Aufgaben

- Betrachten Sie Ihre Gewohnheiten während eines Arbeitstages. Sehen Sie wiederkehrende Aufgaben? Tippen Sie immer wieder die gleiche Folge von Befehlen ein? Versuchen Sie, ein paar Shell-Skripte zu schreiben, um den Prozess zu automatisieren. Klicken Sie immer wieder die gleiche Abfolge von Icons an? Können Sie ein Makro erstellen, das all dies für Sie tut?
- Wie viel Papierkram in Ihrem Projekt kann automatisiert werden? Angesichts der hohen Kosten für Programmierer[6] sollten Sie ausrechnen, wie viel vom Projektbudget die Verwaltungsvorgänge verschlingen. Können Sie den Zeitaufwand für die Erstellung einer automatischen Lösung mit den Kosteneinsparungen rechtfertigen?

43 Schonungsloses Testen

Die meisten Software-Entwickler hassen das Testen. Sie neigen dazu, vorsichtig zu testen und unbewusst die Schwachstellen zu meiden. Pragmatische Programmierer sind anders. Wir werden davon *angetrieben*, unsere Fehler *jetzt* zu finden. So müssen wir die Schande, dass unsere Fehler später von anderen gefunden werden, nicht ertragen.

Fehler zu suchen, ist wie Fischen mit Netzen. Wir benutzen kleine, engmaschige Netze (Unittests), um die kleinen Fische zu fangen, und große, grobe Netze (Integrationstests) für die Killer-Haie. Manchmal schaffen es die Fische zu entkommen. Dann flicken wir die sichtbaren Löcher in der Hoffnung, immer mehr von den glitschigen Defekten zu erwischen, die in unserem Projekt-Pool herumschwimmen.

Tipp 62: ***Testen Sie frühzeitig, häufig und automatisch*** ◄

Wir beginnen mit dem Testen, sobald wir Quelltext haben. Diese mickrigen, kleinen Fische haben die üble Angewohnheit, ziemlich schnell zu gigantischen, Menschen fressenden Haien zu werden, und Haie fangen ist ein ganzes Stück schwieriger. Wir müssen die Tests aber nicht von Hand ausführen.

[6] Für Schätzungen können Sie von einem Durchschnitt von 100000 US-Dollar pro Kopf und Jahr ausgehen – das beinhaltet bereits Zusatzleistungen, Weiterbildung, Büroraum, Verwaltung und so weiter.

Viele Teams entwickeln ausgeklügelte Testpläne für ihre Projekte – manchmal halten sie sich sogar daran. Wir haben aber festgestellt, dass Teams mit automatisierten Tests viel bessere Erfolgsaussichten haben. Tests, die bei jedem Build-Lauf ausgeführt werden, sind viel effektiver als Testpläne, die im Regal verstauben.

Je früher Fehler gefunden werden, desto billiger ist ihre Beseitigung. „Programmiere ein bisschen, teste ein bisschen“ ist ein beliebter Spruch in der SmallTalk-Welt.[7] Wir können uns dieses Mantra aneignen, indem wir die Tests zur selben Zeit schreiben wie den eigentlichen Quelltext (oder sogar vorher).

Genau genommen wird in einem guten Projekt sogar *mehr* Quelltext für Tests geschrieben als für die eigentliche Anwendung. Der Aufwand für das Schreiben der Tests lohnt sich, denn auf lange Sicht kommt es Sie günstiger, und Sie haben damit eine reelle Chance, ein nahezu fehlerfreies Produkt abzuliefern,

Zusätzlich geben Ihnen die erfolgreichen Tests die Zuversicht, dass ein Stück Quelltext wirklich *fertig* ist.

> Tipp 63: ***Das Programmieren ist nicht getan, bis alle Tests erfolgreich waren*** ◄

Dass Sie ein Stück Quelltext fertig getippt haben, bedeutet noch lange nicht, dass Sie Ihrem Chef oder Kunden sagen können, es sei *fertig*. Das ist es nicht. Zunächst einmal ist Quelltext nie wirklich fertig. Noch wichtiger ist aber: Solange er nicht alle verfügbaren Tests besteht, können Sie nicht behaupten, er sei benutzbar.

