Hauptseminar

Management von Softwaresystemen

Technische Universität München
WS 2005/06

Autor : Ceren Alkis
Thema: Standardarchitekturen am Beispiel von QUASAR
Datum: 29.11.2005

Inhaltsverzeichnis:

1. Motivation-Quasar Projekt **3**

2. Komponenten und Schnittstellen **3**
2.1 Komponenten 3
2.2 Schnittstellen 4
2.3 Konfiguration 4
2.3.1 Konfiguration und Implementierung von Klassen 4
2.3.2 Komponenten Verbinden 6
2.3.3 Kompositionsmanager 7

3. Softwarekategorien **10**
3.1 A- und T-Software 10
3.2 Kommunikation zwischen Komponenten verschiedener Kategorien 11
3.3 Softwarekategorien und Komplexität 12

4. Zusammenfassung **16**

5. Literaturangabe **17**

1. Motivation-Quasar-Projekt

Quasar bedeutet Qualitätssoftwarearchitektur und wurde von der sd&m AG definiert. Man kann es als eine Architektur für qualitativhochwertige Software interpretieren. Das ist eine Behauptung von sd&m.

Das Quasar-Projekt begann am 5. Mai 1998 in München und wurde von Prof. Dr. Johannes Siedersleben geleitet. Prof. Dr. Johannes Siedersleben vertrat an der FH Rosenheim das Fachgebiet Software-Engineering und war wissenschaftlicher Leiter von sd&m Research, der Forschungs-abteilung von sd&m (software design & management, München).

Das Ziel mit dem Projekt war die unterschiedliche Architekturideen vieler sd&m Projekte einzusammeln und in eine wieder verwendbare Form zu bringen. Unter dem Namen Quasar werden viele bewährte Architektur Ideen etwas aufbereitet und modernisiert und ein paar neue Ideen werden auch zugefügt.

Quasar hat sich bei sd&m durchgesetzt. Die Schulungen, die sd&m für die eigenen Mitarbeiter veranstaltet hat, wurde eine wichtige Plattform der Kommunikation. Durch die Schulungen wurde die Elemente von Quasar diskutiert und weiterentwickelt. Das Prinzip Denken in Komponenten, Kategorien und Schnittstellen wurde zum Standard geworden.

2. Komponenten und Schnittstellen

2.1 Komponenten

Eine Komponente ist ein in sich gekapselter Teil eines Software-Systems. Die Komponente ist neben der Schnittstelle eine wesentliche Einheit des Entwurfs, der Implementierung und damit der Planung. Jede Komponente besitzt eine verbindlich definierte Aussensicht, ihre Schnittstelle und eine nicht weiter verbindliche Innensicht, nämlich die Implementierung selbst. *Jede Komponente besitzt die folgenden Merkmale:* [Sid04]

- Sie exportiert eine oder mehrere Schnittstellen, die im Sinne eines Vertrags garantiert sind. Jede Komponente, die die Schnittstelle S exportiert, ist eine Implementierung von S.
- Sie importiert andere Schnittstellen. Der Import einer Schnittstelle bedeutet, dass die Komponente die Methoden dieser Schnittstelle verwendet. Sie ist erst lauffähig, wenn alle importierten Schnittstellen zur Verfügung stehen und dies ist Aufgabe der Konfiguration.
- Sie versteckt die Implementierung und kann daher durch eine andere Komponente ersetzt werden, die dieselbe Schnittstelle exportiert.
- Sie eignet sich als Einheit der Wiederverwendung, denn sie kennt nicht die Umgebung, in der sie läuft, sondern macht darüber nur minimale Annahmen.
- Komponenten können andere Komponenten enthalten oder anders ausgedrückt: Man kann neue Komponenten aus vorhandenen Komponenten zusammensetzen.

Komponenten bestehen also aus Modulen und/oder aus anderen Komponenten: Komponenten, die nicht weiter unterteilt sind, nennen wir einfach oder auch Modul; alle anderen sind zusammengesetzt, und zwar aus Subkomponenten.

2.2 Schnittstellen

Schnittstellen verbinden entweder Komponenten untereinander (*Programmschnittstellen* oder kurz *Schnittstelle*) oder Komponenten mit dem Benutzer(*Benutzerschnittstellen*). Jede Komponente kann keine, eine oder mehrere Benutzerschnittstellen anbieten.

Jede Komponente sollte aus zwei Teilen bestehen: [BMRSS98]

- Die Schnittstelle definiert die Funktionalität, die die Komponente zur Verfügung stellt, und beschreibt, wie diese zu benutzen ist. Diese Schnittstelle ist den Klienten der Komponenten zugänglich. Eine exportierte Schnittstelle besteht in der Regel aus Signaturen von Operationen.

