Dr. Horst Kargl

(E-Mail: Horst.Kargl@sparxsystems.eu)
beschäftigt sich seit 1998 mit objektorientierter Modellierung und Programmierung. Bevor er 2008 zu SparxSystems wechselte, war er an der TU Wien als wissenschaftlicher Mitarbeiter in der Lehre tätig und forschte in mehreren Projekten an den Themen e-Learning, Semantic Web sowie modellgetriebener Software Entwicklung. Hierzu dissertierte er und hat sich mit der automatischen Integration von Modellierungssprachen beschäftigt. Während seines PhD Studiums war er bereits als freiberuflicher Mitarbeiter bei SparxSystems tätig. Im September 2008 wechselte er fix als Trainer und Berater zu SparxSystems Software GmbH Central Europa. Seine Schwerpunkte sind Software Architektur, Code Generierung sowie die Anpassungs- und Erweiterungsmöglichkeiten von Enterprise Architect.

Agile Modelle – Modelle ohne zu modellieren

Konzeptionelle Modelle sind meist erwünscht, doch im Projektalltag bleibt oft viel zu wenig Zeit dafür. Dieser Artikel beschreibt einen Ansatz, wie Informationen im Code genützt werden können, um daraus automatisch Modelle abzuleiten, die mehr als nur eine 1:1 Abbildung der Implementierung darstellen. Die generierten Modelle können mit manuell erstellten, abstrakteren Modellen verwoben werden. Dies führt zu einem zusammenhängenden Gesamtmodell.

Motivation

Wer heute im Projektgeschäft tätig ist und nicht von sich behaupten kann, agil zu arbeiten, hat schon fast verloren. Im Software Engineering haben sich in den letzen Jahren einige agile Methoden etabliert. Die wohl bekanntesten davon sind: SCRUM [Sch02], XP [Bec04], FDD [DeL08] und Crystal [Coc04].

Agile Vorgehensmodelle zeichnen sich durch kurze Iterationszyklen aus, an deren Ende auslieferbarer Code steht. Dabei ist es erlaubt, dass sich Anforderungen ändern. Im Gegensatz zu SCRUM, XP und Crystal ist Feature Driven Development (FDD) ein Ansatz, der dem Wasserfallmodell am ähnlichsten ist. Doch egal welche Methode gewählt wird, wie kurz oder lang die Iterationen dauern, wie schnell sich Anforderungen ändern dürfen, eines ist bei allen Ansätzen gleich: es gibt immer einen „Plan“. Je nach Methodik ist der „Plan“ formaler und länger gültig, wie zum Beispiel im fachlichen Gesamtmodell von FDD, oder kurzlebiger und gegebenenfalls reduziert auf fachliche Anforderungen, wie oftmals bei XP.

Ob man gern mit grafischen Modellen arbeitet, oder nicht, bestimmt meist die bisherige Projekterfahrung. In vielen Projekten wird allerdings vom Kunden am Ende des Projektes eine Dokumentation in Form von UML o.ä. gefordert. Modelle können textuell oder grafisch sein. Im folgenden Artikel wird der Begriff Modell immer als Synonym für ein grafisches UML Modell verwendet.

Entwickler, die mit Modellen groß geworden sind, kommen gut damit zurecht und kennen den enormen Mehrwert einer grafischen, abstrakteren Beschreibung des zu realisierenden Systems. Aber auch sie leiden oft unter dem Zeitdruck, am Ende des Tages (z. B. *Daily-Scrum*) eine lauffähige Software auf dem *Build-Server* zu haben. Für ein manuelles Anpassen der Modelle an geänderte Anforderungen bleibt oft keine oder nur wenig Zeit.

Dieser Artikel beschreibt einen Ansatz, um mit möglichst wenig manuellem Modellierungsaufwand und mit viel Automatisierung zu Modellen zu gelangen, die den aktuellen Entwicklungsstand repräsentieren und nach dem *Build* am *Build-Server* eine brauchbare System- und Projektdokumentation liefern.

Modellieren, aber was und wie viel?

