Das Ende moderner Softwareentwicklung

Oliver Szymanski
Selbstständig
Source-Knights.com

Schlüsselworte
Java, Softwareentwicklung, Prozesse, Frameworks, Tools

Einleitung
Anforderungen, die von heutigen Softwareprojekten erfüllt werden sollen, werden immer komplizierter. Die Möglichkeiten, die bessere Hardware und die Evolution von Software bieten, lässt die Bedürfnisse an Software mehr werden. Entwurfsmuster, Frameworks, API-Spezifikationen mit verschiedensten Referenzimplementierungen stellen die Mittel dar, mit denen wir Software bauen. Auch sie und ihre Vielfalt sind in der Vergangenheit gewachsen. Dazu kommen Werkzeuge wie komplexe Entwicklungsumgebungen, diverse Plugin zu ihnen, und Tools, Tools, Tools. Gewürzt wird das ganze mit immer ausgefeilteren Buildprozessen.

Moderne Software
Dies alles unterlag in den letzten Jahren der Evolution. Und generell betrachtet sind alle der genannten Komponenten wertvoll und wichtig für den Softwareentwicklungsprozess. Dennoch, vor lauter Hilfsmitteln und Entwurfsmustern haben viele vergessen, worauf es eigentlich ankommt: Funktionierende Software als Ergebnis, meist mit dem Anspruch erweiterbar und wartbar zu sein. Wen frustrieren nicht Stacktraces, die hunderte von generischen Methodennamen enthalten, obwohl eigentlich nur ein einziger simpler Methodenaufruf vorliegt. Oder die unzähligen Konfigurationsmöglichkeiten dank der eingesetzten Frameworks, die so Vielfältig sind, dass man selten weiss, ob es jetzt konsistent konfiguriert ist. Klar kann man bei all diesen Problemen sagen, dass ein Plugin in der IDE dabei Abhilfe schaffen kann. Aber kann mehr zusätzliche Software wirklich dabei helfen, Probleme mit zuviel Komplexität zu beseitigen?

So sinnvoll und hilfreich z.B. Entwurfsmuster sind, sie wurden in zahlreichen auch namhaften Frameworks ad absurdum geführt. Es gab Zeiten, in denen man Entwicklern erklären musste, wann sie ein Entwurfsmuster nutzen sollen. Heute ist es oft umgekehrt, und man muss ihnen sagen, wo sie es besser lassen.

Komplexität ist meiner Meinung nach die größte Herausforderung, der wir uns in der Softwareentwicklung stellen müssen. Wenn wir nicht lernen, damit sinnvoll umzugehen, läuten wir selbst das Ende der modernen Softwareentwicklung ein und befinden uns in einer Lage nach dem Motto: versuch doch einfach mit dem Hammer da links drauf zu hauen, vielleicht geht es dann. Und leider sehe ich bei immer mehr Kunden solch ein Verhalten, dass es abzustellen gilt.

Husky-Faktor
Intuitive Benutzung, selbsterklärende Features, dass alles ist wünschenswert in Tools, Frameworks, Entwicklungsumgebungen und Programmiersprachen. Von meiner Huskydame habe ich gelernt, dass schnelles Feedback dazu gehört. Erfährt ein Hund nicht innerhalb von 2 Sekunden Belohnung oder Tadel (auf das Wort Strafe verzichte ich in bewusst, jeder vernünftige Hundeerzieher wird es mir danken), lernt er nichts. D.h. wenn ich Heim komme, und die Kleine hat etwas zerstört während ich nicht daheim war, sollte ich keineswegs wütend reagieren. Denn sie könnte es nicht mehr mit ihrer Tat

