



Michael C.
Feathers

Effektives Arbeiten mit **Legacy Code**

Refactoring und Testen bestehender Software

Teil I

Wie Wandel funktioniert

In diesem Teil:

- **Kapitel 1**
Was ist Seam? 21
- **Kapitel 2**
Mit Feedback arbeiten 33
- **Kapitel 3**
Überwachung und Trennung 45
- **Kapitel 4**
Das Seam-Modell 53
- **Kapitel 5**
Tools 69

Software ändern

Code zu ändern, ist etwas Großartiges. Wir verdienen damit unseren Lebensunterhalt. Aber es gibt Methoden, Code zu ändern, die das Leben erschweren, und es gibt Methoden, die es erheblich erleichtern. In der Branche wurde nicht viel darüber geredet. Der Sache am nächsten kommt noch die Literatur über Refactoring. Ich glaube, wir sollten die Diskussion etwas breiter anlegen und überlegen, wie wir in den schlimmsten Situationen mit Code umgehen sollten. Zu diesem Zweck müssen wir uns zunächst näher mit der Mechanik von Änderungen befassen.

1.1 Vier Gründe, Software zu ändern

Der Einfachheit halber möchte ich vier Hauptgründe unterscheiden, Software zu ändern:

1. Eine Funktion hinzufügen
2. Einen Fehler beseitigen
3. Das Design verbessern
4. Die Nutzung von Ressourcen optimieren

1.1.1 Funktionen hinzufügen und Fehler beseitigen

Eine Funktion hinzuzufügen, scheint mir die unkomplizierteste Art von Änderungen zu sein. Die Software zeigt ein Verhalten, und der Anwender erwartet von dem System auch noch ein anderes Verhalten.

Angenommen, wir arbeiteten an einer Webanwendung und ein Manager teilte uns mit, das Unternehmenslogo solle nicht auf der linken, sondern auf der rechten Seite stehen. Wir sprechen mit ihm darüber und stellen fest, dass dies nicht ganz so einfach ist. Er möchte das Logo verschieben, aber erwartet auch andere Änderungen. Es soll beim nächsten Release animiert werden. Gehört dies in die Kategorie »Fehler beseitigen« oder »Neue Funktion hinzufügen«? Das hängt von Ihrem Standpunkt ab. Aus der Sicht des Kunden handelt es sich definitiv um die Beseitigung eines Problems. Vielleicht hat er die Webseite gesehen und ein Meeting mit Mitarbeitern seiner Abteilung veranstaltet; und sie haben beschlossen, das Logo an eine andere Stelle zu setzen und ein wenig mehr Funktionalität zu fordern. Auf der Sicht eines Entwicklers kann die Änderung als vollkommen neue

Funktion eingestuft werden. »Wenn die Abteilung einfach aufhören würde, ständig ihre Meinung zu ändern, wären wir jetzt fertig.« Aber in einigen Unternehmen wird eine Verschiebung eines Logos einfach als Beseitigung eines Fehlers gesehen, ungeachtet der Tatsache, dass das Team dafür umfangreiche neue Arbeit leisten muss.

Man könnte dies leicht als subjektive Einschätzung abtun. Für Sie ist dies ein zu beseitigender Fehler, für mich eine neue Funktion, also was? Leider müssen in vielen Unternehmen Korrekturen von Fehlern und Erweiterungen um neue Funktionen unterschiedlich überwacht und abgerechnet werden, weil es auch noch Verträge, Garantien oder Qualitätsinitiativen gibt. Zwar können wir endlos darüber diskutieren, ob wir Funktionen hinzufügen oder Fehler beheben, aber letztlich müssen Code und andere Artefakte geändert werden. Dieser Streit auf semantischer Ebene, was die Tätigkeit denn nun letztlich sei, maskiert etwas für uns technisch viel Wichtigeres: die Verhaltensänderung eines Systems. Und dabei bedeutet es einen großen Unterschied, ob neues Verhalten hinzugefügt oder altes geändert wird.

