Teil V

Verwaltung und Erhaltung von Software

20 Konfigurationsverwaltung

Software-Entwicklung ist Teamarbeit. Damit alle daran beteiligten Personen geordnet zusammenarbeiten können und die dabei erstellten Dokumente nicht heillos durcheinander abgelegt und geändert werden, müssen organisatorische und technische Vorkehrungen getroffen werden. Diese Regelungen und Maßnahmen fasst man mit dem Begriff der Konfigurationsverwaltung zusammen.

Da sie nicht unmittelbar zur Entstehung des Produkts beiträgt, wird sie oft unterschätzt und vernachlässigt. Dabei stellt sie für die Software-Projekte eine ebenso wichtige Infrastruktur dar wie die Stromversorgung für einen produzierenden Betrieb: Wenn sie vorhanden ist und funktioniert, bemerkt man sie nicht, aber wenn sie unterbrochen ist, geht nichts mehr.

20.1 Grundlagen der Konfigurationsverwaltung

Während der Software-Entwicklung entstehen viele Dokumente und Komponenten. Einige davon werden nur zur Entwicklung selbst benötigt (z. B. ein Testtreiber), andere werden, bevor sie an den Kunden ausgeliefert werden, noch mehrfach korrigiert und geändert (z. B. die GUI-Komponente).

Es ist nahezu unmöglich, dabei den Überblick zu behalten, sofern man dazu nicht spezielle Vorkehrungen schafft. So kommt es immer wieder zu vermeidbaren Fehlern:

- Falsche Komponenten werden integriert.
- Bereits entwickelte Programmteile werden übersehen und erneut entwickelt.
- Testdaten werden einmal verwendet und gehen verloren, obwohl sie mit erheblichem Aufwand entworfen wurden und in der Wartung höchst nützlich wären.
- Größere Änderungen werden begonnen, ohne dass der Stand vor der Änderung archiviert ist; darum ist später weder ein Rückzug noch ein Vergleich möglich.
- Modifikationen an Dokumenten und Komponenten gehen verloren, weil sie von anderen Entwicklern überschrieben werden.

Noch schwieriger wird die Situation, wenn die Entwicklung beendet ist und das System beim Kunden läuft. Jetzt gibt es nur noch wenige Leute, die überhaupt von der Software wissen, und da sie nur noch gelegentlich daran arbeiten, haben sie im Allgemeinen keinen Überblick. Diese Situation ist sehr riskant: Die Integration modifizierter Komponenten führt zu unerwarteten Problemen, weil die Integration des laufenden Systems nicht sorgfältig dokumentiert war. Längst beseitigte Fehler tauchen wieder auf, weil die fehlerhafte Komponente nicht gelöscht wurde. Einem Kunden, der aus einem fernen Land einen Fehler gemeldet hat, kann nicht geholfen werden, weil sich seine Software-Konfiguration nicht ermitteln lässt; man weiß zwar, welches System er hat, aber man kennt nicht die genauen Versionen oder hat sie nicht archiviert.

Die Konfigurationsverwaltung hat den Zweck, diese Schwierigkeiten zu vermindern oder zu vermeiden.

20.1.1 Software-Einheiten

Bevor wir auf die Konfigurationsverwaltung und ihre Ziele eingehen, müssen wir die Begriffe Version, Variante und Konfiguration klären. Um von den unterschiedlichen Dokumentarten zu abstrahieren, führen wir den Oberbegriff *Software-Einheit* ein. Wir definieren:

Software-Einheit — Ein Stück Software, das im Software-Lebenslauf entsteht und für die Entwicklung, den Betrieb oder die Wartung des Software-Systems relevant ist. Eine Software-Einheit kann unabhängig von anderen Software-Einheiten bearbeitet, gespeichert und ersetzt werden. Sie besteht nicht aus kleineren Software-Einheiten, ist also im Sinne der Verwaltung atomar.

Die Unabhängigkeit von anderen Software-Einheiten betrifft nur die Verwaltung, nicht die Schnittstellen.

20.1.2 Versionen

Aus einer Software-Einheit E entsteht eine neue *Version*, wenn in E eine Änderung durchgeführt wird, die diese Einheit in irgendeinem Sinne besser macht, z. B. eine Korrektur, eine Erweiterung oder eine Anpassung an veränderte Bedingungen. Die neue Version (der *Nachfolger*) ersetzt die alte (den *Vorgänger*) und übernimmt deren Bezeichner (genauer: deren festen Teil). Damit die beiden nicht verwechselt werden, gibt es im Bezeichner (typischerweise am Ende) eine ein- oder mehrstufige Versionsnummer, die bei der Bildung einer neuen Version erhöht wird. Beispielsweise ersetzt das neue OS10.4.6 das ältere OS10.4.5. Die Versionen stehen also in einer zeitlichen Ordnung auf ihrem *Entwicklungspfad* hintereinander.

Der IEEE-Standard unterscheidet zwischen Version und Revision; die Revision entsteht, wenn eine Software-Einheit nicht komplett ersetzt, sondern repariert wird (z. B. durch Austausch einzelner Seiten in einem Dokument).

> **version** — (1) An initial release or re-release of a computer software configuration item, associated with a complete compilation or recompilation of the computer software configuration item.
>
> (2) An initial release or complete re-release of a document, as opposed to a revision resulting from issuing change pages to a previous release.
>
> IEEE Std 610.12 (1990)

Wir verzichten darauf, die übrigen einschlägigen Definitionen aus dem IEEE-Glossar zu zitieren, sie erscheinen uns allzu eng.

20.1.3 Varianten

Anders als die Versionen stehen *Varianten* zeitlich nicht *hintereinander*, sondern *nebeneinander*. Wenn Kunde A ein Software-System bekommen hat, das später auch an den Kunden B verkauft werden kann, dann sind meist für B spezifische Anpassungen nötig. Damit entstehen eine Variante und ein neuer Entwicklungspfad. Ebenso entstehen Varianten, wenn eine Software für verschiedene Betriebs- oder Datenbanksysteme variiert wird.

Abbildung 20–1 zeigt schematisch den Zusammenhang zwischen Versionen und Varianten. X2 und X3 sind Versionen von X1. Xa2 und Xb2 sind Varianten von X2. X2 und seine Varianten Xa2 und Xb2 bilden zusammen eine *Variantenfamilie*.

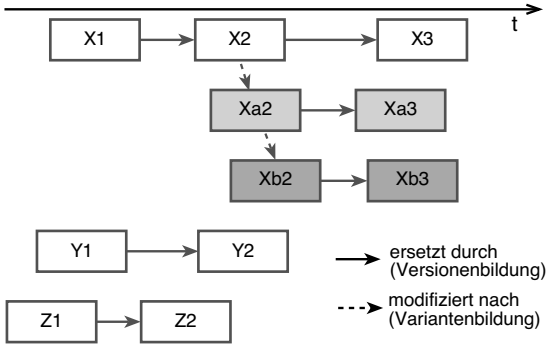

Abb. 20–1 *Die Entstehung von Versionen und Varianten*

20.1.4 Konfigurationen

Eine *Konfiguration* ist eine Menge von Software-Einheiten, die für einen definierten Zweck zusammengestellt sind und zueinander passen, sowie das Geflecht der Beziehungen zwischen diesen Software-Einheiten.

