Standardschnittstellen als nichtfunktionale Variationspunkte: Erfahrungen aus der EPM-Produktlinie

Martin Frenzel, Jörg Friebe, Simon Giesecke, Till Luhmann

BTC Business Technology Consulting AG
Escherweg 5
26121 Oldenburg
{martin.frenzel, joerg.friebe, simon.giesecke, till.luhmann}@btc-ag.com

Abstract: Die Energie-Prozess-Management-Produktlinie (EPM-Produktlinie) der BTC AG baut auf dem gemeinsamen EPM-Architekturstil und der EPM-Referenzarchitektur auf. Deren Anwendung stellt die technische und fachliche Interoperabilität innerhalb der Produktlinie sicher, und ermöglicht es, die effiziente Wartbarkeit, Testbarkeit sowie die erforderliche Variabilität in den nichtfunktionalen Anforderungen und der technischen Umgebung in kundenspezifischen Anwendungslandschaften erzielen zu können. Zur Erreichung dieser Variabilität werden technologieübergreifende Standardschnittstellen als Variationspunkte eingesetzt, die von mehreren Komponenten mit unterschiedlichen nichtfunktionalen Eigenschaften bzw. Anforderungen an die technische Umgebung implementiert werden. Dieser Beitrag berichtet über Erfahrungen und Lösungsansätze für Herausforderungen, die sich in der EPM-Produktlinie gezeigt haben.

1 Einleitung

Die BTC AG bietet ein Portfolio von Softwareprodukten für das technisch-betriebliche Energie-Prozess-Management (EPM) an. Hierzu gehören beispielsweise Anwendungen in den Bereichen Leittechnik für Energieversorgungsnetze, Kraftwerke und Windparks, Advanced Meter Management und virtuelle Kraftwerke.

Auf dieser Ebene werden in verschiedenen Technologien umgesetzte Alt- und Neuprodukte in der EPM-Produktlinie zusammengeführt, um mit Mitteln der Produktlinienentwicklung neue Produkte effizienter und mit höherer Qualität bereitstellen zu können.

Die EPM-Produktlinie dient zunächst als Katalysator für die effektive Wiederverwendung innerhalb des EPM-Produktportfolios. Wiederverwendungsprogramme scheitern häufig an unzureichender technischer und fachlicher Interoperabilität der Softwarebausteine [GAO09]. Zentrale Mittel, um diese Effekte zu verhindern, sind ein gemeinsamer Architekturstil (EPM-Architekturstil), der übergreifende Architekturregeln festlegt, und eine Referenzarchitektur (EPM-Referenzarchitektur), die die funktionalen Zuständigkeiten der wiederverwendbaren Bausteine untereinander und gegenüber den verwendenden Produkten abgrenzt.

Komponentenorientierung und die Verwendung von Standardschnittstellen spielen eine wichtige Rolle innerhalb des EPM-Architekturstils, um die Austauschbarkeit von Komponenten zu erreichen. Diese dienen insbesondere als Variationspunkte hinsichtlich nicht-funktionaler Aspekte. Die Verwendung von Standardschnittstellen bietet Effizienzpotenziale, birgt aber auch Herausforderungen.

Im Folgenden wird zunächst ein Überblick über den Aufbau der EPM-Produktlinie und die Produktlinien-weiten Artefakte gegeben (Abschnitt 2). Anschließend wird ein Überblick über die Architektursichten gegeben, die für die Modellierung von Architekturen in der EPM-Produktlinie verwendet werden (Abschnitt 3). Abschnitt 4 beschreibt die Rolle von Standardschnittstellen in der EPM-Produktlinie. In Abschnitt 5 werden die bislang gemachten Erfahrungen und die dabei identifizierten Herausforderungen, insbesondere im Hinblick auf die Testbarkeit, beschrieben. Abschnitt 6 enthält einen Ausblick auf zukünftige Weiterentwicklungen des Ansatzes.

2 Die EPM-Produktlinie

2.1 Motivation

Innerhalb der EPM-Produktlinie gibt es einzelne Produkte, die teilweise wieder enger zusammenhänge Teilproduktlinien bilden. Die Produkte sind vorwiegend individualisierbare Standardprodukte, d.h. sie werden in Kundenprojekten durch die BTC AG auf die spezifischen Anforderungen des Kunden angepasst. Sie werden in unterschiedlichen Technologien, insbesondere C++ unter Windows und Linux und C#/.NET entwickelt.

Eine Optimierung von Entwicklungskosten, Lieferzeiten und Qualität der den Kunden der BTC angebotenen Produkte und Lösungen soll über die Produkt-übergreifende Wiederverwendung von Komponenten und Schnittstellen erzielt werden. Als Katalysator der Wiederverwendung wirken die Produktlinien-weiten Artefakte EPM-Referenzarchitektur und EPM-Architekturstil zusammen[1], die im Folgenden beschrieben werden.

