Modellgetriebene, Komponentbasierte Softwareentwicklung

Markus Völter, voelter@acm.org, www.voelter.de

- Teil 2 -

Komponentenbasierte Entwicklung und Modellgetriebene Entwicklung passen sehr gut zueinander – dies hat sich im Laufe der letzen paar Projekte in den verschiedensten Domänen gezeigt. In diesem zweiteiligen Artikel möchte ich darauf eingehen, wie man konkret Modellgetrieben und Komponentenbasiert entwickelt. Teil eins hat ein konkretes Metamodell vorgestellt, welches CBD aus den drei wichtigsten Perspektiven beleuchtet. Im zweiten Teil des Artikels möchte ich auf einige typische Variationen dieses Metamodells eingehen, darunter Asynchronität, Events, Layering, und dynamische Verdrahtung.

Einführung

Die Metamodelle, die im Teil eins vorgestellt wurden stellen einen Startpunkt dar, mit denen man in vielen Kontexten einen sinnvollen Start im Rahmen des Projektes hinlegen kann. Allerdings gibt es natürlich eine ganze Reihe von Varianten und Erweiterungen die man sich vorstellen kann, und dich auch in der einen oder anderen Form in diversen Projekten zum Einsatz kamen. Dieser Abschnitt diskutiert einige dieser Varianten. Man beachte: ich diskutiere hier **nicht** Varianten der Umsetzung, sondern Varianten des Metamodells.

Übrigens: die Diskussion dieser Punkte im Rahmen des Projektes führt zu einer ganzen Reihe von Entscheidungen was die Architektur angeht – damit spiele MDSD (hier: das Metamodell) die erwünschte Rolle des Architekturkatalysators!

Interfaces

Zunächst braucht man nicht in allen Projekten **separate Interfaces**. Alternativ kann man die Operationen auch direkt an die Komponente modellieren, statt einer *ComponentInterfaceDepdency* gibt es dann eine *ComponentComponentDependency*, die direkt auf eine andere Komponente zeigt. Dieses Vorgehen ist zunächt eine Vereinfachung, hat aber den Nachteil, dass man „Mengen von Operationen“ nicht separat wieder verwenden kann und es auch nicht möglich ist, dass mehrere Komponenten das gleiche Interface (unterschiedlich) implementieren.

Oft sind auch **Beziehungen zwischen Interfaces** notwendig, in der Art „Wenn ich dieses Interface verwenden möchte, muss ich folgendes auf jeden Fall anbieten“. Dies ist im

Metamodell recht einfach zu modellieren – man muss eben nur die entsprechenden Beziehungen zwischen den betreffenden Interfaces ausdrücken können.

Komponenten

Komponententypen bzw. **Layers** sind eine weitere wichtige architekturelle Eigenschaft. Dies ist deshalb so wichtig, weil sich basierend darauf eine Kontrolle der gültigen Abhängigkeiten implementieren lassen. Abbildung 1 zeigt ein Beispiel. Wenn man diese Typen definiert hat – und im Modell die betreffenden Typen entsprechend instantiiert – so kann man Constraints implementieren, die den mit <<*validDep*>> markierten Meta-Dependencies entsprechen.

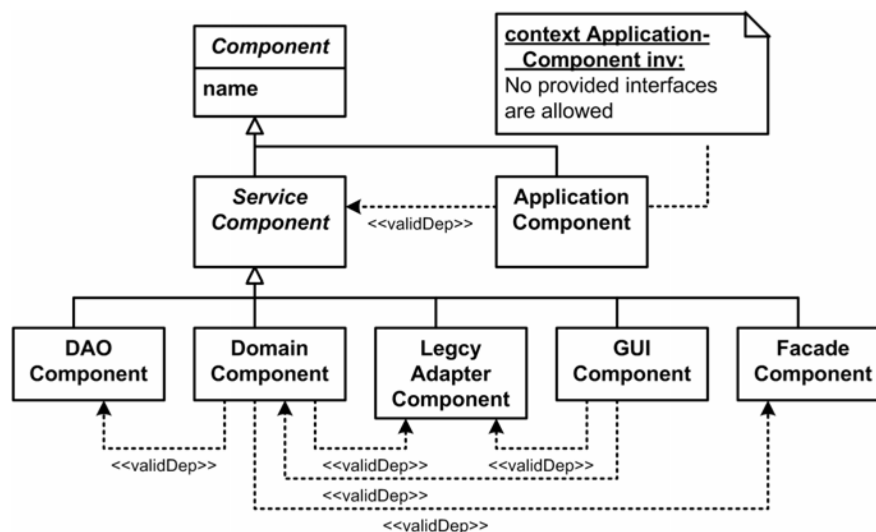

Abbildung 1: Komponententypen

Verwendet man – wie oben – explizite Interfaces, so kann man natürlich nichts über die Abhängigkeiten zwischen Komponententypen direkt sagen, da die Abhängigkeit ja zu einem Interface besteht. Hier muss man entweder auch die entsprechenden Interfacetypen mit den Layern annotieren, oder auf Composition-Ebene (wo die referenziert Komponente ja bekannt ist) überprüfen. Alternativ kann man auch einfach den Typ der Komponente und des Interfaces als Attribut definieren und per Constraintcheck überprüfen, dass nur gültige Beziehungen modelliert werden (siehe Abbildung 2). Eine übliche Constraint ist bspw. auch, dass Komponenten eines bestimmten Layers nur Interfaces eines bestimmten anderen Layers verwenden bzw. anbieten dürfen.