- Die Implementierung enthält den Code für die Funktionalität, die von der Komponente angeboten wird. Die Implementierung kann darüber hinaus auch weitere Funktionen und Datenstrukturen definieren, die nur innerhalb der Komponente benutzt werden. Die Implementierung ist den Klienten nicht zugänglich.

Das wesentliche Ziel dieses Prinzips ist es, die Klienten einer Komponente vor deren Implementierungsdetails zu schützen und ihnen nur die Beschreibung der Schnittstelle und Richtlinien für deren Benutzung zur Verfügung zu stellen. Ausserdem ermöglicht Ihnen dieses Prinzip, die Funktionalität der Komponente unabhängig von der Anwendung in anderen Komponenten zu implementieren.

Schnittstellen gestatten es, die Abhängigkeiten in der Schnittstelle zu konzentrieren und jede Abhängigkeit von der Implementierung zu vermeiden. Natürlich entfalten Schnittstellen ihren Nutzen dann, wenn wirklich ohne Bezug auf die Implementierung nur gegen Schnittstellen programmiert wird.

Die Trennung von Schnittstelle und Implementierung unterstützt auch die Änderbarkeit eines Systems. Eine Komponente lässt sich sehr viel leichter ändern, wenn ihre Schnittstelle von ihrer Implementierung getrennt ist. Diese Trennung bewahrt Klient davor, von jeder Änderung betroffen zu sein. Das Prinzip macht es insbesondere in solchen Fällen, wenn die Schnittstelle einer Komponente von einer Änderung nicht betroffen ist.

2.3 Konfiguration

2.3.1 Konfiguration und Implementierung von Klassen

Die Festlegung der implementierenden Klasse nennen wir Konfiguration. Insgesamt ergibt sich ein immer wieder gleiches Zusammenspiel von Konfiguration, Benutzer und Implementierung. Der Benutzer verwendet eine Schnittstelle, er weiß nichts über die Implementierung. Die Implementierung implementiert die Schnittstelle, sie weiß nichts über den Benutzer.

Die Konfiguration bringt Benutzer und Implementierung zusammen. Sie kennt beide mit Namen, weiß aber nichts über deren Funktion. Benutzer und Implementierung sind voneinander unabhängig; die Implementierung ist austauschbar.

Dies ist ein Beispiel für die perfekte Unabhängigkeit von der Implementierung. *ListUser* läuft mit jeder Klasse, die List implementiert.

```
public class ListUser {

   private List list;
   public ListUser (List list) {
      this.list = list;
   }

   public void foo() {
      Iterator i = list.iterator();

      // tu was mit list
   }
}
```

```
public static void main(String[] args) {

   List list = new ArrayList();
      // oder irgendeine andere Klasse, die List implementiert

      // Liste fuellen
   ListUser lu = new ListUser(list);
   lu.foo();
}
```

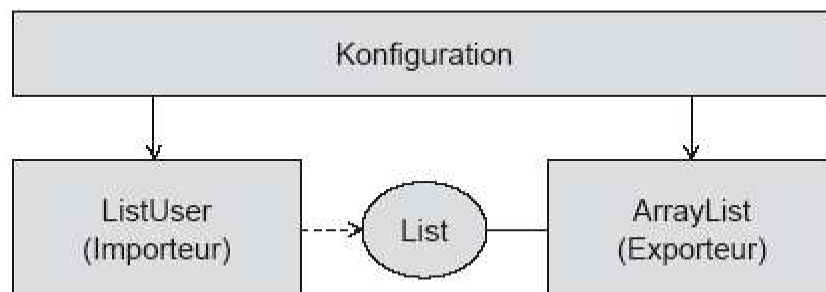

Abb. 1: Konfiguration [Sid04]

2.3.2 Komponeneten Verbinden

Komponenten zerlegen das Gesamtsystem in überschaubare, für sich verstehbare Teile und die Konfigurationen verbinden sie zu einem lauffähigen Ganzen.

