Software-Architekturmuster

Thomas Görtz, Stefan Macke, Patrick Peisker

Praktikum zum Modul Software-Technologie
Prof. Dr. Robra
WS 2007/2008

© 2008

Inhaltsverzeichnis

1 Einführung **1**

2 From mud to structure **3**
- 2.1 Layered 3
- 2.2 Subsumption 4
- 2.3 Pipes and Filters 6
- 2.4 Blackboard 8

3 Verteilte Systeme **9**
- 3.1 Distributed 9
- 3.2 Batch Sequential Systems 9
- 3.3 Repository-Centric Systems 10

4 Interaktive Systeme **13**
- 4.1 Model-View-Controller 13
- 4.2 Rule-Based Systems 14
- 4.3 Frame-Based Systems 16

5 Anpassbare Systeme **18**
- 5.1 Event-Driven Systems 18
- 5.2 Interpreter 19
- 5.3 Disposable 19

1 Einführung

Software-Architekturen beschreiben das höchste Maß der Abstraktion einer Software. Sie bilden ihre fundamentale Struktur und sind damit Grundlage des gesamten Designprozesses.

Muster (*patterns*) für Software-Architekturen lassen sich (grob) in vier Gruppen einteilen:

From mud to structure Diese Muster dienen der Partitionierung von Systemen. Charakteristisch sind eine unüberschaubare Menge von Anforderungen, die in geeignete Subsysteme unterteilt werden müssen. Das Ziel des Entwurfs ist nicht nur ein funktionierendes System, sondern ein System, welches auch in Zukunft noch modifizierbar bleibt.

Vorgestellte Patterns: Layers, Subsumption, Pipes and Filters, Blackboard

Verteilte Systeme Immer mehr Systeme verwenden mehr als nur eine CPU in einem Rechner oder gar mehrere eigenständige Rechenknoten, um ihre Aufgaben zu erfüllen. Gründe hierfür sind:

- Performance und Skalierbarkeit
- Verlässlichkeit
- Ökonomische Gründe und problemtypische Verteiltheit

Vorgestellte Patterns: Distributed, Batch Sequential Systems, Repository-Centric Systems

Interaktive Systeme Ziel dieser Systeme ist es, eine möglichst ergonomische Schnittstelle zum Benutzer bereitzustellen. Anwender sollen den Umgang mit dem System intuitiv und relativ schnell erlernen. Der eigentliche Systemkern, der die Funktionalität beinhaltet, bleibt von der Benutzerschnittstelle unangetastet.

Vorgestellte Patterns: Model-View-Controller (MVC), Rule-Based Systems, Frame-Based Systems

Anpassbare Systeme Systeme entwickeln sich mit der Zeit. Neue Funktionen werden hinzugefügt und existierende verändert. Anpassbare Systeme müssen auf andere Umgebungen portierbar sein und sollten problemlos auf geänderte Kundenwünsche reagieren können. Deswegen sollten Anwendungen von Beginn an auf Modifizierbarkeit hin entworfen werden.

Vorgestellte Patterns: Event-Driven Systems, Interpreter, Disposable

Quellen

- Kaisler, Stephen (2005). *Software Paradigms.* Hoboken, USA: John Wiley & Sons.
- Hübsch, Chris (2000). *Ausdrucksmittel für Systementwürfe und Entwurfsmuster.* Chemnitz: Technische Universität Chemnitz.

2 From mud to structure

2.1 Layered

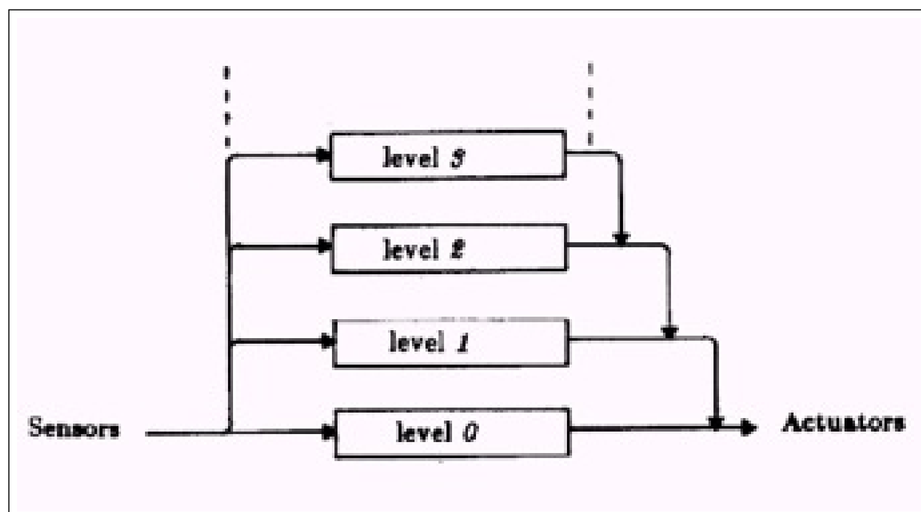
Das System wird realisiert als aufeinander aufbauende Schichten. Jede Schicht bietet Dienste an und verwendet die Dienste der untergeordneten Schicht. Die Konnektoren sind die Protokolle der einzelnen Schichten. Eine mögliche Variante würde den direkten Zugriff auf alle tiefer liegenden Schichten erlauben, allerdings würde die höhere Kopplung den Vorteil der leichten Austauschbarkeit einer Schicht kaputt machen. Ein weiterer Vorteil ist der Aufbau aus Schichten mit zunehmender Abstraktion. Dies führt zu starker Wiederverwendbarkeit: Aufbau einer Schicht auf anderen Schichten (z.B. FTP, Telnet, HTTP auf TCP/IP). Änderungen betreffen höchstens die beiden benachbarten Schichten. Performance-Verluste hat man beim Durchreichen von Information von einer Schicht zu einer nicht direkt benachbarten Schicht. Dies macht eine Abgrenzung und Aufgabenzuordnung der Schichten schwierig. Ein weiteres Einsatzgebiet findet die Architektur bei Netzwerkprotokollen, wie beispielsweise dem ISO-OSI-7-Schichten-Modell.