Wenn von Modellen, vor allem von UML Modellen gesprochen wird, denken viele gleich an 13 Diagramme und sehr viel Aufwand, diese zu erstellen und aktuell zu halten. Nun, eine Devise beim Modellieren besagt: „so wenig wie möglich zu modellieren, aber so viel wie nötig“.

Ein Modell hat mindestens drei Dimensionen. Die erste Dimension beschreibt die Abstraktion des Modells, die zweite Dimension die Details eines Modells. Der wesentliche Unterschied zwischen den Dimensionen Abstraktion und Detail liegt darin, dass bei der Abstraktion mehrere konkrete Konzepte durch ein stellvertretendes Konzept beschrieben werden. Dagegen kann jedes Modell, egal ob abstrakt oder konkret, mit mehr oder weniger Details angereichert sein. Die dritte Dimension ist die Phase, für die das Modell erstellt wird. Grob wird in Analyse, Design und Implementierungsphase unterschieden. Die Phase korreliert meist mit

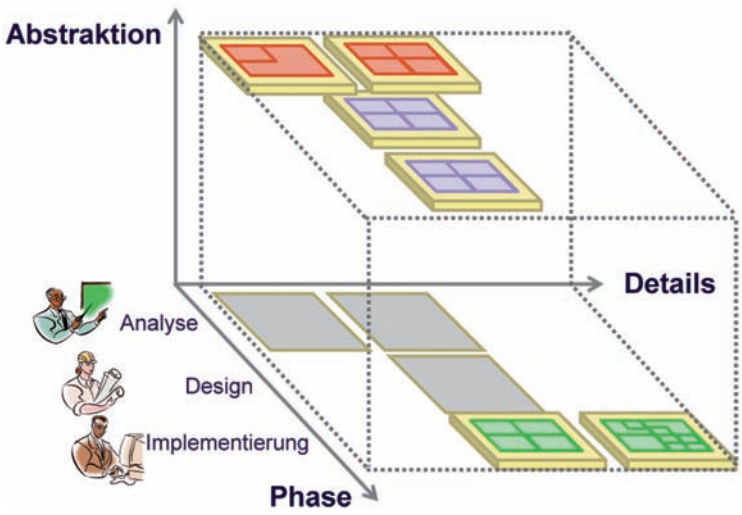

Abbildung 1: Modellierungswürfel

der ersten Dimension. Analysemodelle sind in der Regel abstrakter als Design- oder Implementierungsmodelle, können aber dennoch mit vielen Details angereichert sein! Als vierte Dimension sind die verschiedenen Modelle bzw. Modellierungssprachen (UML:ClassDiagramm, UML:ActivityDiagram, BPMN, etc.) zu erwähnen. Diese vierte Dimension führt zu mehreren Würfeln, wie in Abbildung 1 dargestellt. Ein Würfel steht nun für ein Modell (z. B. UML::ClassDiagram). Ein Modell entspricht genau einem Quadrat in diesem Würfel. Anfangs wird meist ein abstraktes, wenig detailliertes Modell erstellt (Quadrat links oben im Würfel). Dieses Modell kann nun erweitert werden, es wandert von links oben nach rechts unten im Würfel. Dies beschreibt die Evolution eines Modells. Alternativ können auch mehrere, meist verschieden abstrakte Modelle erstellt werden. In Abbildung 1 sind sechs Varianten abgebildet. Jedes Quadrat steht für eine Modellvariante.

Die beiden Dimensionen *Abstraktion* und *Details* erleichtern das Verständnis komplexer Systeme. Ein abstraktes Modell mit wenigen Details ist leicht und schnell erklärt und als Übersichtskarte sowohl für den Entwickler als auch für Projektverantwortliche von großem Nutzen. Durch Verfeinerung dieser Modelle kann durch mehr Details und durch konkretere Konzepte ein tiefes Verständnis für das Problem geschaffen werden. Durch Verlinkung der einzelnen Modelle (Quadrate) wird ein Tracen von den abstrakten Modellen bis zu den Implementierungsmodellen möglich.