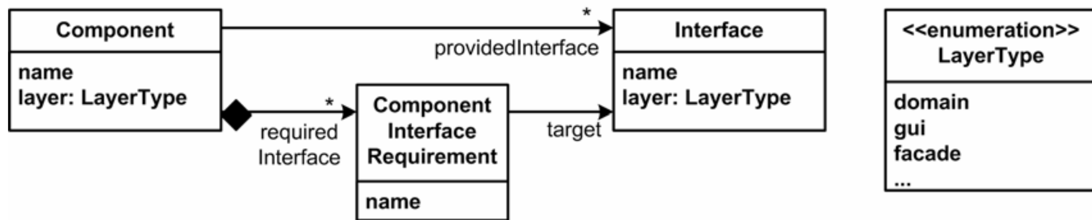

Abbildung 2: Layers

Übrigens reicht es natürlich nicht, im Modell Anhängigkeiten auf Korrektheit zu überprüfen – es muss zusätzlich sichergestellt werden, dass im Implementierungscode nicht weitere, ungewollte Abhängigkeiten erzeugt werden. Wenn das Programmiermodell allerdings sicherstellt, dass nur Abhängigkeiten zu im Modell definierten Artefakten „programmiert" werden können hat man allerdings viel gewonnen. Wie solche Programmiermodelle (oder APIs) aussehen ist allerdings jenseits dieses Artikels.

Ausbau des Typkonzepts

Ein interessanter – und nahe liegender – Punkt ist die **Vererbung zwischen Interfaces**. Dabei ist auch Mehrfachvererbung möglich. Definieren zwei Interfaces gleichnamige Operationen, so kann der Generator den Namenskonflikt erkennen und warnen.

Eine weitere interessante Variante besteht im **Ausbau des Typkonzepts** bei Komponenten. Es gibt derzeit keine Unterscheidung zwischen eine Komponente als „Spieler einer Rolle", der eine Menge von Interfaces anbietet und eine andere Menge verwendet, sowie der Imlementierung dieser Rolle.

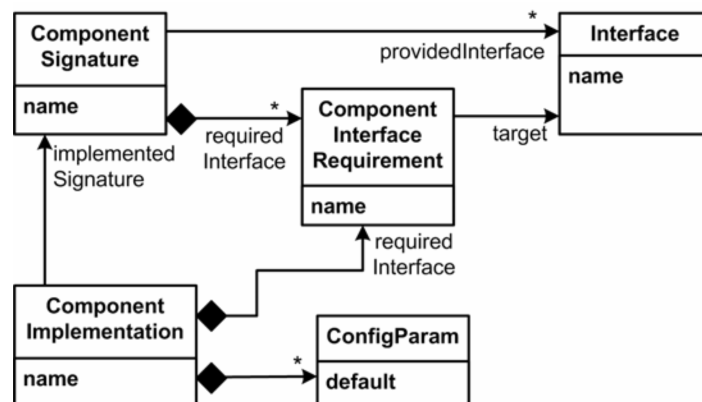

Abbildung 3: ComponentSignature vs. ComponentImplementation

Abbildung 3 soll die Idee illustrieren. Man definiert zunächst eine *ComponentSignature*, die angibt, wie sich eine Komponente „von außen" verhält, man definiert also angebotene und

benötigte Interfaces. Diese Signature hat *keine* Repräsentation in Code! Mehrere *ComponentImplementations* können diese Signature nun unterschiedlich implementieren – natürlich müssen sie dabei den Contract den die Signatur und die Interfaces definieren einhalten. Im hier gewählten Beispiel können Implementierungen weitere Interfaces benötigen; auch die Konfigurationsparameter werden erst hier definiert, was ja auch Sinn macht, da diese ja in aller Regel die Implementierung beeinflussen. Man kann in diesem Sinne noch weitere „Zwischenschichten" bei der Definition von Komponenten einführen, wenn nötig.

Eine sehr mächtige, wenn auch nicht ganz triviale Variante besteht darin, **hierarchische Komponenten** zu erlauben. Eine hierarchische Komponente ist eine, die selbst intern wieder aus Komponenten*instanzen* besteht, die miteinander verdrahtet sind. Dies ist deshalb nicht ganz einfach zu realisieren, da es die bisher klar gezogene Grenze zwischen Komponenten (im Typmodell) und Instanzen (im Compositionmodell) verwischt. Abbildung 4 zeigt das Metamodell.

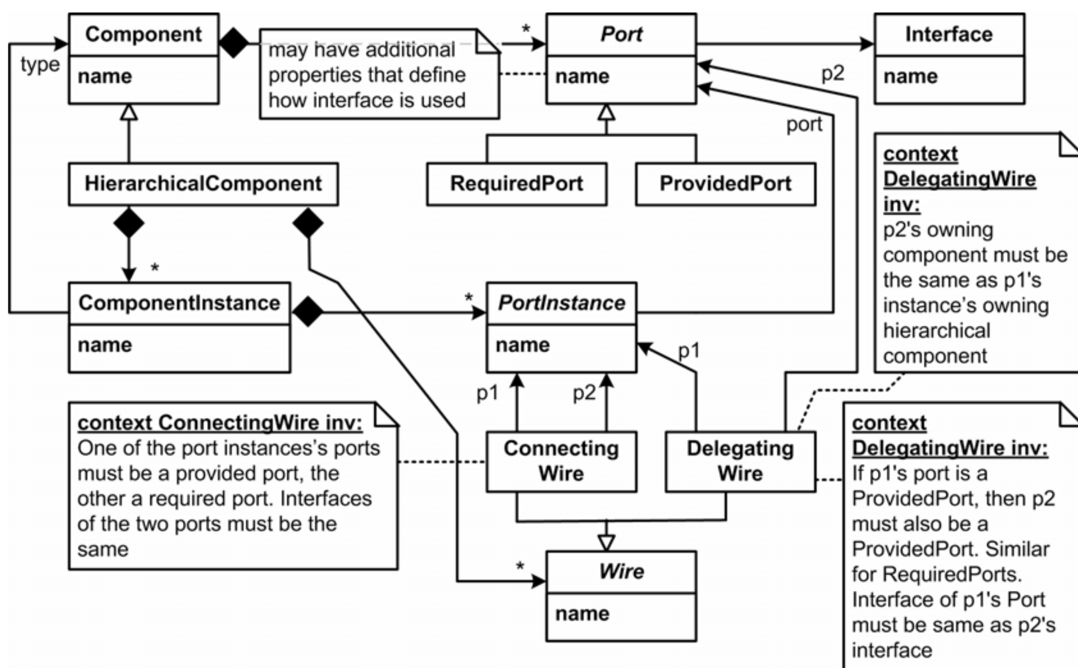

Abbildung 4: Metamodell für hierarchische Komponenten

Zur Verdeutlichung zeigt Abbildung 5 ein Beispielmodell, unter Verwendung der bekannten UML2 Notation. Die *HierachicalComponentA* enthält zwei Komponenteninstanzen, *B* und *C*. Dabei werden die beiden Ports von B und C über einen *ConnectingWire* verbunden. Der andere Port von C sowie der Port *aPort* von *A* werden über einen *DelegatingWire* verbunden – *A* delegiert die Funktionalität, die *aPort* anbietet, an die interne Instanz *C*. *aPort* und *bPort* sind über einen *ConnectingWire* verbunden.