2.2 EPM-Referenzarchitektur

Den Kern der EPM-Referenzarchitektur bildet eine Strukturierung, die sich in einem Zwiebelmodell mit fünf Schichten von Softwarebausteinen[2] darstellen lässt (siehe Abbildung 1). Von innen nach außen handelt es sich zunächst um (1) Basisdienste & Frameworks und (2) EPM-Plattformbausteine. Diese bilden gemeinsam die EPM

[1] Man könnte insgesamt im weiteren Sinne von der Produktlinienarchitektur sprechen, die aber oft im Zusammenhang mit Feature-Modellen betrachtet wird. Feature-Modelle mit funktionalen Variationspunkten spielen auf dieser Ebene eine geringere Rolle, dies ist eher bei den Teilproduktlinien zu finden.

[2] „Softwarebaustein“ wird hier als allgemeiner Oberbegriff für die je nach Architekturstandpunkte unterschiedlichen Modellierungselemente verwendet. Darunter fallen insbesondere Komponenten, Schnittstellen und Module (vgl. Abschnitt 3).

Universal Platform. Darauf folgen (3) Domänen-übergreifende Bausteine, die für mehrere Teilproduktlinien verwendet werden, und (4) Produktlinien-spezifische Bausteine, die jeweils nur für eine Teilproduktlinie verwendet werden. Die Schicht (5) ist geteilt in Produkt-spezifische Bausteine und Projekt-spezifische Bausteine, wobei letztere auf Produkt-spezifischen Bausteinen aufsetzen können, jedoch nicht müssen.

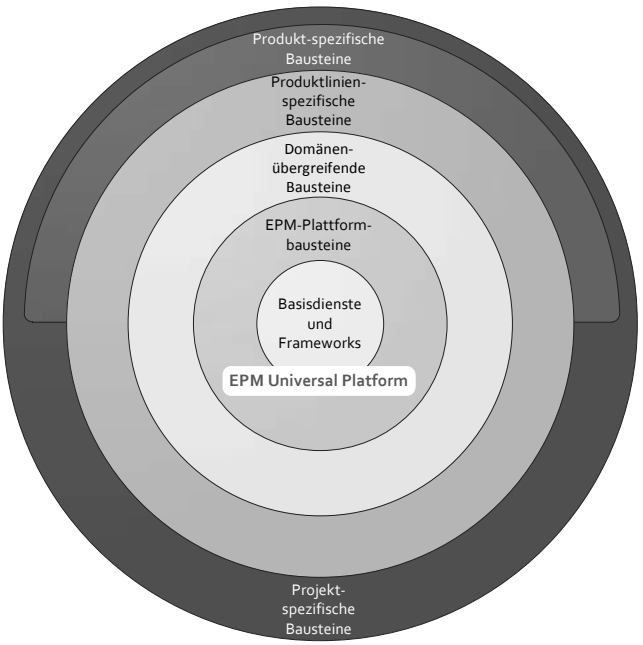

Abbildung 1: Schichtenmodell der EPM-Referenzarchitektur

Für jeden einzelnen Baustein erfolgt in der EPM-Referenzarchitektur eine Definition und Abgrenzung der funktionalen Zuständigkeiten und nichtfunktionalen Eigenschaften, die zur Vermeidung von Überschneidungen mehrerer Bausteine beiträgt und somit die fachliche Interoperabilität sicherstellt. Dies gilt sowohl innerhalb einer Schicht, als auch zwischen den Schichten.

2.3 EPM-Architekturstil

Die Strukturierung der EPM-Referenzarchitektur wird ergänzt durch Schichten-übergreifende und Schichten-spezifische Architekturregeln, die der EPM-Architekturstil festlegt. Diese setzen bewährte Prinzipien in einen Gesamtkontext und entwickeln sie weiter, insbesondere die Ideen von Johannes Siedersleben (Quasar) [Sie04] und Robert C. Martin (SOLID-Prinzipien) [Mar02]. Insgesamt bilden die Regeln den EPM-Architekturstil, der eine von mehreren denkbaren Alternativen zur Realisierung der angestrebten Qualitätseigenschaften darstellt. Die möglichst durchgängige Anwendung dieser Regeln stellen sicher, dass eine technische Interoperabilität sichergestellt ist und Effekte des „architectural mismatch“ [GAO09] vermieden werden. In allen genannten Schichten wird ein konsequent auf Modularisierung und aktivem Management der

Modulabhängigkeiten (physische Softwarestruktur) basierender Software-Entwicklungsansatz verfolgt. In den Schichten (2) bis (5) werden darüber hinaus Komponenten und Schnittstellen (logische Softwarestruktur) definiert, für welche die erste Schicht die gemeinsame Ausführungsinfrastruktur bereitstellt. Im Rahmen dieses Beitrags kann nur ein Überblick über die Architekturstandpunkte (Abschnitt 3) gegeben werden, und anschließend ein Aspekt genauer betrachtet werden, nämlich die Klassifikation von Schnittstellen anhand ihrer Besitzer (Abschnitt 4) und die Erfahrungen mit Standardschnittstellen (Abschnitt 5).