Da Modelle durchaus nützlich sind und der Kunde zwar Dokumentation bekommen will, aber nicht gerne dafür zahlt, ist es notwendig zu entscheiden, welche Modelle in welcher Abstraktion und mit wie vielen Details erstellt werden sollen.

Je nach Projekt ist es sinnvoll, Richtlinien zu erstellen, um die Abstraktion und Details der Modelle festzulegen. Darüber hinaus kann jeder Projektbeteiligte „persönliche" Modelle erstellen, die als Skizze dienen (weitere Quadrate im Würfel). In beiden Fällen kann es leicht passieren, dass die erstellten Modelle „schnell" *out-of-date* sind. Ein mühevolles Nacharbeiten der Modelle ist dann notwendig, um den Gap zwischen Modell und Implementierung des Systems nicht zu groß werden zu lassen. Je abstrakter die Modelle sind, desto weniger schnell werden sie von Implementierungsentscheidungen überholt.

Da der Maßstab an dem ein Projekt gemessen wird, meist ausführbarer Code ist – nicht nur in agilen Projekten – und Entwickler viel Zeit vor ihrer bevorzugten Entwicklungsumgebung (IDE) verbringen, ist es sinnvoll, die im Code vorhandenen Informationen zu nutzen. Je nach Projekt und verwendeten Technologien findet man im Code eine Abbildung der abstrakteren Architekturentscheidungen, sicherlich aber die Implementierung der im Projekt definierten funktionalen Anforderungen. Zusätzlich kann, durch hinzufügen von Metainformationen als Kommentare, direkt beim Codieren auf ein abstrakteres Design verwiesen werden.

Das Modell im Code

Modellgetriebene Softwareentwicklung (MDE) [Schmidt] ist bereits ein etabliertes Thema. Das Ziel dabei ist es, aus möglichst abstrakten, *Platform Independent Models* (PIM) *Platform Specific Models* (PSM) zu transformieren und anschließend aus PSM den Code zu generieren. Natürlich ist auch die gegengesetzte Richtung möglich, also Modelle aus dem Code abzuleiten. Dabei erhält man allerdings in der Regel nicht so abstrakte Modelle, sondern eine 1:1 Abbildung des Codes als Modell (konkretes Implementierungsmodell mit vielen Details, siehe Abbildung 1).

Je nach verwendetem Tool können Struktur- und Verhaltensmodelle *forward* und *reverse engineered* werden. Nahezu jedes ernst zu nehmende Modellierungswerkzeug schafft es, zumindest aus Klassendiagrammen Code zu generieren und diesen mit dem Modell synchron zu halten. Als schwieriger erweist es sich, aus Verhaltensmodellen Code zu generieren und mit dem Modell synchron zu halten. Für diese Aufgabe sind UML Aktivitätsdiagramme und Sequenzdiagramme prädestiniert. Einige Tools unterstützen bereits das Generieren von Code aus Verhaltensmodellen und das nachträgliche Ableiten und Synchronhalten dieser Modelle.

Neben den Strukturmodellen (Klassendiagrammen) stecken oft noch mehr Informationen im Code. Wenn die verwendete Programmiersprache Namensräume unterstützt, können diese herangezogen werden, um automatisch Paketdiagramme generieren zu lassen. Für jeden Namensraum wird ein *UML:Package* angelegt und jede Klasse desselben Namensraums liegt in diesem Paket.

Hat nun eine Klasse aus *Package B* eine Beziehung zu einer Klasse aus *Package A*, ist auf der Paketebene das *Package B* von *Package A* abhängig. Das Paket ist ein Stellvertreter für alle enthaltenen Klassen und somit eine Abstraktion. Alle Beziehungen zwischen den Klassen führen zu Abhängigkeiten im abstrakteren Paketdiagramm und somit zu einer aggregierten Sicht auf das Implementierungs-Klassendiagramm. Das generierte Paketdiagramm ist ein abstraktes Modell der IST-Architektur.