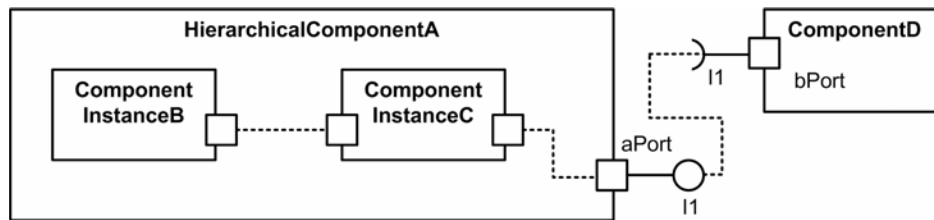

Abbildung 5: Beispielmodell (NICHT Metamodell) für Hierarchische Komponenten

Man beachte dass es falsch wäre, Komponenten (und nicht Instanzen) innerhalb einer Hierarchischen Komponente zu beschreiben. Man stelle sich eine Hierarchische Komponente *ABSSystem* vor. Es besteht aus einer Reihe von Sensorkomponenten, einer „Hauptkomponente" die die Algorithmik enthält sowie eine Radsteuerungskomponente *für jedes Rad*. Die Radsteuerungskomponenten sind die gleichen, jedoch verwendet das *ABSSystem* vier *Instanzen* davon und verdrahtet diese entsprechend mit den Sensoren und der Hauptkomponente.

Eine weitere wichtige Sache sind **Konfigurationsparameter**. Wie bei Kommandozeilenargumenten bei Anwendungen will man eine Komponente ggfs. auch mit Parametern versorgen um eine gewissen Laufzeitflexibilität zu erreichen. Man muss bei diesem Aspekt zwei Dinge berücksichtigen. Die Definition, welche Parameter eine Komponente erwartet muss im Rahmen der Typdefinition festgelegt werden. Die Definition des Wertes für den Parameter kann zu verschiedenen Zeitpunkten statt finden:

- als Default im Typ
- im Rahmen des Composition-Modells statisch
- oder im Rahmen einer Konfigurationsdatei, die dann zur Ladezeit des Systems gelesen wird.

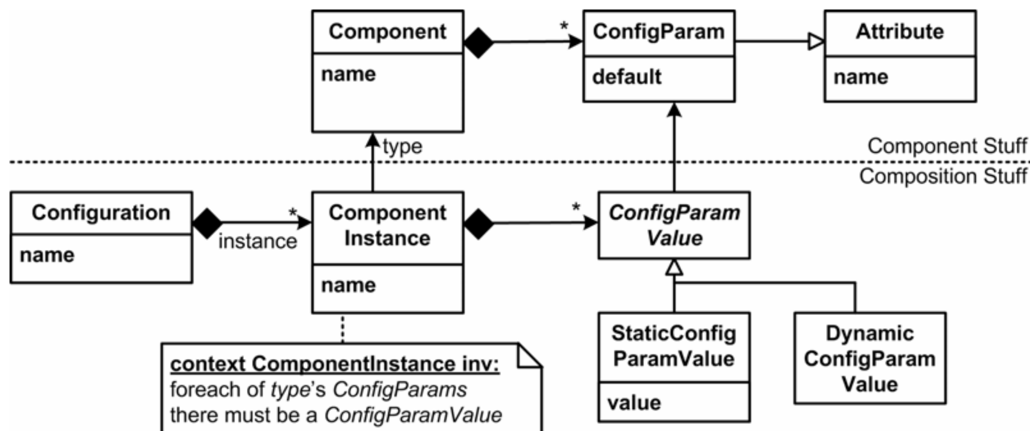

Abbildung 6: Konfigurationsparameter

Abbildung 6 zeigt ein meist angemessenes Metamodell für Konfigurationsparameter. Eine weitere, noch flexiblere Ausbaustufe besteht darin, Konfigurationsparametersätze als eigenständige Modellelemente zu definieren. Komponenten können diese referenzieren, wenn Sie auf die betreffenden Parameter Zugriff haben wollen. Das charmante an dieser Variante ist, dass man sehr flexibel Werte für die Parameter definieren kann:

1. Default, der im Typmodell bei der Definition des Parameters im ParameterSet gesetzt ist
2. Default, der im Typmodell für eine das Set benützende Komponente gesetzt ist
3. Wert, der im Config-File für das ParameterSet gesetzt ist (gilt dann für alle Instanzen aller Typen die das Set referenzieren)
4. Wert, der im Config-File für die Component gesetzt ist (gilt dann für alle Instanzen der Komponente)
5. Wert, der im Config-File für eine Instanz gesetzt ist

Komponenten-Verhalten

Eine ganze Reihe von Dingen ist bei der Implementierung der Komponenten zu beachten (Abbildung 7). Da wäre zunächst der **Lifecycle** der Komponente zu nennen. Der Lifecycle einer Komponente beschreibt, welche Phasen sie im Rahmen ihres Lebens durchläuft. Typischerweise sind dies Dinge wie *initializing*, *active*, *passivated* oder *shuttingDown*. Im Allgemeinen werden solche Lebenszyklen durch eine Zustandsmaschine beschrieben. Der Container managt der Lebenszyklus und erlaubt der Komponente (per Callback) auf Übergänge zwischen diesen Zuständen zu reagieren. Da es sich beim Lebenszyklus einer Komponente um etwas inhärent dynamisches handelt, kann man im Metamodell nicht allzu viel dazu sagen; außer, dass man an der Komponente annotiert, welche

Lebenszyklus-Statemachine die Komponente haben soll – man wird typischerweise eine Reihe solcher vordefinieren.