in Verbindung bringen. Das gilt übrigens auch in dem Fall, dass man sie immer wieder mit der Nase darauf stößt - wörtlich. Sie lernt dann lediglich, dass sie allein daheim ist, und ich wütend nach Hause komme. Das wird sie beim nächsten Mal allein zu Hause wieder in Unruhe ausbrechen lassen wird, und erhöht die Gefahr, dass sie wieder etwas kaputt beisst. Schnelles Feedback ist wichtig. Beim Hund gilt das Prinzip: Reiz, Reaktion, Verstärkung (Lob/Belohnung, Tadel). Am besten mit einer Abfolge unter 2 Sekunden. Auf diese Weise lernt ein Hund, ebenso meine Huskydame. Oft Dinge, die sie nicht lernen soll. Z.B. kann sie sich so beibringen, dass Wurst auf dem Tisch (Reiz) schnell geschnappt (Reaktion) verdammt gut schmeckt (verstärkende Belohnung). Diesem Lerneffekt muss man erst einmal etwas (schnell) entgegensetzen. Ich traue uns Menschen durchaus zu, auch nach 2 Sekunden noch etwas über eine vergangene Tat zu lernen. Allerdings sehe ich in Projekten immer wieder, dass ein Entwickler nach dem Niederschreiben von Codezeilen, dem folgenden Build und anschließendem Deployen seines Projektes nicht mehr weiss, warum der Server ihn mit Fehlermeldungen tadelt. Dann geht die kostenspielige Fehlersuche los. Das ist dann fast so schlimm, wie der Weg zur Erkenntnis, dass die Entwicklungsumgebung keinen simplen Knopf zur Erzeugung einer Archivdatei zur Verfügung stellt. Noch schlimmer, wenn der Server beim Deployen gar keine Fehlermeldungen wirft (sondern erst später beim Nutzen irgendwelches Applikcationsfunktionen). Denn dann ist es wie beim falschen Lerneffekt meiner Hündin. Der Entwickler denkt es ist alles richtig, denn der Server hat sich ja nicht beim Deployen beschwert.

Machen wir es uns nicht selber schwer und versuchen bei der Auswahl unserer Entwicklungsumgebung, Tools und Frameworks auf den Husky-Faktor zu achten: schnelles Feedback ohne falschen Lerneffekt. Das spart Nerven, Zeit und somit Geld. Und lernen wir von den Kindern, was selbsterklärend und intuitiv zu bedienen bedeutet. Zwar wollen wir keine Kinder arbeiten lassen. Aber es kann auch neuen Teammitgliedern bei der Einarbeitung helfen. Und allen ermöglichen sich auf die wesentlichen Dinge, die wir implementieren wollen, zu konzentrieren.

Komplexität

Komplexität liegt auch im Verständnis von Code. Aber von dieser offensichtlichen Komplexität abgesehen, ist Komplexität auch im Laufzeitverhalten (wie lange braucht ein Algorithmus bezogen auf die Eingabemenge, auf der er arbeitet) und Ressourcenbedarf eines Algorithmus zu betrachten.
Die Erfahrung zeigt, dass viele Entwickler wenig über Abschätzungen dieser Arten von Komplexität wissen. Kaum jemand kennt die O-Notation oder weiß, dass manche Algorithmen trotz doppelt so schneller CPU nicht doppelt so schnell arbeiten. Das erkennt man u.a. immer wieder daran, wenn in einem Projekt behauptet wird, dass man dies über Hardware skalieren kann. Ob dies geht, hängt nämlich davon ab, in welche Laufzeitkategorie ein Algorithmus fällt. Die Komplexität eines Algorithmus kann unter seinem Speicherplatzbedarf oder der Laufzeit betrachtet werden.
Ein Algorithmus mit konstanter Laufzeit hat ein Laufzeitverhalten, das auf einem X-Y-Koordinatensystem eine Parallele zur X-Achse zeigt. Die X-Achse steht dabei für die Eingabemenge und wie sie genau zu interpretieren ist hängt davon ab, auf welchen Daten der Algorithmus arbeitet. Die Y-Achse steht für die Komplexität, die der Algorithmus benötigt. Komplexität lässt sich also als Kurve in Abhängigkeit von der Eingabegröße „n“ darstellen. Ein Sortieralgorithmus hat z.B. auf der X-Achse die Anzahl von Objekten die sortiert werden. Also eine Funktion der Form $f(x) = k$ mit k als Konstante. Bei der O-Notation schätzt man solch ein Verhalten ab und kategorisiert einen Algorithmus während man vernachlässigt, wie lange er genau braucht. Es reicht zu Wissen, dass er sich konstant verhält. Daher schreibt man, er hat ein Lauftzeitverhalten von $O(1)$ statt $O(k)$. Ein Sortieralgorithmus, der $O(1)$ braucht, also immer gleich lange läuft egal wieviele Daten er sortiert, ist natürlich nicht in der Realität machbar. Algorithmen, die sich in der Komplexität linear zur Eingabegröße „n“ verhalten, werden mit $O(n)$ klassifiziert. Dies gilt z.B. für eine einfache For-Schleife über die Eingabemenge, die keine rekursiven Aufrufe oder weitere Schleifen enthält. Ein typischer rekursiver Algorithmus hat häufig $O(n^2)$, also quadratische Laufzeit. Warum keine realen Zeitangaben bei Laufzeitabschätzungen? Weil diese Abhängig von der Umgebung sind, auf der der Algorithmus läuft. Hier geht es aber darum, wie der Algorithmus sich verhält. Damit lässt sich z.B. die Frage beantworten, wie lange der