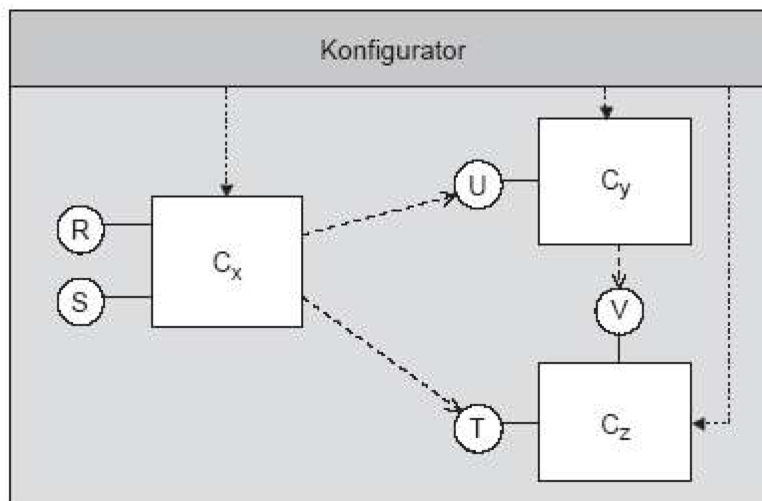

Abb. 2: Konfigurator [Sid04]

Als Beispiel betrachten wir eine Komponente *Cx* mit den exportierten Schnittstellen *R*, *S* und den importierten Schnittstellen *U* und *T*. Die Komponenten *Cy*, *Cz* mögen *U bzw. T* exportieren. Weiterhin importiert *Cy* die von *Cz* exportierte Schnittstelle *V*. Dann sieht die Konfiguration in Java folgendermaßen aus:

```
// Cx, Cy, Cz als Java-Klassen
public class Cx implements R, S { .. }
public class Cy implements U { .. }
public class Cz implements T, V { .. }

// Konfiguration von Cx:
// x, y, z als Komponentenobjekte
Cx x = Cx.getCx();
Cy y = Cy.getCy();
Cz z = Cz.getCz();
x.bindU(y);     // x bekommt eine Implementierung von U
x.bindT(z);     // x bekommt eine Implementierung von T
y.bindV(z);     // y bekommt eine Implementierung von V
```

Nach erfolgreicher Konfiguration ist *x* ein betriebsbereites Exemplar von *Cx,* das nun selbst zur Konfiguration anderer Komponenten zur Verfügung steht.

2.3.3 Kompositionsmanager

Die Aufgabe des Kompositionsmanagers ist eine geeignete Umgebung für die Komponenten herzustellen. Jeder Manager verwaltet eine oder mehrere Komponenten.

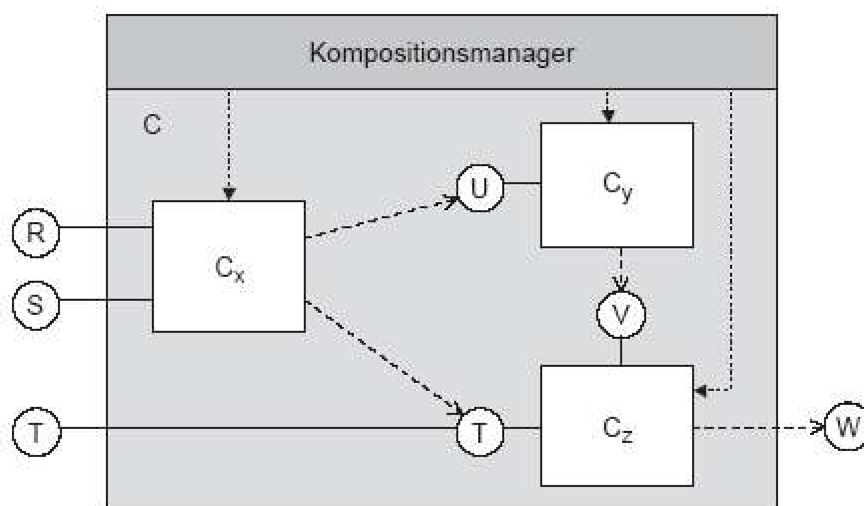

Abb. 3: Kompositionsmanager [Sid04]

Die drei Komponenten *Cx*, *Cy* und *Cz* gemeinsam mit ihrem Kompositionsmanager eine neue Komponente *C*, die die Schnittstellen *R*, *S* und *T* exportiert und *W* importiert. *C* ist eine Komposition von *Cx*, *Cy* und *Cz*.

Der Kompositionsmanager (hier CManager) hat folgenden Aufbau:

```
public class CManager {
   private Cx x;
   private Cy y;
   private Cz z;

   public CManager() {

      // Konfiguration mit lokalen Komponenten
      x = Cx.getCx();
      y = Cy.getCy();
      z = Cz.getCz();
      y.bindV(z);  // y bekommt eine Implementierung von V
      x.bindU(y);  // x bekommt eine Implementierung von U
      x.bindT(z);  // x bekommt eine Implementierung von T
   }

   // nachtraegliche Konfiguration von außen
   public void bindW(W w) {
      z.bindW(w);
   }

   // oeffentlicher Zugang zu den Schnittstellen
   public R getR() { return x; }
   public S getS() { return x; }
   public T getT() { return z; }
}
```

Jede Komposition exportiert also eine Teilmenge der Schnittstellen, die die enthaltenen Komponenten exportieren, und sie importiert genau die Schnittstellen, die lokal nicht versorgt werden können oder sollen. Mit der Methode *bindW* hängt später ein anderer Kompositionsmanager eine geeignete Komponente ein, die *W* exportiert.