Die in vielen Sprachen vorhandenen Annotationen bieten eine weitere

Fundgrube, die genützt werden kann, um Entsprechungen im aus dem Code abgeleiteten Modell zu erstellen.

Werden „eigene“ Annotationen (selbstdefinierte Metainformationen in Kommentaren) im Code verwendet, werden diese genützt, um gezielt weitere abstraktere Modelle aus dem Code zu generieren, bzw. diese mit bereits als Modell vorhandenen Elementen zu assoziieren.

Ein Kommentar bei jeder Klassendeklaration im Code kann z. B. die Zugehörigkeit einer Klasse zu einer Komponente beschreiben. Aus dieser Information kann später ein Komponentendiagramm generiert werden oder, wie in diesem Artikel beschrieben, auf vorhandene Komponenten verlinkt werden.

Der Namensraum bildet somit eine Gruppierung von frei wählbaren Diskriminatoren (z. B. technische/fachliche Sicht der Modelle). Eine Komponente ist die aggregierte und abstrakte Darstellung von Klassen (oder Sub-Komponenten) im fachlichen Kontext.

Manuelle und generierte Modelle zusammenwachsen lassen

Wie bereits erwähnt, sind Modelle in agilen Projekten keine Forderung, aber oft erwünscht. Sie reichen allerdings meist nur bis zu einer gewissen Abstraktion und sollten gerade angemessen viele Details aufweisen, um den Entwickler genügend Informationen an die Hand zu geben, eine zielführende Implementierung zu realisieren (Quadrat in der Mitte des Würfels). Zu viel Aufwand in Modelle zu stecken, ist oft nicht wirtschaftlich.

Ein sehr minimales Modell ist z. B. die Abbildung der Anforderungen, welche eine wesentliche Rolle in agilen Projekten spielen. Diese sind allerdings meist nur textuell oder in diversen Requirements Management (RM)-Tools vorhanden. Denkbar sind auch weitere Modelle wie Komponentendiagramme. Mit ihnen wird eine modularisierte abstrakte Sicht auf das zu realisierende System erstellt. Ressourcen und Verantwortungsbereiche können zugewiesen und gepflegt werden.

Das manuelle Synchronhalten von Modellen und Code ist zeitraubend. Um Zeit zu sparen und die Verantwortlichen bei dieser Aufgabe zu unterstützten, ist eine automatische Zusammenführung von manuell erstellten Analyse/Design Modellen und den aus dem Code *abgeleiteten* Implementierungsmodellen wünschenswert.

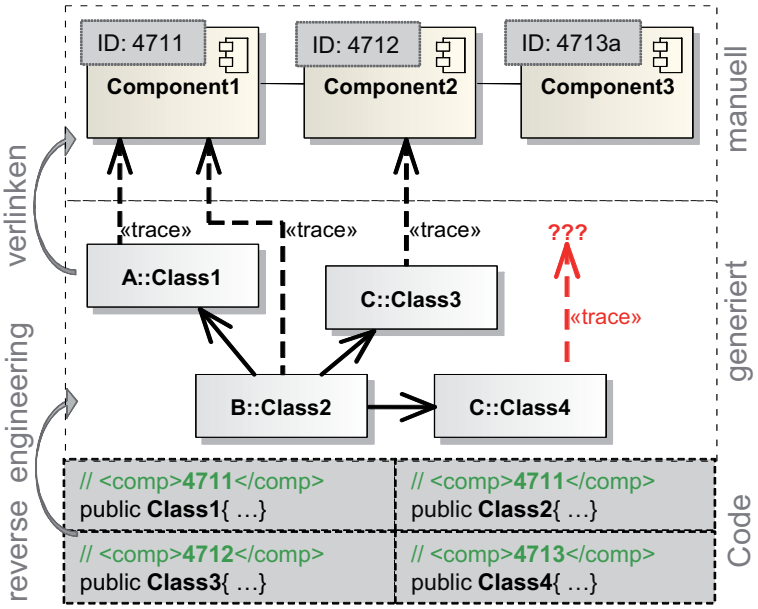

Abbildung 2: Verlinken von Modellen

Diese Zusammenführung basiert auf bestimmten Regeln. Zum Beispiel wird eine Abhängigkeitsbeziehung (oder *trace, abstraction,* etc.) zwischen abstrakten (manuellen) Modellen und automatisch abgeleiteten Modellen erstellt. Das Verlinken von manuellen und automatisch generierten Modellen wird durch ein *Post-Processing* des abgeleiteten Modells durchgeführt.