Bei einer Variantenfamilie bestimmt das dazugehörende Auswahlverfahren, wie viele ihrer Software-Einheiten in einer Konfiguration enthalten sein müssen und sein dürfen. Wenn genau eine Einheit aus der Variantenfamilie gewählt werden muss, spricht man von einer 1-aus-N-Auswahl (z. B. die MySQL-spezifische Komponente zur Speicherung der Daten). Können mehrere Software-Einheiten derselben Variantenfamilie in einer Konfiguration enthalten sein, bezeichnet man dies als N-aus-M-Auswahl. So kann beispielsweise eine Konfiguration einer Webanwendung mehrere Varianten von Browser-spezifischen Komponenten enthalten, wenn diese Anwendung die Eigenschaften unterschiedlicher Browser unterstützen muss.

Natürlich kann dieselbe Software-Einheit in verschiedenen Konfigurationen enthalten sein. Wir sprechen auch bei den Konfigurationen von Varianten; zwei Konfigurationen stellen (Konfigurations-)Varianten dar, wenn sie unterschiedliche Varianten einer Variantenfamilie enthalten. In unserem Beispiel (Abb. 20–1) könnte die Konfiguration A aus den Einheiten Y1, Z1 und Xa2 bestehen, die Konfiguration B aus den Einheiten Y1, Z2 und Xb2.

20.1.5 Probleme mit Versionen und Varianten

In Zusammenhang mit Versionen und Varianten gibt es eine Reihe von Problemen:

- Varianten erschweren die Wartung. Wenn die notwendige Änderung am System seine variantenspezifischen Teile betrifft, muss sie für jede Systemvariante einzeln durchgeführt werden. Darum bemüht man sich, die variantenspezifischen Teile eines Systems möglichst klein zu halten. Eventuell ist deshalb bei der Variantenbildung eine Aufspaltung von Software-Einheiten sinnvoll. Als Faustformel kann man sagen: Der Wartungsaufwand ist proportional der Summe von P_0 und aller P_i, wobei P_0 der Umfang der universellen Einheiten ist, die P_i sind die Umfänge der einzelnen Varianten.
  Eine Variantenbildung mit »copy and paste« hat darum katastrophale Wirkung auf den Wartungsaufwand: P_0 ist null, die P_i sind jeweils so groß, wie vorher das Gesamtsystem war.
- In vielen Fällen können oder wollen nicht alle Kunden auf eine neue Version umsteigen, beispielsweise, weil diese nicht mehr mit älteren Geräten kompatibel ist, die der Kunde weiterhin benutzt. In diesem Fall verwenden also zur selben Zeit verschiedene Kunden verschiedene Versionen auch universeller Einheiten. Der Hersteller muss dann auch die älteren Versionen vorhalten, um Probleme, die bei Kunden auftreten, nachvollziehen zu können.
  Auch bei den Kunden, die nicht umgestellt haben, können Korrekturen oder Anpassungen nötig werden. Das widerspricht aber dem Versionenkonzept, wie es oben definiert wurde. Tatsächlich bilden sich in diesem Fall Varianten, auch wenn sie als Versionen bezeichnet werden. In der Abbildung 20–2 ist die Entstehung einer neuen Version V7.0 aus V6.6 dargestellt. Kunden, die wei-

terhin V6.6 verwenden, bekommen später noch eine korrigierte Fassung V6.7. Damit ist V6.7 faktisch eine Variante von V7. In der Praxis verwendet man mehrstufige Nummern, um bei dieser Variantenbildung wenigstens nicht in Konflikt mit der Namenskonvention zu geraten. Außerdem weist die Inkrementierung der ersten Nummer auf eine erhebliche Änderung hin, die weit über die Fehlerkorrektur (typisch für eine Änderung der letzten Zahl) hinausgeht.

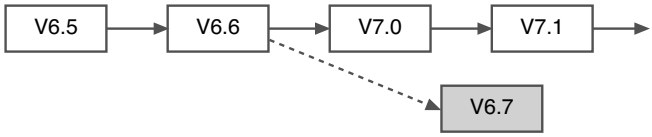

Abb. 20–2 *Weiterführung einer älteren Version als Variante*

Wenn die Gründe entfallen, die zu einer Variantenbildung geführt haben, können die Varianten wieder zusammengeführt werden (z. B. die Varianten a und b in Abb. 20–1), um damit die zukünftige Weiterentwicklung zu erleichtern. Diese Vereinigung ist schwierig, weil in den verschiedenen Varianten inkompatible Komponenten entstanden sein können. Die Kunden erhalten eine neue Version, die – aus ihrer Sicht – gegenüber dem Vorgänger nicht verbessert ist, sondern eventuell neue Mängel zeigt.

20.1.6 Baselines und Releases

Eine *Baseline* ist eine Konfiguration, die so zusammengestellt und geprüft ist, dass sie als stabil gilt und als Bezugspunkt für die weitere Entwicklung geeignet ist. Das *Release* ist eine Baseline, die mit dem Zweck definiert wurde, diese Konfiguration an die Kunden auszuliefern. Eine Baseline *kann*, ein Release *muss* ein lauffähiges System bilden.

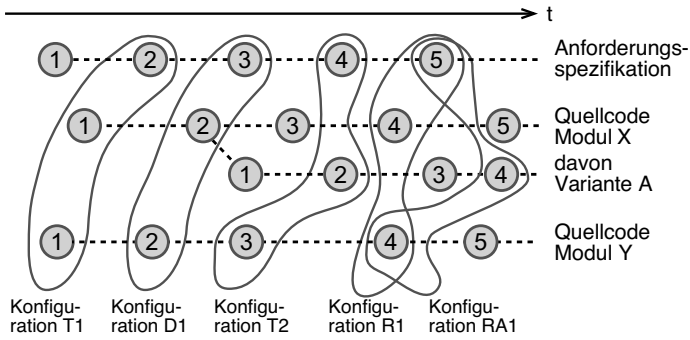

Abb. 20–3 *Versionen, Varianten und Konfigurationen*

Abbildung 20–3 zeigt schematisch die Entwicklungshistorie einiger Software-Einheiten sowie Konfigurationen über diesen Einheiten. Die Konfigurationen T1 und T2 wurden für den Test gebildet, die Konfiguration D1 ist ein Demonstrationsprototyp, die Konfiguration R1 wurde bei verschiedenen Kunden installiert, die Konfiguration RA1 ist kundenspezifisch (Auswahl einer entsprechenden Variante) und wurde nur beim Kunden A installiert. Die Konfigurationen R1 und RA1 sind Releases.

Soll ein Release erstellt werden, dann geschieht das im einfachsten Fall nach folgendem Schema (Popp, 2013):

- Es wird ein Datum für einen sogenannten »Code Freeze« festgelegt. Zum Code Freeze müssen die Arbeiten an allen Software-Einheiten abgeschlossen sein, die für das Release benötigt werden.
- Änderungen an Software-Einheiten müssen nach dem Code Freeze wohlbegründet sein. So wird man erlauben, dass noch inakzeptable Fehler korrigiert werden. Alle erlaubten Änderungen werden durch die Änderungskontrolle überwacht.
- Wenn das Release erfolgreich erstellt und dokumentiert wurde, wird der Code Freeze aufgehoben. Die Arbeiten für das nächste geplante Release können beginnen.

