Architekturprüfung für Modelle
Modell-Erosion effektiv verhindern

Ingo Battis, Sennheiser electronic GmbH & Co. KG
Thomas Eisenbarth, Axivion GmbH

Die Versprechungen der UML-Modellierung mit anschließender Codegenerierung sind signifikant höhere Wartbarkeit, Fehlerfreiheit und Flexibilität im Vergleich zur manuellen Codierung. In diesem Szenario übernimmt das UML-Modell als Implementierungsmodell Aufgaben aus der Codierung. Damit muss man das UML-Modell so behandeln, als ob es die Stelle von Code einnähme und es entsprechend vor Software-Erosion und Implementierungsfehlern schützen. Eine Implementierung muss immer einer Architektur folgen. In unserem Falle muss daher das Implementierungsmodell einer Architektur folgen, die folglich nicht im Implementierungsmodell selbst enthalten sein kann. Erst dadurch kann das Modell auf verschiedene Arten von Korrektheit geprüft werden.

Einsatz-Szenario bei Sennheiser
Für die Entwicklung eines neuen Projekts haben wir uns entschieden, zu einem großen Teil auf die Modellierung mittels UML und die anschließende Generierung von Code aus den Modellen zu setzen. Nur in der Treiberschicht soll manuelle Codierung eingesetzt werden. Zu diesem Zweck werden das Design und die Implementierung vollständig in Rhapsody durchgeführt.
Außerdem pflegen wir eine Architektur, die die Blaupause für Design und Implementierung darstellt.

Hierarchisch gegliederte Modelle
Im betrachteten Projekt erfolgt die Entwicklung der Software als die Erstellung einer Hierarchie von Modellen. Diese Modelle werden durch Relationen ihrer jeweiligen Elemente miteinander verbunden. Eine dieser Relationen nennen wir Verfeinerung. In unserem Beispiel gibt es die Modelle Architektur, Entwurf und Codierung. Die Modelle werden in der Hierarchie von oben nach unten immer detaillierter und dabei immer konkreter. In unserem konkreten Anwendungsfall enthält die Architektur die Schichten, der Entwurf betrachtet Komponenten und weitere Details innerhalb der Schichten, die Codierung schließlich stellt konkrete Elemente der Zielsprache dar, z.B. C-Funktionen.

Die Elemente eines Modells werden in dem hierarchisch darunterliegenden Modell verfeinert. Die Architektur wird manuell zum Entwurf verfeinert, dieser wird in unserem Falle weitestgehend automatisiert durch einen Generator zur Implementierung implizit verfeinert.

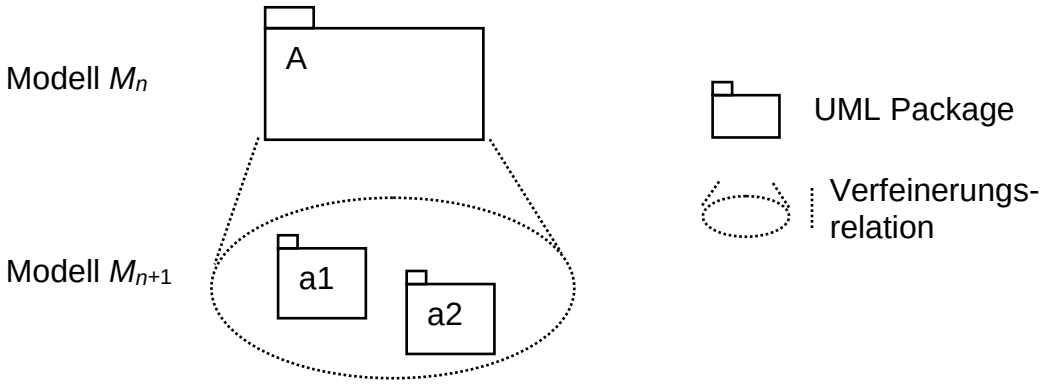

Abb. 1: Verfeinerung eines Modell-Elements: Package **A** in Modell M_n wird verfeinert zu Package **a1** und **a2** in Modell M_{n+1}.