3 Die Architekturstandpunkte des EPM-Architekturstils

Für den EPM-Architekturstil werden vier Architekturstandpunkte im Sinne des ISO-Standards 42010 betrachtet, die unterschiedliche Arten von Softwarebausteine als Modellierungselemente verwenden:

1. Logische Softwarestruktur: Logische Komponenten und Schnittstellen
2. Logische Verteilung: Kompositionskontext und logische Komponenten
3. Physische Softwarestruktur: Module
4. Physische Auslieferung: Deploymenteinheiten, Konfigurationen

Diese werden im Folgenden jeweils kurz beschrieben.

Logische Softwarestruktur: In der logischen Softwarestruktursicht werden *Komponenten* und (Komponenten-)*Schnittstellen* und ihre statischen Abhängigkeiten modelliert. Eine Komponente *implementiert* i.d.R. eine Schnittstelle und *benutzt* eine Menge anderer Schnittstellen. In einem logischen Softwarestrukturmodell kommen keine Abhängigkeiten zwischen Komponenten vor.

Diese Konventionen für die logische Softwarestruktur legen eine gewisse Abbildung auf physische Elemente nahe, zum Beispiel eine auch physische Trennung von Schnittstellen und ihren Implementierungen[3]. Bei der Abbildung bestehen jedoch erhebliche Freiheitsgrade, die grundsätzlich mit beliebigen Technologien umzusetzen sind. Die Granularität von logischen Komponenten und Schnittstellen lässt sich nicht pauschal festlegen, typischerweise ist diese aber gröber als die der Module in der physischen Struktursicht. Insbesondere besteht eine logische Schnittstelle praktisch immer aus vielen Schnittstellen im Sinne einer Programmiersprache.

Logische Verteilung: In der logischen Verteilungssicht können konkrete *Einsatzszenarien* der Komponenten modelliert werden. Hier wird dargestellt, wie Komponenten miteinander zu einer logischen *Konfiguration* verbunden werden, um nach außen einen bestimmten Zweck zu erfüllen. Dabei werden auch logische *Ausführungskontexte* definiert, an deren Grenzen eine physische Verteilung stattfinden kann. Es besteht eine

[3] Zwingend ist dies jedoch nicht, so dass sich auch Legacy-Software auf diese Weise modellieren lässt. Für die Entwicklung neuer Softwarebausteine fordert der EPM-Architekturstil allerdings in der Regel eine solche Trennung.

Wechselbeziehung zwischen der Art der Schnittstellen und den möglichen Einsatzszenarien, da über Ausführungskontexte hinweg keine objektorientierten Schnittstellen, sondern dienstorientierte Schnittstellen eingesetzt werden sollten[4].

Physische Softwarestruktur: In der physischen Softwarestruktursicht sind die Bausteine *Module*, die jeweils implizit oder explizit eine Schnittstelle *exponieren* und die von anderen Modulen exponierten Schnittstellen *nutzen*. Es gibt dabei unterschiedliche *Modultypen*. Eine wichtige Rolle spielen Module, die die logischen Schnittstellen und Komponenten abbilden. Es gibt darüber hinaus aber weitere Modultypen, z.B. Module, die Basisframeworks enthalten, oder solche, die als Adapter zu einem Komponentencontainer dienen. Standardschnittstellen werden generell als eigenständige Module (Schnittstellenmodule) abgebildet, die unabhängig von den Modulen der Komponenten sind, die die Schnittstelle benutzen oder implementieren.

Physische Auslieferung: Die Module werden zu *Auslieferungseinheiten* (z.B. DLLs plus weitere für die Nutzung benötigte Dateien) gebündelt, die für den Betrieb in der Regel durch einen Komponentencontainer instanziert und konfiguriert werden. Die Bündelung soll so erfolgen, dass zur Konfigurationszeit ein Austausch einer Komponente durch eine andere Implementierung derselben Standardschnittstelle bzw. durch eine neue Version derselben Implementierung möglich ist.

4 Standardschnittstellen in der logischen Struktursicht

Schnittstellen können hinsichtlich ihres logischen „Besitzers“[5] (vgl. [Sie04]) unterschieden werden in Importschnittstellen, Exportschnittstellen und Standardschnittstellen. Dies ist in Abbildung 2 dargestellt, die Besitzkontexte sind dort durch die dunklen Kästen markiert. Importschnittstellen werden von einer verwendenden Komponente festgelegt. Exportschnittstellen werden von einer implementierenden Komponente festgelegt. Standardschnittstellen werden unabhängig von verwendenden und implementierenden Komponenten festgelegt.