Abbildung 20–4 zeigt diesen Ablauf an einen Beispiel. Am Release R 1.0 arbeiten

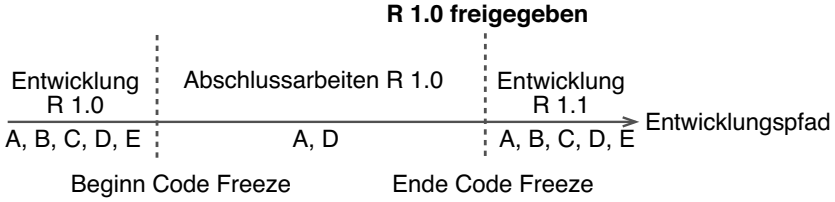

Abb. 20–4 *Erstellung eines Releases im Entwicklungspfad*

bis zum Code Freeze die Entwickler A, B, C, D und E. Um die noch notwendigen Abschlussarbeiten für Release 1.0 zu erledigen, werden nur die Entwickler A und D benötigt (sie arbeiten dann als *Release Engineer*). Die verbliebenen Teammitglieder (B, C und E) können während des Code Freeze jedoch nicht am nachfolgenden Release R 1.1 arbeiten, da der Code nicht verändert werden darf. Sie können erst damit beginnen, nachdem Release 1.0 erstellt wurde.

Will man das vermeiden, müssen die Arbeiten parallelisiert werden. Das erreicht man, wenn man den Entwicklungspfad aufteilt; das wird als »Branching« bezeichnet. Abbildung 20–5 zeigt diese Parallelisierung. Durch das Branching entsteht ein neuer Entwicklungspfad (Release-Pfad). In diesem Pfad werden die Abschlussarbeiten für Release 1.0 von den Entwicklern A und D durchge-

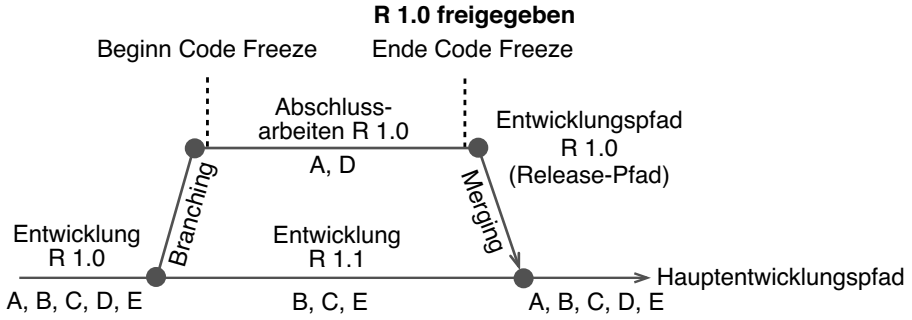

Abb. 20–5 *Erstellung eines Releases in einem eigenen Entwicklungspfad*

führt. Im Hauptentwicklungspfad arbeiten die Entwickler B, C, und E parallel dazu am nachfolgenden Release R 1.1. Die Arbeiten in beiden Pfaden, dem Hauptentwicklungspfad und dem Release-Pfad, starten nach dem Branching mit denselben Versionen der Software-Einheiten; sie werden in beiden Pfaden unabhängig voneinander geändert. Der Code Freeze gilt nur für den Release-Pfad. Alle im Release-Pfad durchgeführten Tätigkeiten, beispielsweise die Konstruktion der ausführbaren Programme, der Test der Programme oder die Paketierung der Software, werden auch als *Release Engineering* bezeichnet.

Die dabei für das Release 1.0 gemachten Änderungen müssen anschließend in den Hauptentwicklungspfad übernommen werden. Dazu werden beide Entwicklungspfade zusammengeführt. Dies wird als »Merging« bezeichnet. Wenn es sich dabei nur um wenige kleine Änderungen handelt, ist das ohne großen Aufwand möglich. Deshalb ist es wichtig, dass alle Software-Einheiten vor dem Code Freeze intensiv geprüft wurden, damit die Anzahl der anschließend noch notwendigen Änderungen gering ist.

Natürlich kann Branching verwendet werden, um Entwicklungspfade für ganz unterschiedliche Zwecke anzulegen. So können dadurch neue Funktionen isoliert vom Hauptentwicklungspfad entwickelt werden. Je weiter die Inhalte der speziellen Entwicklungspfade vom Hauptentwicklungspfad abweichen, desto schwieriger ist das Merging. Deshalb sollte Branching wohl überlegt eingesetzt werden.

20.1.7 Konfigurationsverwaltung

Den systematischen Umgang mit den Konfigurationen eines Systems leistet die Konfigurationsverwaltung. Wir definieren:

Die **Konfigurationsverwaltung** ist diejenige Rolle oder Organisationseinheit, die die Software-Einheiten und Konfigurationen identifiziert, verwaltet, bei Bedarf bereitstellt und ihre Änderungen überwacht und dokumentiert. Dazu gehört auch die Rekonstruktion älterer Software-Einheiten und Konfigurationen.

Die Konfigurationsverwaltung schließt also die Versions- und Variantenverwaltung ein und geht damit über das hinaus, was die Bezeichnung ausdrückt. Die Grundidee der Konfigurationsverwaltung ist simpel:

- Die Konfigurationsverwaltung hat ein Monopol auf die Bereitstellung von Software. Damit wird verhindert, dass unbemerkt Varianten entstehen, unkontrollierte Änderungen stattfinden usw. Alle Entwickler übergeben ihre Software-Einheiten nur an die Konfigurationsverwaltung. Von dort und nur von dort erhalten alle anderen Stellen, vor allem die Prüf- und Integrationsabteilung, die Software. An der Konfigurationsverwaltung führt kein Weg vorbei.
- Alle Software-Einheiten, die von den Entwicklern erstellt und abgeliefert werden, werden eindeutig identifiziert, erfasst, archiviert und vor Änderungen geschützt.
- Überholte Versionen werden nicht einfach überschrieben, sondern aufbewahrt, damit sie später rekonstruiert werden können, sei es, weil man zu einem alten Zustand zurückkehren will, weil ein Kunde mit dieser Version Probleme hat oder weil man feststellen will, wann ein bestimmter Effekt erstmals aufgetreten ist.
- Wenn sich die Entwicklung verzweigt hat, werden alle Varianten bereitgehalten.
- Konfigurationen werden für die Auslieferung zusammengestellt und präzise dokumentiert.
- An allen Software-Einheiten, die der Konfigurationsverwaltung unterstehen, können Metriken erhoben werden (siehe Kap. 14).

Der Aufwand für die Konfigurationsverwaltung ist nicht unerheblich, ihr Nutzen übersteigt den Aufwand aber zuverlässig. Wo die Konfigurationsverwaltung bislang unzureichend ist, lassen sich allein durch eine Verbesserung in diesem einen Punkt erhebliche Einsparungen erzielen.

20.2 Die Aufgaben der Konfigurationsverwaltung

Eine systematisch aufgesetzte und durchgeführte Konfigurationsverwaltung übernimmt die folgenden Aufgaben (Popp, 2013):

- **Auswahl und Beschreibung der Software-Einheiten**
  - Alle Typen von Software-Einheiten, die der Konfigurationsverwaltung unterstellt sein sollen, werden festgelegt (Beispiele siehe Abschnitt 20.3.1).
  - Jeder einzelne Typ muss kurz beschrieben werden, damit jedem Teammitglied klar ist, um was es sich dabei handelt.
  - Zusätzlich muss für jeden Typ ein Schema zur Benennung der konkreten Software-Einheiten festgelegt werden, damit jede Einheit eindeutig identifiziert werden kann (siehe Abschnitt 20.3.1).