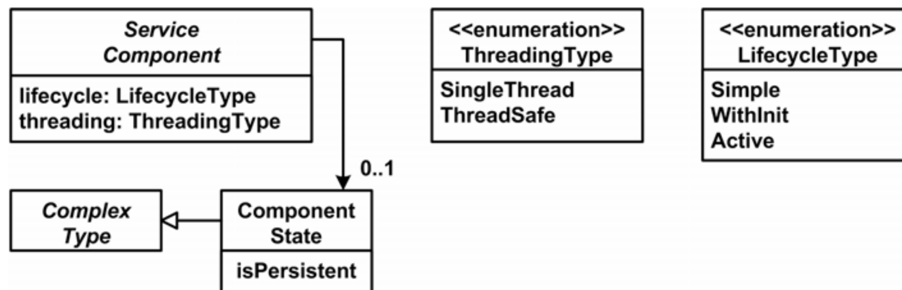

Abbildung 7: Lifecycle, State und Threading

Eine Komponente muss auch definieren, ob sie einen **Zustand** haben soll oder nicht. Wenn ja, muss dieser ggfs. vom Container entsprechend sinnvoll (ggfs. persistent) verwaltet werden.

Weiterhin ist für eine Komponente zu entscheiden, ob sie **threadsafe** sein soll; das bedeutet, dass zur gleichen Zeit mehrere Threads in der Implementierung einer Komponente vorkommen können. Der Entwickler muss dann bei Zugriff auf den Zustand der Komponente (sofern die Komponente welchen hat) entsprechende Synchronisationen einbauen. Die Information ist deshalb im Modell wichtig, weil daraus der die Konfiguration des Containers im Umgang mit der Komponente abgeleitet werden kann.

Wie kommen langsam in die Verlegenheit, recht viele Dinge bei der Charakterisierung einer Komponente definieren zu müssen. Die Komponententypen oben (DomainComponent, DAOComponent, etc.), Lifecycle, Zustand, Threading. Glücklicherweise machen nicht alle Kombinationen dieser Eigenschaften Sinn. Es ist daher sinnvoll, eine begrenzte Menge von Kombinationen zu definieren und denen wiederum „Namen" zu geben; EJB und CCM machen dies beispielsweise. Oft lassen sich auch Komponententypen wie sie oben eingeführt werden mit einer Menge solcher Attribute vorbelegen. Beispiele wären:

- *DAOComponents* haben einen Lifecycle mit Initialisierung (wegen der Acquisation der Datenbankverbindung), sind Stateless und Threadsafe (ist ja kein Problem, redet eh nur mit der Datenbank).
- *DomainComponents* haben einen Lifecycle ohne Initialisierung, sind auch Stateless sind aber nicht threadsafe (da damit die von Hand zu schreibende Implementierung einfacher wird) – der Container muss ggfs. poolen.

Eine besonders interessante Art von Komponenten sind **Aktive Komponenten** – man kann dies als einen speziellen Lifecycle sehen. Solche Komponenten haben eine Art „main" Methode, die entweder

- eine *while(true)* Schleife hat um ewig zu laufen (dann muss der Container der Komponente einen eigenen Thread zur Verfügung stellen), oder
- vom Container zyklisch aktivier wird.

Auf jeden Fall erreicht man mit diesem Mechanismus, dass Komponenten aktiv sein können, und nicht nur Aufrufe von anderen Komponenten entgegen nehmen können.

Asynchronität und Events

In der vergangenenen Diskussion sind wir immer implizit davon ausgegangen, dass die Kommunikation zwischen den Komponenten mittels synchroner Methodenaufrufe geschieht. Manchmal benötigt man allerdings auch **Asynchronität**! Es gibt diverse Patterns, wie Asynchrone Kommunikation sinnvoll durchgeführt werden kann [VKZ04]. Wichtig ist es dabei, dass die Frage, ob asynchron kommuniziert werden soll Auswirkungen auf die Aufruf-API hat. Es muss also bereits wenn die Komponente implementiert wird klar sein, welche Art der Kommunikation verwendet werden soll. Ergo: Eine entsprechende Markierung muss im Typmodell geschehen (siehe Abbildung 8).

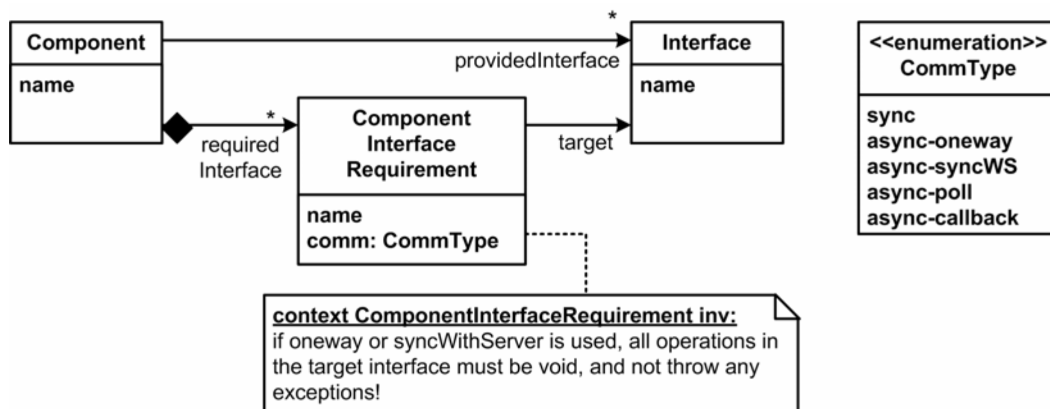

Abbildung 8: Asynchrone Kommunikation

Wichtig ist es, dass das Interface an sich nicht davon betroffen ist, ob es synchron oder asynchron verwendet wird. Auch die implementierende Komponente weiß nichts davon. Lediglich der Client muss das Interface richtig verwenden, es muss also ggfs. ein Proxy generiert werden, der die betreffende API anbietet. Eine Constraint gibt es allerdings noch im Modell zu validieren: *oneway* und *syncWithServer* kann nur für *void*-Operationen die keine Exceptions werfen realisiert werden. Wird dieser Modus also für ein Interface verwendet, welches dieser Bedingung nicht gehorcht, so muss ein Fehler reportet werden.