In Abb. 1 ist als Beispiel die Verfeinerung eines UML Packages aus der Architektur (Modell M_n) in zwei UML Packages im Entwurf (Modell M_{n+1}) dargestellt: Diese Verfeinerung ist später wichtig für die Prüfung. Die Verfeinerung erfolgt manuell und folgt in unserem Beispiel einer Namensregel, was für die Nachvollziehbarkeit zwar wünschenswert, jedoch im Allgemeinen nicht notwendig ist. Die hier beschriebene Vorgehensweise ist übrigens unabhängig von den konkreten Typen der Elemente und somit auch nicht auf die UML festgelegt.

Für unseren Ansatz ist es gleichgültig, in welcher Reihenfolge die Modelle erstellt und verändert werden. Die Modelle können nachträglich verändert und erweitert werden. Es ist somit nicht notwendig, die Modelle und die Verfeinerungen streng sequentiell und top-down (analog zu einem Wasserfallmodell) durchzuführen, eine iterativ-inkrementellen Vorgehensweise ist jederzeit möglich. Tatsächlich nutzt die angewendete Prüfungsmethodik die Modelle und die Verfeinerungsrelationen, jedoch kein Wissen über Reihenfolgen im Entstehungsprozess.

Wenn man UML-Werkzeuge einsetzt, um die hier angesprochenen Modelle zu definieren, dann hat man das Problem, dass in UML selbst nur ein UML-Modell existiert. Diagramme dienen nur der Darstellung, haben aber ansonsten keine Auswirkungen auf die Struktur des UML-Modells.

Um in der Hierarchie der Modelle prüfen zu können, ob eine korrekte Verfeinerung vorliegt, müssen die Modelle disjunkt und über die Verfeinerungsrelation verbunden sein. Ansonsten ist eine Prüfung nicht möglich. Beispiel: Nehmen wir an, die gleichen UML Package-Entitäten werden sowohl in der Architektur als auch im Entwurf genutzt.

In Abb. 2 wird in Modell M_n eine UML Dependency zwischen zwei Packages **A** und **B** eingeführt, weil in M_{n+1} in diesen Packages enthaltene Klassen voneinander ableiten. Genau diese Abhängigkeit hätte man so bereits in der Architektur einfügen können. Es lässt sich nicht mehr prüfen, ob diese Abhängigkeit korrekt von der Architektur in den Entwurf verfeinert worden ist.

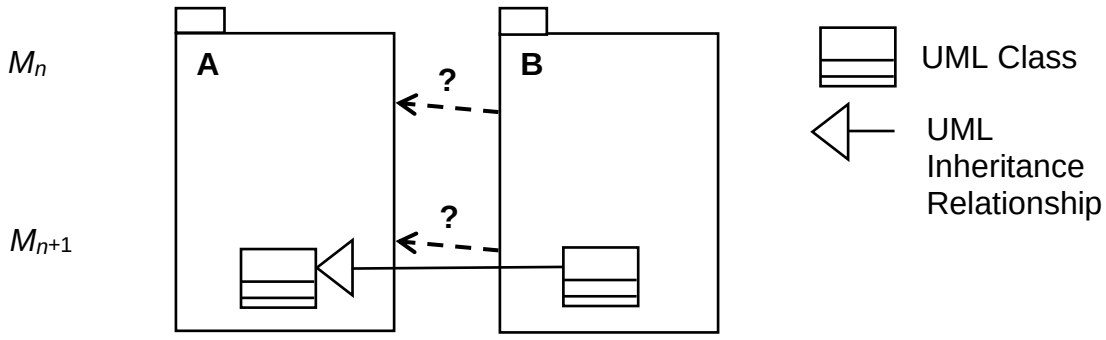

Abb. 2: Packages **A** und **B** sind sowohl in Modell M_n als auch M_{n+1} vorhanden und die Verfeinerung ist implizit gegeben.

Das Beispiel zeigt auch, dass ein System, bei dem eine Änderung in einem Modell automatisch zu einer Änderung im anderen Modell führt, keine Konsistenzprüfung zwischen den Modellen zulässt. Oft wird diese Kopplung als Vorteil gesehen („das Gesamtmodell stimmt immer"), es ist aber nur aus dem Gesichtspunkt des Reverse-Engineerings und des simplen Programmverstehens sinnvoll („ich sehe, was ist"). Aus der Sicht der Planung, der Nachvollziehbarkeit und des abstrakten Programmverstehens („ich sehe, wo die Unterschiede liegen zwischen dem was ist, und dem was sein sollte") ist dieses Vorgehen kontraproduktiv.