- **Speicherung der Software-Einheiten und Versionskontrolle**
  - Alle Software-Einheiten werden in einer zentralen Konfigurationsdatenbank abgelegt. Diese ist ein wichtiger Bestandteil der technischen Infrastruktur. Alle Teammitglieder haben Zugriff und müssen sie nutzen.
  - Alle Software-Einheiten sind eindeutig identifiziert.
  - Unberechtigte Zugriffe auf Software-Einheiten werden verhindert.
  - Die erstellten Versionen und Varianten aller Software-Einheiten werden über lange Zeiträume verwaltet. Nur Versionen, die nicht in Konfigurationen eingesetzt und vor längerer Zeit ersetzt wurden (Version 3 der Komponente X in Abb. 20–3), könnten gelöscht werden, wenn sie nicht aus speziellen Gründen aufzubewahren sind. Bei geeigneter Speichertechnik (nur die Änderungen werden gespeichert) ist das aber kaum sinnvoll.
- **Konfigurationskontrolle**
  - Es können beliebige Konfigurationen des Software-Systems mit genau definierten Eigenschaften erstellt werden, wenn die benötigten Software-Einheiten vorliegen.
  - Es ist sichergestellt, dass die Bestandteile einer Konfiguration konsistent, also miteinander verträglich sind.
- **Konstruktion ausführbarer Programme**
  - Der Prozess, um aus dem Code ausführbare und installierbare Programme zu erzeugen, wird mit den Konfigurationsinformationen gesteuert und ist automatisiert.
- **Änderungskontrolle**
  - Alle Änderungen an Software-Einheiten werden überwacht, alle Prüfresultate verwaltet (siehe Abschnitt 21.4).
  - Alte Versionen können wiederhergestellt werden.
- **Koordination der Teamarbeit**
  - Die Zusammenarbeit der Entwickler oder Entwicklergruppen wird durch die gemeinsame Referenzumgebung und durch die Konflikterkennung oder -vermeidung unterstützt (siehe Abschnitt 20.4.1).

20.3 Benennung und Identifikation von Software-Einheiten

20.3.1 Die Benennung von Software-Einheiten

Wie bereits erwähnt, muss zu Beginn eines Projekts festgelegt werden, welche Typen von Software-Einheiten der Konfigurationsverwaltung unterstellt sein sollen. In vielen Fällen kommen dafür die folgenden Typen, unabhängig von der konkreten Aufgabenstellung, infrage:

- Begriffslexikon
- Anforderungsspezifikation und Systementwurf
- Entwurfsbeschreibung für Komponenten
- Benutzungshandbuch und Installationsvorschrift
- Testvorschrift und Testdaten für Einzeltest, Komponententest, Integrationstest und Systemtest
- Test- und Review-Berichte
- Code, Testcode und Konfigurationsdateien

Mit Ausnahme von Test- und Review-Berichten können alle diese Software-Einheiten im Laufe des Projekts geändert werden (siehe Tab. 12–1 auf S. 265). Manchmal ist es auch notwendig, benutzte Bibliotheken oder Werkzeuge in die Konfigurationsverwaltung aufzunehmen.

Wie bei technischen Zeichnungen erweist sich auch bei der Software ein *Bezeichnungsschema* zur Kennzeichnung der Software-Einheiten als sinnvoll. Ein Bezeichnungsschema für Software-Einheiten basiert auf beschreibenden Kürzeln und Nummern. Die primitivste Bezeichnung besteht aus einem Kürzel und einer Nummer. Das andere Extrem ist eine Kette von Kürzeln ohne Nummer. Tabelle 20–1 zeigt Beispiele.

Bezeichnungsschema	Beispiel	Bestandteile
linear	DSG.00013	Produkt (DSG) und laufende Nummer
hierarchisch	DSG.EDA.EB	Produkt (DSG), Teilsystem (EDA), Art der Software-Einheit (Entwurfsbeschreibung, EB)

Tab. 20–1 *Bezeichnungsschemata für Software-Einheiten (ohne Versionsnummer)*

Beide Verfahren haben Nachteile: Kürzel verlieren oft ihren Sinn, dann sind es nur noch unverständliche Chiffren. Zudem können bei einer Änderung des Entwurfs Einheiten in andere Umgebungen geraten. Sie müssen dann umbenannt werden, was einigen Aufwand verursacht. Nummern lassen keine Rückschlüsse auf den Inhalt zu und werden leicht verwechselt und verdreht. Um die Versionen einer Einheit zu unterscheiden, hängt man in jedem Fall an die Bezeichnung noch eine Versionsnummer an.

Bei der Programmierung verwendet man »sprechende« Bezeichner. Dieses Prinzip sollte man aber nicht schematisch auf die Kennzeichnung der Software-Einheiten übertragen, denn diese werden oft in umfangreichen Listen, Konfigurationstabellen usw. geführt, die durch lange Bezeichner unübersichtlich werden. Zudem ist eine Software-Einheit in fast allen Fällen zu komplex, um wirklich durch einen Bezeichner, und sei er auch sehr lang, ausreichend charakterisiert zu werden.

Beim Code ist es zweckmäßig, die Programmeinheit (d. h. den Namen der Komponente, Klasse o. Ä.) als Dateinamen zu verwenden. Auch bei allen anderen

Dokumenten sollte man dafür sorgen, dass das Dokument nicht nur unter seinem Namen abgelegt ist, sondern diesen Namen auch explizit enthält, sodass man ohne Mühe feststellen kann, welches Dokument man gerade vor sich hat. Die Konsistenz dieser Angabe mit der Bezeichnung sollte automatisch überwacht werden.

Abbildung 20–6 zeigt ein Beispiel für die vollständige Kennzeichnung einer Software-Einheit: Sie hat die Bezeichnung `DSG.EDA.EB` und die Versionsnummer `02`. Das Kürzel `EB` zeigt an, dass es sich um eine Software-Einheit vom Typ »Entwurfsbeschreibung« handelt. Beschrieben wird die Komponente `EDA` (Erkennung der Änderungen), die Teil des Produkts Datensicherung (`DSG`) ist. Wenn Varianten entstehen, kommt ein weiteres Kürzel zur Kennzeichnung der Variante hinzu.

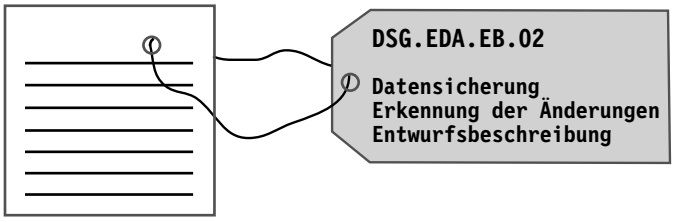

Abb. 20–6 *Kennzeichnung einer Software-Einheit*

20.3.2 Identität und Verwaltung von Software-Einheiten

Werkzeuge für die Konfigurationsverwaltung sorgen automatisch dafür, dass Versionsnummern fortlaufend vergeben werden und nicht versehentlich zwei verschiedene Software-Einheiten denselben Bezeichner bekommen.

