7 Die Übel der Wiederholung

Captain Kirks bevorzugter Trick, um wild gewordene, künstliche Intelligenz auszuschalten, war, dem Computer zwei widersprüchliche Informationen zu geben. Dummerweise kann dieses Vorgehen auch Ihren Quelltext in die Knie zwingen.

Als Programmierer sammeln, organisieren, pflegen und nutzen wir Wissen. Wir dokumentieren es in Spezifikationen, wir erwecken es in Quelltext zum Leben und wir nutzen es beim Testen.

Leider ist Wissen nicht stabil. Es verändert sich – meist sehr schnell. Ihr Verständnis der Anforderungen ändert sich eventuell nach einer Besprechung mit Ihrem Kunden. Die Regierung ändert ein Gesetz und lässt damit Geschäftslogik veralten. Tests zeigen, dass der gewählte Algorithmus nicht funktioniert. All diese Instabilitäten führen dazu, dass wir einen Großteil unserer Zeit damit verbringen, das Wissen in unseren Systemen neu zu organisieren und neu zu formulieren.

Die meisten Leute denken, dass Wartung anfängt, wenn die Anwendung ausgeliefert wird, und dass Wartung bedeutet, Fehler zu beseitigen und Funktionen zu verbessern. Wir glauben, dass diese Leute Unrecht haben. Programmierer befinden sich in einer ständigen Wartungsphase. Unser Verständnis ändert sich täglich. Während wir entwerfen und implementieren, erhalten wir neue Anforderungen. Manchmal ändert sich das Umfeld. Egal aus welchem Grund auch immer: Wartung ist keine abgeschlossene Aktivität, sondern ein routinemäßiger Bestandteil des gesamten Entwicklungsprozesses.

Wenn wir Wartung betreiben, müssen wir die Repräsentationen der Dinge – in der Anwendung verstreute Wissenskapseln – finden und ändern. Das Problem dabei ist, dass Wissen in Spezifikationen, Prozessen und Programmen leicht dupliziert werden kann. Wenn wir das in Programmen tun, lösen wir damit einen Albtraum bei der Wartung aus, der lange vor der Auslieferung seinen Lauf nimmt.

Wir sind überzeugt, dass es nur einen einzigen Weg zu zuverlässiger, verständlicher und wartbarer Software gibt. Wir nennen ihn: DRY-Prinzip.

> Jedes Stück Wissen muss eine einzige, eindeutige und maßgebliche Repräsentation in einem System haben.

Wieso nennen wir das DRY?

Tipp 11: ***DRY – Don't Repeat Yourself (dt. Wiederholen Sie sich nicht)*** ◄

Die Alternative dazu ist, dasselbe Stück Wissen an zwei oder mehr Stellen zu formulieren. Wenn eine Stelle geändert wird, müssen Sie daran denken, auch die andere zu ändern. Sonst zwingt ein Widerspruch Ihr Programm genauso in die Knie wie die Computer der Außerirdischen. Die Frage ist nicht, ob Sie daran denken, sondern wann Sie es vergessen.

Das DRY-Prinzip wird in diesem Buch immer wieder auftauchen, meist in Zusammenhängen, die nichts mit Implementierung zu tun haben. Wir sind der Überzeugung, dass es eines der wichtigsten Werkzeuge im Repertoire eines Pragmatischen Programmierers ist.

In diesem Abschnitt werden wir die Probleme umreißen, die durch Wiederholungen entstehen, und allgemeine Strategien vorschlagen, um damit umzugehen.

Wie entstehen Wiederholungen?

Die meisten Wiederholungen, die wir entdeckten, fallen in eine der folgenden Kategorien:

- **Erzwungene Wiederholung.** Software-Entwickler meinen, keine Wahl zu haben. Die Umstände scheinen die Wiederholung zu erfordern.
- **Unabsichtliche Wiederholung.** Software-Entwickler bemerken nicht, dass sie Informationen wiederholen.
- **Wiederholung aus Ungeduld.** Software-Entwickler werden faul und bauen Wiederholungen ein, weil es ihnen einfacher erscheint.
- **Wiederholungen durch mehrere Entwickler.** Verschiedene Software-Entwickler in einem Team (oder in verschiedenen Teams) wiederholen dieselbe Information.

Lassen Sie uns diese vier Ursachen von Wiederholungen genauer betrachten.