Neben asynchroner Kommunikation gibt es auch die Problematik der Notifikation durch **Events**. Nun sind Events ein potentiell großes Thema – man schaue sich nur Standards wie den CORBA Event/Notification Service an. Wir wollen es hier sicherlich nicht vollständig abhandeln ☺. Was man in der Praxis aber schon oft braucht, ist die Möglichkeit, einfache Notifikationen zwischen Komponenten auszutauschen, und zwar auch und gerade, um den „offiziell erlaubten" Depdendency-Richtungen entgegen zu laufen. Abbildung 9 zeigt das Metamodell.

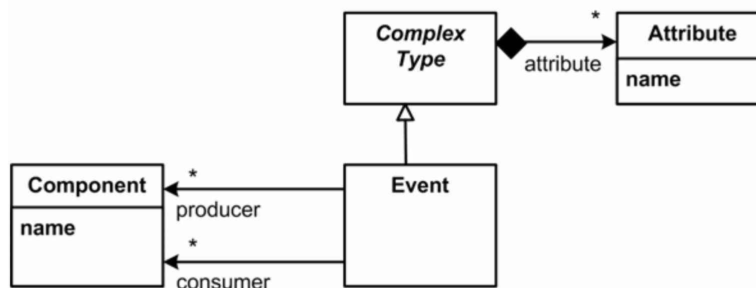

Abbildung 9: Events

Dies bedarf vermutlich etwas Erläuterung. Was dieses Metamodell besagt, ist, dass man im Modell Events definieren kann, die eine Menge von Attributen haben. Außerdem muss angegeben werden, wer dieses Event erzeugen, und wer es empfangen soll (daraus können dann entspr. APIs und Verteilungsinfrastrukturen generiert werden). Was hier im Metamodell nicht notiert wird (weil es für alle Events gilt und daher nicht modelliert werden sollte) ist, dass man jedes Event nach dem tatsächlichen Producer fragen kann – also auf Instanzebene! Die Verdrahtung der Eventproduzenten und -empfänger passiert nicht dynamisch (Observer oder so), sondern eben durch die Definitionen im Modell.

Weitere Strukturierungsmöglichkeiten

Eine weitere Geschichte die man oft benötigt sind weitere Strukturierungsmöglichkeiten, neben den reinen Komponenten. Insbesondere die Definition von **Subsystemen** oder sog. **Business Components** wird oft nötig. In beiden Fällen definiert man eine neue Abstraktion, die den Zweck hat, Mengen von Komponenten organistorisch zusammen zu schließen. Ein Subsystem ist damit einfach eine (flache) Menge von Komponenten, Interfaces und Datentypen. Business Components sind im Prinzip dasselbe, haben aber eine innere Struktur. Beispielsweise könnte man definieren, dass jede Business Component mindestens eine *FacadeComponent* besitzen muss; der Zugriff auf diese Business Component darf dann nur über diese Facade geschehen (siehe Abbildung 10)

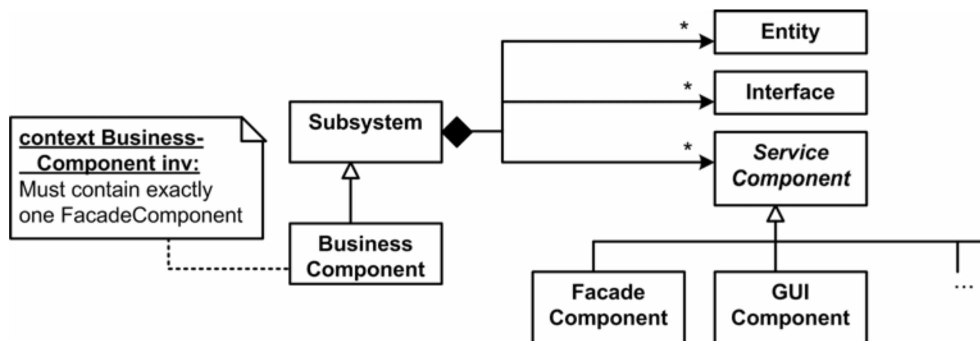

Abbildung 10: Subsysteme und Business Components

Alternativ könnte man prinzipiell auch der BusinessComponent direkt ein Interface zuordnen – das „öffentliche" Interface eben dieser ganzen BusinessComponent. Man beachte dabei aber, dass man nun aus der BusinessComponent als Mittel zur Strututierung des Systems ein laufzeitrelevantes Artefakt macht – man muss nun auf einmal Instanzen von BusinessComponents deployen können! Dies macht alles erheblich schwieriger und man ist dann schon fast wieder bei Hierarchischen Komponenten (siehe oben). Die Sache bleibt einfacher, wenn man nur die „normalen" Komponenten laufzeitrelevant sein lässt.

Daten

Eine ganze Reihe von Dingen kann man bei den Datenstrukturen machen (Abbildung 11). Zunächst ist es oft sinnvoll zwischen **Entitäten und Dependent Types** zu unterscheiden. Entitäten sind Daten, die eine eigene Identität haben (später dann auch eigene DAO-Komponenten bekommen, s.u.). Dependent Types haben keine eigene Identität und werden immer von einer Entität besessen, deren DAO die Dependent Types auch automatisch mit verwaltet. Übrigens ist es sinnvoll, die **DAO Komponenten** (Untermetatyp *DAOComponent*) automatisch durch Modelltransformationen und Generierung zu erzeugen, siehe [VM05c]. Alle Komponenten die eine Entity benützen, bekommen automatisch eine *ComponentInterfaceDependency* auf das Interface der generierten *DAOComponent*, somit können sie die entspr. Entitäten auch laden/speichern/löschen.