Konfiguration mit XML

```xml
<?xml version="1.0"?>
<configuration>

  <!-- alle beteiligten Schnittstellen -->
  <interface name="R"/>
  <interface name="S"/>
  <interface name="T"/>
  <interface name="U"/>
  <interface name="V"/>
  <interface name="W"/>

  <!-- Komponentendeklarationen -->
  <component name="Cx">
     <export interface="R"/>
     <export interface="S"/>
     <import interface="U"/>
     <import interface="T"/>
  </component>

  <component name="Cy">
     <export interface="U"/>
     <import interface="V"/>
  </component>

  <component name="Cz">
     <export interface="T"/>
     <export interface="V"/>
     <import interface="W"/>
  </component>

  <!-- Komposition -->
  <composition name="C">
     <!-- Komponentenobjekte -->
     <object name="x" component="Cx"/>
     <object name="y" component="Cy"/>
     <object name="z" component="Cz"/>

     <!-- interne Bindungen -->
     <binding interface="U" importer="x" exporter="y"/>
     <binding interface="T" importer="x" exporter="z"/>
     <binding interface="V" importer="y" exporter="z"/>

     <!-- extern sichtbare Schnittstellen -->
     <export interface="R"/>
     <export interface="S"/>
     <export interface="T"/>
     <import interface="W"/>
  </composition>
</configuration>
```

Konfigurationscode ist stereotyp und fehleranfällig und deshalb liegt es nahe, einen Generator zu erstellen. Dieser braucht die Konfigurationsinformation in Form einer einfachen XML-Syntax.

3. Softwarekategorien

Jedes Stück Software gehört zu einer oder mehreren Kategorien, die angeben, welches Wissen in der Software enthalten ist und wovon die Software abhängt. Kleine Softwareeinheiten sollten möglichst nur zu einer einzigen Kategorie gehören, sich um ein einziges Problem kümmern. Das ist das Prinzip der Trennung der Zuständigkeiten. Software, die zu verschiedenen Kategorien gehört, ist schlecht und sollte vermieden werden. Man sollte also anstreben, dass *einfache* Komponenten (Module) zu genau einer Kategorie gehören.

Es gibt keinen Algorithmus, der ein System in Komponenten aufteilt. Der Weg zu einer guten und vor allem nachvollziehbaren Komponentenbildung führt über die Analyse der Abhängigkeiten und die Definition von Softwarekategorien.

Quasar schlägt vor: Analysiere zuerst die Softwarekategorien und zerlege danach auf dieser Basis das System in Komponenten, die von vornherein zur richtigen Kategorie gehören. Softwarekategorien sind die Vorstufe zu Komponenten: Die richtigen Kategorien sind Schubladen, in denen man gefundene Komponenten ablegt. Man schaut sich die Kategorien der Reihe nach an und überlegt, welche Komponenten und welche Schnittstellen man in der jeweiligen Kategorie unterbringen könnte.

3.1 A- und T-Software

Grundlage jeder Kategorie ist die Kategorie 0, zu der alles gehört, was ohne weiteres im gesamten System zur Verfügung steht. Bei betrieblichen Informationssystemen gibt es neben der Kategorie 0 zwei wichtige Kategorien, die in jedem Projekt vorkommen: Software, die sich nur mit der Anwendung befasst, aber frei ist von Technik, hat die Kategorie A; Software, die mindestens ein technisches API kennt, hat die Kategorie T.

Software, die sich unterschiedlich schnell ändert, sollte in verschiedenen Modulen untergebracht werden. Dahinter steht die Absicht, verschiedene Änderungen unabhängig voneinander jeweils lokal in verschiedenen Modulen durchzuführen.

Wer A und T ohne weitere Vorkehrungen vermengt, der erhält Software der unreinen Kategorie AT. Sie ist bestimmt durch Anwendung und Technik; das ist leider eine häufig anzutreffende Form. Sie ist schwer zu warten, widersetzt sich Änderungen, kann kaum wieder verwendet werden und ist daher zu vermeiden, es sei denn, es handelt sich um R-Software.

0-Software kann man ohne Bedenken einsetzen; sie schafft keine ungewünschten Abhängigkeiten. Daher behalten A-Software und T-Software ihre Kategorie selbst dann, wenn sie 0-Software aufrufen: A = A + 0; T = T + 0.