Verhalten ist der wichtigste Aspekt von Software. Es ist der Grund, warum Anwender Software verwenden. Anwender lieben es, wenn wir Verhalten hinzufügen (vorausgesetzt, es leistet, was sie wirklich wollten), aber wenn wir Verhalten ändern oder entfernen, das sie benötigen (Fehler einführen), verlieren wir ihr Vertrauen.

Fügen wir in unserem Unternehmenslogo-Beispiel Verhalten hinzu? Ja; denn nach der Änderung wird das System ein Logo auf der rechten Seite anzeigen. Beseitigen wir Verhalten? Ja; denn es gibt kein Logo mehr auf der linken Seite.

Betrachten wir einen schwierigeren Fall. Angenommen, ein Kunde wolle ein Logo rechts auf einer Webseite anzeigen, aber es gäbe kein Logo auf der linken Seite, mit dem wir anfangen könnten. Ja; wir fügen Verhalten hinzu; aber entfernen wir auch Verhalten? Wurde an der Stelle, an der das Logo erscheinen soll, irgendetwas anderes angezeigt?

Ändern wir Verhalten, fügen wir Verhalten hinzu, oder beides?

Wir können eine Unterscheidung treffen, die für uns als Programmierer nützlicher ist. Wenn wir Code modifizieren müssen (und HTML zählt in diesem Fall als Code), könnten wir Verhalten ändern. Wenn wir nur Code hinzufügen und ihn aufrufen, fügen wir oft Verhalten hinzu. Betrachten wir ein anderes Beispiel, eine Methode einer Java-Klasse:

```
public class CDPlayer
{
    public void addTrackListing(Track track) {
```

```
    ...  
  }  
  ...  
}
```

Die Klasse enthält eine Methode, mit der wir Track-Listings (etwa mit den Songs einer CD) hinzufügen können. Fügen wir eine Methode hinzu, mit der wir Track-Listings ersetzen können:

```
public class CDPlayer  
{  
    public void addTrackListing(Track track) {  
        ...  
    }  
  
    public void replaceTrackListing(String name, Track track) {  
        ...  
    }  
    ...  
}
```

Haben wir mit dieser Methode neues Verhalten zu unserer Anwendung hinzugefügt oder haben wir Verhalten geändert? Weder noch. Eine Methode hinzuzufügen, ändert Verhalten erst, wenn die Methode irgendwie aufgerufen wird.

Ändern wir den Code erneut. Wir wollen einen neuen Button in die Benutzerschnittstelle des CD-Players einfügen und ihn mit der `replaceTrackListing`-Methode verknüpfen. Damit fügen wir das Verhalten hinzu, das wir in der `replaceTrackListing`-Methode spezifiziert haben; aber wir ändern auch Verhalten auf subtile Weise; denn die Benutzerschnittstelle wird mit diesem neuen Button etwas anders dargestellt. Möglicherweise dauert es eine Mikrosekunde länger, bis es komplett angezeigt wird. Es scheint fast unmöglich zu sein, Verhalten hinzuzufügen, ohne zugleich vorhandenes Verhalten bis zu einem gewissen Grad zu ändern.

1.1.2 Das Design verbessern

Das Design zu verbessern, ist eine weitere Art von Software-Änderung. Wir wollen die Software umstrukturieren, etwa damit sie wartungsfreundlicher wird, wobei ihr Verhalten im Allgemeinen bewahrt werden soll. Wird dabei Verhalten, vielleicht aus Versehen, entfernt, bezeichnen wir dies oft als Bug (Fehler). Einer der Hauptgründe, warum viele Programmierer nicht versuchen, das Design zu verbessern, liegt darin, dass dabei leicht Verhalten verloren gehen, beschädigt oder unerwünscht verändert werden kann.