Quellen

- Wallwitz, Fabian (2001). *Objektorientierte Architekturen, Frameworks und Architekturmuster*. Universität Stuttgart.[1]

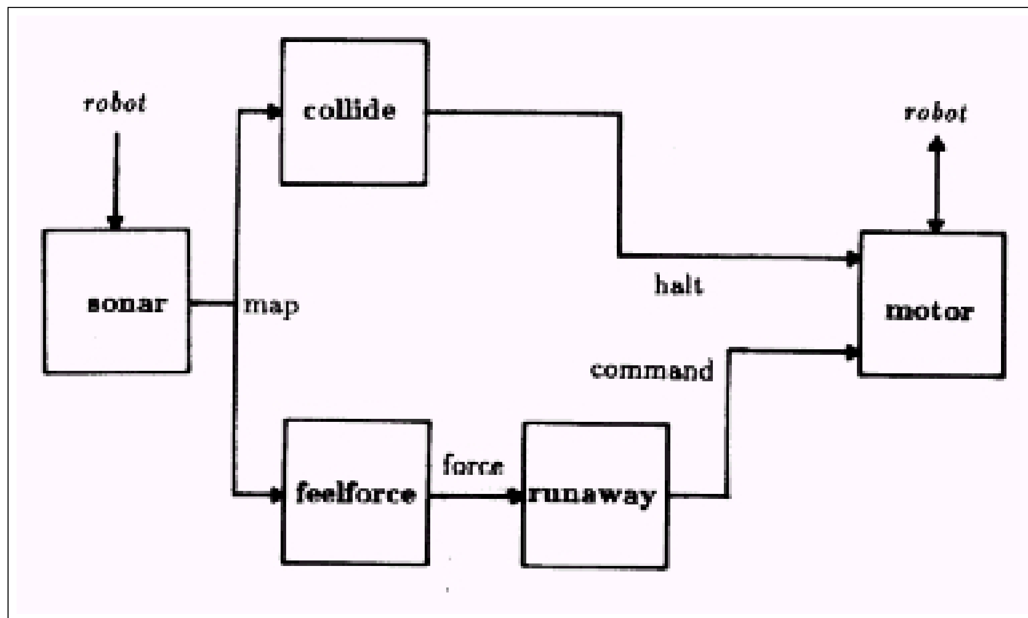
2.2 Subsumption

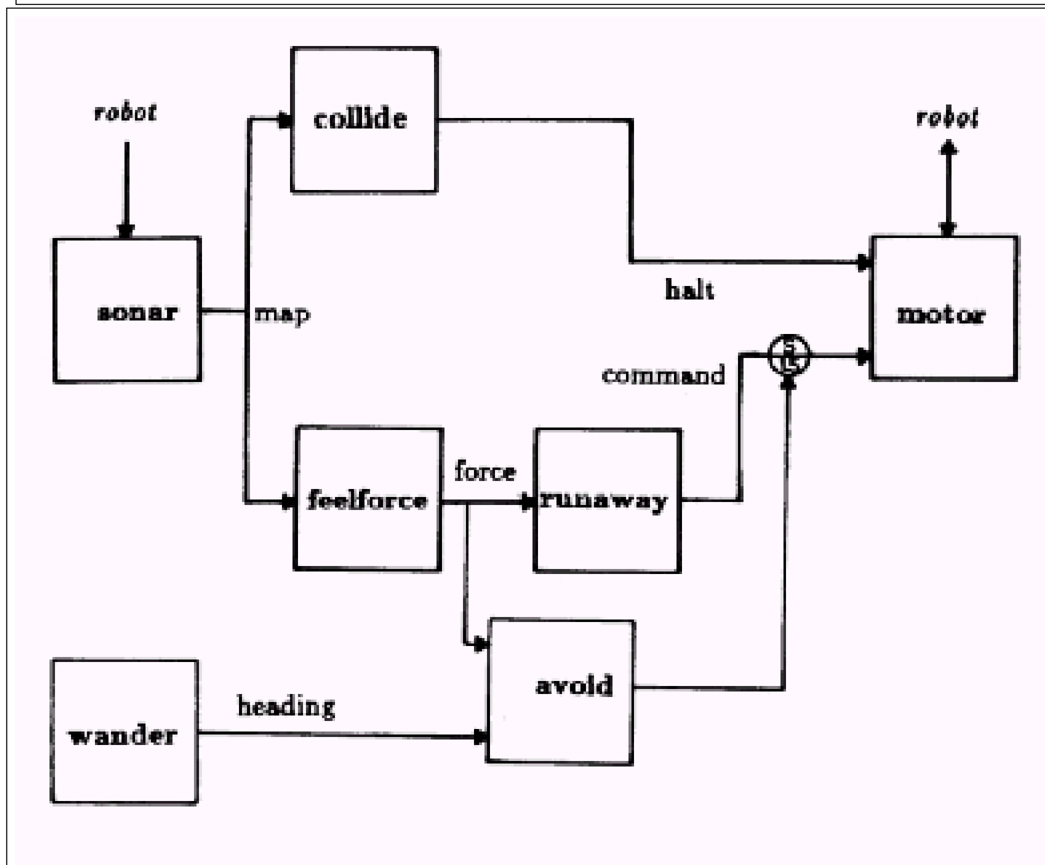
Das Subsumption-Pattern wird bei der Entwicklung von Robotern verwendet. Es besteht aus mehreren Schichten, die aufeinander aufbauen. Dabei werden von den unteren Schichten zunächst die elementaren Funktionen wahrgenommen (wie z.B. Hindernis ausweichen), die dann in den darüberliegenden Schichten um immer komplexere Funktionen ergänzt werden. Die Schichten verarbeiten dabei asynchron die Signale der Sensoren des Roboters, sodass unter anderem ein Ausfall einer höheren Schicht keine Auswirkungen auf darunterliegende Schichten hat.