Exportschnittstellen sind im Allgemeinen der mit Abstand verbreiteteste Fall: Es wird eine Komponente entwickelt, die implizit oder explizit eine Schnittstelle zur Nutzung durch andere Komponenten anbietet. Auch wenn eine Komponenten-orientierte Entwicklung die explizite Trennung von (Export-)Schnittstelle und Komponente fordert, ist damit noch nicht automatisch eine strenge Trennung des Lebenszyklus von Komponente und Schnittstelle verbunden: Wenn bei der Entwicklung der Komponente eine Änderung der Schnittstelle zweckmäßig erscheint, wird diese vorgenommen, ggf. eingeschränkt durch gewisse Zusicherungen über die verlässliche Evolution an die Nutzer. Die Schnittstelle wird aber nicht als eigenständiges Artefakt gepflegt.

Bei Standardschnittstellen ist dies anders. Ihre Evolution wird nicht durch die Entwickler einer einzelnen Implementierung bestimmt, sondern unterliegt einer eigenständigen

[4] vgl. [Sie04] und Martin Fowler’s First Law of Object Distribution: Don’t distribute your objects [Fow03].
[5] Diese Unterscheidung ist nicht zu verwechseln mit der Benutzt- vs. Implementiert-Beziehung zwischen Komponenten und Schnittstellen. Diese Zusammenhänge sind voneinander unabhängig.

Weiterentwicklung. In die Überlegungen zur Weiterentwicklung fließen sowohl Anforderungen von Nutzern und Implementierern ein. Datenbankzugriffsschnittstellen wie ODBC stellen Standardschnittstellen dar, bei der Nutzer, Schnittstelle und Implementierungen sogar typischerweise von drei jeweils unterschiedlichen Organisationen entwickelt werden.

Plug-in-Schnittstellen sind typischerweise als Importschnittstellen ausgeprägt; dies wird beispielsweise in Eclipse von verschiedenen Teilkomponenten verwendet; die Plug-ins sind dann nur in einer bestimmten Komponente nutzbar. Standard-Plug-in-Schnittstellen, die von mehreren Komponenten angeboten werden, sind jedoch ebenfalls denkbar.

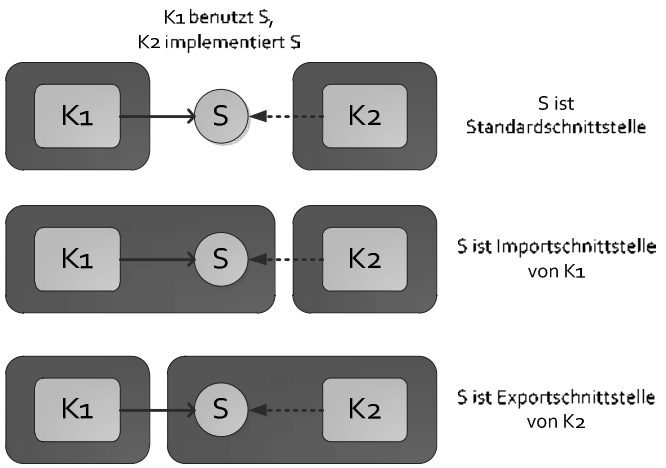

Abbildung 2: Besitzer von Schnittstellen

Für Standardschnittstellen gelten besondere Qualitätsanforderungen (vgl. [Sie04]):

- **Abgeschlossenheit:** Die Schnittstelle muss in sich abgeschlossen sein, d.h. sie darf keine weiteren Schnittstellen verwenden, ansonsten bilden diese lediglich eine logische Schnittstelle, die aus mehreren Teilschnittstellen besteht. Die benötigten Datentypen müssen innerhalb der Schnittstelle definiert werden. Ausnahmen gelten für allgemein nutzbare Bausteine (vgl. [Nag90]) der Schicht (1) Basisdienste und Frameworks (vgl. Abbildung 1).
- **Dokumentation:** Es existiert eine Qualitäts-gesicherte Dokumentation der Schnittstelle und ihrer Operationen.
- **Validierbarkeit:** Es steht eine Testsuite zur Validierung beliebiger Implementierungen zur Verfügung, die definierte Anforderungen an die Testabdeckung erfüllt.
- **Implementierbarkeit:** Mittels mindestens zwei Implementierungen muss demonstriert worden sein, dass die Schnittstelle implementierbar ist. Zu diesen Implementierungen gehören eine „Dummy“-Implementierung und eine echte, produktiv nutzbare Implementierung.