+	0	A	T	R
0	0	A	T	R
A		A	AT	./.
T			T	./.
R				R

Abb. 4: A- und T-Software [Sid04]

3.2 Kommunikation zwischen Komponenten verschiedener Kategorien

Man kann folgende Sichtbarkeitsregel formulieren: Für Software einer hohen Kategorie ist Software von verfeinerten Kategorien sichtbar. Das heißt, Kategorie 0 ist für alle sichtbar.

Für je zwei Kategorien *a* und *b* gibt es im Kategoriegraphen mindestens einen ersten gemeinsamen Vorfahren; das ist die höchste Ebene der Kommunikation und die speziellste Kategorie, über die sich *a* und *b* verständigen können.

Dasselbe etwas formaler: Es sei *K* eine Komponente der Kategorie *a*. Dann darf *K* Schnittstellen einer anderen Kategorie *b* genau dann exportieren oder importieren, wenn *a* eine Verfeinerung von *b* ist. Komponenten einer hohen Kategorie dürfen also Schnittstellen einer niedrigeren Kategorie importieren und exportieren; insbesondere sind 0-Schnittstellen völlig unproblematisch.

Es seien *K* und *H* zwei Komponenten der unabhängigen Kategorien *a* bzw. *b*. Dann führt der einzige legale Weg der Kommunikation über eine Schnittstelle *S* einer Kategorie *c*, die sowohl von *a* als auch von *b* verfeinert wird.

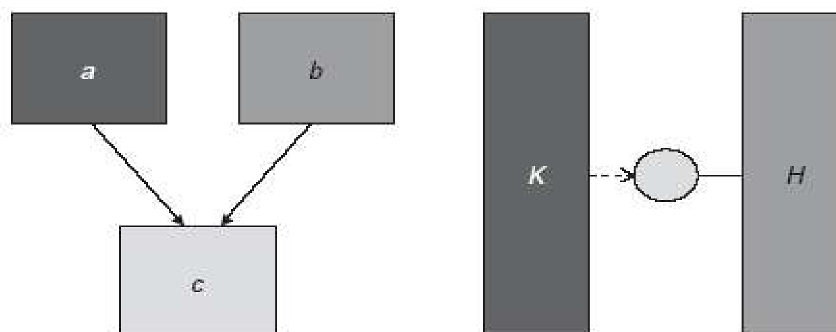

Abb. 5: Kommunikation zwischen Komponenten verschiedener Kategorien [Sid04]

3.3 Softwarekategorien und Komplexität

Kategorien sind teilgeordnet: Jede Kategorie kann eine oder mehrere andere Kategorien verfeinern. Die Kategorien eines Systems ordnet man im zyklenfreien Kategoriegraphen an. Die Wurzelkategorie ist die Kategorie 0 – das ist das allgemeine Grundwissen, das überall vorhanden ist und dessen Verwendung keine unerwünschten Abhängigkeiten erzeugt. Jede Softwarekategorie ist eine direkte oder indirekte Verfeinerung der Kategorie 0.

Eine Softwarekategorie *a* nennen wir *rein*, wenn es im Kategoriegraphen genau einen Weg von *a* zur Kategorie 0 gibt; im anderen Fall heißt sie *unrein*. Reine Kategorien entstehen durch stückweise Verfeinerung der Kategorie 0; unreine Kategorien vermengen zwei oder mehr Kategorien (rein oder unrein).

Jede Komponente und jede Schnittstelle gehört zu einer oder mehreren Kategorien. Ziel ist dabei, dass jeder Modul (einfache Komponente) und jede Schnittstelle möglichst wenigen Kategorien angehört, und zwar solchen, die sich im Kategoriegraphen möglichst weit unten befinden: Je komplizierter oder spezieller ein Thema ist, desto weniger Software sollte sich damit befassen.

Beispiel: Schafkopfen mit dem Computer [Sid04]

Man kann unter dem Beispiel Schafkopfen, das ein vorallem in Bayern verbreitetes Kartenspiel ist, die Softwarekategorien besser betrachten. Die Frage ist, wie man dieses Spiel in Kategorien teilen kann.

Die Softwarekategorie Kartenspiel hat das Wissen, dass es sich um ein Kartenspiel mit einer festen Anzahl von Personen handelt, die der Reihe nach drankommen. Sie wird durch die Kategorie Schafkopf spezialisiert, welche die Regeln des Schafkopfspiels enthält. Getrennt von der Kategorie Schafkopf gibt es die Kategorie Schafkopfstategien, die von Schafkopf spezialisiert ist. Weil man in Kenntnis von Schafkopfregeln viele unterschiedliche Schafkopfsstrategien formulieren kann.