Die oberen Schichten können das Verhalten der unteren Schichten bewusst steuern (mittels *suppression*- und *inhibition*-Signalen), indem sie gezielt Daten unterdrücken oder ändern. So kann der Roboter z.B. von Schicht 1 angewiesen werden, ein Hindernis zu umgehen, während Schicht 0 normalerweise einfach die entgegengesetzte Richtung einschlagen würde, um dem Hindernis auszuweichen.

[1] http://www.iste.uni-stuttgart.de/ps/Lehre/HS_OO_Entwurf/Wallwitz.pdf





Die einzelnen Schichten sind als Netzwerk endlicher Automaten implementiert, die um Timer ergänzt werden, um Aktionen nach einer bestimmten Zeit ausführen zu können. Durch das schnelle Verarbeiten der Sensorsignale durch die fest program-

mierten endlichen Automaten kann der Roboter in Echtzeit auf seine Umwelt reagieren. Er reagiert quasi reflexartig auf äußere Änderungen, wie dies auch Insekten tun, die das Vorbild für diese Architektur sind. Leider ist der Roboter in der Subsumption-Architektur auf diese Reflexe beschränkt, kann sich also kein internes Abbild seiner Umwelt erstellen, was für intelligenteres Verhalten nötig wäre.

Quellen

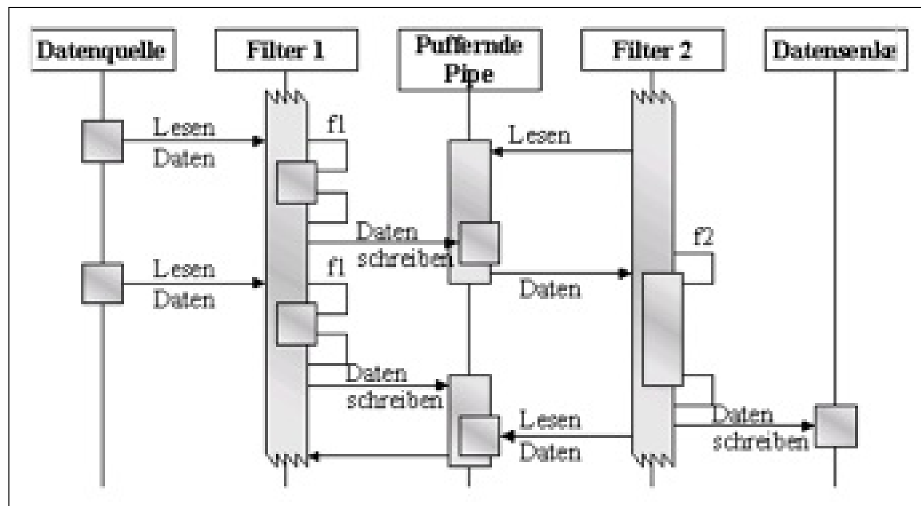
- Bergmann, Kai (1998). *Ansätze von Rodney Brooks, Subsumption Architecture.*[2]
- *Subsumption architecture with Robolab 2.5.*[3]

2.3 Pipes and Filters

Durch Pipes and Filters kann die Aufgabe eines Systems als Kombination von einzelnen sequenziellen Teilschritten (Filter) implementiert werden. Ein Datenstrom fließt dabei durch die *Pipeline* zwischen den einzelnen Filtern und wird Schritt für Schritt weiterverarbeitet. Jeder Filter übernimmt dabei genau eine Aufgabe, die er unabhängig von den anderen Filtern umsetzt (die Filter wissen nicht, dass andere Filter exisiteren). Die Ausgabedaten des einen Filters (Datenquelle) sind dabei gleichzeitig die Eingabedaten des folgenden Filters (Datensenke). Dabei erfolgt eine kontinuierliche Verarbeitung des Datenstroms ohne Zwischenspeicherung.

Filter können im Wesentlichen drei Aktivitäten mit den Daten durchführen: Anreichern, Verfeinern und Umwandeln. Man unterscheidet zwischen aktiven und passiven Filtern. Erstere laufen in eigenen Prozessen/Threads, letztere werden erst zur Laufzeit aktiviert (mittels Push- oder Pull-Mechanismus). Aktive Filter müssen von einer Pipe synchronisiert werden (z.B. durch Pufferung der Daten), während bei passiven Filtern die Pipe eigentlich nicht erkennbar ist (s.u.: Unix-Shell).