Erzwungene Wiederholung

Manchmal scheinen wir zu Wiederholungen gezwungen zu sein. Projektstandards schreiben eventuell Dokumente vor, die Informationen duplizieren oder Informationen aus dem Quelltext wiederholen. Verschiedene Zielplattformen können eigene Programmiersprachen, Bibliotheken und Entwicklungsumgebungen erforderlich machen. Programmiersprachen erzwingen selbst gewisse Strukturen, die Informationen duplizieren. Wir alle haben Situationen kennen gelernt, in denen man gegen Wiederholungen scheinbar machtlos war. Dennoch gibt es häufig Mittel und Wege, jedes Stück Wissen dem DRY-Prinzip folgend an einer einzigen Stelle zu formulieren und uns damit das Leben zu erleichtern. Hier sind einige davon:

Verschiedene Repräsentationen einer Information. Bei der Implementierung benötigen wir dieselbe Information oft in verschiedenen Repräsentationsformen. Sei es, dass wir eine Client-Server-Anwendung mit unterschiedlichen Programmiersprachen auf Client und Server schreiben und gemeinsame Strukturen in beiden beschreiben müssen, oder wir brauchen Klassen, die das Schema einer Datenbank widerspiegeln. Vielleicht schreiben wir an einem Buch und wollen Ausschnitte aus Quelltexten einfügen, die wir aber auch kompilieren und testen wollen.

Mit etwas Einfallsreichtum kann man im Allgemeinen den Zwang zu Wiederholungen umgehen. Häufig ist ein einfacher Filter oder Quelltextgenerator die Lösung dafür. Strukturen in verschiedenen Sprachen können mit einem einfachen Quelltextgenerator aus gemeinsamen Metadaten bei jedem Build-Lauf neu erzeugt werden (ein Beispiel finden Sie in Abbildung 3.4.) Klassendefinitionen können automatisch aus einem vorhandenen Datenbankschema erzeugt werden oder aus den Metadaten, aus denen schon das Schema erzeugt wurde. Die Quelltextbeispiele in diesem Buch werden beispielsweise jedes Mal, wenn der Text formatiert wird, von einem Präprozessor eingefügt. Der Trick daran ist, daraus einen aktiven Vorgang zu machen: Einmalige Generierung funktioniert nicht, denn dann sind wir wieder dazu gezwungen, Informationen zu duplizieren.

Dokumentation im Quelltext. Programmierern lehrt man, ihren Quelltext zu dokumentieren: Guter Quelltext habe viele Kommentare. Leider erzählt ihnen niemand, *wieso* Quelltext Kommentare braucht. Schlechter Quelltext *verlangt* besonders viele Kommentare.

Das DRY-Prinzip besagt, konkretes Wissen in den Quelltext zu schreiben, wo es hin gehört, und Kommentare für abstraktere Erklärungen zu nutzen. Ansonsten wiederholen wir Wissen und bei jeder Veränderung müssen sowohl Quelltext als auch Kommentare angepasst werden. Die Kommentare werden unweigerlich veralten und falsche Kommentare sind schlimmer als gar keine (siehe auch *Es geht ums Schreiben*, Seite 233).

Dokumentation und Quelltext. Man schreibt erst Dokumentation und dann Quelltext. Etwas ändert sich und man ergänzt die Dokumentation und dann den Quelltext. Dokumentation und Quelltext repräsentieren beide dasselbe Wissen. Wir alle wissen, dass wir in der Hitze des Gefechts bei drohenden Terminen und ungeduldigen Kunden dazu neigen, das Dokumentieren aufzuschieben.

David hat einmal an einem internationalen Telex-Vermittlungssystem gearbeitet. Verständlicherweise verlangte der Kunde eine ausgiebige Testspezifikation und bestand darauf, dass die Software bei jeder Auslieferung alle Tests erfolgreich absolviert. Um sicher zu gehen, dass die Tests die Spezifikation genau wiedergeben, hat das Team sie aus dem Spezifikationsdokument generiert. Wenn der Kunde die Spezifikation ergänzt hat, änderten sich automatisch die Tests. Als das Team den Kunden erst einmal von der Zuverlässigkeit der Methode überzeugt hatte, dauerte die Generierung der Akzeptanztests üblicherweise nur wenige Sekunden.

Sprachprobleme. Viele Programmiersprachen erzwingen erhebliche Wiederholungen im Quelltext, meistens wenn die Schnittstelle eines Moduls von seiner Implementierung getrennt wird. In den Header-Dateien von C und C++ werden Namen und Typinformation von exportierten Variablen, Funktionen und (im Falle von C++) Klassen wiederholt. Object Pascal dupliziert diese Information sogar in derselben Datei. Wenn Sie Remote Procedure Calls (RPC) oder CORBA [URL 29] verwenden, müssen Sie Informationen über die Schnittstelle in der Schnittstellendefinition und im dazugehörigen Quelltext formulieren.