Um die Integrität der Architektur zu gewährleisten, haben wir uns entschlossen, die Architektur in Enterprise Architect zu modellieren, das Design jedoch in Rhapsody. Auf diese Weise ist bereits technisch ausgeschlossen, dass ungewollt Beziehungen aus einem Modell in ein anderes gelangen.

Die Architekturprüfung an sich

Die Prüfung zwischen zwei Modellen in der Modellhierarchie verläuft als Graphenoperation, bei der jeder Entität und Beziehung aus dem konkreteren Modell eine Entität bzw. Beziehung aus dem abstrakteren Modell zugeordnet wird [1]. Gelingt dies nicht, so liegt ein Verstoß vor. Dabei können sich folgende Situationen ergeben:

Bei der Erstellung der Verfeinerung (siehe Abb. 3):

- Ein abstraktes Element wird nicht konkreter verfeinert
- Ein konkretes Element ist keine Verfeinerung eines abstrakteren.
-

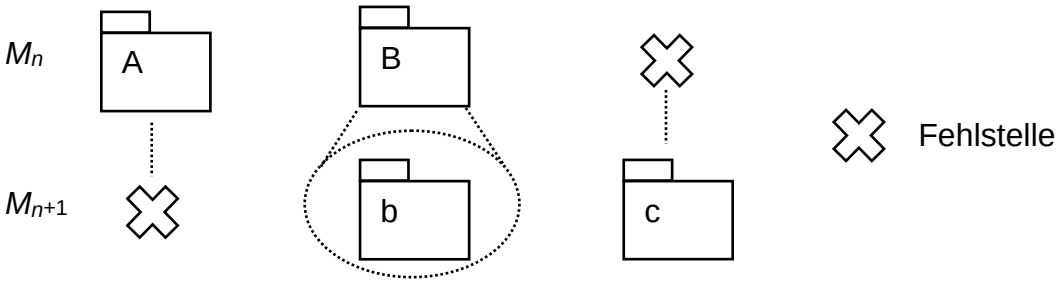

Abb 3: **A** wird nicht konkreter verfeinert, **c** ist keine Verfeinerung einer abstrakten Entität.

Jede Verfeinerung zwischen zwei Modellen M_n und M_{n+1} muss die Relationen des Modells M_n im Modell M_{n+1} in der Verfeinerung beibehalten. Wenn dies nicht der Fall ist, dann ist die Abbildung zwischen den Modellen nicht sinnvoll. Diese Situation bedeutet einen Modellierungsfehler, siehe Abb. Z.

Während des eigentlichen Prüfvorgangs (siehe Abb. Z):

- Es gibt eine Relation **a** $\rightarrow$ **b** in M_{n+1}, wobei **a** eine Verfeinerung von **A** und **b** eine Verfeinerung von **B**, jedoch gibt es keine Relation **A** $\rightarrow$ **B** in M_n (diese Situation nennen wir Divergenz).
- Es gibt eine Relation **A** $\rightarrow$ **B** in M_n, jedoch keine Verfeinerung **a** von **A** und Verfeinerung **b** von **B**, so dass **a** $\rightarrow$ **b** gilt (diese Situation nennen wir Absenz)

Für die Architekturprüfung müssen wir festlegen, welche UML-Elemente (Entitäten und Beziehungen) jeweils in der Architektur und im Entwurf verwendet werden dürfen und wie sie einander entsprechen.
Beispielsweise entspricht eine Dependency mit Stereotype „use“ in der Architektur unseres Fallbeispiels einer ganzen Reihe von Dependencies und Associations im Design, z.B. Generalization, use-Depencency und Association.
Die Verfeinerungsrelation zwischen Architektur und Entwurf wird durch ein Python-Skript berechnet, welches die Festlegung einliest und die Verfeinerungsrelationen darauf basierend erstellt. In unserem Fall ist die Verfeinerungsrelation durch Namensregeln gegeben: Ein Package namens „A“ in der Architektur entspricht genau einem Package namens „PkgA“ im Entwurf. Diese Relation ist als 1:1-Relation besonders einfach, aber im Allgemeinen sind 1:*n*-Relationen möglich.
Die oben aufgeführten Fehlerbedingungen beim Erstellen der Verfeinerung werden ebenfalls diagnostiziert. Dadurch können Fehlbedienungen und Fehlinterpretationen im Tooleinsatz aufgedeckt werden.

