonalität

Wenn in der Geometrie zwei Vektoren orthogonal zueinander stehen, sind sie voneinander komplett unabhängig.

In der Softwareentwicklung wurde diese Idee als wichtiges Grundprinzip übernommen: Wenn an einem Element eine Änderung vorgenommen wird, sollen davon möglichst wenige oder keine anderen Elemente beeinflusst werden. Code soll möglichst modular aufgebaut sein, aufgeteilt in kleine Einheiten die klar definierte Funktionalität bieten.

4.4 Good Enough

Die Qualität eines Produktes hängt von vielen Faktoren ab. Kosten, Zeit und Umfang eines Projektes spielen eine Rolle. Am wichtigsten ist es dabei, diese Faktoren auszubalancieren, sodass die Qualität *für den vorgesehenen Zweck* ausreichend ist und den Endbenutzer zufriedenstellt. Sind die Kosten am Schluss viel höher oder der Umfang viel kleiner als erwartet, ist bessere Qualität der erbrachten Leistung meist keine Rechtfertigung. *Know when to stop.*

4.5 Reversibility

Es gibt keine finalen Entscheidungen. Unser Code sollte damit klarkommen.

4.6 Weitere Konzepte

Listen to your lizard brain: Auf unsere Instinkte zu hören wird einem als Mensch heutzutage schnell abtrainiert. Das Unterbewusstsein als solches kann aber sehr oft hilfreich sein, und kann auch mittels verschiedenen Tricks wieder trainiert werden. Der einfachste und wohl gleichzeitig schwierigste davon ist, sich bei Problemen besser mit etwas völlig anderem abzulenken, anstatt sich immer weiter in ein sprichwörtliches Loch zu graben.

Programming by Coincidence: Etwas, das zwar funktioniert aber nicht wirklich verstanden wird, kann im nächsten Moment zu neuen Problemen führen, mit denen man nicht rechnet. Testen ist besser als annehmen.

5 Code

Wenn es um effektive Arbeitstechniken geht, wie *pragmatische Projekte* umgesetzt werden können, werden folgende Punkte diskutiert.

5.1 Tracer Bullets

Leuchtende Projektile sollen im Militär anzeigen, ob man die Zielscheibe auch wirklich trifft oder nicht. Genauso können beim Entwickeln kleine Einheiten von Funktionalität verwendet werden, um die End-zu-End Kommunikation quer durch alle Layers einer Architektur zu testen und zu garantieren. Wird das Ziel nicht erreicht, können unkompliziert Anpassungen in der Strategie oder im Toolset vorgenommen werden.

5.2 Prototypes

Prototypen sind lauffähige Systeme, die gebaut werden um bestimmte Aspekte eines Projektes auszuloten, neue Technologien zu testen oder Risiken zu minimieren -

ohne dass dieser mit dem Produktivsystem etwas zu tun hat. Punkte wie Korrektheit, Komplettheit, Robustheit und Coding-Stil können bewusst vernachlässigt werden.

5.3 Methodik

You can't write perfect Software. Um dieser Tatsache entgegenzuwirken, gibt es verschiedenste Grundlagen, die beachtet werden sollten.

Design by Contract: Methoden stellen Preconditions an den Aufrufer. Werden diese eingehalten, garantieren Sie gewisse Anforderungen an das Resultat, die Postconditions. Class Invariants sind Garantien, die dem Aufrufer einer Methode gemacht werden über den Status der Daten vor und nach dem Methodenaufruf.

Semantic Invariants: Manche Vorgehensweisen sind in bestimmten Kontexten implizit oder explizit vorgegeben. Beispielsweise wird bei Finanztransaktionen im Fehlerfall immer zugunsten des Kunden entschieden.

Crash Early: Im Fehlerfall ist es immer besser, die Aktion abzubrechen anstatt mit möglicherweise fehlerhaften Daten weitere Aufrufe zu tätigen. Der Fehler bleibt damit isoliert an Ort und Stelle des Geschehens.

Assertive Programming: Assertions können verwendet werden, um sicherzustellen dass das Unmögliche nicht eintritt. Fälle, die aus Sicht des Programmierers "garantiert nie vorkommen", können optimal abgefangen werden mittels einem Assertion Check. Somit ist sofort klar, falls irgendwo ein Fehler aufgetreten ist und der Unmögliche Fall dadurch eingetreten ist.