Wir müssen drei Hauptaspekte des projektweiten Testens betrachten: Was muss gestestet werden? Wie muss es getestet werden? Wann muss es getestet werden?

Was muss getestet werden?

Sie müssen einige wichtige Arten von Softwaretests durchführen:

- Unittests
- Integrationstests
- Validierung und Verifikation
- Ressourcenverbrauch, Fehlersituationen und Wiederherstellung
- Performance-Tests
- Usability-Tests (Tests auf Bedienerfreundlichkeit)

[7] Bei eXtreme Programming [URL 45] wird dieses Konzept „fortlaufende Integration und automatisiertes Testen“ genannt.

Diese Liste ist in keiner Weise vollständig und spezielle Projekte werden auch verschiedene andere Tests durchführen müssen, aber sie ist ein guter Ausgangspunkt.

Unittests

Ein *Unittest* ist Quelltext, der ein Modul auf die Probe stellt. Wir haben dieses Thema bereits in *Einfach testbar*, Seite 179, behandelt. Es ist die Basis aller anderen Testarten, die wir in diesem Abschnitt behandeln. Wenn die Teile schon einzeln nicht funktionieren, werden sie sicher auch zusammen nicht besser arbeiten. Alle verwendeten Module müssen ihre eigenen Unittests bestehen, bevor Sie weitermachen können.

Sobald alle relevanten Module ihre eigenen Unittests bestanden haben, sind Sie bereit für den nächsten Schritt. Sie müssen testen, wie die Module im System interagieren.

Integrationstests

Integrationstests prüfen, ob die wesentlichen Teilsysteme des Projekts funktionieren und gut miteinander zusammenspielen. Mit vernünftigen Verträgen und sorgfältigen Tests können Integrationsprobleme leicht entdeckt werden. Anderenfalls wird die Integration ein fruchtbarer Nährboden für Fehler. Genau genommen sind Integrationsprobleme die allergrößte Fehlerquelle in einem System.

Integrationstests sind in der Tat nur eine Erweiterung der Unittests. Man testet jetzt eben, ob ganze Subsysteme ihren Vertrag erfüllen.

Validierung und Verifikation

Sobald Sie eine ausführbare Benutzeroberfläche oder einen Prototypen haben, müssen Sie sich vor allem eine Frage stellen: Die Anwender haben Ihnen gesagt, was sie wollen, aber ist es auch das, was sie brauchen?

Erfüllt es die funktionalen Anforderungen des Systems? Auch das muss getestet werden. Ein fehlerfreies System, das die falschen Fragen beantwortet, ist nicht besonders sinnvoll. Finden Sie heraus, wie Anwender das System benutzen, und prüfen Sie, wie die Daten der Anwender von den Testdaten der Programmierer abweichen (siehe Linienziehen im Abschnitt *Fehlersuche* auf Seite 82).

Ressourcenverbrauch, Fehlersituationen und Wiederherstellung

Nachdem Sie jetzt einen ziemlich guten Eindruck davon haben, dass sich das System unter idealen Bedingungen korrekt verhält, müssen Sie untersuchen, wie es unter *realen* Bedingungen aussieht. In der realen Welt haben Programme keine unbeschränkten Ressourcen. Unter den Dingen, deren Grenzen Ihr Quelltext spüren wird, sind:

- Speicher
- Festplattenplatz

- CPU-Durchsatz
- Zeit
- Festplattendurchsatz
- Netzwerkdurchsatz
- Farbpalette
- Bildschirmauflösung

Sie testen vermutlich auf Fehler bei der Zuteilung von Festplatten- und Speicherplatz, aber wie oft testen Sie auch auf die anderen? Passt Ihre Anwendung auf einen Bildschirm mit 640 × 480 Pixeln und 256 Farben? Sieht sie bei 1600 × 1200 Pixeln und 24 Bit Farbtiefe wie eine Briefmarke aus? Läuft die Stapelverarbeitung durch, bevor das Backup startet?

Einschränkungen der Umgebung, wie die Bildschirmeinstellung, kann man erkennen und sich darauf einstellen. Aber man kann nicht allen Fehlersituationen ausweichen. Wenn das Programm feststellt, dass kein Speicher mehr frei ist, sind die Optionen begrenzt. Sie haben eventuell nicht mehr genug Ressourcen, um noch irgendetwas anderes zu tun, als abzubrechen.