Als Visualisierungsmittel gibt es spezielle Kategorien. Unabhängig von der GUI Bibliothek legt die Kategorie KartenspielGUI fest, wie ein Kartenspiel am Bildschirm aussehen muss. Zum Beispiel in der Mitte stehen die ausgespielten Karten, an den vier Seiten die Blätter der Spieler, so dass jeder Spieler nur seine eigene Karten sehen kann.

Als eine Verfeinerung von GUI Bibliothek und KartenspielGUI bildet man die Kategorie KartenspielGUISwing, damit man im Kenntnis von der Kartenspiele die Darstellungsmöglichkeiten erweitert .

Damit die stattgefundenen Spiele gespeichert werden können, braucht man eine weitere Kategorie Dateisystem, die die Software der Dateispeicherung versteckt. So können die Spiele später eingelesen werden.

Als Grundlage jeder Kategorie steht die Kategorie 0 im untersten Schicht. Wenn das ganze System in Java programmiert wird, stehen in der Kategorie 0 die Java-Pakete *java.lang* und *java.util*.

Dadurch kriegen wir die Kategorien aus der Anwendung(Schafkopf, Schafkopfstrategie), aus der Technik(Dateisystem) und eien Kombination(KartenspielGUISwing).

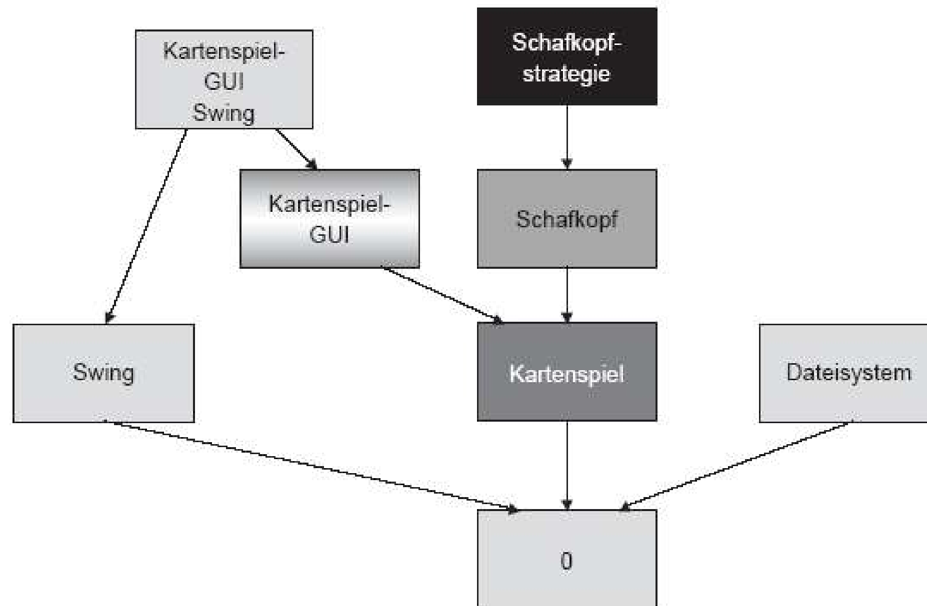

Abb. 6: Schafkopfen mit dem Computer [Sid04]

Im zweiten Schritt kann man die Komponenten anhand der Kategorien bilden und zuordnen.

In der Kategorie Schafkopf definieren wir die Komponente Schafkopfregeln, welche die Schnittstelle G implementiert. Sie antwotet auf die Frage, welche Karten man spielen darf. Die andere Komponente Schafkopfstrategie von der Kategorie Schafkopfstrategie, welche die Schnittstelle H implementiert, gibt die Antwort auf die Frage, welche Karte soll ich spielen.

Die beiden Schnittstellen G und H gehören zur Kategorie Kartenspiel, weil es im Allgemeinen nur um Kartenspiel nicht speziell um Schafkopf geht. Das heisst, man für ein anderes Kartenspiel dieselben Schnittstellen verwenden, dass man sie ganz anderes implementiert.

Die Komponenten *Schafkopfregeln* und *Schafkopfstrategien* informieren sich über die Schnittstelle A von der Komponente *Kartenspielinfo* der Kategorie *Kartenspiel*. Kartenspielinfo enthält den Spielzustand und die Historie.

Über die Schnittstelle D, die von *Kartenspielsteuerung* implementiert wird, geben die Spieler bekannt, welche Karten sie spielen; die *Kartenspielsteuerung* kann ein Spiel starten, unterbrechen und zurücksetzen.