Trotzdem kommt es immer wieder vor, dass die Mechanismen zur eindeutigen Kennzeichnung der Software-Einheiten unterlaufen werden. Ursachen können Schlampereien der Entwickler, Fehler bei der Konfigurationsverwaltung oder auch Notreparaturen sein, die fern vom Standort des Software-Herstellers vorgenommen wurden, weil eine korrekte Bearbeitung des Fehlers viel zu lange gedauert hätte. Wenn ein Ingenieur irgendwo im Urwald eine Anlage in Betrieb nehmen soll und das nur schafft, indem er ein Programm verändert (und sei es im Maschinencode), dann wird er das tun. Wenn endlich alles funktioniert, wird er die Notreparatur höchstwahrscheinlich vergessen, und niemand weiß, dass eine Variante entstanden ist. Einige Monate später wird eine neue Version der Software zur Anlage geschickt und von den Betreibern installiert. Natürlich tritt nun wieder der Fehler auf, an den sich niemand mehr erinnert hat.

Mit einer Prüfsumme kann man Veränderungen praktisch sicher erkennen. Einfaches Aufaddieren reicht aber nicht aus, weil damit Umstellungen unerkannt bleiben können; man verwendet einen effizienten Hashing-Algorithmus, der den vollständigen Inhalt der Datei verarbeitet. Die Prüfsumme sollte bei allen Ausga-

ben und Anfangsmeldungen angezeigt werden und leicht zu überprüfen und abzufragen sein.

Die Prüfsumme kann man auch zur Identifikation binärer Dateien verwenden. Verteilt man die Software als Quellcode, kann man mithilfe der Prüfsumme feststellen, ob die ausführbare Form des Programms am Zielort richtig generiert wurde.

Die Informationen, die die Konfigurationsverwaltung speichern und verknüpfen muss, sind umfangreich und extrem kritisch hinsichtlich Verlust oder Verfälschung. Darum muss eine geeignete *Konfigurationsdatenbank* benutzt werden, auf die direkt aus den Entwicklungsumgebungen der Entwickler zugegriffen werden kann. In der Konfigurationsdatenbank werden alle Informationen zu den Software-Einheiten und zu den definierten Konfigurationen abgelegt. So werden beispielsweise der Status der einzelnen Software-Einheiten und die Beziehungen zwischen ihnen gespeichert. Diese Daten dienen nicht nur zur Generierung von Konfigurationen, sondern für die Projektleitung und die Qualitätssicherung auch als Informationsquelle über den aktuellen Stand.

20.4 Arbeitsumgebungen für die Software-Bearbeitung

Bevor Software installiert und betrieben werden kann, muss sie bereitgestellt und ausgeliefert werden. Der Prozess der Auslieferung von Software lässt sich anschaulich durch die dazu notwendigen Aktivitäten beschreiben, die in verschiedenen logisch getrennten Umgebungen stattfinden.

20.4.1 Das Zusammenspiel der Umgebungen

Zwischen Entwicklung und Auslieferung der Software ist die von der Konfigurationsverwaltung beherrschte *Referenzumgebung* gesetzt. Sie sorgt dafür, dass nur geprüfte Software-Einheiten als Bausteine verwendet werden.

Abbildung 20–7 zeigt das Prinzip. Ein Entwickler erstellt, modifiziert und überprüft Software-Einheiten in seiner persönlichen *Entwicklungsumgebung*. Fertige Einheiten übergibt er der Referenzumgebung. Dort werden neue oder veränderte Software-Einheiten mit dem Status »ungeprüft« versehen und gespeichert. Sie sind damit der Konfigurationsverwaltung unterstellt, d. h., niemand kann sie dort verändern, auch nicht ihr Entwickler. In der *Prüfumgebung* werden die Software-Einheiten geprüft. Dabei werden andere, bereits geprüfte Einheiten sowie Hilfsprogramme (Testtreiber, Platzhalter) verwendet. Eine Einheit, in der Fehler entdeckt werden, erhält den Status »verworfen«; sie wird nicht verwendet, sondern ihrem Entwickler zur Korrektur zurückgegeben. Werden keine Fehler entdeckt, so wird der Status auf »freigegeben« verändert; diese Einheit kann nun an die *Integrationsumgebung* weitergegeben werden.

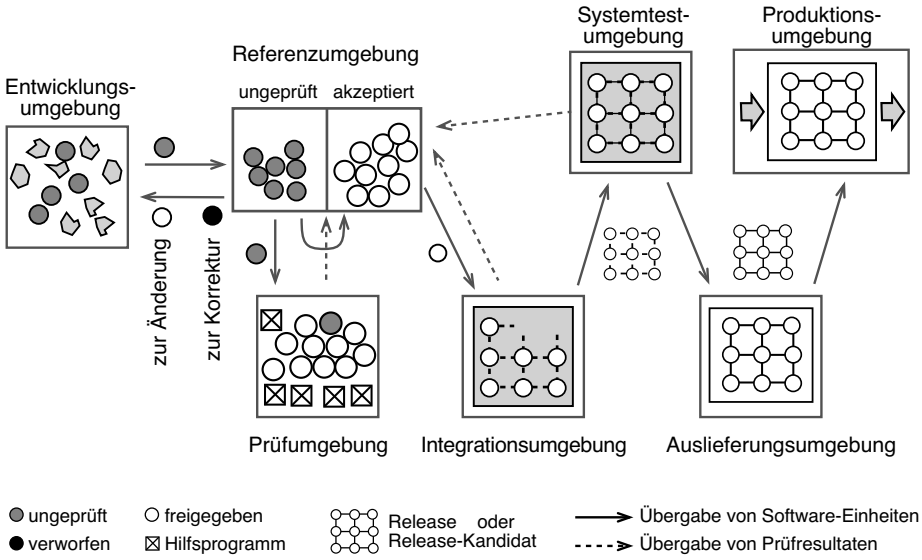

Abb. 20–7 *Die Umgebungen der Software-Bearbeitung*

Dort wird das System aus freigegebenen Einheiten schrittweise zusammengebaut. Jeder Integrationsschritt wird durch einen Test geprüft. Lässt sich eine Einheit nicht fehlerfrei integrieren oder zeigen sich im Test Fehler, ist sie oder eine der bereits integrierten Einheiten nicht in Ordnung. In diesem Fall muss der Fehler lokalisiert werden; die fehlerhafte Einheit bekommt den Status »verworfen«. Erst wenn sie korrigiert und ohne Befund geprüft ist, kann die Integration fortgesetzt werden.

Mit dem letzten Integrationsschritt ist ein vollständiges System entstanden. Der Systemtest erfolgt anschließend in der *Systemtestumgebung*. Fehler, die im Systemtest auffallen, haben die gleiche Wirkung wie die aus dem Integrationstest. Ist das System ohne Befunde getestet, so wird daraus ein Release-Kandidat, der an die *Auslieferungsumgebung* geht.

Meist werden dort auch freigegebene Updates (siehe Abschnitt 21.2.3) für ältere Releases bereitgestellt. Jede Software-Lieferung erfolgt aus der Auslieferungsumgebung. Wurde ein Release-Kandidat freigegeben, dann wird daraus ein Release. Das Release wandert dann in die *Produktionsumgebung*, wo es produktiv eingesetzt wird. Diese Umgebung kann beim Kunden, aber auch bei einem speziellen Betreiber liegen (z. B. in der Cloud).

Durch Werkzeuge und Organisation ist sichergestellt, dass Einheiten nur auf die oben beschriebene Weise, also vom Entwickler und nach Prüfung ohne Beanstandung, in die Referenzumgebung gelangen kann. Aus anderen Umgebungen fließen nie Einheiten in die Referenzumgebung zurück.