Wenn das System versagt,[8] tut es das dann auch würdevoll? Wird es versuchen, so gut wie eben möglich seinen Zustand zu sichern und den Verlust von Arbeit zu vermeiden? Oder wird es dem Benutzer eine „Allgemeine Schutzverletzung“ oder einen „Core Dump“ entgegenschmettern?

Performancetests

Performance-, Stress- und Lasttests sind sicherlich ein ebenfalls wichtiger Aspekt für das Projekt.

Fragen Sie sich selbst, ob die Software die Performance-Anforderungen unter realen Bedingungen erfüllt: mit der erwarteten Anzahl von Benutzern, Verbindungen oder Transaktionen pro Sekunde. Ist sie skalierbar?

Bei einigen Anwendungen benötigen Sie vielleicht sogar spezielle Test-Hardware und -Software, um die Last realistisch simulieren zu können.

Usability-Tests

Usability-Tests unterscheiden sich von den bisher angesprochenen Testarten. Sie werden mit echten Anwendern unter realen Umgebungsbedingungen durchgeführt.

Betrachten Sie Usability als die menschlichen Faktoren. Gab es Missverständnisse bei der Anforderungsanalyse, die ausgeräumt werden müssen? Passt die Software so gut

[8] Unser Lektor hat darum gebeten, den Satz in „Falls das System versagt, ...“ zu ändern. Wir haben uns gewehrt.

zu den Benutzern, dass sie zu deren verlängerter Hand wird? (Wir wollen nicht nur, dass sich unsere Werkzeuge unseren Händen anpassen, genauso sollen sich die Werkzeuge, die wir selbst herstellen, anderen Händen anpassen.)

Wie bei Validierung und Verifikation müssen Sie Usability-Tests so früh wie möglich durchführen. Nur so bleibt Zeit für Korrekturen. Bei größeren Projekten kann es sinnvoll sein, Experten für diese menschlichen Faktoren hinzuzuziehen. (Auf jeden Fall macht es Spaß, mit den halb durchlässigen Spiegeln zu spielen.)

Bei den Usability-Kriterien zu versagen, ist genauso ein Fehler wie eine Division durch Null.

Wie soll getestet werden?

Wir haben betrachtet, *was* getestet werden soll. Jetzt lenken wir unsere Aufmerksamkeit auf das *Wie*:

- Regressionstests
- Testdaten
- Grafische Benutzeroberflächen testen
- Tests testen
- Sorgfältig testen

Regressionstests

Ein Regressionstest vergleicht die Ausgabe des aktuellen Tests mit älteren (oder bekannten) Ausgaben. Damit können wir feststellen, ob die Fehlerbeseitigung von heute vielleicht Teile kaputt gemacht hat, die gestern noch funktioniert haben. Das ist ein wichtiges Sicherheitsnetz, und es reduziert unerfreuliche Überraschungen.

Alle Tests, die wir bisher erwähnt haben, können als Regressionstests durchgeführt werden. So stellen wir sicher, dass man bei der Entwicklung von neuem Quelltext nicht den Boden unter den Füßen verliert. Mit Regressionstests kann man Performance, Verträge, Korrektheit und vieles mehr überprüfen.

Testdaten

Woher bekommen wir die Daten für all diese Tests? Es gibt nur zwei Arten von Daten: reale Daten und synthetische Daten. Wir brauchen beide, weil ihre unterschiedliche Natur auch unterschiedliche Fehler in unserer Software aufdecken wird.

Reale Daten kommen aus einer echten Quelle. Sie sind eventuell in einem realen System gesammelt worden, dem eines Wettbewerbers, oder mit irgend einem Prototypen. Sie stellen typische Anwenderdaten dar. Die Überraschung kommt mit der Erkenntnis, was *typisch* bedeutet. Sie werden Fehler und Missverständnisse bei der Anforderungsanalyse am ehesten aufdecken.

Testen von Entwurf und Methodik

Kann man den Entwurf des Quelltextes selbst oder die Methode, mit der die Software erstellt wird, testen? Ja, man kann es, einigermaßen. Man tut das durch Analyse von *Metriken*, Maße für verschiedene Aspekte des Quelltexts. Die einfachste (und meist auch langweiligste) Metrik ist *lines of code* (dt. Anzahl der Quelltextzeilen) – wie groß ist der Quelltext?