[2] http://www.igi.tugraz.at/STIB/WS98/Bergmann/einleitung.htm
[3] http://www.convict.lu/Jeunes/Subsumption.htm



Beispiele Unix-/Linux-Shell: Text-Stream in der Pipeline

```
1 cat LocalSettings.php | grep "^ *[#|//]" | wc -l
```

(Anzahl Kommentarzeilen in PHP-Datei ausgeben)

```
1 du | sort -rn | head -5
```

(Die fünf größten Dateien ausgeben)

```
1 ps -e | grep " p" | awk '{ print $1 }' | xargs kill
```

(Alle Prozesse stoppen, die mit *p* beginnen)

Beispiele Microsoft PowerShell: *Echte* Objekte in der Pipeline

```
1 Gwmi Win32_Service | where {$_.State ?eq "Running" } | select Name,
    StartMode
```

(Name und Startmodus aller laufenden Dienste ausgeben)

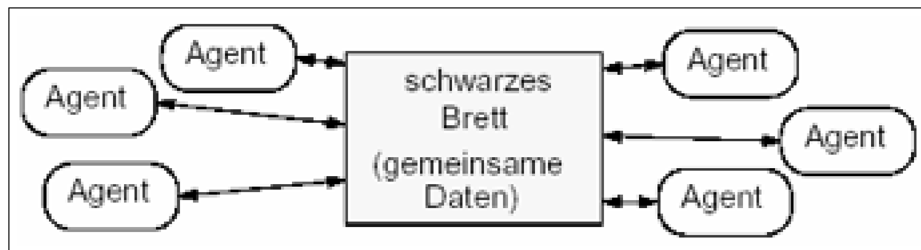
```
1  get−process p∗ | stop−process
```

(Alle Prozesse stoppen, die mit *p* beginnen)

Quellen

- Krutscher, Andreas (1997). *Pipes und Filter.*[4]
- Avgeriou, Paris (2005). *Architectural Patterns Revisited – A Pattern Language.*[5]

2.4 Blackboard



Das Schwarzes Brett dient als zentrale Datenstruktur. *Agenten* verarbeiten vorhandenes und bringen neues Wissen. Eine Steuerung entscheidet, welcher Agent die Bedingung zum Ermitteln von neuem Wissen erfüllt und somit das Programm der Lösung einen Schritt näher bringen könnte, dann aktiviert es den Agenten. Die Zugriffe von Agenten auf das schwarze Brett stellen die Konnektoren da. Die Agenten sind völlig entkoppelt und können auch zur Laufzeit hinzugefügt und ausgetauscht werden, ohne dass andere Agenten betroffen sind. Die parallele Ausführung von Agenten ist ebenfalls möglich. Das Programmverhalten von Systemen, für die solch eine Architektur eingesetzt wird, ist hochgradig nichtdeterministisch und daher schwer prüfbar. Im Bereich Robotersteuerung und Mustererkennung (Bild, Ton, Sprache, Schrift) wird aufgrund der nichtdeterministischen Problemlösung die Black Board Architektur häufig verwendet.

Quellen

- Wallwitz, Fabian (2001). *Objektorientierte Architekturen, Frameworks und Architekturmuster.*[6]

[4] http://www.fh-wedel.de/~si/seminare/ws97/Ausarbeitung/3.Krutscher/archmu1.htm
[5] http://www.infosys.tuwien.ac.at/Staff/zdun/publications/ArchPatterns.pdf
[6] http://www.iste.uni-stuttgart.de/ps/Lehre/HS_00_Entwurf/Wallwitz.pdf

3 Verteilte Systeme

3.1 Distributed

Beim Distributed Computing werden einzelne Komponenten einer Software auf unterschiedliche (verteilte) Systeme ausgelagert. Solche Systeme können über ein Netzwerk (wie das Internet) verbunden, aber auch innerhalb eines physikalischen Systems verteilt sein (mehrere Prozessoren). Zu den Vorteilen dieser Architektur zählen die hohe Ausfallsicherheit, wenn gleiche Komponenten auf mehreren Systemen laufen, und ebenso die sehr gute Lastverteilung. Werden von den einzelnen Komponenten möglichst einfache Dienste angeboten, erhöht sich auch deren Wiederverwendbarkeit (Extrembeispiel: SOA). Nachteile sind z.B. der Overhead für die Kommunikation und Synchronisation der Komponenten untereinander oder der Umgang mit ausgefallenen Komponenten.

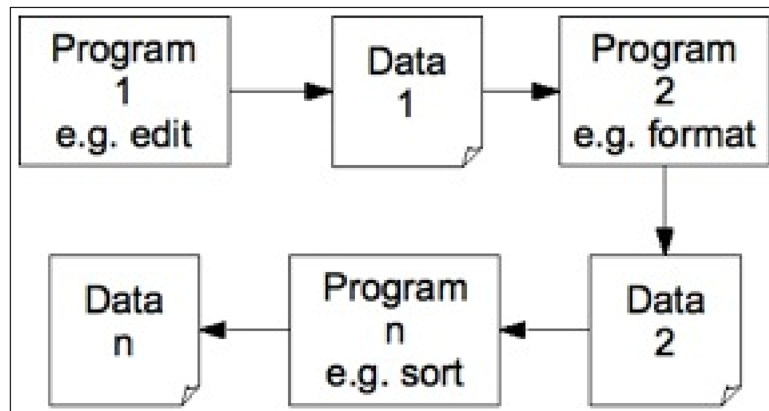
Quellen

- Portland Pattern Repository (2007). *Distributed System.*[7]
- Portland Pattern Repository (2007). *Distributed Computing.*[8]

3.2 Batch Sequential Systems

Batch Sequential Systems sind vergleichbar mit dem *Pipes and Filters*-Muster. Diese Systeme bestehen aus einer Sammlung von Programmen, die mehr oder weniger linear angeordnet sind und sequentiell ausgeführt werden. Das zu lösende Problem wird in verschiedene Stufen (Stages) zerlegt. Jeder Schritt wird zu Ende ausgeführt (*Run to Completion*) bevor der nachfolgende gestartet wird. Die Steuerung der Programmabfolge wird meistens durch eine Skript-Sprache des verwendeten Betriebssystems realisiert (z.B. Unix Shell). Daten zwischen den einzelnen Stufen werden durch Dateien ausgetauscht.