Transforming Programming: Mit dem Fokus darauf, dass ein Programm grundsätzlich einen gegebenen Input in einen anderen Output umwandelt, kann der Ablauf eines Programms auch als Abfolge von verschiedenen Transformationen auf gegebenen Daten angesehen werden.

Resource Balancing: Das Öffnen und Schliessen von Handles auf Dateien oder anderen Ressourcen muss immer koordiniert ablaufen. Die Verantwortung dazu muss klar definiert sein, auch und besonders im Fehlerfall.

Test to code: Macht man sich im Voraus Gedanken darüber, wie eine Funktionalität getestet werden soll, funktioniert in vielen Fällen die effektive Umsetzung einfacher und besser. Man betrachtet den Code nicht als Autor, sondern als Nutzer. Das Ziel ist klarer, die Umsetzung kann in kleinen Schritten erfolgen und das Resultat ist bereits getestet.

Refactoring: Immer dann, wenn man etwas gelernt hat. Wenn einem schlechtes Design oder Duplizierung auffällt. Wenn sich Anforderungen verändert haben. Wenn die Tests grün sind. Refactoring, das zu einfacherem und klarerem Code beiträgt, sollte wann immer möglich ausgeführt werden.

Outrunning your Headlights: Ein Appell, mit möglichst kleinen Schritten vorwärts zu kommen, um ganz im agilen Sinne den Kurs jederzeit korrigieren zu können. Wie beim Autofahren, wo man auf Sichtweite jederzeit anhalten können muss.

Stay safe out there: Heutzutage kann nicht mehr davon ausgegangen werden, dass sich niemand für eine Applikation interessiert, und man sie darum nicht schützen muss. Alles, was öffentlich erreichbar ist oder aus unsicherer Quelle stammt, sollte als verwundbar oder potenziell böswillig angeschaut werden, und erst nach gründlicher Überprüfung in der Software weiterverwendet werden. Kryptographie auf keinen Fall selber implementieren, hier auf gängige und getestete Lösungen setzen.

5.4 Bend or Break

Während das Prinzip *Easier To Change* besagt, dass unsere Entscheidungen immer auf möglichst einfach anpassbare Software abzielen sollten, geht dieses Kapitel darauf ein, wie diese Flexibilität erreicht werden kann.

Decoupling: Wenige Abhängigkeiten bilden zwischen verschiedenen Modulen, resultiert meist in *high cohesion*, was einen guten Zusammenhalt innerhalb eines Moduls beschreibt.

Globale Daten vermeiden: Globale Daten verbinden alle Module, die darauf zugreifen. Abhängigkeiten entstehen, Testing wird schwieriger, der Code wird schwieriger verständlich.

Unit Tests als Gradmesser: Wenn für einzelne Unit Tests verschiedenste Elemente aus dem Code eingebunden werden müssen, ist das ein schlechtes Zeichen.

Inheritance Tax: Auch Vererbung zwischen Klassen schränkt die Anpassbarkeit von Code ein, und sollte meist vermieden werden. Alternativen sind *Interfaces*, *Delegation* und *Mixins*.

Configuration: Mit externen Quellen für Konfigurationsdaten können Applikationen auf einfache Art und Weise an verschiedene Umgebungen und Kunden angepasst werden. Dies kann mittels Textfiles in gängigen Formaten wie JSON oder YAML erreicht werden, oder die Informatinen können in der Datenbank abgelegt und eventuell mit einem geeigneten Interface zugänglich gemacht werden.

5.5 Events

In der realen Welt geschehen Dinge, und wir reagieren darauf. Software sollte idealerweise dem selben Prinzip folgen, wenn reale Gegebenheiten abgebildet werden sollen. Dies kann auf verschiedene Arten implementiert werden, eine sehr elegante kann das Design einer *Finite State Machine* sein. Diese definiert verschiedene Zustände, Ereignisse und dazugehörige Übergänge bzw. Aktionen. Zwei weitere Varianten sind *Publish/Subscribe* oder *Streams*.

5.6 Concurrency

Mit dem Begriff *Temporal Coupling* wird die zeitliche Abhängigkeit zwischen verschiedenen Aktionen beschrieben: A muss ausgeführt werden, erst dann kann B gestartet werden. Solche Einschränkungen sind offensichtlich nicht wünschenswert. Effiziente Parallelisierung wird verunmöglicht.

Shared State is Incorrect State: Greifen mehrere Akteure auf dieselben Information zu und verändern diese auch, kann davon ausgegangen werden, dass Fehler passieren.