Es gibt eine Vielzahl weiterer Metriken, mit denen man Quelltext analysieren kann. Dazu gehören:

- McCabe Zyklomatische Komplexität Metrik (misst die Komplexität von Entscheidungsstrukturen)
- Eingangsfächer (Anzahl der Basisklassen) und Ausgangsfächer (Anzahl von Subklassen) der Vererbung (engl: inheritance fan-in and fan-out)
- Result Set (Ergebnismenge; siehe *Entkopplung und das Demeter-Gesetz*, Seite 130)
- Klassenkopplungsverhältnis (engl: class coupling ratios; siehe [URL 48]).

Einige Metriken sind so entworfen, dass sie gut/schlecht Aussagen treffen. Andere liefern nur relative Vergleichswerte. Das heißt, man berechnet die Metriken für jedes Modul im System und vergleicht, wie einzelne Module gegen die anderen abschneiden. Dabei werden die üblichen statistischen Methoden (wie Mittelwert und Standardabweichung) verwendet.

Synthetische Daten werden künstlich erzeugt, eventuell sogar unter bestimmten statistischen Einschränkungen. Verschiedene Gründe machen die Verwendung von synthetischen Daten notwendig:

- Man braucht sehr viele Daten, eventuell sogar mehr, als eine echte Probe liefern könnte. Man kann echte Daten als Basis für die Generierung einer größeren Datenmenge verwenden.
- Sie brauchen Daten, um die Grenzfälle zu prüfen. Diese Daten können völlig synthetisch sein: Datumsfelder mit dem 29. Februar 1999, riesige Datensätze und Adressen mit ausländischen Postleitzahlen.
- Sie brauchen Daten, die bestimmte statistische Eigenschaften aufweisen. Wollen Sie wissen, was passiert, wenn jede dritte Transaktion fehlschlägt? Erinnern Sie sich an den Sortieralgorithmus, der im Schneckentempo läuft, wenn er mit vorsortierten Daten gefüttert wird. Sie können Daten in zufälliger oder sortierter Reihenfolge eingeben, um diese Schwächen aufzudecken.

Grafische Benutzeroberflächen testen

Für das Testen grafischer Benutzeroberflächen braucht man oft spezialisierte Werkzeuge. Häufig basieren sie auf einem einfachen Aufnahme/Wiedergabe-Modell von

Ereignissen oder steuern die Benutzeroberfläche über spezielle Skripte. Manche Werkzeuge verwenden auch eine Kombination von beidem.

Weniger ausgeklügelte Werkzeuge erzwingen eine enge Kopplung zwischen der getesteten Softwareversion und dem Testskript: Wenn Sie einen Dialog verschieben oder eine Schaltfläche verkleinern, findet sie der Test eventuell nicht mehr und schlägt fehl. Die meisten modernen Testwerkzeuge für grafische Benutzeroberflächen verwenden verschiedene Methoden, um dieses Problem zu umgehen und sich selbstständig an kleinere Layoutänderungen anzupassen.

Man kann aber nicht alles automatisieren. Andrew hat an einem Grafiksystem gearbeitet, mit dem man verschiedene nicht-deterministische, visuelle Effekte für die Simulation natürlicher Phänomene erzeugen und darstellen konnte. Leider konnte man bei den Testläufen die Ausgaben nicht einfach mit Bildern aus vorhergehenden Läufen vergleichen, weil sie *jedes* Mal unterschiedlich sein sollten. In Situationen wie diesen haben Sie vermutlich keine andere Wahl, als sich auf manuelle Interpretation der Testdaten zu verlassen.

Einer der vielen Vorteile von modularem Quelltext (siehe *Entkopplung und das Demeter-Gesetz*, Seite 130) ist modulares Testen. Bei Anwendungen mit grafischer Oberfläche beispielsweise sollte der Entwurf so entkoppelt sein, dass die Anwendungslogik auch *ohne* grafische Benutzeroberfläche getestet werden kann. Diese Idee ähnelt dem Testen von Teilkomponenten. Wenn die Anwendungslogik überprüft ist, ist es einfacher, Fehler zu lokalisieren, die sich erst in der Benutzeroberfläche zeigen (wahrscheinlich liegt der Fehler dann im Quelltext der Benutzerschnittstelle).