Neue oder geänderte Anforderungen machen es immer wieder notwendig, bereits freigegebene Software-Einheiten zu verändern. Es entstehen neue Versionen, die ebenfalls den beschriebenen Weg durchlaufen.

20.4.2 Realisierung der Umgebungen

In den beschriebenen Umgebungen werden Werkzeuge eingesetzt. In der Referenzumgebung wird immer ein Werkzeug zum Verwalten der Versionen, Varianten und Konfigurationen verwendet. Solche Werkzeuge werden als *Versionskontrollsysteme* bezeichnet. Alle heute zur Verfügung stehenden Werkzeuge wie beispielsweise GIT[1] gehen auf das Mitte der Siebzigerjahre an den Bell Laboratories entwickelte »System Source Code Control System« (Rochkind, 1975) und das von Walter Tichy an der Purdue University entwickelte System »Revison Control System« zurück (Tichy, 1982). Das Versionskontrollsystem realisiert die Konfigurationsdatenbank und kontrolliert den Zugriff auf bereits abgelegte Software-Einheiten. Ein weiteres Werkzeug verwaltet die Änderungsanträge und dient dazu, sie systematisch zu bearbeiten. Querverweise zu den geänderten Software-Einheiten werden aktualisiert.

Die Entwicklungsumgebung besteht aus einer Vielzahl von Werkzeugen, die zur Analyse, zum Entwurf, zur Codierung, zum Testen und zur Fehlersuche verwendet werden. Moderne Progammierwerkzeuge bieten einen einfachen und integrierten Zugriff auf das benutzte Versionskontrollsystem der Referenzumgebung.

In der Integrationsumgebung werden oft Werkzeuge genutzt, um teilintegrierte Systeme automatisch zusammenzubauen, sowie Werkzeuge, die den Integrationstest automatisiert durchführen. In der Prüfumgebung und in der Systemtestumgebung werden die Werkzeuge eingesetzt, die für die unterschiedlichen Prüfungen und die Verwaltung der Prüfresultate benötigt werden. Die Auslieferungsumgebung kann im einfachsten Fall durch eine Ordnerstruktur realisiert werden, in der die freigegebenen Releases und Updates abgelegt werden. Besser ist es aber, diese in einem speziellen Code-Repository abzulegen, auf das kontrolliert zugegriffen werden kann.

Es sollte klar sein, dass viele Probleme des Software Engineerings nicht durch Werkzeuge gelöst werden, sondern durch Methoden und organisatorische Maßnahmen. Die Konfigurationsverwaltung bedarf der Organisation; Werkzeuge sind nur dann nützlich, wenn die Organisation funktioniert.

---

1 Linus Torvalds, einer der Initiatoren des Betriebssystems Linux, startete 2005 die Entwicklung des Versionskontrollsystems GIT, um das Linux-Kernel-Projekt zu unterstützen. »GIT« (dt. Dummkopf) ist ein Scherz von Torvald. Viele Software-Hersteller verwenden GIT für die Software-Verwaltung.

20.5 Automatisierte Software-Auslieferung

In Abschnitt 20.4.1 haben wir auf Basis der Umgebungen den Prozess beschrieben, um Software auszuliefern.

Eine wichtige Aufgabe der Konfigurationsverwaltung ist es, diesen Prozess so vollständig wie möglich zu automatisieren (Abschnitt 20.2). Dazu muss die Konfigurationsverwaltung in der Lage sein, aus Software-Einheiten automatisiert ausführbare und installierbare Programme oder ganze Systeme zu bauen. Diese Aufgabe leistet der sogenannte *Build-Prozess*, dessen Ergebnis auch als »Build« bezeichnet wird. Bass, Weber und Zhu beschreiben diesen Prozess folgendermaßen:

> **build process** – The process of creating an executable artifact from input such as source code and configuration information. It primarily consists of compiling source code and packaging all files that are required for execution. Once the build is complete, a set of automated tests is executed that test whether the integration with other parts of the system uncovers any error.
>
> Bass, Weber, Zhu (2015)

Im ersten Schritt des Build-Prozesses werden die notwendigen Code-Dateien ermittelt. Dazu werden die technischen Abhängigkeiten der Programmbestandteile analysiert, die in sogenannten Build-Skripten oder make-Dateien angegeben werden. Die Code-Dateien werden in der richtigen Reihenfolge übersetzt und zu einem ausführbaren Programm zusammengebaut. Dessen Integration mit anderen Programmen wird anschließend durch automatisierte Tests geprüft.

20.5.1 Die kontinuierliche Integration

Die kontinuierliche Integration (Continuous Integration, CI) ist eine Team-Praktik im Extreme Programming (Abschnitt 10.6.4) und eine spezielle Art der inkrementellen Integration (Abschnitt 19.2.2).

Kontinuierliche Integration bedeutet, dass immer dann in der Integrationsumgebung eine vollständig integrierte Version des Systems erstellt und getestet wird, wenn eine Änderung in den Hauptentwicklungspfad eingebucht wurde. Dies ist nur möglich, wenn die Integration automatisiert durchgeführt wird.

Um zu vermeiden, dass der Build in der Integrationsumgebung fehlschlägt, sollte zuerst lokal in der Entwicklungsumgebung (EU) integriert und getestet werden, die dazu um ein lokales Versionskontrollsystem (VCS) und eine Integrationsumgebung erweitert werden muss. Erst dann wird die Änderung in das VCS der Referenzumgebung (RU) eingebucht. Abbildung 20–8 basiert auf Sommerville (2020) und zeigt schematisch und vereinfacht diesen Prozess.

Der geänderte Code wird zuerst in das lokale VCS der Entwicklungsumgebung eingebucht. Anschließend werden der aktuelle Stand des VCS der Referenzumgebung und das VCS der Entwicklungsumgebung zusammengeführt, damit alle seit der letzten Integration von anderen Entwicklern hinzugefügten Änderun-

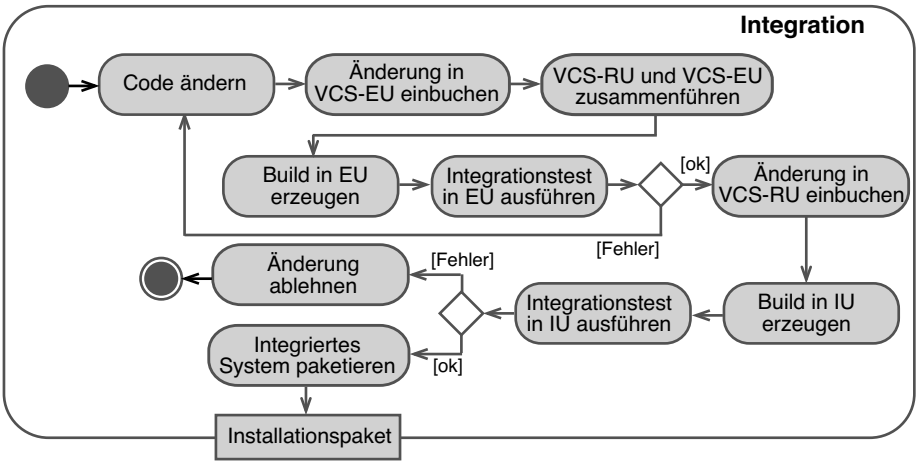

Abb. 20–8 *Kontinuierliche Integration (UML-Aktivitätsdiagramm)*

gen auch im lokalen VCS enthalten sind. Dann werden alle Code-Dateien übersetzt, und es wird ein Build erzeugt. Anschließend wird in der Entwicklungsumgebung der Integrationstest ausgeführt. Werden dabei Fehler aufgedeckt, behebt sie der Entwickler und startet den Prozess erneut. Wenn der lokale Integrationstest keine Fehler aufdeckt, wird die Änderung in das zentrale VCS der Referenzumgebung eingebucht. Dort werden noch einmal die Integrationstests in der Integrationsumgebung (IU) ausgeführt, die in der Regel keine Fehler mehr aufdecken sollten. Ist das aber der Fall, dann wird die Änderung abgelehnt. Ansonsten wird das integrierte und getestete System paketiert, damit es installiert werden kann. Natürlich müssen auch die in der Entwicklungsumgebung durchgeführten Integrationstests aktuell sein und vorher aus dem VCS der Referenzumgebung bezogen werden.