[7] http://c2.com/cgi-bin/wiki?DistributedSystem
[8] http://c2.com/cgi-bin/wiki?DistributedComputing



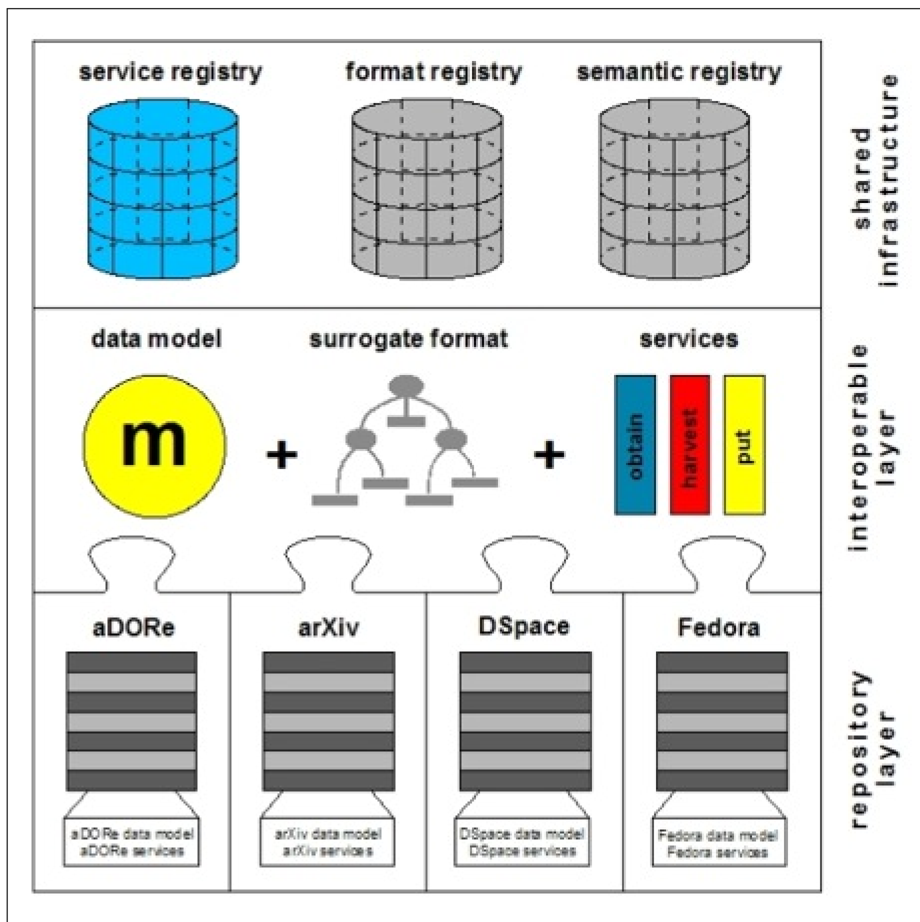
Quellen

- Kaisler, Stephen (2005). *Software Paradigms*. Hoboken, USA: John Wiley & Sons.
- Hübsch, Chris (2000). *Ausdrucksmittel für Systementwürfe und Entwurfsmuster*. Chemnitz: Technische Universität Chemnitz.

3.3 Repository-Centric Systems

Softwaresysteme mit großem Umfang haben natürlicherweise eine große Anzahl an Komponenten, welche eine große Anzahl an Daten-Artefakten erzeugen. Eine Repository-zentrierte Architektur kann hierbei mittels eines oder mehrerer zentraler Datenlager (Repositories), die Daten-Artefakte konsistent halten. Dienstschichten sorgen dabei für den Zugriff.

Repository-Centric System mit mehreren Repositories im Layered Pattern aufgebaut.



Quellen

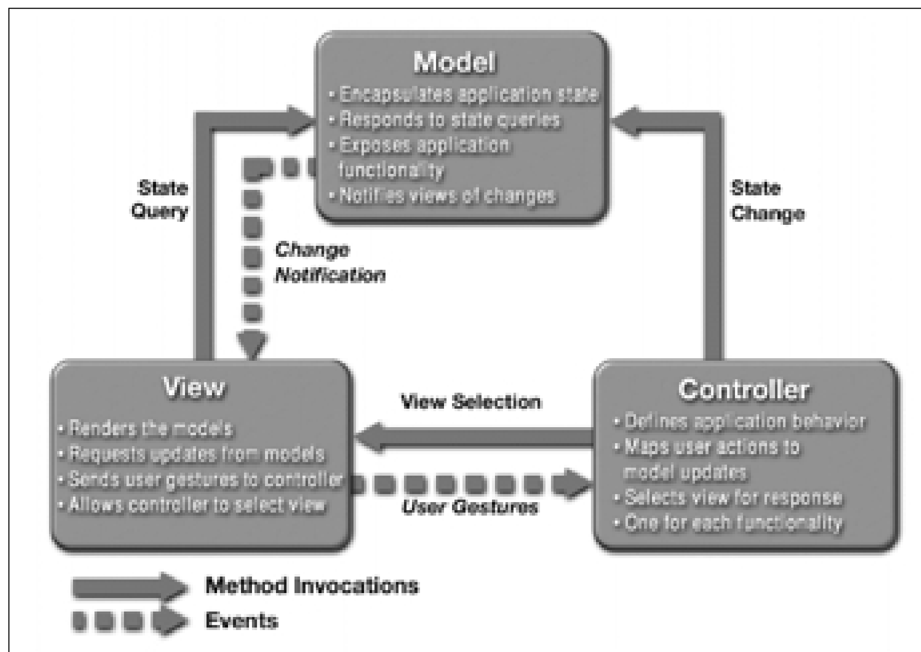
- D-Lib Magazine (2006). *An Interoperable Fabric for Scholarly Value Chains.*[9]