Tests testen

Wenn niemand perfekte Software schreiben kann, sind auch perfekte Tests unmöglich. Wir müssen die Tests testen.

Betrachten Sie unsere Testreihen als aufwändiges Sicherheitssystem, das einen Alarm auslösen soll, wenn ein Fehler auftritt. Wie kann man ein Sicherheitssystem besser testen als mit einem Einbruchsversuch?

Haben Sie einen Test geschrieben, der einen bestimmten Fehler erkennen soll, müssen Sie den Fehler bewusst *verursachen* und sicher gehen, dass der Test Alarm schlägt. Das gewährleistet, dass der Test den Fehler auch erkennt, wenn er wirklich auftritt.

Tipp 64: Testen Sie Ihre Tests durch Sabotage ◄

Wenn Sie es mit Testen wirklich ernst meinen, sollten Sie vielleicht einen *Projekt-Saboteur* ernennen. Seine Aufgabe ist es, in einer gesonderten Kopie des Quelltextes absichtlich Fehler einzubauen, die von den Tests entdeckt werden müssen.

Gehen Sie beim Schreiben von Tests sicher, dass der Alarm los geht, wenn er soll.

Sorgfältig testen

Wenn Sie überzeugt sind, dass Ihre Tests korrekt sind und auftretende Fehler von Ihnen gefunden werden, woher wissen Sie dann, ob Sie Ihren Quelltext sorgfältig genug getestet haben?

Die Antwort lautet: „Sie wissen es nicht“ und werden es auch nie. Es gibt aber Produkte auf dem Markt, die helfen können. Die Werkzeuge zur *Abdeckungsanalyse* beobachten Ihren Quelltext während des Testens und verfolgen, welche Quelltextzeilen ausgeführt wurden und welche nicht. Diese Werkzeuge geben Ihnen ein grobes Gefühl dafür, wie umfassend Ihre Tests sind, aber erwarten Sie keine 100%ige Abdeckung.

Selbst wenn Sie jede Quelltextzeile erwischen, zeigt das nicht das ganze Bild. Wichtig ist die Anzahl der Zustände, die Ihr Programm annehmen kann. Zustände sind nicht gleichbedeutend mit Quelltextzeilen. Nehmen Sie beispielsweise an, Sie hätten eine Funktion, die zwei Integer erwartet, von denen jede eine Zahl zwischen 0 und 999 sein kann.

```
int test(int a, int b) {
  return a / (a + b);
}
```

In der Theorie hat dieser Dreizeiler 1000000 logische Zustände, von denen 999999 korrekt funktionieren und einer nicht (wenn `a + b = 0` ist). Allein die Tatsache, dass diese Quelltextzeile ausgeführt worden ist, zeigt das nicht. Sie müssen alle möglichen Zustände des Programms identifizieren. Dummerweise ist das im Allgemeinen ein *echt hartes* Problem. Hart im Sinne von: „Die Sonne wird ein kalter, harter Brocken sein, bevor Sie es gelöst haben.“

> *Tipp 65:* ***Testen Sie Zustandsabdeckung, nicht Quelltextabdeckung*** ◄

Selbst bei guter Quelltextabdeckung haben die verwendeten Daten noch einen riesigen Einfluss. Den größten Einfluss überhaupt hat vielleicht die *Reihenfolge*, in der Sie den Quelltext durchlaufen.

Wann muss getestet werden?

In vielen Projekten neigt man dazu, das Testen auf die letzte Minute zu verschieben. Genau dorthin, wo es von der scharfen Kante einer Deadline[9] abgeschnitten wird. Wir müssen viel früher damit anfangen. Sobald irgendein Stück Anwendungs-Quelltext existiert, muss es getestet werden.

[9] dead line \ded-lain\ (1864): eine Linie im oder um ein Gefängnis herum. Beim Überschreiten der Linie riskiert der Gefangene, erschossen zu werden. *(Webster's Collegiate Dictionary)*

Die meisten Tests sollten automatisch durchgeführt werden. Mit „automatisch“ meinen wir, dass auch die *Ergebnisse* der Tests automatisch interpretiert werden. (In *Alles automatisch*, Seite 218 , finden Sie mehr zu diesem Thema.)