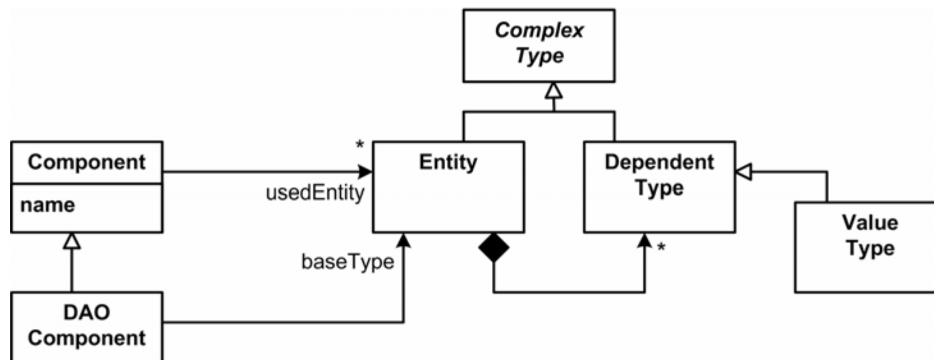

Abbildung 11: Erweitertes Datenmodell

Eine interessante Frage in großen Systemen, ist, wer Datentypen verwenden darf. Auch Datentypen, die durch das ganze System verteilt verwendet werden bedeuten Abhängigkeiten. Diese haben zur Folge, dass die Datentypen zum einen inhaltlich mit dem ganzen System angestimmt werden müssen; zusätzlich ergeben sich insbes. auch beim Deployment Probleme, der Code für die Datentypen muss ja überall wo der Typ verwendet wird, vorhanden sein – was wiederum bei Änderungen viel Neubauen und – Deployen zur Folge hat! Insofern ist es wichtig, den **Besitz und Scope von Datentypen** zu definieren. Defaultmäßig kann jede Komponente jeden Datentyp verwenden. Welche Komponente nun welchen typ *tatsächlich* verwendet (und ihn damit im entspr. Deployment-Package benötigt oder auf das Datentyp-Deployment-Package Zugriff haben muss) kann leicht anhand der Methodensignaturen der referenzierten Interfaces herausbekommen werden. Werden Subsysteme oder Business Components verwendet, kann man die Verwendung eines Datentyps auch leicht auf ein Subsystem oder eine Business Component beschränken – auch eine Zuordnung zu Layern oder Komponententypen bzw. bestimmtem Komponenten leicht möglich. Wie auch immer man es macht, es ist sichergestellt, dass man – wenn nötig – Kontrolle über die Datentypen und die damit erzeugten Abhängigkeiten erhalten kann. Insbesondere für das Deployment ist dies, wie gesagt, essentiell.

Verdrahtung

Eine weitere interessante Variante ist die Frage, wie die Verdrahtung zwischen den Komponenteninstanzen hergestellt wird. Standardmäßig wird als Ziel einer eine direkte Referenz einer anderen Instanz angegeben und es sind für jede *ComponentInterfaceRequirement* ein *Wire* nötig. An dieser Stelle gibt es zwei Dinge zu variieren.

Zum einen können *ComponentInterfaceRequirements* **optional** sein. Das bedeutet, dass man in Metamodell der Klasse *ComponentInterfaceRequirement* ein boolean-Attribut *isOptional* gibt. Wenn dies im Modell auf *true* gesetzt ist, ist es möglich, dass für das betreffende

Requirement kein *Wire* definiert wird – und der Anwendungsentwickler muss im Implementierungscode damit rechnen, dass die Referenz *null* ist.

Der andere interessante Aspekt betrifft **dynamische Verdrahtung**. Bisher sind wir davon ausgegangen, dass die Verdrahtung von Komponenteninstanzen statisch geschieht, also als Teil der Systemspezifikation in einem Modell fest vorgegeben kann. Damit sind keine Laufzeitvarianten mehr möglich. Dies ist für viele Fälle allerdings zu rigide. Es ist oft nötig, die Verdrahtung zur Laufzeit zu ändern (bspw. im Falle von Failover), mehrere Möglichkeiten anzugeben (zwecks Lastverteilung) oder komplett dynamisch zur Laufzeit mittels eines Directory Services ein passendes Ziel für einen *Wire* zu finden. Wie aus der Diskussion klar wird, handelt es sich dabei um etwas, was zur Laufzeit stattfindet (auch die statische Verdrahtung findet, genau genommen, zur Laufzeit statt). Man muss nur dafür sorgen, dass (a) zur Laufzeit die betreffenden Informationen vorhanden sind, und (b) ein entspr. Algorithmus zur Laufzeit die Verdrahtung durchführt. Für letzteres kann man davon ausgehen, dass die Plattform eine entspr. Verdrahtungsmöglichkeit zur Verfügung stellt (eben bspw. durch Verwendung eines Directories oder durch Discovery in P2P Systemen). Was die Informationen angeht, so muss man eben passende zur Verfügung stellen – das Metamodell muss dadurch mittels entspr. Attribute erweitert werden. Fall a) in Abbildung 12 zeigt wie man zwei Ziele (eins als Backup) angeben kann, Fall b) zeigt, wie man spezifiziert, dass die gesuchte Komponente ein bestimmtes Interface anbieten muss und wie man weiterhin ein Query spezifiziert, welches dann vor Directory ausgewertete wird um eine passende Instanz zu finden.

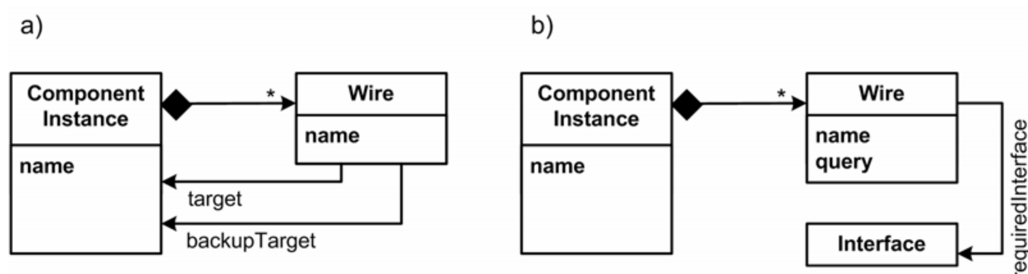

Abbildung 12: Dynamische Verdrahtung

Manchmal ist es auch notwending, bestimmte Verdrahtungen nicht vornehmen zu müssen – also **automatische Verdrahtung** zu ermöglichen. Beispiel: angenommen, wie bauen ein Embedded-System wo bestimmte Komponenten von Zeit zu Zeit von einem Timer aktiviert werden können. Nun ist es ein gängiges Pattern, Dienst wie einen Timer selbst als Komponente auf der Middleware-Schicht zu implementieren. Man braucht also eine Verdrahtung des Timers mit der zu aktivierenden Komponente. Nun will man das – eben gerade für solche Infrasturkturdinge wie z.B. einen Timer – nicht jedes Mal explizit verdrahten. In einem solchen Fall bietet es sich an, ein Interface als *autowire* zu markieren. Wenn eine Komponente dieses Interface verwendet, sucht der Generator für jede Instanz,