Die Komponenten *realerSpieler* und *virtuellerSpieler* implementieren die Schnittstelle *E* mit der Operation: »Du bist dran, spiele aus«! Der virtuelle Spieler fragt die *Schafkopfstrategie,* was er machen soll, und antwortet sofort; beim realen Spieler geht die Kontrolle ans GUI. Der reale Spieler kann selbst entscheiden, was er spielt oder ebenfalls die Hilfe der *Schafkopfstrategie* in Anspruch nehmen.

Mit Hilfe der *Kartenspielverwaltung* kann man Spiele speichern, alte Spiele laden und erneut ablaufen lassen.

In der Kategorie *KartenspielGUI befinden sich die* Komponenten zur Präsentation von Kartenspiel, Verwaltung und Spieler. Diese beschaffen sich über die Schnittstellen *A*, *B* und *C* die zu präsentierenden Informationen und geben sie an eine Swingkomponente weiter.

Damit haben wir die Komponenten beschrieben, deren Kategorie von vornherein feststeht.

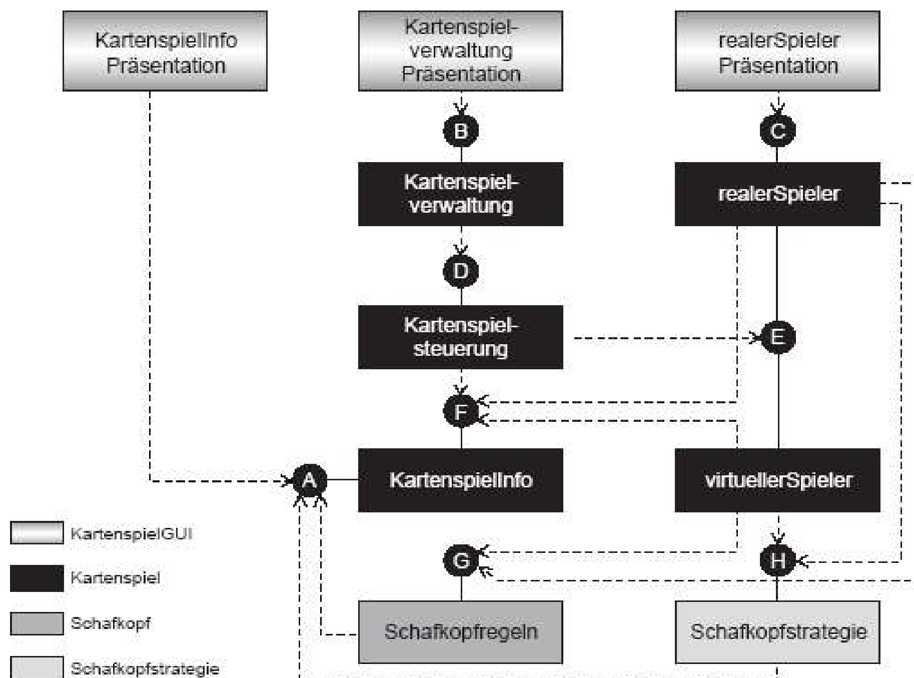

Abb. 7: Schafkopfen mit dem Computer [Sid04]