Wir testen gerne so häufig wie möglich und immer, bevor wir Quelltext ins Versionskontrollsystem einchecken. Einige dieser Systeme, wie Aegis, können das automatisch. Ansonsten tippen Sie einfach

```
% make test
```

Normalerweise ist es kein Problem, Regressionstests mit allen einzelnen Unittests und Integrationstest so oft wie eben nötig ablaufen zu lassen.

Einige Tests können aber nicht einfach so häufig ausgeführt werden. Lasttests beispielsweise brauchen eventuell spezielle Einstellungen und Gerätschaften sowie etwas Händchenhalten. Diese Tests werden seltener durchgeführt, vielleicht wöchentlich oder monatlich. Es ist aber wichtig, dass Sie regelmäßig und in einem festen Zeitplan durchgeführt werden. Wenn etwas nicht automatisch durchgeführt werden kann, müssen Sie sicherstellen, dass es mit allen notwendigen Ressourcen fest eingeplant wird.

Das Netz enger ziehen

Schließlich wollen wir noch das allerwichtigste Konzept beim Testen enthüllen. Es ist naheliegend, und praktisch jedes Lehrbuch weist darauf hin, aber aus irgendeinem Grund wenden es die meisten Projekte nicht an.

Wenn ein Fehler durch das bestehende Netz von Tests schlüpft, müssen Sie einen neuen Test hinzufügen, um ihn beim nächsten Mal zu entdecken.

Tipp 66: *Finden Sie Fehler nur einmal* ◄

Wenn ein menschlicher Tester einen Fehler findet, sollte es das *letzte* Mal gewesen sein, dass ein Mensch das machen muss. Die automatisierten Tests müssen so geändert werden, dass von da an auf diesen speziellen Fehler hin geprüft wird. Die Prüfung erfolgt jedes Mal, ohne Ausnahme, egal wie trivial der Fehler auch sein mag und wie bestimmt der Programmierer auch behauptet: „Oh, das wird nie wieder passieren.“

Es wird wieder passieren, und wir haben nicht die Zeit zur Jagd auf Fehler, die automatisierte Tests für uns finden könnten. Wir müssen unsere Zeit nutzen, neuen Quelltext zu schreiben – und neue Fehler.

Verwandte Abschnitte

- *Der Hund hat meinen Quelltext gefressen*, Seite 2
- *Fehlersuche*, Seite 82
- *Entkopplung und das Demeter-Gesetz*, Seite 130
- *Refaktorisieren*, Seite 173
- *Einfach testbar*, Seite 179
- *Alles automatisch*, Seite 218

Aufgaben

- Können Sie Ihr Projekt automatisch testen? Viele Teams müssen mit „Nein“ antworten. Wieso? Ist es zu schwierig, die richtigen Ergebnisse festzulegen? Wäre es dann nicht auch schwierig, den Auftraggebern klar zu machen, dass das Projekt „fertig“ ist?

  Ist es zu schwierig, die Anwendungslogik unabhängig von der grafischen Benutzeroberfläche zu testen? Was sagt das über die grafische Benutzeroberfläche aus? Was über Kopplung?

44 Es geht ums Schreiben

Die blasseste Tinte ist besser als das beste Gedächtnis.
Chinesisches Sprichwort

Normalerweise verschwenden Software-Entwickler nicht viel Zeit auf Dokumentation. Im besten Fall ist es eine unangenehme Notwendigkeit und im schlechtesten wird es als niedrig priorisierte Aufgabe behandelt, die das Management am Projektende vielleicht vergessen wird.

Pragmatische Programmierer nehmen Dokumentation als integralen Bestandteil der Software-Entwicklung an. Das Schreiben von Dokumentation wird einfacher, wenn man die Arbeit nicht mehrfach macht oder dabei Zeit verschwendet, und indem man die Dokumentation in Reichweite aufbewahrt, wenn möglich im Quelltext selbst.

Das sind nicht besonders originelle und neue Gedanken. Die Idee, Quelltext und Dokumentation zu verheiraten, taucht unter anderem in Donald Knuths Arbeit über „Literate Programming“ und dem Javadoc-Werkzeug von Sun auf. Wir wollen die Zweiteilung von Quelltext und Dokumentation herunterspielen. Stattdessen sehen wir beides als