Actors and Processes: Die Kommunikation zwischen verschiedenen Akteuren und Prozessen geschieht via Nachrichten. State wird nur innerhalb von Nachrichten oder von jedem Akteur privat gehalten. Globaler State entfällt somit.

6 Kommunizieren

Kommunikation eines Programmierers kann sein wie man sich im Code ausdrückt (Prinzipien wie DRY, ETC, ..), aber natürlich auch auf sprachliche Art.

Meetings mit Kunden, Kollegen und Vorgesetzten machen einen wichtigen Teil der Tätigkeit aus und entscheiden meist über Erfolg oder Misserfolg eines Projekts.

Deshalb sollten auch hier die eigenen Fähigkeiten gepflegt und gestärkt werden. Wichtige Grundsätze dazu:

- Kenne dein Publikum
- Wisse, was du hinüberbringen willst
- Wähle den richtigen Moment
- Wähle eine passende Ausdrucksart
- Beziehe dein Publikum mit ein
- Höre zu
- Bleibe keine Antworten schuldig
- Bringe Optionen statt Ausreden
- **Ja sagen**, wenn man sich sicher ist, dass etwas in der gewünschten Zeit oder auf die gewünschte Weise umsetzbar ist.
- **Nein sagen**, wenn eine Deadline nicht sicher einzuhalten ist, man mehr Zeit zum abklären oder einschätzen braucht oder auf Unterstützung angewiesen ist.
- **Dokumentation** dort platzieren, wo sie gelesen wird.
  Das *wie* soll durch den Code selbsterklärend sein, das *warum* kann als Kommentar eingefügt werden.

7 Working in a (pragmatic) Team on (pragmatic) Projects

7.1 Estimating

Schätzwerte in der Softwareentwicklung sind sehr wichtig aber auch sehr schwierig, vor allem für unerfahrene Entwickler.

Die geforderte Genauigkeit und die angegebene Grösseneinheit sind voneinander abhängig, beispielsweise wird bei *anderthalb Monaten* viel mehr Ungenauigkeit erwartet oder in Kauf genommen, als bei einer Angabe von *30 Arbeitstagen.*

Die Basis für gute Schätzungen ist Erfahrung. Sind Anforderungen, Konzepte und verwendete Tools bekannt, ist die Antwort wahrscheinlich ziemlich genau. Andernfalls ist es praktisch unmöglich, eine Schätzung abzugeben, ohne mehr Zeit in Abklärungen zu investieren. Dies sollte immer klar kommuniziert werden.

Weiter sind Schätzwerte auch immer abhängig von verschiedenen äusseren Einflüssen, diese sollten genauso mit einbezogen werden. Eine gute Schätzung für alles ausser ganz simplen Anforderungen beinhaltet deshalb stets mehrere Szenarien, von optimistisch bis pessimistisch.

7.2 Requirements

Hier lautet der Grundsatz: *Niemand weiss genau, was er will.* Der Kunde weiss meist genausowenig, was eine Software alles bieten kann.

Als Softwareentwickler gehört es deshalb zur Aufgabe, mit dem Kunden zusammen herauszufinden, möglicherweise über zahlreiche Iterationen, ob das Produkt wirklich den Anforderungen entspricht, bzw. wie diese Anforderungen denn überhaupt aussehen. Unklarheiten müssen angesprochen und bereinigt werden.

Idealerweise kann der Entwickler sich mit den späteren End-Usern direkt austauschen, um deren Arbeitsweise und Bedürfnisse zu erlernen.

Ein Projekt-Glossar kann helfen, dass alle Beteiligten dieselbe Sprache sprechen.

7.3 Solving Impossible Puzzles

Wenn ein Problem nicht lösbar erscheint, hilft es meist, einen Schritt zurück zu machen. Sind die Annahmen zu allen Voraussetzungen richtig? Gibt es Variablen, die verändert werden könnten?

Oft hilft eine Pause, Ablenkung, oder das Problem einer anderen Person (oder der Rubber Duck) zu erklären.

7.4 Working together

Methoden wie Pair- oder Mob Programming helfen in verschiedenen Hinsichten: der Lerneffekt für unerfahrene Programmierer ist gross, der sofortige Qualitäts-Review durch andere Personen verbessert das Resultat, und mögliche Probleme, die alleine viel Zeit in Anspruch nehmen würden, können im Team meist insgesamt schneller gelöst werden.

8 *agile thinking*

Stetiges Hinterfragen der Situation, der Entscheidungen und der Methoden.
Requirements und Feedback im Loop abholen.

Und nicht vergessen, in Stress-Situationen: