Evolution von Softwaresystemen

Konfigurationsmanagement – Einführung, Extensionale und Intensionale Systeme

Von: Patrick Dreher

Literatur: „Version models for Software configuration management“, Reidar Conradi and Bernhard Westfechtel, ACM Computing Surveys, 30(2), 1998

1. Einführung

In der Softwareentwicklung werden Daten der verschiedensten Kategorien benötigt oder erstellt. So fallen während des Entwicklungsprozesses Daten wie Design, Spezifikation und Quellcode von Programmen, sowie begleitende Dokumentation an. Dies kann im Laufe eines Projektes eine Menge von Daten zur Folge haben, ja zu einer regelrechten „Datenflut“ führen. Eben um diese Informationen zu speichern, zu ordnen und zu verwalten benötigt man ein Software Configuration Management System. Es soll ebenfalls eine Infrastruktur für kooperatives Arbeiten innerhalb eines Projektes schaffen. Zusammenfassend ist zu sagen, dass das Software Configuration Management die Aufgabe hat die Entwicklung während allen Phasen des Projektes mit Hilfe von Werkzeugen und Standards zu unterstützen und zu kontrollieren. Man erhofft sich als Ergebnis eine Verbesserung der Qualität und Minimierung von Kosten und Fehlern während der Entwicklung.

2.Product Space

Der Product Space beschreibt die Struktur eines Software Produktes ohne Versionierung. Das heißt wir betrachten das Software Produkt unter der Annahme, dass nur eine Version existiert. Er besteht aus Software Objekten und Beziehungen zwischen diesen und wir durch Graphen repräsentiert, deren Knoten und Kanten für die Software Objekte und deren Beziehungen stehen.

2.1. Software Objekte

Software Objekte stellen das Ergebnis einer Entwicklungsaktivität dar. Software Configuration Management Systeme müssen alle Typen von Software Objekten verwalten. Software Objekte können verschiedene Repräsentationen haben, abhängig von den Typen von Tools, die auf ihnen verwendet werden. Diese Repräsentationen haben Einfluss auf die Funktionalitäten eines SCM, z.b. liefert der „diff“-Befehl bei Textfiles sich unterscheidende Textzeilen, wogegen bei anderen Typen entsprechend andere syntaktische Einheiten geliefert werden. Unabhängig von der Repräsentation unterscheidet man zwischen „domain-specific“, speziell zugeschnitten auf bestimmte Arten von Software Objekten, und „domain-independent“ Modellen, machen keine Annahmen über die Arten der Software Objekte. Hierbei ist die Versionsidentifikation ein wichtiger Aspekt; so muss jedes Software Objekt eine eindeutige Identifikation, OID (object identifier), besitzen. Man unterscheidet zwischen externer OID, vom User zugewiesen, und interner OID, vom System intern benutzt. Man

unterscheidet Software Objekte weiterhin zwischen „source“ und „derived objects“. „Source objects“ wurden von Menschen mit Hilfe von interaktiven Tools erstellt und „derived objects“ werden automatisch durch Tools erstellt. Die Klassifizierung in „source“ und „derived“ ist abhängig von den verfügbaren Tools. Außerdem können Software Objekte teilweise „derived“ oder teilweise manuell konstruiert werden. Der Prozess des Erstellen von „derived objects“ aus „source objects“ oder anderen „derived objects“ nennt man „system building“. Dieses unterliegt sogenannten „build rules“, daher spricht man auch von „build dependencies“ und „source dependencies“.

2.2.Beziehungen

Software Objekte können durch verschiedene Arten von Beziehungen miteinander verbunden sein. Man unterscheidet hier vor allem zwischen Komposition und Abhängigkeit.

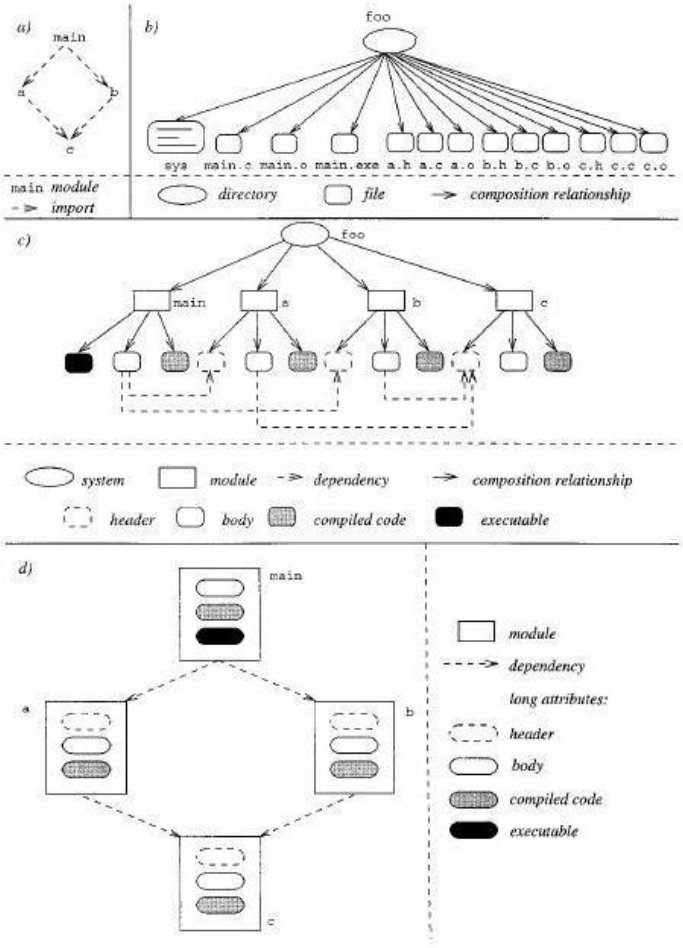
Kompositionsbeziehungen werden benutzt, um Software Objekte hinsichtlich deren Körnung zu organisieren. Objekte, die man zerlegen kann, heißen „composite objects“ (zusammengesetzte Objekte). Ein solches „composite object“ wird definiert als ein Objekt, das einen Teilgraphen des Produktgraphen repräsentiert. Es dient als Einheit bezüglich struktureller Operationen (z.b. copy oder delete), Abstraktion (Kapselung von Komponenten), Konkurrenzkontrolle (Sperren einer Komponente eines „composite objects“ schließt das Sperren aller Komponenten mit ein) und Versionskontrolle. Die Objekte an den Blättern der Kompositionshierarchie heißen atomare Objekte, die aber immer noch intern strukturiert sind. Die Wurzel einer Kompositionshierarchie heißt Software Produkt.

Abhängigkeitsbeziehungen stellen gerichtete Beziehungen zwischen Objekten dar, die unabhängig von Kompositionsbeziehungen sind. Hierbei gibt es ein „dependent“ und ein „master“ Objekt. Abhängigkeit bedeutet, dass der Inhalt des „dependent“ konsistent zu dem des „master“ bleiben muss, d.h. wenn der „master“ geändert wurde muss der „dependent“ eventuell angepasst werden.

2.3. Repräsentation des Product Space

Hier wird zur Erklärung ein Beispiel Software Produkt namens „Foo“ benutzt, die in C implementiert ist.

a) zeigt die Module von „Foo“ und deren „import“-Abhängigkeiten. Das oberste Modul „main“ importiert aus den beiden Modulen „a“ und „b“, diese beiden wiederum importieren aus Modul „c“.
b) „Foo“ ist hier in einem File System gespeichert: jedes Modul wird repräsentiert durch mehrere Files. Die Endungen .h, .c, .o und .exe bezeichnen „header“-Files, “body“-Files, compiliertem Code und „executable“ Files. Abhängigkeiten und „build-steps“ sind in einem Textfile gespeichert: „sys“
c) Genau wie in b) haben wir immer noch einen Kompositionsbaum, dessen Blätter mit einzelnen Files korrespondieren. Abhängigkeiten werden hier nicht mehr in einem separaten Textfile gespeichert, sondern der Baum wird durch diese Abhängigkeiten ergänzt. (ähnelt der logischen Struktur aus b), ergänzt durch die Informationen aus a))
d) Hier werden alle Files, die ein Modul bilden, werden zu einem Objekt zusammengefasst. Außerdem wird nur ein einzelner Typ von Beziehung benutzt: „source-dependencies“ zwischen Modulen. (ähnelt der logischen Struktur in a), ergänzt durch die Informationen aus b))



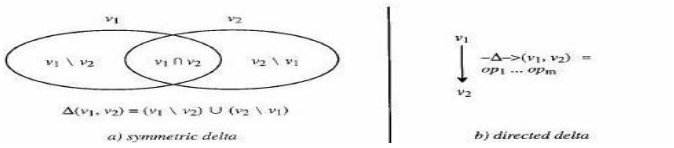
3.Version Space

Das Versionsmodell definiert Items, die versioniert werden, die gemeinsamen Eigenschaften aller Versionen, die „deltas“, Unterschiede zwischen zwei Versionen eines Items, die Art und Weise, wie die Menge der Versionen organisiert werden, die passende Repräsentation der Versionsmenge, führt verschiedene Dimension von Entwicklung ein, Revisionen und Varianten und bietet Operationen zum Wiederherstellen früherer Versionen, sowie zum Konstruieren neuer Versionen.

3.1. Versionen, versionierte Items und Deltas

Definition: Eine Version v repräsentiert den Zustand eines, in der Entwicklung befindlichen, Items i und wird charakterisiert durch ein Paar v=(ps,vs), wobei ps und vs einen Zustand im Product Space beziehungsweise Version Space bezeichnen.
Ein versioniertes Item ist ein Item, das der Versionskontrolle unterworfen wird.
Versionierung kann mit verschiedener Granularität angewendet werden und benötigt ein eindeutiges Kriterium, welches angibt, ob zwei Versionen zu demselbem Item gehören.
Alle Versionen eines Items besitzen gemeinsame Eigenschaften, die man Invarianten nennt, z.b. unversionierte Attribute oder Beziehungen. Versionen teilen semantische Eigenschaften
Jede Version muss eindeutig identifizierbar sein, durch einen VID (version identifier). Versionen besitzen virtuell nur einen gemeinsamen OID.
Der Unterschied zwischen zwei Versionen heißt Delta. Es gibt zwei verschiedene Definitionen von Delta:

- Symmetrisches Delta zweier Versionen v_1,v_2 = (v_1\$v_2$) ∪ ($v_2$\$v_1$)
- gerichtetes Delta (auch Veränderung oder Change genannt) = eine Folge von Operationen die auf v_1 angewendet zu v_2 führen.



3.2. Extensionales und intensionales Versionieren

Extensionales Versionieren:
Die Versionsmenge V ist explizit gegeben durch Aufzählen aller Versionen: $V=\{v_1,v_2,...,v_n\}$. Alle Versionen sind explizit gegeben und typischerweise durch Nummern gekennzeichnet. Ein User öffnet eine Version v_i, führt an dieser Änderungen durch und speichert sie (meist) als v_{i+1}. Alte Versionen können unveränderlich („immutable“) gemacht werden, diese wird bereits von vielen Systemen automatisch bei früheren Versionen gemacht. Außerdem bietet extensionales Versionieren das Wiederherstellen früherer Versionen.

Intensionales Versionieren:
Die Versionsmenge V wird durch ein Prädikat c definiert: $V=\{v \mid c(v)\}$. Das Prädikat c definiert Bedingungen, die erfüllt werden müssen und Versionen sind somit implizit gegeben. Eine spezifische Version wird durch ihre Eigenschaften beschrieben und wird als Ergebnis einer Anfrage („query“) generiert.

Extensionales und intensionales Versionieren schließen sich in keinster Weise gegenseitig aus, sondern können und sollen in einem SCM kombiniert werden.

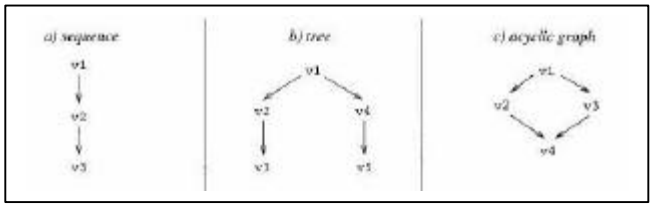
3.3. Revisionen, Varianten

Versionen, die ihren Vorgänger ersetzen sollen heißen Revisionen. Sie entwickeln sich entlang der Zeitachse und aus verschiedenen Gründen, wie z.b. Beseitigung von Bugs, Erweitern von Funktionalitäten, Anpassung an äußere Änderungen,... .
Version, die gedacht sind zu koexistieren, heißen Varianten, z.b. für Software Produkte auf mehreren Systemen.

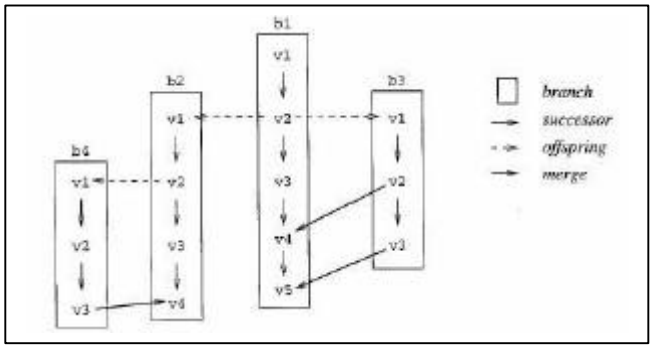
3.4. Repräsentation des Version Space

Viele SCM benutzen Versionsgraphen als Repräsentation des Version Space. Solche Versionsgraphen werden in Verbindung mit extensionaler Versionierung angewendet, da man für einen Versionsgraphen eine explizit gegebene Versionsmenge benötigt. Hier gibt es verschiedene Ansätze:

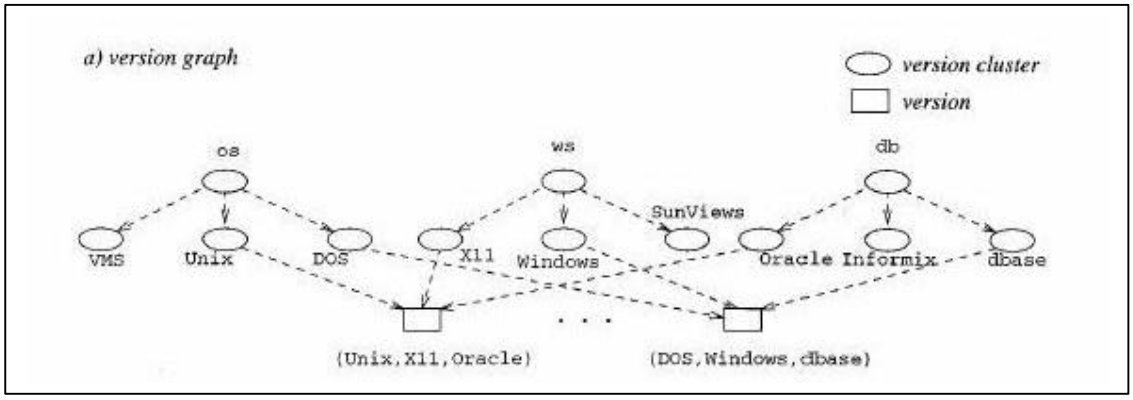
- „one-level-organization“: Ein Graph besteht aus einer Menge von Versionen verbunden durch eine einzige Art von Verbindungen (Nachfolger). Solch ein Graph repräsentiert die Entwicklungsgeschichte eines versionierten Items. Er kann je nach auferlegten Beschränkungen verschiedene Formen besitzen: eine Folge von Versionen (im beschränktesten Falle (nur Nachfolger der aktuellsten Version)), ein Versionsbaum (es dürfen auch Nachfolger von Versionen gebildet werden, die nicht an den Blättern liegen) und ein azyklischer Graph (eine Version darf mehrere Vorgänger besitzen).



- „two-level-organization“: Ein Graph besteht aus Zweigen, die ihrerseits wiederum aus einer Folge von Revisionen bestehen. Hier gibt es mindestens zwei Arten von Beziehungen: Nachfolger (innerhalb eines Zweiges) und Ursprung (zwischen Zweigen). Mischen bedeutet hier, Veränderungen, die in einem Zweig durchgeführt wurden, können in einen anderen Zweig fortgesetzt werden. Die Zweige werden nicht verbunden, sondern existieren weiter



- „multidimensional-variation“: Versionen werden in Cluster organisiert, um Klassifikationshierarchien zu konstruieren.



3.5. State-based und Change-based

Versionsmodelle, die ihr Augenmerk auf die Zustände eines versionierten Items legen heißen "State-based". „State-based" Versionierung nennt die Veränderung nicht, das bedeutet sie müssen aus dem Graphen hergeleitet werden, was eventuell sehr schwierig sein kann. „State-based" extensionale Versionierung wird z.B. von RCS benutzt, „State-based" intensionale Versionierung kann z.B. durch „conditional compilation", wobei die Zustände Tupeln entsprechen, die die entsprechenden Werte der Präprozessorvariablen enthalten.
Versionsmodelle, die Versionen durch Ausdrücke von Veränderungen beschreiben heißen „Change-based". Im Falle von „change-based" extensionaler Versionierung ist die Versionsmenge V weiterhin explizit definiert und jede Version wird beschrieben durch Änderungen, somit dienen die Änderungen hauptsächlich als Dokumentation. Bei „change-based" intensionaler Versionierung wird eine Version v konstruiert durch Anwenden einer Folge von Änderungen $c_1,..,c_n$ bezüglich einer Basis b: $v= c_n(...(c_1(b)...)$ und Änderungen können frei kombiniert werden. Somit ist klar, dass „state-based" versus „change-based" unabhängig von extensional versus intensional ist.
Der Change Space, das heißt der Version Space strukturiert in Form von Änderungen, kann verschieden repräsentiert werden:

- Matrixrepräsentation: Zeilen und Spalten korrespondieren mit Versionen und Änderungen, und ein Kreis gibt an, dass in der Version der entsprechenden Zeile die Veränderung der entsprechenden Spalte ausgeführt wurde.
- Versionsgraph mit expliziten Änderungen: zeigt explizit, wann bestimmte Änderungen angewendet wurden

4. Zusammenspiel zwischen Product Space und Version Space

Wir haben bisher den Product Space, in dem es von jedem Item nur eine Version gibt, und den Version Space, in dem nur einzelne Items versioniert wurden. Nun betrachten wir die „versioned object base“, die den Product Space und den Version Space kombiniert. Das Versionsmodell legt fest welche Items, mit welcher Körnung versioniert werden, in welcher Beziehung Versionen verschiedener Items zueinander stehen und welches Modell zur Repräsentation benutzt wird.

5. Intensionales Versionieren

Intensionales Versionieren beschäftigt sich vorrangig mit der Konstruktion neuer Versionen aus eigenschaftsbasierten Beschreibungen. Es ist von großer Bedeutung für große Version Spaces. Es muss „Combinability“, das heißt jede Version muss auf Verlangen konstruiert werden können, und Konsistenzkontrolle, konstruierte Versionen müssen Vorgaben einhalten, gewährleisten. Hierbei entsteht ein Problem, da die Anzahl der möglichen Versionen schnell ins unermessliche steigt. Die Herausforderung besteht darin Konsistenzkontrolle zu bieten, während die „Combinability“ weiterhin unterstützt wird. „Change-based“ Versionierung reduziert das „Combinability“ Problem durch Gruppierung logisch zusammenhängender Änderungen. Konfigurationsregeln werden benutzt, um konsistente Versionen zu konstruieren. Konfigurationsregeln werden klassifiziert in „built-in rules“, fest vorgegeben und nicht veränderbar, und „user-defined rules“. Konfigurationsregeln bestehen aus einem Versionsteil und einem Produktteil. Der Versionsteil bezieht sich im Revision Space auf die Zeitdimension, im Variant Space auf die Werte der Varianten Attribute und im Change Space spezifizieren sie eine Version in Form von Änderungen, die angewendet wurden. Der Produktteil beschreibt, auf welchen Teil eines Produktes sich die Konfigurationsregel bezieht. Konfigurationsregeln können in Striktheitsklassen eingeteilt werden. Es gibt „Constraint“ (obligatorische Regel: muss erfüllt werden und jede Verletzung hat Inkonsistenz zur Folge), „Präferenz“ (optionale Regel: wird nur angewendet, wenn sie erfüllt werden kann) und „Default“ (optionale Regel: Schwächer als Präferenz). Des weiteren können Regeln zusätzlich explizit oder implizit, durch die Reihenfolge, mit Prioritäten versehen werden. Ein Konfigurator konstruiert eine Version durch Auswertung der Konfigurationsregeln.
„Merge-Tools“ kombinieren Versionen oder Änderungen und können wie folgt klassifiziert werden:

- „raw merging“: ein Change wird in einem anderen Kontext angewendet
- „2-way-merging“: vergleicht zwei verschiedene Versionen und mischt diese in eine Version; bei Unterschieden muss der User die geeignete Alternative wählen
- „3-way-merging“: vergleicht ebenfalls Versionen, aber wenn ein Change nur in einer Version ausgeführt wurde, wird er automatisch aufgenommen; andernfalls gibt es einen Konflikt, der entweder automatisch oder manuell gelöst wird.

Von einer Inkosistenz spricht man, wenn eine Änderung nicht durchgeführt werden kann und ein Konflikt ist vorhanden, wenn es zwei gegensätzliche Änderungen gibt. „Merge-Tools“ können zusätzlich durch den semantischen Level, in dem Mischen ausgeführt wird, charakterisiert werden. Man spricht deshalb z.b. von textuellem Mischen, syntaktischem Mischen oder semantischem Mischen.

6.Fazit

Zum Schluss wollen wir noch einmal kurz resümieren, worin der Unterschied zwischen extensionalen und intensionalen Systemen besteht. Extensionales Versionieren legt das Hauptaugenmerk auf das Wiederherstellen früherer Versionen, wogegen intensionales Versionieren sich vor allem auf das Konstruieren neuer Versionen konzentriert. Da bei extensionalen Systemen die Versionenmenge explizit gegeben ist, ist es hier wichtig sämtliche Daten effizient zu speichern, da die anfallende Datenmenge sehr groß werden kann. Bei intensionalen Systemen liegt die Herausforderung darin, Konsistenzkontrolle zu bieten, während die „Combinability“ weiterhin erhalten bleibt. Intensionale Versionierung wird besonders bei großen Version Spaces angewandt.
Abschließend bleibt zu sagen, dass in einem Software Configuration Management System beide Arten kombiniert werden sollten. Dies wird häufig in der Form realisiert, dass die Komponenten extensional und die Produktversionen intensional versioniert werden.