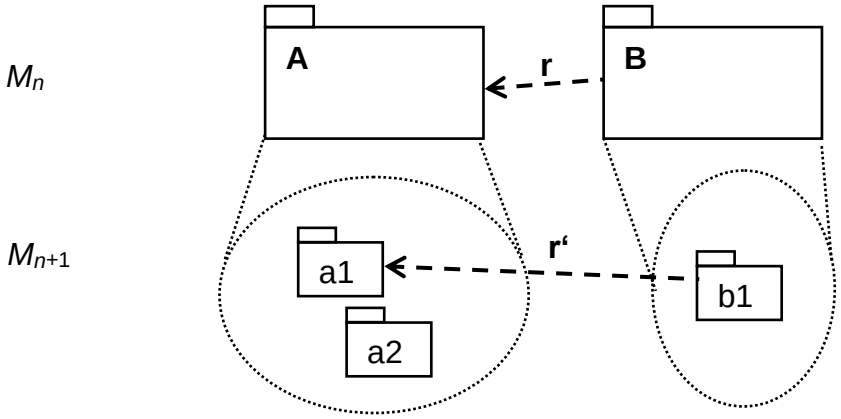

Abb. 4: Konvergenz: **A** und **B** aus Modell M_n werden verfeinert zu **a1**, **a2** und **b1**, die Relation **r** zwischen **A** und **B** wird verfeinert zu der Relation **r‘** zwischen **a1** und **b1**.

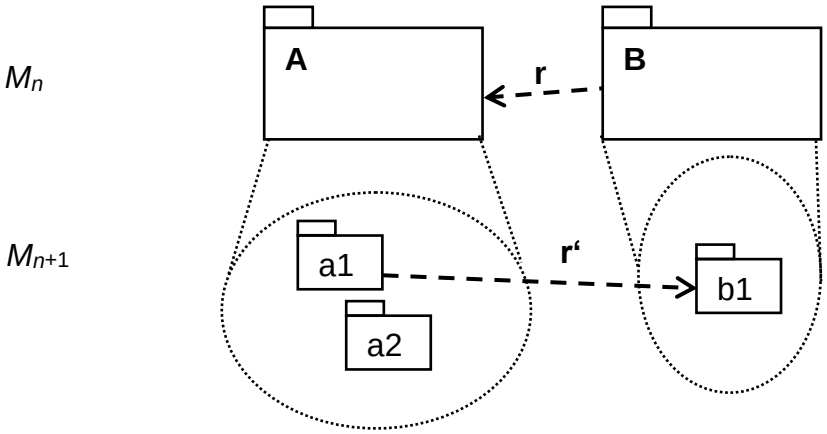

Abb. 5: Absenz: Die Relation **r** zwischen **A** und **B** in Modell M_{n+1} wird in Modell M_{n+1} zwischen **a1**, **a2** und **b1** nicht verfeinert. Divergenz: Stattdessen taucht eine Relation **r‘** von **a1** nach **b1** auf, obwohl keine entsprechende Relation zwischen **A** und **B** in M_n vorliegt.

Erfolg des Einsatzes: Beispiele für aufgedeckte Erosion

In der täglichen Arbeit hat sich die Prüfung bewährt und Situationen aufgedeckt, die andernfalls zu einer langfristigen Erosion der Architekturbeschreibung geführt hätten, siehe Abb. 6.