Architektur eines Informationssystems

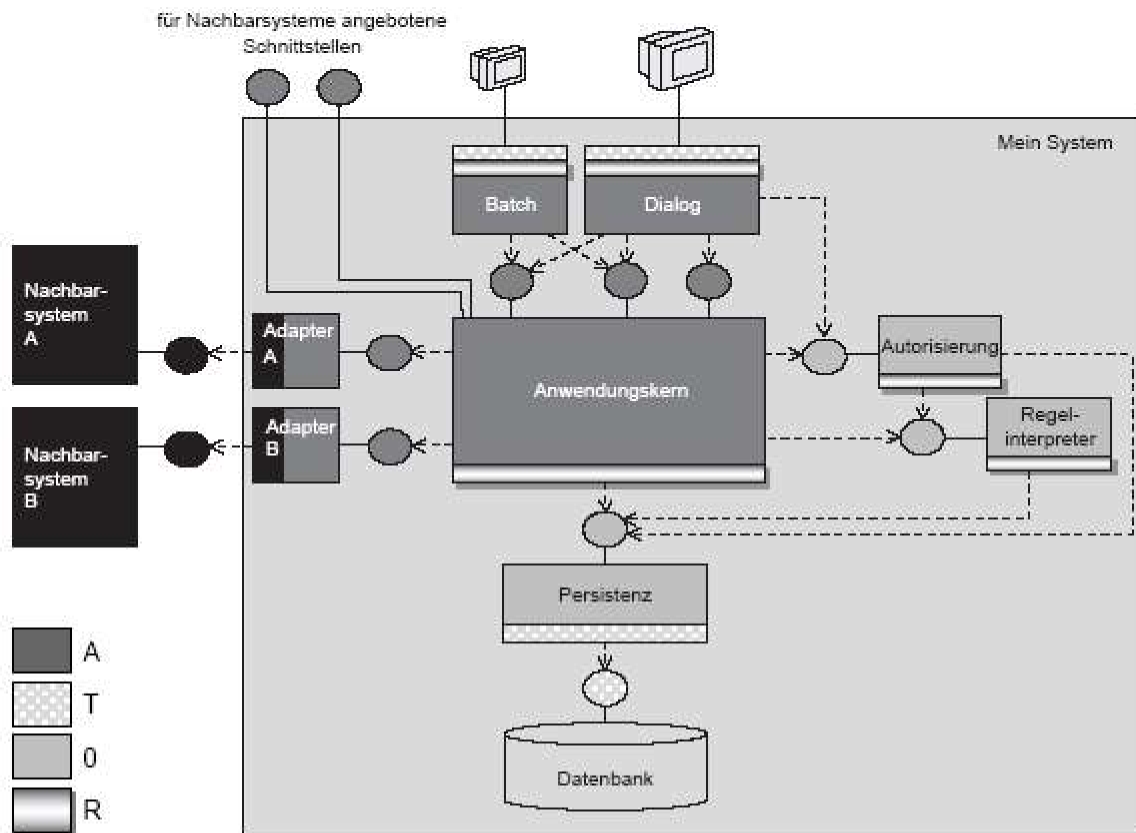

Abb. 8: Architektur eines Informationssystems [Sid04]

Informationssysteme sind nach dem drei Schichten-Prinzip aufgebaut. Das Beispielsystem importiert zwei Nachbarsystemschnittstellen . Damit man das System in Gang setzen kann, muss man erst den Zugang zu den Nachbarsystemen herstellen.

Das System exportiert eine GUI-Schnittstelle und den Administrationsdialog für den Stapelbetrieb, ausserdem zwei Schnittstellen zur Verwendung durch andere Nachbarsysteme.

Der Kern eines jeden Informationssystems ist der Anwendungskern. Er enthält die fachliche Intelligenz des Systems.

Man darf die Nachbarsysteme niemals direkt aufrufen, stattdessen definiert man für jedes Nachbarsystem eine Schnittstelle als fachliche Abstraktion. Das ist die Sicht des Anwendungskerns auf das Nachbarsystem. Diese Schnittstelle hat die Kategorie A und ein Adapter agiert als Mittler zwischen beiden Systemen.

Viele Systeme benutzen Standardkomponenten der Kategorie 0, weil sie technikfrei und keinen Bezug zu einer konkreten Anwendung besitzen. Im Beispiel sind dies eine Autorisierungs-komponente und ein Regelinterpreter.

Wir programmieren objektorientiert, aber wir speichern unsere Daten in relationalen Datenbanken. Die Persistenzkomponenten vermitteln zwischen objektorientierten und relationalen Welt. Allgemein exportiert eine Persistenzkomponente Schnittstellen der Kategorie 0. Neben dem Anwendungskern besitzen auch die Autorisierungs und Regelinterpreter persistenze Daten, und deshalb importieren alle drei die Persistenzschnittstelle.

Der Anwendungskern exportiert Anwendungsschnittstellen, und zwar Anwendungsfälle und Abfragen. Das sind die Schnittstellen, gegen die man Batch- und Dialogprogramme schreibt. In dieser Weise ist der Anwendungskern in beiden Umgebungen verwendbar.

4. Zusammenfassung

Quasar definiert ein Verständnis von Softwarequalität. Die Grundprinzipien von Quasar sind Trennung der Zuständigkeiten, Denken in Schnittstellen und Komponenten. Beim Finden von Komponenten und bei der Kontrolle von Abhängigkeiten sind die Softwarekategorien eine wichtige Hilfe.

Meiner Meinung nach ist der wichtigste Vorteil dieses Projekts die Einfachheit. Man kann mit einer Reihe technikneutraler Begriffe über Softwarearchitektur sprechen.

Dagegen ist Quasar aus wissenschaftlicher Sicht unfertig. Viele Definitionen sind unformell. Zum Beispiel muss der Wiederverwendbarkeitskonzept tiefer beschrieben werden.

Literaturangabe:

[Sid04] Johannes Siedersleben: Moderne Softwarearchitekturen, Dpunkt, 2004

[Sid02] Johannes Siedersleben: Softwaretechnik, Hanser Verlag, 2002

[BMRSS98] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: Pattern-orientierte Softwarearchitektur, Addison-Wesley, 1998