- **Verlässlichkeit:** Eine Standardschnittstelle sollte nur selten geändert werden. Insbesondere sollten Änderungen, die nicht abwärtskompatibel sind, nur sehr selten vorgenommen werden. In solchen Fällen muss für einen definierten Zeitraum die alte Version noch unterstützt werden und Implementierungen der alten Schnittstelle während dieses Zeitraums auch noch gewartet werden.

Neben diesen Anforderungen mit technischen Auswirkungen gelten auch noch Anforderungen hinsichtlich der Veröffentlichung in der EPM-Referenzarchitektur und der Akzeptanz durch den angestrebten Nutzerkreis. Die Erfüllung dieser Qualitätskriterien ist Bedingung für ihre effiziente Nutzung. Eine kritische Rolle dabei spielt die Verlässlichkeit. Voraussetzung dafür ist einerseits, dass die Standardschnittstellen einen Reifungsprozess durchlaufen haben, und andererseits, dass die Nutzungsprofile der Nutzer nicht zu stark divergieren, was durch die eingeschränkte Anwendungsdomäne der nutzenden Produkte und Lösungen innerhalb der Produktlinie erreicht wird.

Eine für die Schicht der EPM-Plattformbausteine (vgl. Abbildung 1) spezifische Regel des EPM-Architekturstils fordert, dass vorrangig Standardschnittstellen definiert werden, zu denen je nach Erfordernis mehrere implementierende Komponenten existieren. Dies kann sowohl zeitlich parallel als auch versetzt gelten: Zeitlich parallel angebotene Implementierungen derselben Schnittstelle können unterschiedliche nicht-funktionale Anforderungen bzw. unterschiedliche technische Rahmenbedingungen erfüllen bzw. verschiedene externe Systeme anbinden. Zeitlich versetzt angebotene Implementierungen derselben Standardschnittstelle können bei einer Weiterentwicklung der Implementierungstechnologie zum Einsatz kommen, die sich auf die Nutzer der Standardschnittstelle nicht auswirkt. In der EPM-Referenzarchitektur wird festgehalten, welche konkreten Standardschnittstellen und Komponenten ihre Implementierungen es gibt bzw. geben soll.

5 Erfahrungen und Herausforderungen

In diesem Abschnitt werden einige Erfahrungen aus der EPM-Produktlinie im Zusammenhang mit Standardschnittstellen und die sich daraus ergebenden Herausforderungen diskutiert, sowie Lösungsansätze skizziert.

Grundsätzlich sind diese Herausforderungen von Standardschnittstellen nicht neu gegenüber der korrekten Verwendung von Schnittstellen im Allgemeinen. Ihre Auswirkungen sind im Zusammenhang mit Standardschnittstellen jedoch weitreichender.

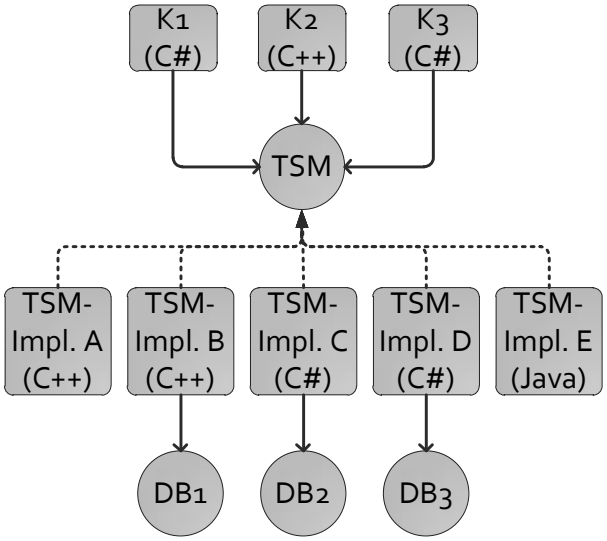

Abbildung 3: Nutzer und Implementierer der TSM-Schnittstelle

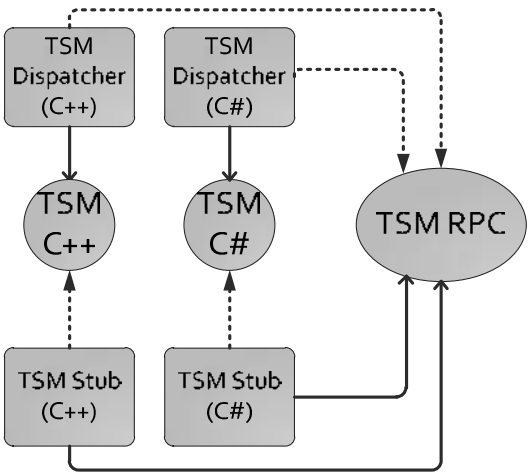

Abbildung 4: Technologie-spezifische Ausprägungen der TSM-Schnittstelle

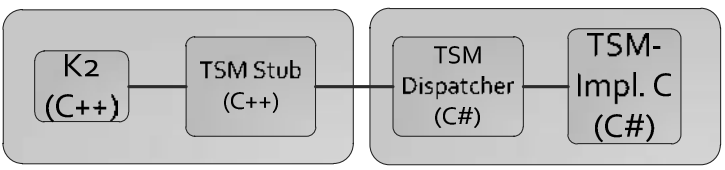

Abbildung 5: Beispiel eines Einsatzszenarios der TSM-Komponenten mit Kompositionskontexten