Der Prozess, Design zu verbessern, ohne Verhalten zu ändern, wird als *Refactoring* bezeichnet. Es basiert auf der Idee, dass wir Software wartungsfreundlicher machen können, ohne ihr Verhalten zu ändern, wenn wir Tests schreiben, mit

denen wir kontrollieren, dass das vorhandene Verhalten nicht geändert wird, und in kleinen Schritten vorgehen, um dies nach jedem Schritt zu verifizieren. Entwickler säubern schon seit Jahren den Code vorhandener Systeme; aber erst in den letzten Jahren hat sich das Refactoring verbreitet. Es unterscheidet sich von allgemeinen Säuberungen darin, dass wir nicht einfach risikoarme Änderungen wie etwa eine Umformatierung von Quellcode oder invasive und riskante Dinge wie etwa das Umschreiben ganzer Code-Fragmente vornehmen, sondern dass wir eine Reihe kleiner struktureller Änderungen vornehmen und dabei von Tests unterstützt werden, die das Ändern des Codes erleichtern. Die Essenz des Refactorings besteht darin, Verhalten zu bewahren, während funktionale Änderungen Verhalten modifizieren.

1.1.3 Optimierung

Optimierung ähnelt dem Refactoring, verfolgt aber ein anderes Ziel. Bei beiden wird die Funktionalität nicht geändert, aber beim Refactoring wird die Programmstruktur geändert, während bei der Optimierung die Nutzung von Ressourcen (Zeit, Speicherplatz usw.) verbessert wird.

1.1.4 Alles im Überblick

Doch ähnelt das Refactoring der Optimierung tatsächlich viel stärker als dem Hinzufügen von Funktionen oder der Beseitigung von Fehlern? Refactoring und Optimierung haben gemeinsam, dass die Funktionalität invariant bleibt, während etwas anderes geändert wird.

Im Allgemeinen können wir bei der Arbeit an einem System drei verschiedene Aspekte ändern: Struktur, Funktionalität und Ressourcenverbrauch.

Was ändert sich normalerweise und was bleibt im Wesentlichen konstant, wenn wir vier unserer verschiedenen Arten von Änderungen vornehmen (ja, oft ändern sich alle drei Aspekte, aber wir wollen das Typische betrachten):

	Funktion hinzufügen	Fehler beseitigen	Refactoring	Optimierung
Struktur	Ändert sich	Ändert sich	Ändert sich	-
Funktionalität	Ändert sich	Ändert sich	-	-
Ressourcenverbrauch	-	-	-	Ändert sich

Oberflächlich sehen sich Refactoring und Optimierung sehr ähnlich. Sie halten die Funktionalität invariant. Aber was passiert, wenn wir neue Funktionalität separat betrachten? Wenn wir eine Funktion hinzufügen, führen wir eine neue Funktionalität ein, aber ohne vorhandene Funktionalität zu ändern.

	Funktion hinzufügen	Fehler beseitigen	Refactoring	Optimierung
Struktur	Ändert sich	Ändert sich	Ändert sich	-
Neue Funktionalität	-	Ändert sich	-	-
Ressourcenverbrauch	-	-	-	Ändert sich

Beim Hinzufügen von Funktionen, beim Refactoring und bei der Optimierung bleibt die vorhandene Funktionalität konstant. Und wenn wir das Beseitigen von Fehlern genauer betrachten, stellen wir zwar fest, dass wir damit Funktionalität ändern; aber die Änderungen sind oft sehr klein, verglichen mit der insgesamt vorhandenen Funktionalität, die nicht geändert wird.

Das Hinzufügen von Funktionen und das Beseitigen von Fehlern ähneln stark dem Refactoring und der Optimierung. In allen vier Fällen wollen wir Funktionalität oder Verhalten ändern, aber gleichzeitig viel mehr bewahren (siehe Abbildung 1.1).

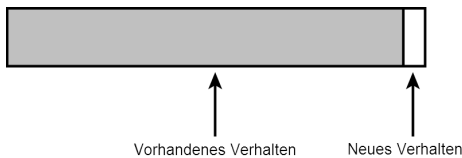


Abb. 1.1: Verhalten bewahren

Was bedeutet diese detaillierte Analyse der möglichen Änderungen für unsere praktische Arbeit? Positiv betrachtet scheint sie uns zu sagen, worauf wir uns konzentrieren müssen. Wir müssen dafür sorgen, dass die kleine Anzahl der Dinge, die wir ändern, korrekt geändert werden. Negativ betrachtet lernen wir, dass dies nicht das Einzige ist, auf das wir uns konzentrieren müssen. Wir müssen herausfinden, wie wir den Rest des Verhaltens bewahren können. Dazu gehört leider mehr, als einfach den Code in Ruhe zu lassen. Wir müssen Gewissheit haben, dass sich das Verhalten nicht ändert, und das kann sehr schwierig sein. Das Verhalten, das wir bewahren müssen, ist normalerweise sehr umfangreich, aber das ist nicht das Problem. Das Problem ist, dass wir oft nicht wissen, in welchem Umfang Verhalten durch unsere Änderungen gefährdet ist. Andernfalls könnten wir uns auf dieses Verhalten konzentrieren und den Rest ignorieren.