die das Interface verwendet nach einer passenden Instanz, die das Interface anbietet und verdrahtet diese. Fortgeschrittenere Generatoren können damit dann automatisch ableiten, welche „Infrastrukturkomponenten“ (ein Komponententyp, s.o.) zusätzlich deployt werden müssen, um das System lauffähig zu machen. Übrigens ist es auch hier sinnvoll, das Konzept der Ports zu verwenden – weil man in einem Fall wie dem Timer oft einen Timer mit *vielen* zu aktivierenden Komponenten verbinden muss. Diese Kardinalitäten lassen sich gut am Port annotieren (1:1-Port, 1:n-Port, etc).

Mehr Hardware

Eine ganze Reihe von sinnvollen Erweiterungen betrifft den Systemaspekt. Da wären zunächst mal **Container/Knotentypen**. Oft besteht ein System aus verschiedenen Knotentypen (z.B. Client auf Eclipse, Server auf Spring). Durch Angabe dieses Sachverhalts im Modell können wieder bestimmte Dinge gecheckt werden, bspw. dass *DAOComponents*, die für die Persistenz zuständig sind, nur auf dem Server liegen dürfen. Selbstverständlich kann auch die Codegenerierung vom Knotentyp abhängig sein. Will man Komponenten auf einer Eclipse Runtime laufen lassen, so wird man anderen Code „um die Komponentenimplementierung herum“ generieren müssen als bei Spring (man beachte: wenn man’s richtig macht, ist der Implementierungscode nicht anzupassen!).

Ein weiterer wichtiger Aspekt sind **Netzwerke bzw. Busse**. Aktuell gehen wir davon aus, dass, wenn Instanzen auf verschiedenen Knoten/Containern liegen, ein Remoting-Mechanismus Verwendung findet, um trotzdem zu kommunizieren. Leider ist das oft nicht so einfach… Manchmal gibt’s zwischen zwei Knoten gar kein Netzwerk, bzw. es gibt nur ein Netzwerk mit bestimmtem Restriktionen (Bandbreite, Timing, …) sodass nicht jede Kommunikation darüber ablaufen kann (bzw. zumindest für das Netzwerk passender Remoting-Code generiert werden muss). Auch diesen Aspekt sollte man im Metamodell unterbringen. Siehe Abbildung 13.

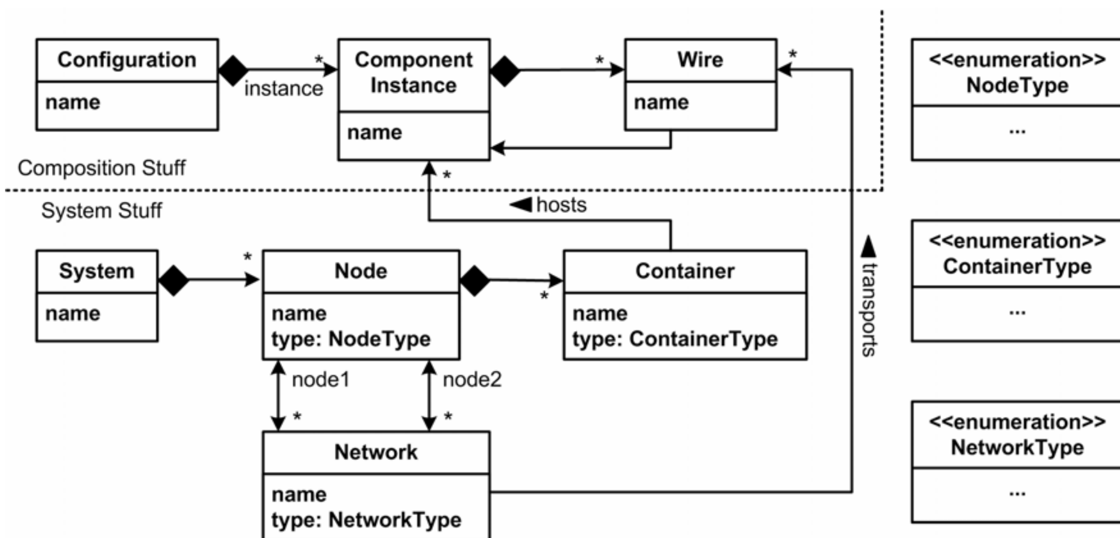

Abbildung 13: Containertypen und Netzwerke

Man beachte, dass in diesem Metamodell ein Netzwerk weiß, welche *Wires* es „transportiert“ und nicht anders herum. Dies ist wieder der Abhängigkeitshierarchie zwischen den Viewpoints geschuldet (siehe **Fehler! Verweisquelle konnte nicht gefunden werden.**). Für den Lokalfall (Testumgebung) läuft ja sowieso alles auf einem Knoten/Container, und Netzwerke spielen keine Rolle.

Versionierung

Als ein letzter Punkt sei noch die **Versionierung** angesprochen. Ein System entwickelt sich ja in aller Regel weiter – auch nach der ersten Auslieferung. Man wird also Komponenten durch neue – buggefixte oder anderweitig modifzierte – ersetzen. Bei so einer Ersetzung ist es wichtig zu wissen, welche Teile des Systems zusammenbrechen, wenn man eine bestimmte Komponente ersetzt. Könnte man Aussagen darüber machen, dann wäre das ersetzen einer Komponente mit erheblich weniger Risiko verbunden. Besonders relevant ist diese Frage natürlich vor allem bei Systemen mit dynamischer Verdrahtung, weil eben *keine* statische Prüfung möglich ist, die uns sagt ob das System noch lauffähig ist.