5.1 Standardschnittstellen in der EPM-Produktlinie: Beispiel Zeitreihen

In der im Aufbau befindlichen EPM-Produktlinie gibt es bereits verschiedene Standardschnittstellen bzw. Vorläufer von Standardschnittstellen mit verschiedenen Implementierungen. Am weitesten gereift ist eine Standardschnittstelle für den Zugriff auf Zeitreihendaten (Time Series Management, kurz: TSM). Es existieren mehrere Komponenten als Implementierungen dieser Schnittstelle, die unterschiedliche Arten der Datenhaltung verwenden (siehe Abbildung 3). Dadurch können sie in unterschiedlichen technischen Umgebungen eingesetzt werden (insbesondere eine Verwendung verschiedener Datenbanken, in der Abbildung als DB1, DB2 und DB3 bezeichnet, alternativ) und weisen unterschiedliche nichtfunktionale Eigenschaften, z.B. hinsichtlich Performance und Datensicherheit, auf. Funktional sind die verschiedenen Implementierungen hingegen grundsätzlich identisch.

Die Schnittstelle existiert in Ausprägungen in C++, C# und Java, und es existieren auch Implementierungen in jeder dieser Technologien. Ein Nutzer der Schnittstelle kann über einen Remote-Zugriffs-Mechanismus die Implementierungen jeder Technologie verwenden (siehe Abbildung 4 und Abbildung 5). In Abbildung 4 ist dies beispielhaft für C++ und C# dargestellt. Je Technologie gibt es einerseits einen sog. Stub, der die Technologie-spezifische Ausprägung der Schnittstelle implementiert, d.h. beispielsweise einem lokalen C++-Nutzer die C++-TSM-Schnittstelle anbietet. Auf der anderen Seite gibt es einen (ebenfalls Technologie-spezifischen) sog. Dispatcher, der eine beliebige konkrete Implementierung seiner Technologie nutzen kann. Stub und Dispatcher kommunizieren über ein Technologie-übergreifend eindeutiges RPC-Protokoll, in diesem Falle das TSM-RPC-Protokoll. Im übertragenen Sinne kann man sagen, dass der Dispatcher diese „Schnittstelle“ implementiert und der Stub diese nutzt. In Abbildung 5 ist eine konkrete Konfiguration (ein Modell der logischen Verteilungssicht) eines C++-Nutzers und einer C#-Implementierung mit dazwischen liegendem Stub und Dispatcher dargestellt. Die Technologie des linken Konfigurationskontexts ist natives C++ mit einem C++-Komponentencontainer, der rechte Konfigurationskontext C#/.NET mit Spring.NET.

5.2 Parametrisierung von Tests gegen Standardschnittstellen

Standardschnittstellen stellen für die Testbarkeit eine besondere Herausforderung dar, bieten aber auch erhebliche Effizienzpotenziale. In einer Testhierarchie bewegen sich diese in der Regel auf einer mittleren Ebene zwischen Systemintegrationstests und modulbezogenen Unit-Tests einer bestimmten Implementierung (bei einfachen Implementierungen, die nur aus einem Modul bestehen, können diese ggf. auch die Rolle von Unit-Tests übernehmen). Eine einmal entwickelte Testsuite kann mit geringem Aufwand für jede Implementierung eingesetzt werden, wenn der Test mit einer Komponente parametrisierbar ist. Dies gilt zumindest dann, wenn die Tests keine impliziten Annahmen über eine bestimmte Implementierung enthalten, und die Implementierungen die Schnittstelle korrekt implementieren, insbesondere das Liskov’sche Substitutionsprinzip [LW94] erfüllen.

5.3 Erweiterte Parametrisierung von Tests mit Testtreibern

Typische Standardschnittstellen können unterschiedlich stark bestimmt ausgeprägt sein (eine vollständige Spezifikation kommt bei den von uns betrachteten Systemen nicht vor). In dem hier betrachteten Fall ist es so, dass es für den Basisteil der Schnittstelle keine funktionalen Unterschiede zwischen den Implementierungen geben sollte. Dies gilt in vielen Fällen aber nicht: Einige Implementierungen akzeptieren bestimmte Parameterwerte, die von anderen nicht akzeptiert werden. Ein typisches Beispiel hierfür ist, dass nur bestimmte Implementierungen einer (Programmiersprach-)Schnittstelle, die zur selben Komponente gehören, akzeptiert werden. Um dieses Problem in den Griff zu bekommen, müssen Tests nicht nur mit der Komponente, sondern mit einem Testtreiber (test subject) parametrisiert werden, der die für den Test benötigten Parameter erzeugen kann.