Algorithmus benötigt, wenn die Anzahl an eingehenden Daten sich verdoppelt. Bei quadratischer Laufzeit wäre dies 4x so hoch. Das gleicht dann auch keine doppelt so schnelle CPU aus.
Man sollte immer versuchen effiziente Algorithmen zu entwickeln. Dies sind Algorithmen, die mit polynomiellem Aufwand lösbar sind. Also wenn die Kurve mit einem Polynom angegeben/abgeschätzt werden kann. Probleme, die „schlechtere“ Laufzeiten haben, gelten in der theoretischen Informatik als „praktisch nicht lösbar“. Darauf basiert u.a. Kryptographie. Die erforderlichen Algorithmen um einen Schlüssel zu knacken sind meist nicht effizient, und dies bedeutet, dass sie einfach viel zu viele Jahre laufen müssten, um die Schlüsselgrößen zu verarbeiten.

Direct-Call-Pattern
Als Anregung möchte ich ein Entwurfsmuster von David Tanzer und mir erwähnen, das vielleicht noch nicht alle kennen: das Direct-Call-Pattern. Es soll zum Schmunzeln anregen. Und danach zum Nachdenken.

Einführung:
2 Objekte wollen miteinander kommunizieren. Einer ist der Aufrufer (Caller), der andere der Aufgerufene (Callee). Für den Aufgerufenen ist das völlig in Ordnung und der Aufrufer macht alles erforderliche vor und nach dem Aufruf gerne selbst.

Anforderungen:
- 2 Objekte, manchmal 1 Objekt in 2 Rollen (Aufrufer, Aufgerufener)
- Sonst nichts

Lösung:
Der Aufrufer ruft den Aufgerufenen direkt auf. Kein Proxy, Interceptior oder sonst ein Vermittlerobjekt sind im Aufruf involviert. Wirklich, bloss ein Aufruf von einem vertrauenswürdigen Freund zum anderen. Vielleicht loggt eine dritte Partei den Aufruf, aber das lässt sich heutzutage schwer vermeiden.

Pros:
- Schneller Aufruf
- Saubere und kurze Stack Traces
- Weniger Verwirrung
- WYSIWYG

Contra:
- Viele "WTF ist die Dependency Injection" und "wo finde ich den Interceptor" Kommentare
- Verärgert das nicht benötigte Annotationen- und Aspektorientierungs-Zeug
- Andere können den Code verstehen, den man geschrieben hat
- Weniger Geld mit Software Support

Nutzen wenn:
- Kein Geld für Application Server vorhanden
- Keine Zeit um Frameworks zu debuggen
- Man sich nicht nach langen Zeiten voller exklusivem Consulting sehnt
- Projekt-Laufzeit länger als einige Wochen/Monate (und man soll sich selbst ohne Aufpreis um den Support kümmern)

Fazit
Wir müssen an Lösungen arbeiten und nicht an neuen Problemen. Darauf sollte moderne Softwareentwicklung abzielen. Evolution bedeutet nämlich auch, das Komponenten, die nicht für unseren Fall nicht effizient sind, zurücktreten. Dies gilt auch, wenn sie zwar eine Lösung bieten, aber es simplere, für uns im Hinblick auf unsere Anforderungen effizientere Möglichkeiten gibt. Und es

bezieht sich ebenso auf Standardlösungen der Industrie. Nicht jedes Projekt benötigt immer den vollständigen Enterprise-Stack an verfügbaren Komponenten.

Kontaktadresse:
Oliver Szymanski
Freiberufler, Source-Knights.com
E-Mail oliver.szymanski@source-knights.com
Internet: Source-Knights.com