Dieses *Post-Processing* erstellt nach dem Ableiten der Modelle aus dem Code (siehe Abbildung 2) alle zusätzlichen abstrakten Modelle bzw. verlinkt sie zu bereits vorhandenen manuellen Modellen. Aus den Kommentaren im Code wird jede Klasse mit bereits vorhandenen, korrespondierenden Komponenten (manuelle Modelle) verlinkt. Die Korrespondenz wird mittels ID der manuellen Komponenten und Annotationen im Code erstellt. Das aus den Namensräumen generierte Paketdiagramm ist in Abbildung 3 ersichtlich.

Implikationen und Nutzen

Gibt es nun manuell erstellte Modelle ohne Link zu Modellen, die aus dem Code generierten wurden, ist dies ein Indiz dafür, dass das manuelle Modell und die Implementierung auseinander laufen und das manuelle Modell oder der Code angepasst werden müssen (siehe Abbildung 2). Jede Änderung am Modell führt zu einer Änderung seiner ID. In Abbildung 2 wurde die *Component3* verändert und ihre ID von 4713 auf 4713a geändert. Wird diese Änderung nicht im Code berücksichtigt (Anpassen der Annotation), führt dies beim Post-Processing zu einem fehlenden Link vom generierten zum manuellen Modell.

Ohne diesen Mechanismus müssen diese Korrespondenzen gedanklich durchgespielt werden oder sind schlicht unmöglich aufzulösen, da einfach die Informationen fehlen, welche Implementierung mit welchen abstrakten Modellen korrespondieren sollten.

Eine Sichtung des Gesamtmodells (manuell und generiert) dient als Architektur-Check und beantwortet Fragen wie: „Sind

Abbildung 3: Generiertes Paketdiagramm

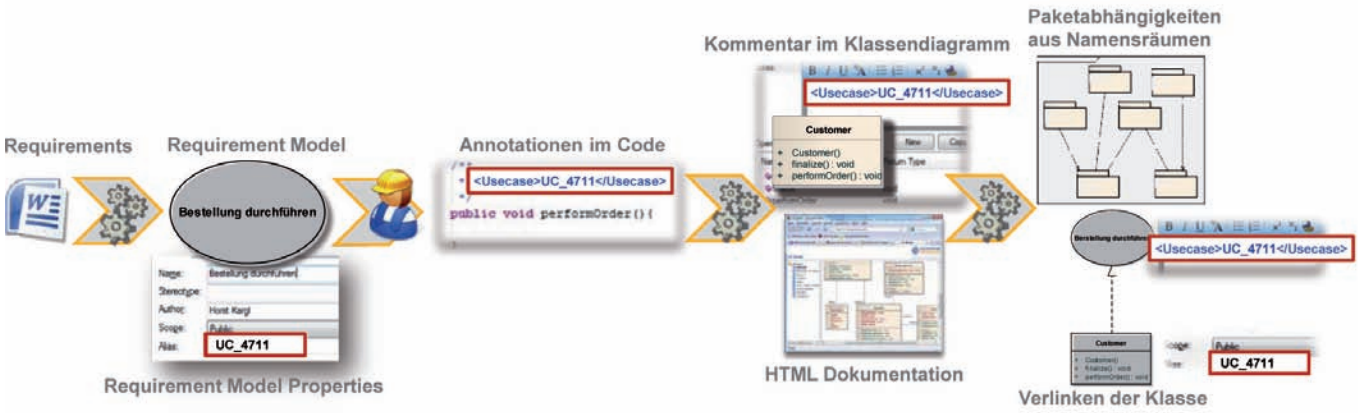

Abbildung 4: Automatisierungs-Workflow

meine Klassen in den richtigen Komponenten?“, „Entspricht der Code noch der modellierten Gesamtarchitektur?“, „Ist meine Architektur noch zielführend?“.