Es gibt keine einfache Lösung, um die Anforderungen einer Programmiersprache zu überwinden. Einige Entwicklungsumgebungen verbergen die Existenz von Header-Dateien, indem sie diese automatisch erzeugen. Object Pascal erlaubt beispielsweise Abkürzungen für wiederholte Funktionsdeklarationen. Trotzdem muss man üblicherweise mit dem klar kommen, was man hat. Auf die meisten Wiederholungsfehler von Programmiersprachen wird man zumindest frühzeitig hingewiesen. Eine Header-Datei zum Beispiel, die nicht zur Implementierung passt, wird im Allgemeinen zu einem Fehler beim Kompilieren oder Linken des Programms führen.

Denken Sie auch an die Kommentare in Header-Dateien und deren Implementierung. Es ist absolut sinnlos, Kommentare in beiden Dateien zu wiederholen. Dokumentieren Sie Belange der Schnittstellen in den Header-Dateien und dokumentieren Sie die Details, die man für den Aufruf nicht kennen muss, in der Implementierung.

Unabsichtliche Wiederholung

Manchmal entsteht Wiederholung aus Fehlern im Entwurf.

Lassen Sie uns ein Beispiel aus der Logistikbranche betrachten. Nehmen wir an, dass ein Lastwagen neben anderen Attributen einen Typ, ein Kennzeichen und einen Fahrer hat. Passend dazu besteht eine Liefertour aus einer Fahrtroute, einem Lastwagen und einem Fahrer. Mit diesem Verständnis schreiben wir dann ein paar Klassen.

Aber was passiert, wenn Sabine sich krankmeldet und wir den Fahrer wechseln müssen? Sowohl `Lastwagen` als auch `Liefertour` enthalten einen Fahrer. Welchen ändern wir? Diese Wiederholung ist offensichtlich schlecht. Normalisieren Sie das entsprechend der zugrunde liegenden Geschäftsregeln. Braucht ein Lastwagen wirklich zwingend einen Fahrer oder braucht die Liefertour einen? Vielleicht brauchen wir auch eine dritte Klasse, die beides mit einem Fahrer verknüpft. Unabhängig von der Lösung dieses Problems sollten Sie solche nicht-normalisierten Datenstrukturen vermeiden.

Es gibt eine weniger offensichtliche Form nicht-normalisierter Daten, die bei voneinander abhängigen Datenelementen auftritt. Betrachten Sie dieses Beispiel einer Klasse an, die eine Linie repräsentiert:

```
class Line {
 public:
  Point  start;
  Point  end;
  double length;
};
```

Auf den ersten Blick sieht diese Klasse vernünftig aus. Eine Linie hat offensichtlich einen Anfang und ein Ende, und damit auch immer eine Länge (selbst wenn diese Null ist). Aber es gibt eine Wiederholung, denn die Länge ist durch Anfang und Ende definiert. Wenn Sie einen der beiden Punkte verändern, ändert sich auch die Länge. Es ist besser, die Länge zu berechnen.

```
class Line {
 public:
  Point  start;
  Point  end;
  double length() { return start.distanceTo(end); }
};
```

Zu einem späteren Zeitpunkt entscheiden Sie sich eventuell aus Gründen der Performance bewusst für die Verletzung des DRY-Prinzips. Das geschieht häufig, wenn Daten zwischengespeichert werden, um aufwändige Operationen zu vermeiden. Der Trick dabei ist, die Auswirkungen einzugrenzen. Die Verletzung wird nach außen nicht sichtbar, nur die Methoden in der Klasse müssen sich darum kümmern, die Dinge in Ordnung zu halten.

```
class Line {
 private:
  bool   changed;
  double length;
  Point  start;
  Point  end;

 public:
  void setStart(Point p) { start = p; changed = true; }
  void setEnd(Point p)   { end   = p; changed = true; }

  Point getStart(void)   { return start; }
  Point getEnd(void)     { return end;   }

  double getLength() {
    if (changed) {
      length  = start.distanceTo(end);
      changed = false;
    }
    return length;
  }
};
```

Nebenbei veranschaulicht dieses Beispiel ein wichtiges Thema in objektorientierten Programmiersprachen wie Java oder C++: Verwenden Sie immer Zugriffsmethoden zum Lesen und Schreiben von Attributen.[1] Das erleichtert es, nachträglich Funktionalität wie zum Beispiel das Zwischenspeichern (Caching) von Werten einzubauen.

Wiederholung aus Ungeduld

Jedes Projekt steht unter Zeitdruck – eine Macht, die die Besten unter uns zu Abkürzungen verleitet. Wenn Sie ein Unterprogramm brauchen, das einem existierenden ähnelt, geraten Sie in die Versuchung, das Original zu kopieren und ein paar Änderungen zu machen. Sie brauchen eine Konstante für die maximale Punktzahl? „Wenn ich die Header-Dateien ändere, muss das ganze Projekt neu kompiliert werden. Vielleicht sollte ich an dieser Stelle besser ein Literal verwenden, und dort, und da auch noch“, denken Sie sich. Brauchen Sie eine Klasse wie jene in der Java-Klassenbibliothek? Der Quelltext ist verfügbar, wieso also nicht kopieren und modifizieren (ungeachtet der Lizenzproblematik).