[9] http://www.dlib.org/dlib/october06/vandesompel/10vandesompel.html

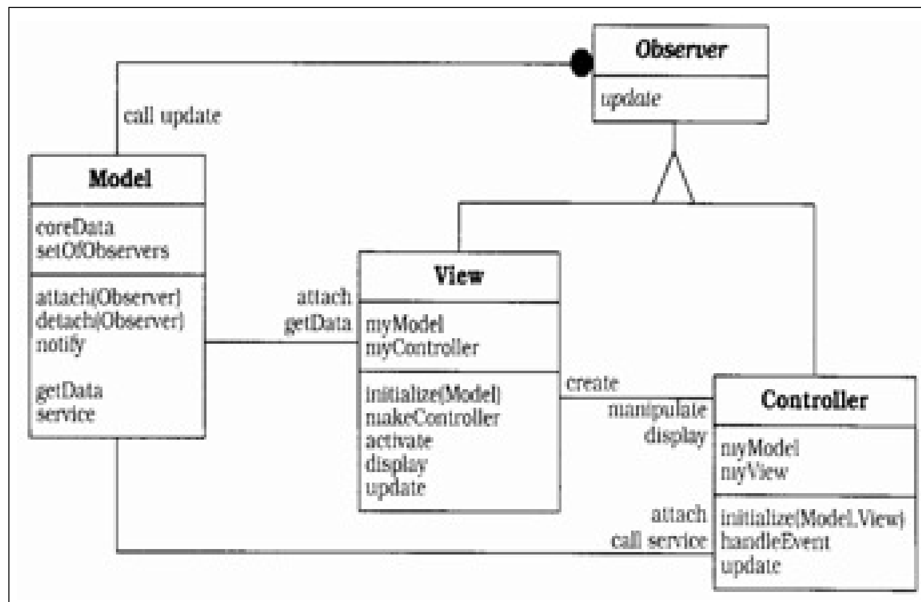
4 Interaktive Systeme

4.1 Model-View-Controller

Das MVC-Pattern ist eine spezielle Variante des Layers-Pattern, die sich aus den drei Schichten Datenhaltung (Model), Programmlogik (Controller) und Präsentation (View) zusammensetzt. Das Model ist dabei für die Speicherung der Daten zuständig und stellt Zugriffsmethoden für diese bereit, der/die View(s) übernimmt die Darstellung der Daten und der Controller regelt, welcher View aufgerufen wird und wie Datenmanipulationen am Model vorgenommen werden.



Die Schichten sind mittels des Observer-Patterns lose gekoppelt, wodurch die einzelnen Komponenten leicht austauschbar werden. Durch die strikte Trennung der Präsentation von den übrigen Schichten, ist es möglich, auf dieselbe Anwendung über unterschiedliche Clients (etwa GUI, Webinterface oder XML) zuzugreifen, ohne die Programmlogik anpassen zu müssen.



Das MVC-Pattern findet insbesondere in der Webentwicklung Anwendung (z.B. Ruby on Rails), wo es oft durch ein weiteres Pattern, den *Front Controller* erweitert wird, einen zentralen Controller, der sämtliche Anfragen an die Anwendung bearbeitet. Da Controller und View meistens eng gekoppelt sind (z.B. Ereignissteuerung der GUI) werden sie häufig als eine Einheit gesehen. Diese Abwandlung des MVC-Patterns wird Document-View-Pattern genannt.

Quellen

- Sun Microsystems (2002). *JavaBlueprints Model-View-Controller.*[10]
- Wallwitz, Fabian (2001). *Objektorientierte Architekturen, Frameworks und Architekturmuster.*[11]

4.2 Rule-Based Systems

Regelbasierte Systeme (*Rule-Based Systems*, RBS) sind historisch eine der ersten Formen von KI-Systemen. Sie bestehen aus:

- Einer Datenbank von Fakten (*Facts*), der Faktenbasis. Der Arbeitsspeicher dient als *Kurzzeitgedächtnis* (Short-Term Memory Buffer) und enthält eine

[10] http://java.sun.com/blueprints/patterns/MVC-detailed.html
[11] http://www.iste.uni-stuttgart.de/ps/Lehre/HS_OO_Entwurf/Wallwitz.pdf

Menge von Grundaussagen, die den momentanen Zustand des modellierten Weltausschnitts beschreiben.