Die Idee ist es nun, im Modell Angaben darüber zu machen in welchem Verhältnis Interfaces, Komponenten oder Entitäten zueinander stehen. Daraus können dann Constraints abgeleitet werden, die sicherstellen, dass das System noch funktioniert, wenn entsprechende Artefakte getauscht werden. Das Metamodell sieht sehr einfach aus – siehe Abbildung 14.

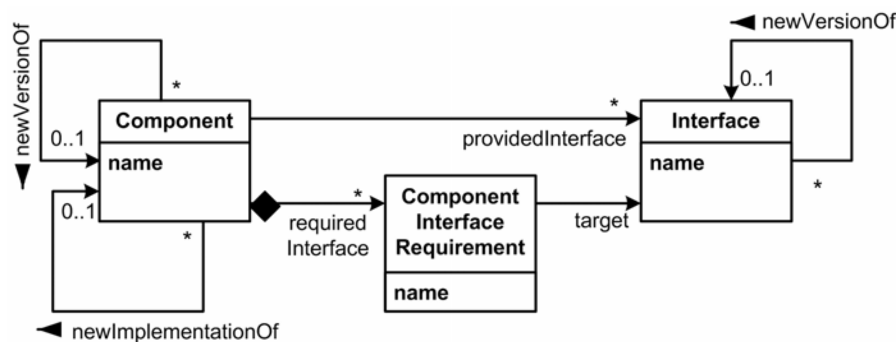

Abbildung 14: Versionierung

Man kann nun die folgenden Constraints definieren:

- Wenn ein *InterfaceB* deklariert, es sei eine *newVersionOf InterfaceA,* dann erbt es automatisch alle Operationen von *InterfaceA* und darf maximal noch weitere dazu nehmen. Damit ist Subtypkompatibilität gewahrt, und überall, wo bisher *InterfaceA* zum Einsatz kam, kann nun gefahrlos *InterfaceB* verwendet werden. Um die Deployment-Kompatibilität zu wahren, kann man des Weiteren fordern, dass keine zusätzlichen komplexen Datentypen als Argumente verwendet werden.

- Definiert eine *ComponentB* sie sei eine *newImplementationOf ComponentA*, dann kann *ComponentB* keine eigenen Interface-Dependencies definieren, da sie automatisch alle Interfaces-Dependencies von *ComponentA* erbt. Es ist damit lediglich möglich eine neue Implementierung – üblicherweise zwecks Bugfixing – zu liefern. Damit kann *ComponentB* überall dort eingesetzt werden, wo bisher *ComponentA* verwendet wurde.

- Schlussendlich erfordert die Beziehung *ComponentB newVersionOf ComponentA*, dass *ComponentB* die Interfaces von *ComponentA* anbietet (oder neue Versionen vorhandener dieser), und zusätzlich Interfaces. Verwendet werden dürfen aber keine zusätzlichen Interfaces. Das bedeutet, dass die *ComponentB* in der gleichen *Umgebung* lauffähig ist wie *ComponentA*, aber zusätzliche Dienste anbietet.

Die nächste Stufe wäre, dass eine *ComponentB* auch zusätzliche Interfaces required. Damit ist aber keine Kompatibilitätsaussage mehr vorhanden, weswegen *ComponentB* dann einfach eine neue Komponente ist. Eine spezielle Markierung ist nicht mehr sinnvoll.

Natürlich beziehen sich die hier gemachten Kompatibilitätsaussagen nur auf die im Modell überprüfbaren Aspekte einer Komponente – und das umfasst standardmäßig nun eben gerade *nicht* deren Implementierung. Das bedeutet, dass Fehler oder Inkompatibilitäten die von einer anderen Implementierung herrühren mit diesem Mechanismus nicht erkannt werden. Ein interessanter Ansatz um dieses Problem (ein Stück weit) in den Griff zu bekommen sind Pre- und Postconditions, die man mit einem *Interface* assoziieren kann. Die

Infrastruktur – bzw. generierte Proxies – stellen sicher, dass zur Laufzeit die entspr. Constraints überprüft werden. Wichtig sind dabei zwei Dinge:

- Man muss sicherstellen, dass die Constraints tatsächlich mit dem *Interface* assoziiert sind. Wenn also mehrere Komponenten das gleiche Interface implementieren, so müssen für die Komponenten auch die gleichen Constraints geprüft werden.
- Bei Vererbung von Interfaces bzw. bei oben geschilderter Versionsevolution muss definiert werden, in welcher Form sich die Constraints verändern. Das Liskov-Substitution-Principle fordert, das Preconditions verodert werden und die Postconditions verundet.

Übrigens haben wie diesen Ansatz – Versionierung inkl. Constraints – bereits in mehreren Projekten umgesetzt.

Wiederverwendung des Metamodells und der Varianten?

Meine Behauptung ganz zu Anfang des ersten Artikels war ja, dass dieses Metamodell – mit der einen oder anderen oben beschriebenen Variation – in quasi jedem Projekt zum Einsatz kommt; zumindest als Startpunkt. Es wäre also schön, wenn man das Metamodell direkt verwenden könnte, beispielsweise in einer für den openArchitectureWare Generator passenden Form. Es stellt sich allerdings dann die Frage, wie man die Variationen die oben diskutiert wurden zum Ausdruck bringt. Eine Lösung dieses Problems besteht darin, den auf Featuremodellen basierenden Ansatz zur Variantenbildung von Modellen (beschrieben in [MV05]) auf Ebene des Metamodells anzuwenden. Technisch spricht nichts dagegen. Müsste man eigentlich mal machen. Freiwillige vor ☺.

Vielen Dank an Eberhard Wolff für das Feedback zu dem Artikel.

Referenzen

[MV05] Markus Völter, Variantenmanagement im Kontext von MDSD, http://www.voelter.de/data/articles/MDSDAndVariants.pdf