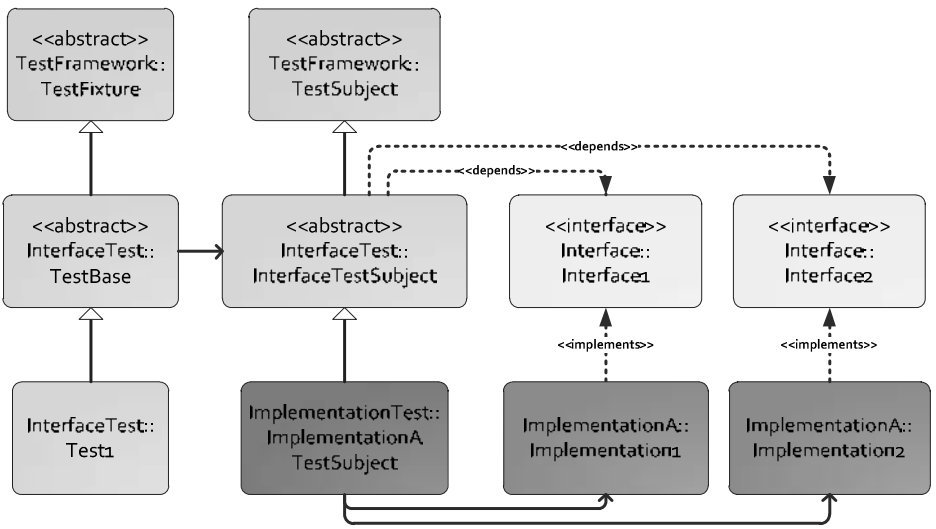

Abbildung 6: Muster für die Trennung von Test und Testtreiber

In Abbildung 6 ist die Klassenstruktur des zugrunde liegenden Entwurfsmusters dargestellt. Das Testframework stellt typischerweise eine abstrakte Basisklasse für Tests `TestFixture` zur Verfügung, ggf. auch eine abstrakte Basisklasse `TestSubject`. `InterfaceTestSubject` ist eine abstrakte Fabrik für die Schnittstelle `Interface1` und `Interface2`; `ImplementationATestSubject` ist eine zugehörige konkrete Fabrik, die von konkreten Implementierung von `Interface1` und `Interface2` abhängt. Es gibt eine abstrakte Testklasse `TestBase` je `InterfaceTestSubject`, die Initialisierung und Deinitialisierung des Testtreibers übernimmt. Dazu kann es mehrere konkrete Testklassen, (in der Abbildung `Test1`) geben, die dasselbe `InterfaceTestSubject` verwenden. Die Verknüpfung von konkretem `TestSubject` und konkretem Test wird erst bei der Instanzierung hergestellt.

5.4 Strukturierung von Standardschnittstellen und zugehörigen Tests

Unterschiede zwischen mehreren Implementierungen einer Schnittstelle, die sich auf die Testbarkeit auswirken, können weitere Ursachen haben: Einerseits kann es sich um unterschiedliche Teilmengen von unterstützten Operationen handeln, die nach Möglichkeit durch mehrere unabhängige physische *Teilschnittstellen* (ggf. auch unabhängige logische Schnittstellen) abgebildet werden können. Dies hat einen weiteren positiven Nebeneffekt: Wie in Abbildung 7 dargestellt, bietet dies dann auch die Möglichkeit, eine generische, aber möglicherweise wenig effiziente Implementierung der erweiterten Schnittstelle nur auf Grundlage der Basis-Schnittstelle anzubieten. Ein Nutzer kann die Implementierung A mit der Schnittstellen-Erweiterung nutzen, obwohl Implementierung A diese selbst nicht implementiert.

Handelt es sich um spezielleres Verhalten derselben Operationen, kann dies durch eine *Schnittstellenhierarchie* abgebildet werden. Für Schnittstellen-basierte Tests ergeben sich Konsequenzen: Im ersten Fall muss der Test mit der Erwartung über die Unterstützung von Teilschnittstellen parametrisiert werden. Im zweiten Fall kommt neben der Schnittstellenhierarchie eine parallele Testhierarchie zum Einsatz. Nur mit der allgemeinsten Testebene ist es unmöglich, aussagefähige Tests mit einer vollständigen Code-Abdeckung zu realisieren.