Denken Sie an den abgedroschenen Spruch „Abkürzungen dauern länger“, wenn Sie diese Versuchung spüren. Sie sparen sich jetzt vielleicht ein paar Sekunden, allerdings zum Preis eines potenziellen Verlusts von Stunden in der Zukunft. Denken Sie an die Probleme um das Jahr-2000-Fiasko. Viele davon sind durch die Bequemlichkeit von Software-Entwicklern entstanden, die die Größe von Datumsfeldern nicht definiert und auf einheitliche Bibliotheken für Datumsfunktionen verzichtet haben.

Wiederholung aus Ungeduld ist einfach zu entdecken und zu behandeln, aber es erfordert Disziplin und den Willen, vorsorglich Zeit zu investieren, um zukünftigen Ärger zu vermeiden.

Wiederholung durch mehrere Entwickler

Diese Form der Wiederholung ist vielleicht am schwersten zu entdecken und zu vermeiden. Sie tritt zwischen den Software-Entwicklern in einem Projekt auf. Ganze Sammlungen von Funktionalität können dadurch unbeabsichtigt dupliziert werden, Jahre lang unentdeckt bleiben und so zu Problemen bei der Wartung führen. Wir haben aus erster Hand von einem U.S. Bundesstaat erfahren, der auf die Jahr-2000-Tauglichkeit überprüft wurde. Die Untersuchung förderte 10000 Programme zu Tage, die alle eigene Plausibilitätsprüfungen für die Sozialversicherungsnummer implementiert hatten.

[1] Die Verwendung von Zugriffsmethoden folgt Meyers *Prinzip des einheitlichen Zugriffs* [Mey97d], das besagt: „Alle Dienste, die ein Modul anbietet, sollen in einheitlicher Notation zur Verfügung gestellt werden, die nicht verrät, ob sie mittels Speicherung oder Berechnung implementiert sind.“

Auf abstrakter Ebene können Sie dem Problem mit einem sauberen Entwurf, einem starken technischen Projektleiter (siehe *Pragmatische Teams*, Seite 212) und einer klaren Aufteilung der Verantwortlichkeiten im Entwurf begegnen. Auf der Ebene von Modulen ist das Problem jedoch heimtückischer. Allgemein verwendete Funktionalität, die nicht eindeutig in einen Verantwortungsbereich fällt, kann unzählige Male implementiert werden.

Wir sind der Meinung, dass die Förderung aktiver und häufiger Kommunikation zwischen Software-Entwicklern der beste Weg ist, diesem Problem zu begegnen. Richten Sie Foren für die Diskussion allgemeiner Probleme ein. (In vergangenen Projekten haben wir private Usenet-Foren eingerichtet, damit Software-Entwickler Ideen austauschen und Fragen stellen konnten. Das bietet eine unaufdringliche Möglichkeit zur Kommunikation – sogar über Standorte hinweg – und hält dennoch alles Gesagte fest.) Ernennen Sie ein Teammitglied zum Projekt-Bibliothekar, der den Wissensaustausch fördern soll. Halten Sie einen zentralen Platz in der Verzeichnisstruktur des Projekts für Hilfsfunktionen und Skripte frei. Betonen Sie, wie wichtig es ist, den Quelltext und die Dokumentation anderer zu lesen, entweder informell oder in Reviews. Sie schnüffeln dabei ja nicht herum, sondern lernen von ihnen. Bedenken Sie, dass der Einblick gegenseitig ist. Regen Sie sich nicht darüber auf, wenn andere Leute *Ihren* Quelltext studieren.

Tipp 12: **Machen Sie Wiederverwendung einfach** ◄

Bemühen Sie sich, eine Umgebung aufzubauen, in der es einfacher ist, Existierendes zu finden und wiederzuverwenden, als es selbst zu schreiben. *Wenn es nicht einfach ist, werden es die Leute nicht tun.* Wenn Ihnen Wiederverwendung nicht gelingt, riskieren Sie Wiederholungen.

Verwandte Abschnitte

- *Orthogonalität*, Seite 31
- *Textbearbeitung*, Seite 91
- *Quelltextgeneratoren*, Seite 94
- *Refaktorisieren*, Seite 173
- *Pragmatische Teams*, Seite 212
- *Alles automatisch*, Seite 218
- *Es geht ums Schreiben*, Seite 233