Durch ein manuell erstelltes Paketdiagramm mit Abhängigkeiten zwischen Paketen, kann eine SOLL Architektur beschrieben werden. Durch eingezeichnete Abhängigkeitsbeziehungen wird beschrieben, wie die einzelnen Klassen der unterschiedlichen Pakete verbunden sein dürfen. Durch ein durch die Referenzen zwischen den einzelnen Klassen automatisch generiertes Paketdiagramm mit den aktuellen Abhängigkeitsbeziehungen (Abbildung 3), wird die IST Architektur abgebildet. Durch einen SOLL/IST Vergleich können so Architekturfehler leicht erkannt werden.

Einige der von Robert Martin definierten Architekturprinzipien (*SOLID*-Prinzipien) [Robert C. M.] können ebenfalls in das UML Modell mit aufgenommen werden. Konkret sind es Heuristiken, welche in diversen Tools bereits vorhanden sind (Checkstyle [Check] Plugin für Eclipse, XDepend [XDep]). Durch Einfärben von Klassen, Interfaces und Assoziationen können Verletzungen des *Single-Responsibility-Principle* (Klasse mit zu vielen Assoziationen hat zu viele Zuständigkeiten), *Interface-Segregation-Principle* (Interfaces mit zu vielen Operationen) und *Dependency-Inversion-Principle* (zu enge Kopplung durch zu viele Konstruktor-Aufrufe) auch im Modell visualisiert werden. Eine grafische Visualisierung der Verletzung solcher Kennzahlen anhand von UML Diagrammen fördert Strategien zur Auflösung und Umorganisierung der Architektur.

Praxisbeispiel

Die Firma LieberLieber Software GmbH hat, basierend auf dem Modellierungswerkzeug Enterprise Architect von SparxSystems, einen vergleichbaren Ansatz realisiert. Als manuelle Modelle dienen Requirements, welche im RM-Tool (Word) verwaltet werden. Klassendiagramme und Paketdiagramme werden generiert. Die Zuordnung von Requirements zur konkreten Implementierung wird automatisch erstellt, wobei jedes Requirement direkt zur implementierenden Operation nachverfolgt werden kann.

Abbildung 4 beschreibt den generellen Workflow, Abbildung 5 das verwendete Tool-Setting. Es ist zu erkennen, dass es im Workflow lediglich einen manuellen Interaktionspunkt gibt: wenn der Entwickler die Anforderungen implementiert.

Die Requirements werden aus Word importiert (es ist auch möglich Excel-Listen zu importieren bzw. direkt RM-Tools zu integrieren). Nun stehen die Requirements in Enterprise Architect (EA) als Requirement Modell (Requirement oder Use Case) zur Verfügung. Im EA wird für jedes Requirement eine eigene ID generiert (siehe Requirements Model Properties, Abbildung 4). Will der Entwickler lieber mit einer Liste als Vorlage arbeiten, können alle Requirements als RTF Dokument ausgegeben werden. Beim Codieren muss nun lediglich der Entwickler zu den Operationen (oder Klassen) die Requirement-ID des dafür zuständigen Requirements (Use Case) als Kommentar annotieren (siehe Annotation im Code, Abbildung 4).