1.2 Riskante Änderungen

Um Risiken zu verändern, müssen wir drei Fragen stellen:

1. Welche Änderungen müssen wir vornehmen?

2. Wie erfahren wir, dass wir sie korrekt vorgenommen haben?
3. Wie können wir sicher sein, dass wir nichts beschädigt haben?

Wie viele Änderungen können Sie sich leisten, wenn Änderungen riskant sind?

Die meisten Teams, mit denen ich gearbeitet habe, arbeiten mit einem sehr konservativen Risikomanagement. Sie minimierten die Anzahl der Änderungen ihrer Code-Basis. Manchmal handeln sie nach der Maxime: »Wenn es nicht kaputt ist, fass es nicht an.« In anderen Teams werden Änderungen »kleingeredet«. Die Entwickler sind einfach sehr vorsichtig, wenn sie Änderungen vornehmen: »Was? Sie erstellen dafür eine andere Methode?« Antwort: »Nein, ich füge nur die Code-Zeilen direkt hier in die Methode ein, wo ich gleichzeitig den Rest des Codes sehen kann. Ich muss weniger editieren, und es ist sicherer.«

Es ist verlockend zu denken, wir können Software-Probleme minimieren, indem wir sie ignorieren; aber leider holt uns die Wirklichkeit immer ein. Wenn wir vermeiden, neue Klassen und Methoden zu erstellen, werden die vorhandenen immer größer und unübersichtlicher. Wenn Sie ein umfangreiches System ändern, müssen Sie damit rechnen, dass es eine Weile dauert, mit dem Arbeitskontext vertraut zu werden. Gute und schlechte Systeme unterscheiden sich auch dadurch, dass Sie bei guten nach dieser Lernphase ein Gefühl der Sicherheit haben und sich zutrauen, die Änderungen erfolgreich vorzunehmen. Wenn Sie dagegen schlecht strukturierten Code nach der Lernphase ändern wollen, haben Sie eher das Gefühl, von einer Klippe zu springen, um einem Tiger zu entkommen. Sie zögern diesen Schritt immer weiter hinaus: »Bin ich bereit dafür? Nun, mir bleibt wohl nichts anderes übrig.«

Änderungen zu vermeiden, hat auch andere negative Konsequenzen. Wer Code nicht ändert, verliert oft die Fähigkeit dafür. Eine große Klasse in Teile zu zerlegen, kann ziemlich anstrengend sein, wenn Sie es nicht mehrfach pro Woche tun. Andernfalls wird es zur Routine. Sie erkennen immer besser, was kaputtgehen kann und was nicht, und die Arbeit geht viel leichter von der Hand.

Die letzte Konsequenz, Änderungen zu vermeiden, ist Angst. Leider haben viele Teams eine unglaubliche Angst vor Änderungen; und jeden Tag wird es schlimmer. Oft merken die Mitglieder gar nicht, wie viel Angst sie haben, bis sie bessere Techniken kennen lernen und die Angst langsam nachlässt.

Jetzt haben Sie erfahren, dass es schlecht ist, Änderungen zu vermeiden; aber welche Alternativen gibt es? Eine Alternative besteht einfach darin, sich mehr anzustrengen. Vielleicht können wir mehr Entwickler einstellen, damit alle genügend Zeit für Studium und Analyse des Codes haben und die Änderungen »richtig« durchgeführt werden. Sicher, mehr Zeit und bessere Analysen machen Änderungen sicherer. Oder etwa nicht? Wie kann ein Team nach allen Analysen sicher sein, ob es alles richtig verstanden hat?