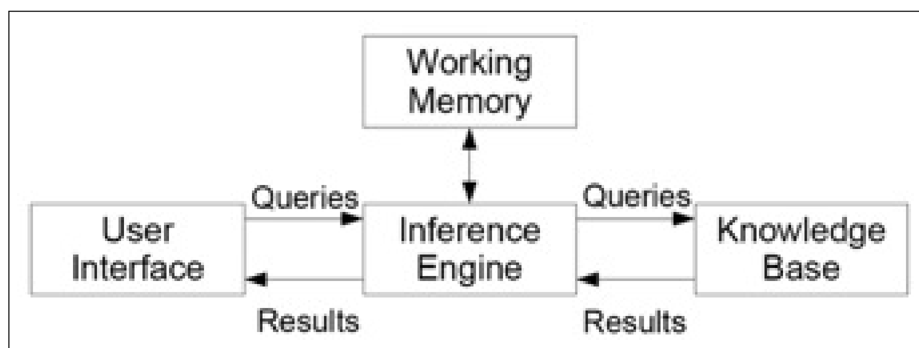
- Einer Menge von Regeln (*Rules*), der Regelbasis. Die Anwendung einer Regel verändert im allgemeinen den Inhalt des als *Kurzzeitgedächtnis* bezeichneten Arbeitsspeichers.
- Einem Kontrollsystem mit Regelinterpreter (Interpreter).

Regeln liegen in der Form: *WENN ... DANN ... SONST (IF THEN ELSE)* vor.

Beispiel *WENN* Herdplatte heiß *UND* kein Topf auf Herd *DANN* schalte Herd aus.

Der *WENN*-Teil der Regel heisst Prämisse, der *DANN*-Teil Konklusion. Eine einzelne Regel stellt eine Wisseneinheit dar und eine Menge von Regeln zusammen mit einer Ausführungsstrategie stellt eine Art von Programm für die Lösung eines Problems oder einer Problemklasse dar.

Die Aufgabe des Kontrollsystems ist die Identifikation geeigneter Regeln, das Anwenden dieser, sowie die Aktualisierung der Datenbank. Auswahlmechanismen für die nächste anzuwendende Regel sind entweder datengetrieben (Vorwärtsverkettung, *Forward Chaining*), zielgetrieben (Rückwärtsverkettung, *Backward Chaining*) oder eine Kombination dieser beiden Möglichkeiten. Der Benutzer erhält Zugriff auf das System durch eine Benutzeroberfläche mit vordefinierter Syntax.



Quellen

- Henke, Friedrich von (2004). *Einführung in die Künstliche Intelligenz*. Ulm: Universität Ulm.
- Wikipedia (12/2007). *Regelbasiertes System*.[12]

[12] http://de.wikipedia.org/wiki/Regelbasiertes_System

4.3 Frame-Based Systems

Frame-Based Systems sind Wissensbasierte Systeme die *Frames* benutzen. Die Idee wurde 1975 in einem Seminarpapier mit dem Namen *A framework for representing knowledge* von Marvin Minsky veröffentlicht. Die Verwendung von Frames für Wissensbasierte Systeme galt damals als Gegenströmung zu verwendeten Formalismen der Regelbasierten Systemen.

Frames sind Konstrukte zur Wissensrepräsentation. Sie erfassen (clustern) alle relevanten Informationen eines Konzepts oder einer Situation. Frames können als *Objekte ohne Methoden* angesehen werden. Sie besitzen eine hierarchische Vererbungsstruktur, Attribute (Slots) und übergeordnete Frames können ihre aktuellen Slot-Werte (Filler), an untergeordnete Frames vererben.

Beispiel Frames eines Stammbaums:

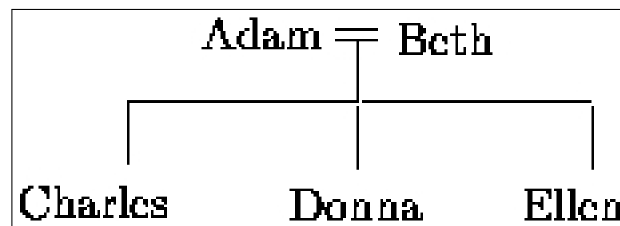
Adam:
Geschlecht: M
Ehepartner: Beth
Kind: (Charles Donna Ellen)

Beth:
Geschlecht: F
Ehepartner: Adam
Kind: (Charles Donna Ellen)

Charles:
Geschlecht: M
Ehepartner: ()
Kind: ()

Donna:
Geschlecht: F
Ehepartner: ()
Kind: ()

Ellen:
Geschlecht: F
Ehepartner: ()
Kind: ()



Quellen

- Nebel, Bernhard (1999). *Frame-Based Systems*. Cambridge, USA: Massachusetts Institute of Technology.
- Hewett, Michael (1996). *Introduction to Frame Based Knowledge Representation*. Austin, USA: University of Texas.
- Wikipedia (01/2008). *Frames*.[13]

[13] http://de.wikipedia.org/wiki/Frames

5 Anpassbare Systeme

5.1 Event-Driven Systems

Ereignisgesteuerte Systeme oder auch ereignisbasierende Systeme (*Event-Driven Architecture*, EDA) basieren auf dem Zusammenspiel der Komponenten durch Nachrichten (Events). Diese Architektur ermöglicht die Kommunikation verschiedener Threads oder Prozesse in lose-gekoppelten Systemen, wie z.B. in einer SOA. Ereignisse können intern (z.B. Änderungsbenachrichtigungen) oder extern (z.B. Benutzereingaben) erzeugt werden. Die Verarbeitung von Ereignissen wird Ereignisbehandlung (Event Handling) genannt.