Die kontinuierliche Integration deckt nicht nur Integrationsfehler frühzeitig auf, sie schafft auch eine Qualitätskultur. Weil jedes Teammitglied vermeiden will, dass ein Build fehlschlägt, werden die Änderungen sehr sorgfältig getestet, bevor sie in das VCS der Referenzumgebung eingebucht werden.

20.5.2 Die kontinuierliche Bereitstellung

Die kontinuierliche Bereitstellung (Continuous Delivery, CD) erweitert die kontinuierliche Integration. Sie stellt sicher, dass erfolgreich integrierte Systeme auch bereit sind, um an Kunden ausgeliefert zu werden.

Das bedeutet, dass das integrierte System in der Produktionsumgebung getestet werden muss, um auszuschließen, dass Umgebungsfaktoren zu Fehlern oder sogar zu Systemausfällen führen. Dazu muss die Produktionsumgebung in der Systemtestumgebung nachgebildet werden. Wenn die Produktionsumgebung durch

einen Container definiert ist, kann ein völlig gleicher Container in der Systemtestumgebung (STU) genutzt werden. Dieser Container enthält das zu testende System und alle Ressourcen, beispielsweise Bibliotheken, Hilfsprogramme und Daten, die es zur Laufzeit benötigt.

Neben Systemtests, die die Funktion testen, werden oft auch Lasttests durchgeführt, die prüfen, ob sich das System bei steigender Last, z. B. durch viele gleichzeitige Benutzer, wie spezifiziert verhält.

Abbildung 20–9 zeigt die wesentlichen Schritte bei einer kontinuierlichen Bereitstellung. Zuerst wird die Systemtestumgebung erstellt, dann wird das vor-

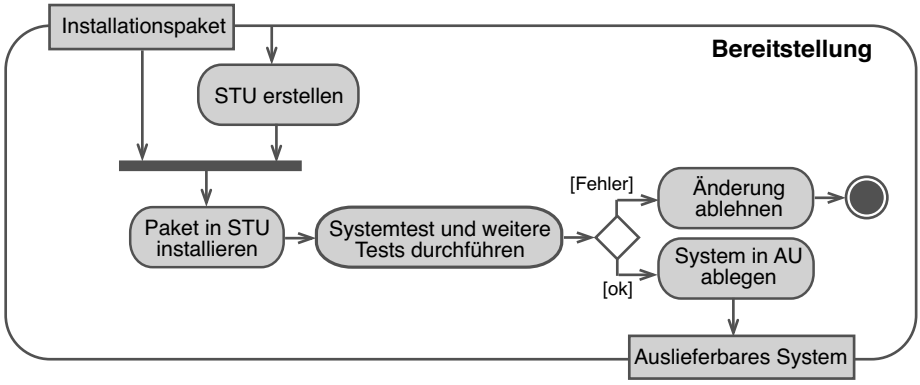

Abb. 20–9 *Kontinuierliche Bereitstellung (UML-Aktivitätsdiagramm)*

liegende Installationspaket, das das integrierte System enthält, darin installiert. Decken die in der Systemtestumgebung durchgeführten Tests Fehler auf, dann wird die Änderung abgelehnt und im VCS der Ausgangszustand wiederhergestellt. Decken sie keine Fehler auf, kann das System in der Auslieferungsumgebung (AU) abgelegt werden und steht dort als Release-Kandidat zur Verfügung. Die Installation von Release-Kandidaten in den Produktionsumgebungen der Kunden bedarf einer entsprechenden Entscheidung durch die Kunden und wird manuell angestoßen.

20.5.3 Die kontinuierliche Auslieferung

Bei der kontinuierlichen Auslieferung (Continuous Deployment, CD) geht man noch einen Schritt weiter. Durch diesen Prozess wird jede Änderung, die alle Teststufen der kontinuierlichen Integration und Bereitstellung fehlerfrei durchlaufen hat, automatisiert in den Produktionsumgebungen der Kunden installiert und in Betrieb genommen. Dies geschieht, ohne dass die Kunden das in jedem Einzelfall erlauben müssen.

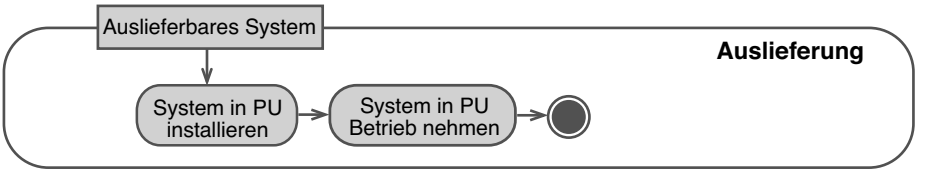

Abb. 20–10 *Kontinuierliche Auslieferung (UML-Aktivitätsdiagramm)*

Dazu wird das System aus der Auslieferungsumgebung in die Produktionsumgebung (PU) gebracht und dort installiert. Anschließend wird die dort vorhandene Vorgängerversion des Systems durch die neue Version im Betrieb ersetzt (Abbildung 20–10).

In der Literatur werden die Begriffe »Continuous Delivery« und »Continuous Deployment« häufig als Synonyme verwendet. Eine Trennung, wie sie in Bosch (2017) und Sommerville (2020) vorgenommen wird, scheint uns sinnvoll und angebracht.

Die automatisierten Tests sind die wichtigsten Elemente dieser Prozesse. Sie müssen das System auf jeder Stufe intensiv prüfen und den Entwicklern eine hohe Sicherheit geben, dass die bereitgestellte oder sogar ausgelieferte Software auch den Qualitätsanforderungen entspricht. Ohne hochwertige Tests birgt die automatisierte Bereitstellung und Auslieferung massive Risiken. Die Investition in die Entwicklung und Pflege der Tests ist dementsprechend hoch.

Eine kontinuierliche Auslieferung ist jedoch nicht immer sinnvoll, in vielen Fällen sogar nicht erlaubt. Während Software wie E-Mail-Systeme oder viele Webanwendungen kontinuierlich ausgeliefert werden können, ist das beispielsweise bei medizinischer Software nicht gestattet. Diese darf nur nach einer aufwendigen Zertifizierung ausgeliefert und genutzt werden.

Weiter gehende Informationen zu den Prozessen und Techniken der kontinuierlichen Integration, Bereitstellung und Auslieferung finden sich in den Büchern von Humble und Farley (2011) und Duvall, Matyas und Glover (2007).