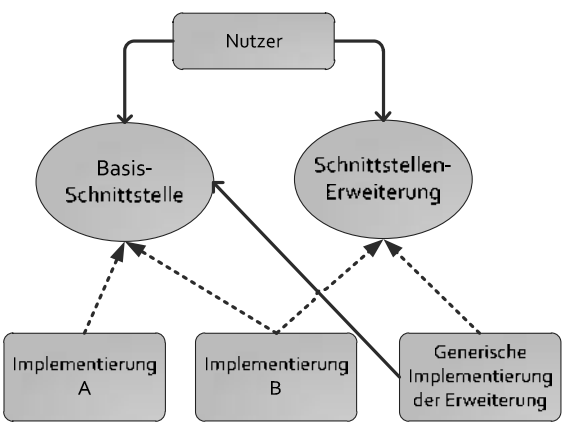

Abbildung 7: Basis-Schnittstelle und Schnittstellen-Erweiterung

5.5 Remote-Fähigkeit von Standardschnittstellen

Eine wichtige Frage ist die Remote-Fähigkeit von Standardschnittstellen. Bei der hier betrachteten Zeitreihenzugriffs-Schnittstelle ist es naheliegend, diese entfernt zu nutzen. Für den gewählten Ansatz der Technologie-übergreifenden Nutzung ist dies sogar zwingend. Eine Remote-Nutzung ist nie transparent möglich: Offensichtlich gilt dies für das Zeitverhalten, aber auch das im programmiersprachlichen Sinne funktionale Verhalten ist betroffen. Die Remote-Fähigkeit ist daher beim Entwurf der Schnittstelle zu berücksichtigen, sie kann nicht ohne weiteres nachträglich hinzugefügt werden, da

dann alle Nutzer angepasst werden müssen. Neben der gröberen Granularität der Operationen und dem Verzicht auf Verteilung auf Objekt-Ebene gilt dies insbesondere für das Fehlerverhalten. Zunächst muss dieses korrekt sein, d.h. es dürfen bei einer Remote-Verwendung nicht zusätzliche, in der Schnittstelle unspezifizierte Ausnahmen erzeugt werden. Es ist ebenfalls nicht hilfreich, in der Schnittstelle pauschal zu spezifizieren, dass generische Ausnahmen erzeugt werden können („throws Exception"), da dies einem Nutzer unmöglich macht, geeignet auf Ausnahmen zu reagieren. Vielmehr ist für die Remote-Nutzung dem Aufrufer die Möglichkeit zu geben, im Falle von Zeitüberschreitungen, Verbindungsabbrüchen und ähnlichem, je nach Bedarf zu reagieren. Hierfür müssen geeignete Konventionen festgelegt werden, so dass ein Nutzer nicht für jede genutzte Standardschnittstelle oder gar jede Operation ein anderes Fehlerbehandlungskonzept implementieren muss.

6 Ausblick

In dem genannten Beispiel hat sich darüber hinaus gezeigt, dass eine sehr hohe Testabdeckung (Implementierung A: 99%) einer Implementierung nicht automatisch auch zu einer ähnlich hohen Testabdeckung anderer Implementierungen führt (Implementierung B: 68%, Implementierung C: 40%). Theoretisch lassen sich hierfür leicht Gründe finden, z.B. die Prüfung von Vorbedingungen in internen Methoden einer Klasse, die durch einen Test der öffentlichen Methoden unmöglich vollständig geprüft werden können. Praktisch stellt sich die Frage, wodurch dies in welchem Ausmaß bei den konkreten Komponenten verursacht wird, und ob sich dieser Effekt ggf. durch geeignete Konventionen gezielt vermindern lässt. Die konsequente Nutzung von Code Contracts (z.B. .NET Code Contracts) könnte hierbei helfen, da hiermit ein definierter Weg zur Spezifikation von Vor- und Nachbedingungen zur Verfügung steht.

In einer weiteren Ausbaustufe der EPM-Produktlinie ist die Etablierung eines einheitlichen Plug-in-Konzepts geplant, um auch die funktionale Variabilität einzelner Produkte effizienter umsetzen zu können.

Literaturverzeichnis

[Fow03] Martin Fowler: Patterns für Enterprise Application-Architekturen, mitp, 2003.

[GAO09] David Garlan, Robert Allen, and John Ockerbloom: Architectural Mismatch: Why Reuse Is Still So Hard, IEEE Software 26(4), July/August 2009, pp. 66–69.

[LW94] Barbara Liskov, Jeannette Wing: A behavioral notion of subtyping, ACM Transactions on Programming Languages and Systems 16(6), November 1994, pp. 1811–1841.

[Mar02] Robert C. Martin: Agile Software Development, Prentice-Hall, 2002.

[Nag90] Manfred Nagl: Softwaretechnik. Methodisches Programmieren im Großen, Springer, 1990.

[Sie04] Johannes Siedersleben: Moderne Softwarearchitektur, Dpunkt Verlag, 2004.