Der nächste Schritt passiert, wenn der Entwickler seinen Code *committet* (im Fall der Fa. LieberLieber in ein *SVN Repository*). Der Build Server (*Team City*) erstellt einen neuen *Build*. War der *Build* erfolgreich, ist dies der Trigger für das EA-Plugin, aus dem Code die Modelle abzuleiten und in ein EA-Projekt aufzunehmen,

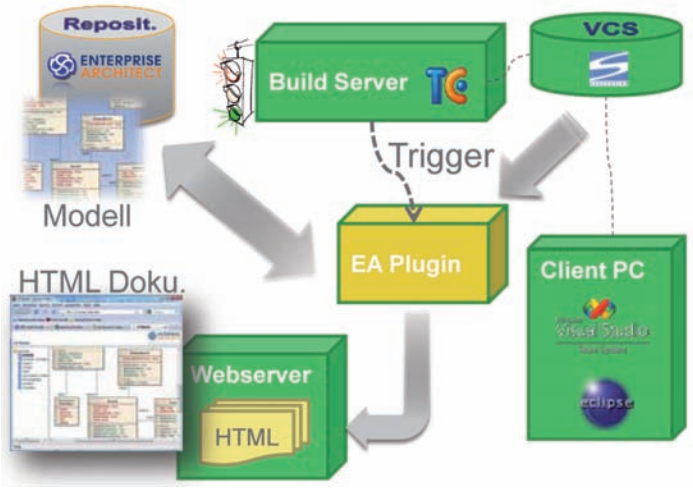

Abbildung 5: Tool-Setting

sowie das *Post-Processing* zu starten.

Beim *Post-Processing* der generierten Modelle wird ein Paketdiagramm mit deren Abhängigkeiten erstellt. Die im Modell zur Verfügung stehenden Requirement_IDs dienen beim *Post-Processing* der Zuordnung der im Modell vorhandenen Requirements zu den realisierenden Klassen (**Abbildung 4**, Verlinken der Klasse).

Anschließend wird der Reportgenerator von EA gestartet und eine HTML-Dokumentation auf dem internen Webserver abgelegt. Somit ist nach jedem *Commit*, der zu einem erfolgreichen *Build* führt, automatisch die UML-Dokumentation am internen Webserver verfügbar.

Durch die automatisch erstellten Diagramme und die durch Enterprise Architect zur Verfügung gestellten Features ist es nun ein leichtes, den Überblick über die Implementierung zu behalten, ohne dafür Zeit zum Modellieren bzw. Anpassen der Modelle zu benötigen. In diesem Ansatz werden aktuell noch keine Komponenten aus Annotationen generiert. Einer Erweiterung steht allerdings nichts im Wege.

Fazit

Wer wenig Zeit zum Modellieren hat, auf Modelle allerdings nicht verzichten möchte, ist mit dem beschriebenen Ansatz, seine Modelle zu generieren und mit manuell erstellten Modellen zu verlinken, auf dem richtigen „agilen" Weg. Die eingepflegten Automatismen nehmen viel lästige Arbeit ab und unterstützen die Analyse von Soll- und Ist-Architektur. Benutzerspezifische Kommentare (Annotationen) im Code bieten viele Möglichkeiten, abstraktere Modelle aus dem Code zu generieren bzw. mit schon vorhandenen abstrakten Modellen automatisch zu verlinken. Der beschriebene Ansatz und die gezeigte Tool-Kette unterstützt bei der Einhaltung von Design-Richtlinien und beim Auffinden von Regelbrüchen und erleichtert das Arbeiten in Agilen Projekten. Das automatische Erstellen der Implementierungsmodelle und deren Verlinkung mit manuellen Modellen, führt zu einem agilen Gesamtmodell. ■

Referenzen

[Robert C. M.] Robert C. Martin, „Agile Software Development", Prentice Hall

[Check] http://eclipse-cs.sourceforge.net/

[XDep] http://www.xdepend.com

[Sch02] K. Schwaber, M. Beedle, Agile Software Development with Scrum, Prentice Hall, 2002.

[Bec04] K. Beck eXtreme Programming explained – embrace change, Addison-Wesley, 2004.

[DeL08] J. DeLuca, Web-Seite zu Feature Driven Development, siehe: www.featuredrivendevelopment.com.

[Coc04] A. Cockburn, Cristal Clear: A Human-Powered Methodology for Small Teams, Addison-Wesley, 2004.

[Schmidt] Schmidt, D.C., "Model-Driven Engineering". IEEE Computer, Schmidt, February 2006 http://www.cs.wustl.edu/~schmidt/PDF/GEI.pdf