20.5.4 Realisierung von Bereitstellungs- und Auslieferungsprozessen

Die beschriebenen Prozesse, um Systeme kontinuierlich zu integrieren, bereitzustellen und auszuliefern, werden technisch durch sogenannte CI/CD-Pipelines realisiert. Diese Pipelines sind immer projekt- und/oder produktspezifisch. Bevor eine Pipeline mithilfe von Werkzeugen und Skripten implementiert werden kann, muss der Prozess, den die Pipeline ausführen soll, modelliert werden. Es gibt verschiedene Ansätze und Notationen, um solche Modelle zu erstellen; eine standardisierte Notation gibt es aktuell (2022) noch nicht.

In der Literatur, beispielsweise in Humble und Farley (2011), bestehen solche Modelle typischerweise aus Abschnitten, Aktivitäten und Entscheidungen; die im

Prozess verarbeiteten und erzeugten Software-Einheiten werden als »Artefakte« bezeichnet. Steffens, Lichter und Döring (2018) verfeinern Aktivitäten in Transformationen und Prüfungen. Wir erhalten damit die folgenden Modellierungselemente:

- *Abschnitte* modularisieren die recht komplexen CI/CD-Prozesse; sie werden als »stages« bezeichnet. Ein Abschnitt fasst alle die Aktivitäten zusammen, die benötigt werden, um eine Teilaufgabe im Prozess zu realisieren.
- *Aktivitäten* erledigen grundlegende Aufgaben im Prozess, z. B. beschaffen sie Artefakte, die die Pipeline benötigt, oder speichern erzeugte Artefakte an einer vorgegebenen Stelle.
- *Transformationen* realisieren die verarbeitenden Aktivitäten. Sie wandeln Eingabe-Artefakte in Ausgabe-Artefakte um. Beispielsweise wird durch die Transformation »Übersetzen« Code in Bytecode transformiert.
- *Prüfungen* sind Aktivitäten, die Artefakte auf Basis von Kriterien bewerten und das Ergebnis der Prüfung in einem Bericht dokumentieren. Der Einzeltest kann als Prüfung modelliert werden, wenn entsprechende Qualitätskriterien für diese Testart vorgegeben sind, z. B. der Grad der erzielten Zweigüberdeckung.
- *Entscheidungen* modellieren im Prozess Schritte, an denen entschieden wird, ob der Prozess fortgeführt werden kann oder beendet werden muss. Entscheidungen basieren typischerweise auf den Ergebnissen von Prüfungen.

Abbildung 20–11 zeigt das Pipeline-Modell eines sehr einfachen, nicht vollständigen Bereitstellungsprozesses. Pipelines sind immer Teil des sogenannten »Delivery-Systems«. Das ist eine Software, die Pipelines startet und die Informationen aufnimmt, verarbeitet und entsprechend weiterleitet, die durch die Pipelines erzeugt werden.

Die dargestellte Beispiel-Pipeline besteht aus drei Abschnitten: Bauen, Testen, Bereitstellen. Nachdem alle Code-Dateien dem Versionskontrollsystem (VCS) entnommen wurden, werden sie übersetzt. Anschließend wird der Code statisch analysiert. Wenn die Prüfung zeigt, dass die vorgegebenen Qualitätsanforderungen eingehalten wurden, wird der Code paketiert. Damit ist der erste Abschnitt beendet. Im folgenden Abschnitt wird lediglich der Einzeltest durchgeführt. Falls dieser keine Fehler zeigt und die Qualitätsanforderungen für Einzeltests erfüllt sind, wird im dritten Abschnitt das erstellte Paket auf einem Paket-Server installiert und bereitgestellt. Der erfolgreiche Durchlauf der Pipeline wird dem Delivery-System gemeldet.

Immer wenn Prüfungen nicht bestanden werden, werden die Fehler und erstellten Berichte dem Delivery-System gemeldet, danach bricht die Pipeline ab. Die Pipeline wird auch immer dann abgebrochen, wenn eine Aktivität nicht erfolgreich abgeschlossen werden konnte, z. B. wenn der Code nicht übersetzt werden kann. Diese Kontrollflüsse sind in der Abbildung nicht visualisiert. Alle

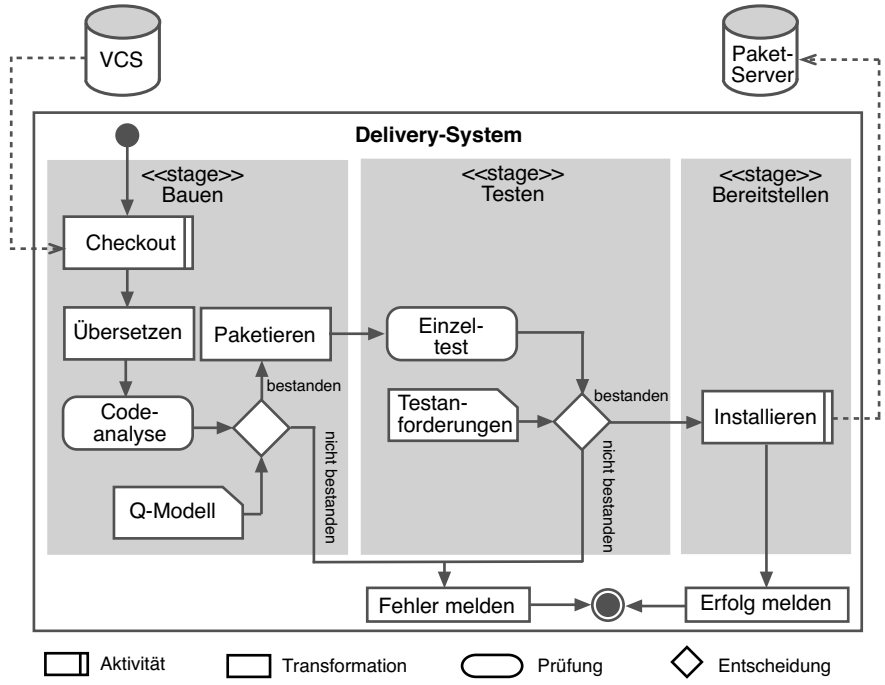

Abb. 20–11 *Ein Modell einer einfachen Pipeline*

Aktivitäten, die ausgeführt werden, melden Statusinformationen an das Delivery-System. Diese können anschließend ausgewertet werden. Moderne Delivery-Systeme stellen Funktionen zur Analyse und Visualisierung dieser Informationen zur Verfügung.

Reale Modelle bestehen aus sehr vielen Aktivitäten mit komplexen Abhängigkeiten. Bei der Entwicklung der Modelle werden, wenn immer möglich, Aktivitäten parallelisiert, um die Laufzeit der Pipelines zu reduzieren, das steigert die Komplexität der Modelle zusätzlich. Die Modelle selbst werden in XML-artigen Sprachen angegeben. Für die Realisierung der Modelle steht eine Reihe von kommerziellen und Open-Source-Werkzeugen zur Verfügung.

Das Delivery-System ist aufgrund seiner Aufgabe, Software für die Kunden bereitzustellen oder sogar bei den Kunden zu installieren und in Betrieb zu nehmen, von besonderer Bedeutung. Wenn sich Fehler zeigen, sei es im Delivery-System, sei es in den Modellen der Pipelines oder in deren Implementierungen, kann im schlechtesten Fall keine Software ausgeliefert werden. Deshalb muss die Entwicklung des Delivery-Systems und der Pipelines nach den gleichen Standards und Qualitätsvorgaben erfolgen, die auch für die Software gelten, die die Pipelines bereitstellen oder ausliefern sollen. Ein durchgängiger Engineering-Ansatz dazu fehlt aber noch.