Ereignisse beinhalten im Nornmalfall Angaben über die Quelle und Art des Ereignisses (Typ) sowie je nach Anwendungfall einen Zeitstempel und eine eindeutige ID. Ereignisse sind keine Nachrichten, die zielgerichtet an bestimmte Komponenten versandt werden. Sie können in einer oder mehreren Stellen im System Aktionen auslösen, aber auch völlig unbeachtet verworfen werden.

Feste Bestandteile einer EDA sind eine Ereignis-Quelle (Event Source), ein Ereignis-Vermittler (Event Dispatcher) und ein Ereignis-Verarbeiter (Event Handler)

Quellen

- Wikipedia (12/2007). *Ereignisgesteuerte Architektur.*[14]
- OSE (12/2007). *Event Driven Systems.*[15]
- Wikipedia (12/2007). *Event Driven Architecture.*[16]

[14] http://de.wikipedia.org/wiki/Ereignisgesteuerte_Architektur
[15] http://ose.sourceforge.net/browse.php?group=library-manual&entry=events.htm
[16] http://en.wikipedia.org/wiki/Event_Driven_Architecture

5.2 Interpreter

Ein Interpreter wird verwendet, um zur Laufzeit eine Sprache (mit eigener Syntax und Grammatik) zu interpretieren und in Funktionsaufrufe umzuwandeln. Der Interpreter parst die Sprachanweisungen (Scripts) und führt diese aus. Dies geschieht, indem der Interpreter die Anweisungen in die für seine Plattform gültigen Funktionsaufrufe übersetzt. Die Scripts sind somit auf verschiedenen Systemen lauffähig, sofern es für diese einen Interpreter gibt. Somit stellt der Interpreter eine Abstraktion der tatsächlichen Plattform dar (ähnlich einer virtuellen Maschine).

Das Interpreter-Pattern ist eigentlich ein Entwurfsmuster, kann aber auch als Architekturmuster verstanden werden, wenn es in einem größeren Zusammenhang, nämlich dem der Softwarearchitektur, gesehen wird. Wie erwähnt stellt es hier eine Abstraktionsschicht über dem Betriebssystem und der Hardware dar.

Quellen

- Avgeriou, Paris (2005). *Architectural Patterns Revisited – A Pattern Language.*[17]

5.3 Disposable

Das Disposable Pattern beschreibt eine deterministische Methode zur Objekt-Deinitialisierung in Softwaresystemen ohne starke Typbindung mit automatischer Speicherbereinigung (Garbage Collection).

Die automatische Speicherverwaltung dient in Softwaresystemen dazu, den Speicher nicht mehr referenzierter Datenobjekte freizugeben. Zudem können Objekte durch die sogenannte Finalisierung mittels der automatischen Speicherbereinigung deinitialisiert werden. Dies geschieht asynchron.

Die Finalisierung birgt aber folgende Probleme:

Die Finalisierung ist nicht deterministisch. Es gibt keine definierte Finalisierungsreihenfolge. Daher kann es geschehen, dass während der Finalisierung auf andere Objekte zugegriffen wird, die ebenfalls der Finalisierung unterworfen sind, zu diesem Zeitpunkt aber überhaupt nicht mehr existieren. Es gibt je nach Implementierung (z.B. in der Programmiersprache Java) keine Garantie dafür, dass die Finalisierungsroutine von der automatischen Speicherbereinigung überhaupt aufgerufen wird. An

[17] http://www.infosys.tuwien.ac.at/Staff/zdun/publications/ArchPatterns.pdf

dieser Stelle kommt das Disposable Pattern zum Tragen. Im Gegensatz zur asynchronen Finalisierung ist das Disposable Pattern deterministisch.

Am Beispiel von C# wird mittels des Shortcuts using am Ende des Gültigkeitsbereichs eines Objekts die Dispose Methode aufgerufen:

```
1  static void Main(string[] args)
2  {
3     using (StreamWriter writer = new StreamWriter(File.OpenWrite("Test.txt")))
4     {
5        writer.Write("Some value");
6     }
7  }
```

Der C# Kompiler erweitert das zu folgendem Code:

```
1  static void Main(string[] args)
2  {
3     StreamWriter writer = new StreamWriter(File.OpenWrite("Test.txt"));
4  
5     try {
6        writer.Write("Some value");
7     }
8     finally {
9        writer.Dispose();
10    }
11 }
```

Die Dispose Methode muss in diesem Fall überschrieben werden, damit die Finalisierung der automatischen Speicherbereinigung nicht zuschlägt und versucht das freigegebene Objekt ein weiteres Mal zu frei zu geben.

```
1  public class MyDisposable : IDisposable
2  {
3     public void Dispose()
4     {
5        Dispose(true);
```

```
6     }
7
8     ~MyDisposable()
9     {
10      Dispose(false);
11    }
12
13    private void Dispose(bool disposing)
14    {
15    // Wird die Dispose Methode aufgerufen, muss die Finalisierung unterdückt
          werden.
16    if (disposing)
17      GC.SuppressFinalize(this);
18
19      // Jeden ungenutzen Speicher freigegben
20      // ...
21    }
22  }
```

Quellen

- Laban, Jerome (2007). *The Disposable Pattern, Determinism in the .NET World.*[18]
- Wikipedia (12/2007). *Automatische Speicherbereinigung.*[19]

[18] http://blogs.labtech.epitech.net/blogs/jaylee/archive/2004/08/16/844.aspx
[19] http://de.wikipedia.org/wiki/Automatische_Speicherbereinigung