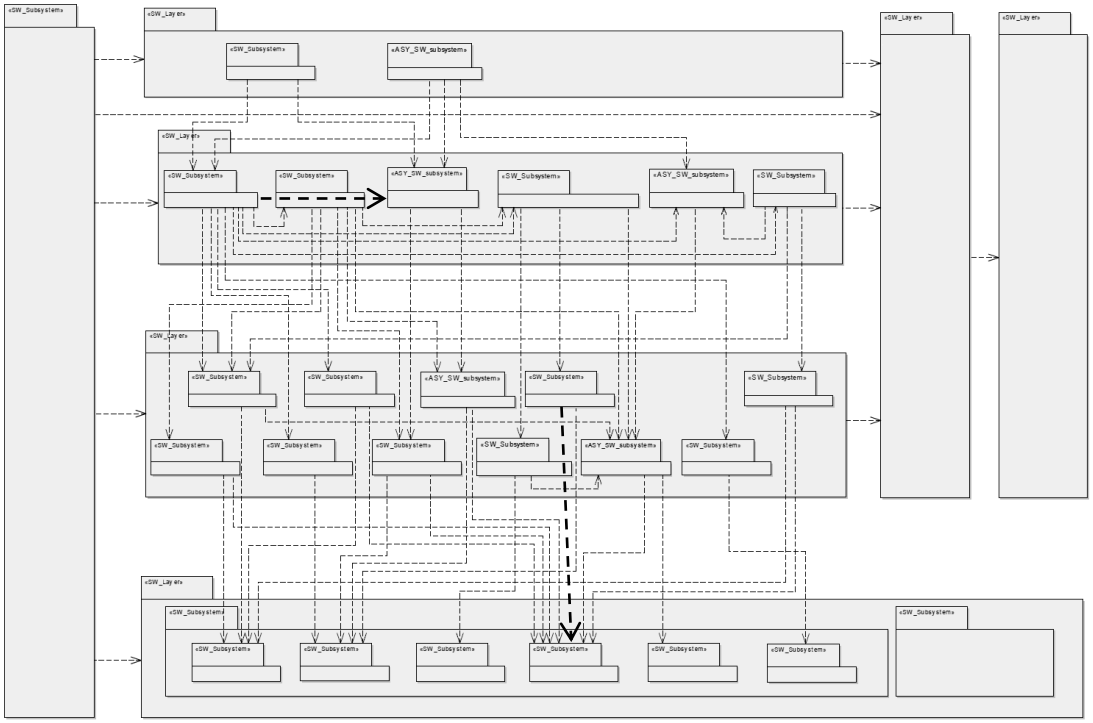

Abb. 6: Die in Enterprise Architect modellierte Architektur überlagert mit zwei Befunden (dick), wo der Entwurf zusätzliche Relationen besitzt.

Die Analysen werden in einer Continuous Build-Umgebung ausgeführt. Die einzelnen notwendigen Schritte sind:

- Import der Architektur aus Enterprise Architekt
- Import des Designs aus Rhapsody
- Vorbereitende Schritte: Interpretation der Modelle, Berechnen der Verfeinerungsrelation
- Durchführung der Prüfung
- Visualisierung der Ergebnisse

Die Imports, vorbereitenden Schritte und die Prüfungen werden mit Hilfe der Axivion Bauhaus Suite durchgeführt. Visualisiert werden die gefundenen Verstöße entweder graphisch oder, im täglichen Einsatz, anhand des Dashboards der Axivion Bauhaus Suite. Dank des prozessbegleitenden Vorgehens ist es direkt möglich, die fraglichen Stellen architekturkonform umzubauen bzw. die entsprechend Architektur anzupassen.

Literatur

[1] Koschke, Rainer; Simon, Daniel: Hierarchical Reflexion Models. In: Proc. of the Working Conference on Reverse Engineering, IEEE Computer Society Press, 2003.

Autoren

Ingo Battis ist System Architekt in der Professional Systems Division bei der Sennheiser electronic GmbH & Co. KG. Die Schwerpunkte seiner Tätigkeit in der Produktentwicklung sind die Erstellung von Software Architekturen, die Steuerung eines Software-Entwicklungsteams im agilen Umfeld und die SW Integration. Herr Battis beschäftigt sich außerdem mit Code Generatoren und Build-Prozess Automatisierung. Nach dem Studium der technischen Informatik folgten mehrjährige Stationen bei Thomson Multimedia und Bosch Car Multimedia. Herr Battis hat 15 Jahre Berufserfahrung in der embedded Software Entwicklung und dem Bereich der verteilten Systeme.

Herr Eisenbarth ist Gründer und Geschäftsführer der Axivion GmbH, die in Kooperation mit den Unis Stuttgart und Bremen Analysewerkzeuge rund um das Thema Entwicklung und Qualitätssicherung von Software entwickelt und vertreibt. Durch seine langjährige Tätigkeit auf dem Gebiet der Software-Analyse und -Wartung kennt Herr Eisenbarth eine Vielzahl von Architekturen, resultierenden Implementierungen und damit gekoppelten Vorgehensweisen.